Deploying Concurrent Applications on Heterogeneous Multiprocessors

Andrew Christopher Mihal

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-145
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-145.html

November 10, 2006

Deploying Concurrent Applications on Heterogeneous Multiprocessors

by

Andrew Christopher Mihal

B.S. (Carnegie Mellon University) 1999

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Kurt Keutzer, Chair
Professor Edward Lee
Professor Robert Jacobsen

Fall 2006

The dissertation of Andrew Christopher Mihal is approved:

_____

Chair Date

_____

Date

_____

Date

University of California, Berkeley

Fall 2006

Deploying Concurrent Applications on Heterogeneous Multiprocessors

Andrew Christopher Mihal

Abstract

Deploying Concurrent Applications on Heterogeneous Multiprocessors

by

Andrew Christopher Mihal

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kurt Keutzer, Chair

The now-common phrase “the processor is the NAND gate of the future” begs the questions: “What kind of processor?” and “How to program them?” For many the presumption is that the natural building block is a common and successful general-purpose RISC processor and that programming will be done in C. Numerous programmable multiprocessor systems of this type have been built, but they are notoriously hard to program.

What makes programming hard? One proposal is that poor methodologies for modeling and implementing concurrency are to blame. Embedded applications have multiple styles of concurrency on several levels of granularity. In order to meet performance goals, all of this concurrency must be considered in the implementation. C does not provide good abstractions for concurrency. This forces designers to use ad hoc techniques to divide the application into a set of interacting C programs that exploit the architecture’s capabilities. This manual approach is slow and error-prone, and it inhibits effective design-space exploration.

Domain-specific languages are successful at modeling concurrency more formally. This improves productivity, but it also exposes a more fundamental problem. For many programmable multiprocessors, there is an inherent mismatch between the concurrency capabilities of the architecture and the concurrency requirements of the application. This problem is the *concurrency implementation gap*. If programmable processors are to find success in future systems, designers must rethink how they design architectures in addition to the way they program them.

This dissertation takes a three-part approach to this challenge. Designers use separate abstractions for modeling applications, capturing the capabilities of architectures, and mapping applications onto architectures. Models of computations and actor-oriented design provide a mechanism

for correctly modeling concurrent applications. The Sub-RISC paradigm provides efficient, cost-effective architectures and exports their capabilities. A disciplined mapping methodology leads to correct implementations and enables effective design space exploration. With this approach, designers can solve the concurrency implementation gap and realize the benefits of programmable multiprocessors.

_____

Professor Kurt Keutzer
Dissertation Committee Chair

ID: PA092249	891226

UNDER PENALTY OF LAW THIS TAG NOT TO BE REMOVED EXCEPT BY THE CONSUMER

This dissertation contains

65 % NEW RESEARCH
20 % FRESH RESULTS
10 % RECYCLED RESEARCH
5 % DUCK FEATHERS

CONTENTS STERILIZED

Reg. No. 012986★ Permit No. 070650(NJ)

Certification is made by the manufacturer that the materials in this dissertation are described in accordance with law.

Distributed by:
Andrew Mihal
545Q Cory Hall
UC Berkeley
Berkeley, CA 94720

IL # 19800721 MADE IN USA

Contents

List of Figures

List of Tables

Acknowledgments

Without the fellowship and advice of my colleagues, this dissertation would not be possible. For listening to my ideas, critiquing presentations, co-authoring papers, and making the office such an enjoyable place to work, I would like to thank: Christopher Brooks, Adam Cataldo, Bryan Catanzaro, Elaine Cheong, Jike Chong, Thomas Feng, Matthias Gries, Jörn Janneck, Yujia Jin, Matt Moskewicz, Steve Neuendorffer, “Squire Wills” Plishker, Kaushik “The Destroyer” Ravindran, N. R. Satish, Christian Sauer, Niraj Shah, Martin Trautmann, Kees Vissers, and Scott Weber. I would like to thank Herman Schmit for helping me choose the right graduate school. I thank my committee, Kurt Keutzer, Edward Lee, Robert Jacobsen, and Jan Rabaey for sharing their experience and wisdom with me.

Chapter 1

A Crisis in System Design

The history of electronic system design methodologies is marked by design discontinuities. A design discontinuity occurs when an established methodology is replaced by a new methodology. Designers wish to create more complex systems, and they find that the new methodology allows them to handle the increased complexity with greater productivity and provides better quality of results.

Figure 1.1 shows the major design discontinuities to date. The life cycles of individual methodologies are characterized by the s-shaped curves found in the figure. Early in the cycle, a methodology is a work-in-progress that is not well supported by tools. Consequently, its productivity is seen as lower than existing methodologies. This is the bottom of the s-curve. As the methodology gains traction and designers begin to accept the advantages of the new approach, a period of rapid growth begins. Tools that support the methodology are actively developed. Eventually, system complexity reaches the limits of the tools, and the underlying methodology is not able to provide answers to these problems. This is the top of the s-curve. Designers start to experiment with alternative methodologies, even if the corresponding tools are immature.

This cycle has repeated several times. Originally, systems were designed by drawing explicit blueprints of transistors and interconnects. Standard cell libraries and place-and-route algorithms allowed designers to replace geometric circuit design with gate-level circuit design. Later, synthesis algorithms allowed designers to work at the register-transfer level (RTL) instead of the gate level. In each of these design discontinuities, designers gave up some control over fine implementation details in exchange for an abstraction that enabled them to create more complex systems with higher productivity.

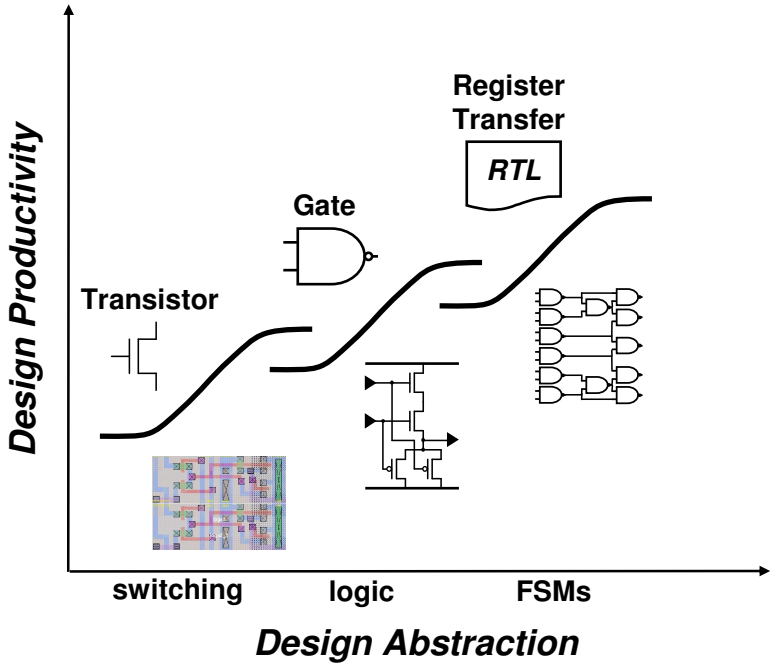


Figure 1.1: Design Discontinuities in Electronic Design Automation

1.1 The Quadruple Whammy

Manufacturing capability has increased to the point where it is possible to assemble billions of transistors on a single chip. Designers want to leverage this capability to put more complex functionality into embedded systems. However, it is clear that the capabilities of RTL design methodologies have leveled off as designers move toward gigascale systems. There are four major factors that make RTL design difficult.

First, there is the challenge of dealing with the enormous number of transistors. The basic design entity in RTL synthesis methodologies is the finite state machine (FSM). Designers build systems by making compositions of FSMs. This block is too fine-grained for gigascale systems. In order to use up a billion transistors, architects have to assemble a huge number of FSMs. The complexity of the coordination and communication among these machines is overwhelming.

Second, manufacturing variability and deep sub-micron effects make the design of each transistor harder. It has become difficult to get predictable results from an RTL design flow [15]. The problems of exponentially more transistors and individual transistor variability mesh together to result in higher manufacturing costs. This includes the costs of masks, packaging, and testing.

High costs pressure designers to get the design right the first time. This exacerbates the third

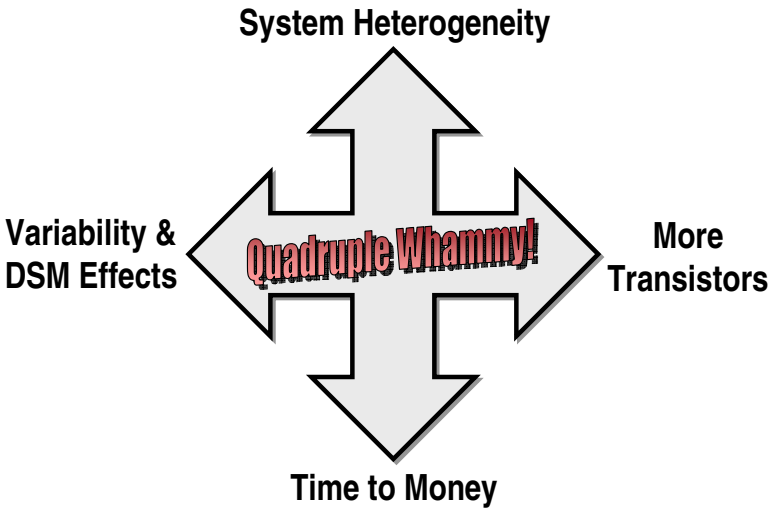


Figure 1.2: The Quadruple Whammy

design dilemma: shrinking design windows. If the design is not right the first time, not only does the business lose the manufacturing costs, but it also loses to competitors the market opportunity to sell the product. Additionally, in the time it takes to do a re-spin of a design, the product may become obsolete.

Exponentially more transistors also means that there will be a multiplicity of different kinds of circuits on a chip. This system heterogeneity is the fourth aspect of the RTL design crisis. A simple divide-and-conquer approach is insufficient because the interactions between heterogeneous components are just as important as the functionality of the components themselves.

System heterogeneity also means that architects will have to choose from a large set of implementation alternatives while designing a system. The choices to be made cover the gamut from high-level structural decisions that affect the entire system to choosing the parameters of individual components such as buses and caches. There are too many possibilities to make an a priori assessment of which options will lead to a better system. Thus, thorough *design space exploration* is necessary to find a solution that meets the desired criteria.

With ever-shrinking design windows, there is limited time for design space exploration. Commonly, producing one functional design is considered success. There is seldom confidence that the delivered design is as good as it can be.

Each of these challenges is difficult by itself, and together they negatively synergize to hit designers with a “quadruple whammy” (Figure 1.2). These factors are driving the current design discontinuity. A new methodology is desperately needed to replace RTL design.

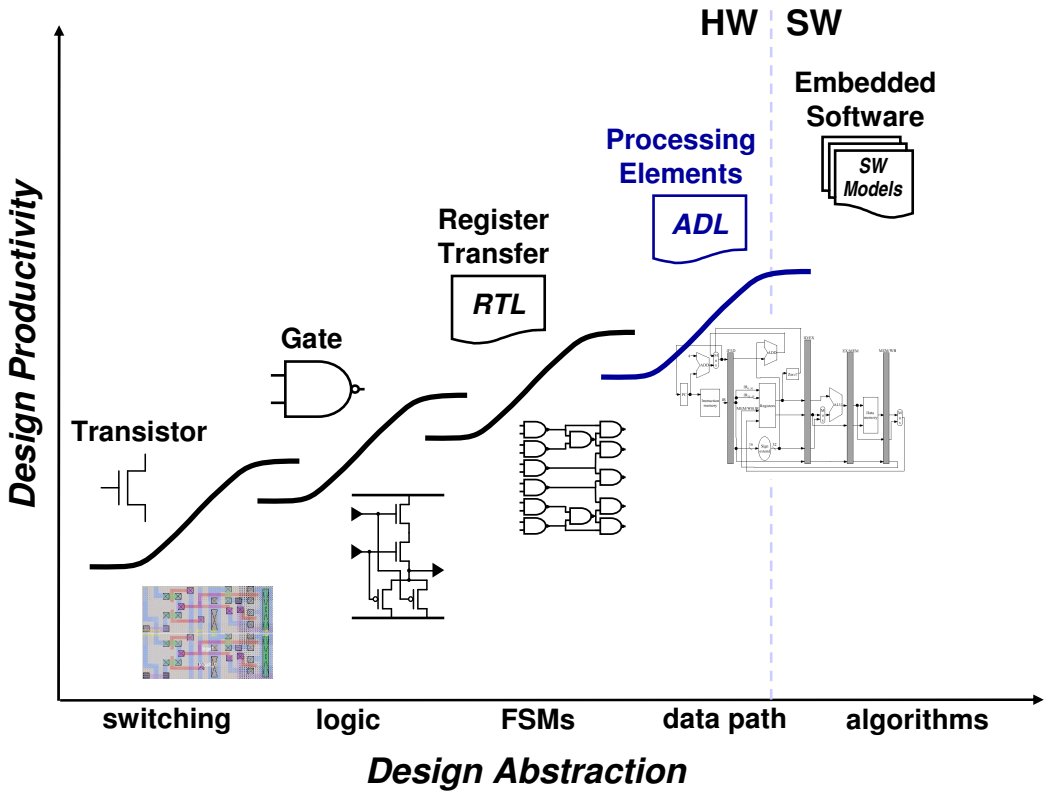


Figure 1.3: The Programmable Multiprocessor Design Discontinuity

1.2 The Promise of Programmable Multiprocessors

Just as in the previous design discontinuities, the next design methodology will ask designers to give up control over fine implementation details in exchange for a more productive work flow. The new methodology will provide an abstraction based on a coarser-grained architectural component that can be composed into complex systems.

RTL design is design at the level of individual finite state machines. A proposed solution to the RTL design crisis is to replace these state machines with programmable processing elements. Designers will no longer be able to fine-tune the implementation of individual state machines. Instead, they will assemble programmable elements to make heterogeneous multiprocessor architectures. These systems are also sometimes referred to as programmable platforms [69]. Figure 1.3 shows how programmable multiprocessors fit into the design discontinuity s-curve diagram.

Heterogeneous multiprocessors are an attractive design point for several reasons. First, they can reduce the costs and risks of RTL design. Programmability adds a software aspect to the design flow. It is much faster and cheaper to make changes to software than it is to make changes

to hardware. Downloading new software onto a chip does not require remanufacturing the chip. Designers no longer have to get it right the first time. Functionality can instead be built up in an iterative fashion, and problems can be fixed as they are found. Iterative development is a way to perform application-level design space exploration. Programmers can experiment with several different software architectures easily.

Programmable systems are less risky than hard-coded RTL designs because they increase the longevity of a design. A single system can be reprogrammed to serve several different applications within an application domain. Also, a programmable system can be upgraded in the field to follow a trendy, fast-moving application domain. These types of reuse allow high manufacturing costs to be amortized over a longer period.

Second, heterogeneous multiprocessors improve designer productivity while maintaining a reliable and predictable path to implementation. A datapath is a coarser block than a state machine, allowing architects to construct more complex systems in a shorter amount of time. The physical design challenges that plague RTL methodologies are constrained to stay within the datapath basic blocks. Outside of this boundary, the performance and behavior of a datapath is well-characterized by the instruction set architecture (ISA) and the clock rate. Programmers can reliably predict the performance of a software application using computer science instead of physics.

Third, system heterogeneity offers the possibility of high performance. Designers can assemble a variety of architectural components with custom features that benefit a particular application domain. Architectural design space exploration can find a solution that balances application-specific performance goals with the proper amount of general-purpose reusability.

1.3 The Reality of Programmable Multiprocessors

Programmable multiprocessors have already appeared in commercial embedded designs. In the network processing domain, Intel's IXP series of network processors is a well-known example [26]. This architecture is only one of large diversity of network processors [105]. The rise of field-programmable gate arrays (FPGAs) as system platforms is another example. Xilinx's Virtex architecture contains multiple programmable PowerPC cores [64]. An FPGA can also be used as a medium for implementing multiprocessor systems composed of soft processor basic blocks (e.g. Altera NIOS, Xilinx MicroBlaze) [67]. In the graphics processing domain there is the (recently canceled) Intel MXP5800 multiprocessor architecture. The CELL processor, a joint effort of IBM, Sony and Toshiba, is expected to gain more traction [95].

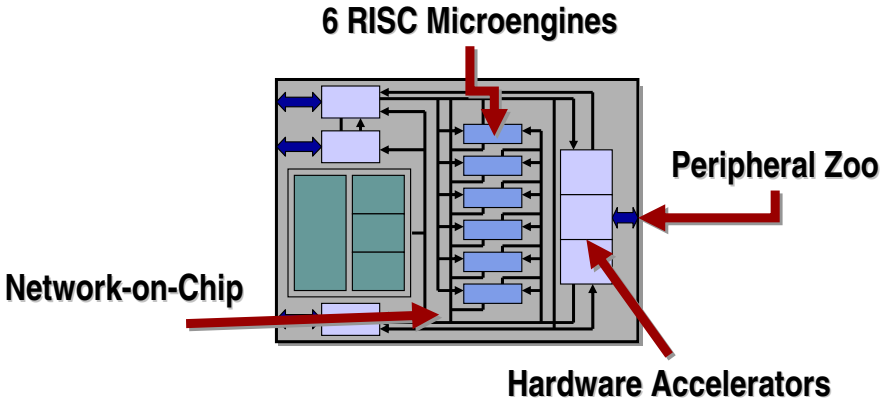


Figure 1.4: Intel IXP1200 Network Processor

Later in this dissertation, the focus is placed on network processing examples. Therefore the IXP1200 network processor is highlighted as a canonical example of a heterogeneous multiprocessor. Its architecture is shown in Figure 1.4. The IXP1200 contains six RISC-like *microengines* that perform the bulk of the network processing tasks. The architecture is heterogeneous because it contains a diversity of architectural elements in addition to the microengines. An XScale RISC processor is included to serve in a supervisory role. Hardware function units exist to accelerate certain network processing computational tasks. There is a diverse on-chip network for moving data around. This network consists of a variety of buses and point-to-point connections with different bandwidths and protocols. Finally, there is a zoo of peripherals that interact with the on-chip programmable elements.

On its face, the IXP1200 seems to have a desirable set of features for network processing applications. However, once application designers started to use it, the architecture quickly achieved notoriety as being difficult to program. Craig Matsumoto wrote in EE Times magazine:

> "Intel Corp. . . . chose a "fully programmable" architecture with plenty of space for users to add their own software - but one that turned out to be difficult to program." [81]

1.3.1 Blame the Programmers

What makes programming so difficult? Some point their finger at the application designers. Modern embedded applications are full of concurrency. An implementation must exploit this concurrency in order to meet the desired performance goals. This criticism says that poor methodologies for modeling and implementing concurrency are to blame for the programming difficulties.

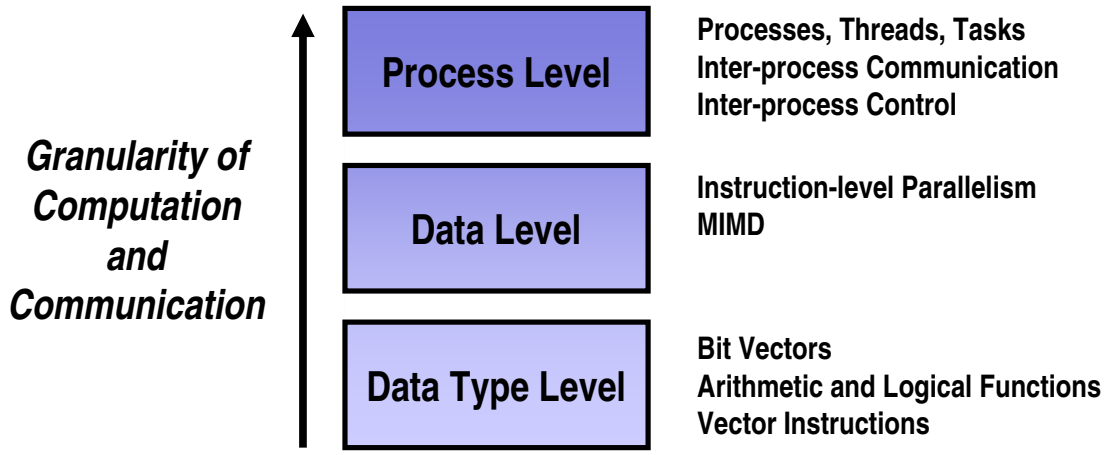


Figure 1.5: Levels of Concurrency for Applications and Architectures

Multiple Granularities of Concurrency

Concurrency appears in embedded applications at several different levels of granularity. These are detailed in Figure 1.5. In this classification, granularity refers to the size and complexity of the computations, and the amount, scale and frequency of the communication between the computations.

Process-level concurrency is the coarsest-grained category. This includes the application's processes, threads and tasks. When an application can take advantage of multiple flows of control then it is said to exhibit process-level concurrency. When programmers use libraries like Pthreads they are trying to exploit this process-level concurrency.

Process-level concurrency also refers to the communication and control patterns among concurrent processes. Shared memory and message passing are two common modes of process-level communication. Control patterns are the way the processes execute relative to each other. They may be asynchronous, where process-level control is independent of process-level communication. Alternatively, the control pattern may be related to the communication of data between processes.

Data-level concurrency is concurrency within a process. Here, the size and complexity of computations is smaller. An application exhibits data-level concurrency if it can perform computation on individual pieces of data in parallel. Instruction-level parallelism (ILP) is one form of data-level concurrency. A multiple-issue or MIMD-style architecture is a target that can be used to exploit an application's data-level concurrency. The scale of communication between parallel computations at this level is smaller than that of process-level concurrency. At this level the dependencies between individual datums are of interest.

Figure 1.6: Network Security Gateway Deployment Scenario

Data-level concurrency is orthogonal to process-level concurrency. Applications can and will exhibit both types.

The finest-grained category is datatype-level concurrency. Whereas data-level concurrency is concurrency among datums, datatype-level concurrency is concurrent computation *within* datums. Specifically, this category refers to the types of arithmetic and logical operations the application desires to perform on datums that are vectors of bits. Operations such as additions and CRC checksums perform parallel computation on individual bits and exchange information at the bit level. It is usually taken for granted that this level of concurrency is implemented directly in hardware.

Heterogeneous Concurrency

An embedded application will exhibit concurrency at all three of these levels. Furthermore, modern applications will exhibit multiple different styles of concurrency at each of the three levels. This is called heterogeneous concurrency.

For example, consider the following network processing application as a thought experiment. A small office or home office of the future will contain numerous networked devices and a broadband Internet uplink. A router is required to share Internet access among all of the internal devices. To provide security, the router should inspect packets thoroughly to search for possible attacks. This network security gateway is shown in Figure 1.6.

There are two major components to the network security gateway application. First, the device must perform packet header processing to perform routing, connection sharing and quality of service guarantees. Second, the device must perform packet body inspection for the security facet of the application. Each of these facets has different computational requirements and exhibits different styles of process-level, data-level, and datatype-level concurrency.

In header processing, each packet can be processed independently. The application is interested in manipulating packets spatially (e.g. forwarding packets to the proper destination) and temporally (e.g. queuing packets to respond to bursts and to shape packet streams). These are aspects of the application's process-level concurrency. Packet processing applications also manipulate a variety of header fields. There are custom datatype-level operations such as checksum calculations and bit manipulations. Independent header fields can be modified in parallel, which is a form of data-level concurrency.

In packet body processing, the basic process-level unit is a packet flow. A related stream of packets between the same source and destination must be correlated and assembled according to a given protocol's fragmentation rules. The contents of these packets are then treated as a single stream of text characters and are searched for interesting patterns. This style of process-level concurrency is different from that in header processing because the packets are not independent. The styles of data-level and datatype-level concurrency are different as well. There are no checksum operations or bit field manipulations. A homogeneous stream of characters is fed through an exact string matching or regular expression matching algorithm.

In order to implement the network security gateway application, the programmer must consider not only multiple granularities of concurrency but also multiple styles of concurrency at each level of granularity.

Modeling Concurrency with a Domain Specific Language

Current best practice is to begin by creating a high-level model of the application. This allows designers to perform introductory application-level design-space exploration. The model is meant to guide the process of refinement to an actual implementation and can serve as a "golden model" to verify correctness at later stages.

Since the application contains so much concurrency, designers may wish to use a domain-specific language (DSL) to make it easy to capture the concurrency. For packet header processing, Click is a well-known DSL [72].

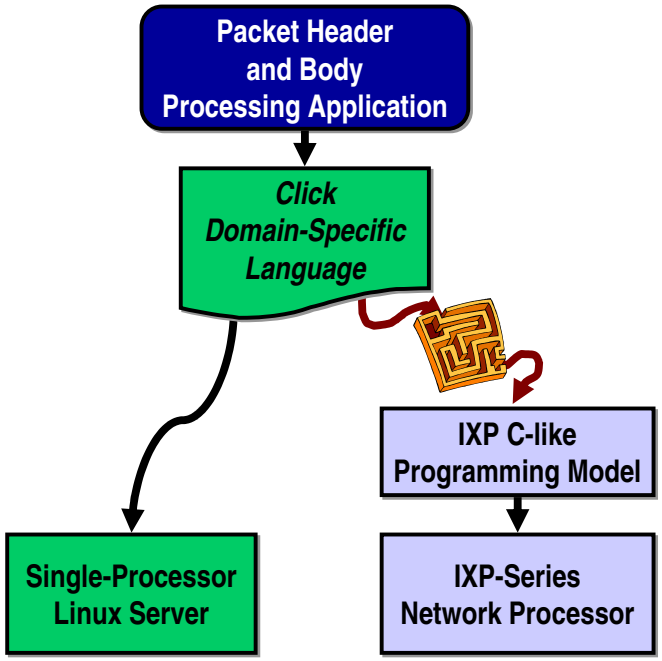


Figure 1.7: Programming a Network Processor Starting From a DSL

Unfortunately, the refinement path from Click to implementation on a network processor such as a member of the IXP series is not clear. Figure 1.7 illustrates this scenario. The existing tools and methodologies that are packaged with the Click language target implementation on standard uniprocessor Linux machines [71]. Click is converted into a C++ implementation that is compiled and run on the single processor.

The programming model for the IXP network processors is different. Intel provides a C-like programming model for their architectures where designers can program each processing element individually. The C++ language is not supported, but this is the least of the problems. Designers must turn the abstract description of process-level concurrency in the Click model into an implementation that fully leverages multiple parallel microengines, each of which runs multiple threads. Scheduling and partitioning must be done manually. Designers must determine how to best utilize shared resources such as memories and hardware accelerators. On top of this there are numerous idiosyncrasies of the architecture that must be dealt with. For the IXP1200, shared control status registers and a transmit FIFO that manages interactions with off-chip Ethernet MACs turned out to be an unexpected bottleneck [107].

These complications make the refinement process starting with a Click model slow and error-prone. Instead of performing iterative design, programmers spend most of their time on the path

between the DSL and the low-level programming model used for implementation. Getting one functional implementation is often considered success. Even minor modifications to the original Click application require lengthy software rewrites, so there is seldom time for adequate design-space exploration.

The ability to perform iterative application design is supposed to be one of the major wins of programmable architectures. When making even one implementation is so difficult, this benefit is ruined. The path from Click to the network processor's programming model truly is the maze of complexity shown in the figure.

Some may question the benefits of using the DSL in the first place, and advocate starting with the architecture's programming model directly. In addition to the problems stated above, the Click language is only well-suited for the header processing component of the application. It does not assist the programmer with the packet body processing facet. Why use it at all? Without a high-level model of the application's concurrency, programmers have only a mental model, or at best a diagram on a whiteboard, to guide them in the implementation process. The only realization of the specification is in the form of the low-level program code. This makes it even harder to experiment with changes to the structure of the application. The DSL approach is imperfect, but it is better than the alternative.

1.3.2 Blame the Architects

The preceding criticism says that heterogeneous multiprocessors are difficult to program because programmers have poor methodologies for modeling and implementing the various styles of concurrency that are critical to the application. An alternative explanation is to place the blame on the hardware architects. There are two reasons. The first claim is that architects do a poor job of communicating the machines' capabilities for implementing heterogeneous concurrency to programmers through programming models. Second, architects are accused of constructing machines that are not good matches for the requirements of the application domains. This section explores both parts of this criticism in detail.

Poor Communication of Architectural Capabilities

An architecture's capabilities are supposed to complement the requirements of the application domain. The role of a programming model is to communicate these capabilities (and the inevitable caveats) to the programmers so they can exploit the architecture fully. The programming model must

balance visibility toward the features that are necessary to obtain high performance, and opacity toward the minutiae that get in the way of productive application deployment.

In current practice, programming models for multiprocessor architectures call for programmers to write code for each processing element individually. Typically this is done in assembly language, or in a C language dialect with compiler intrinsics.

The C language was designed to balance visibility and opacity for sequential general-purpose processors. It is ill-suited for heterogeneous multiprocessors because it does not provide any abstractions for concurrency. Like embedded applications, programmable multiprocessors exhibit multiple styles of concurrency on several levels of granularity. Architectural concurrency follows the same categorization as application concurrency (as described in Section 1.3.1), but the details are different.

- *Architectural Process-Level Concurrency:* Multiple programmable elements (PEs) run in parallel and communicate over an on-chip network. The network topology affects how well certain PEs can communicate with others. Different buses utilize different protocols to provide different communication semantics. The control semantics of the PEs can vary as well. Some PEs may run sequential programs individually, while others only execute in response to communication events. These patterns of computation, communication, and control describe an architecture's process-level concurrency.

  The C language provides the abstraction of only a single thread of control and a single monolithic memory. There is no concept of multiple simultaneous processes or of communication channels between processes. There is no concept of separate memories.

  This dissertation argues that thread libraries such as Pthreads are better defined as new languages rather than as simple C libraries. Although a Pthreads program looks like a C program and can be compiled with a C compiler, the behaviors of many of the statements do not match those of standard C statements. For example, consider the statement *pthread_create*. This appears to be a normal C function call, but it does not behave like a C function call. Instead of jumping to a subroutine and returning, this statement forks the thread of control. A programmer who does not know this and tries to interpret the *pthread_create* statement as an ordinary function call will not be able to understand how the program works.

- *Architectural Data-Level Concurrency:* A heterogeneous multiprocessor has a variety of PEs with different issue widths, register file configurations, superscalar features and MIMD,

SIMD, and VLIW properties. These characteristics define how an architecture is able to perform computation on multiple pieces of data within a process simultaneously. This is the architecture's data-level concurrency.

C source code does not express data-level parallelism explicitly, and thus programmers cannot see the hardware's ability to perform multiple instructions simultaneously. Much research has been done to extract this sort of implicit parallelism from C code. This is difficult because C obfuscates data dependencies with ambiguous pointer semantics.

- *Architectural Datatype-Level Concurrency:* PEs and hardware accelerators have different bit widths (e.g. integer, floating-point, fixed-point, and custom data types), instruction sets and ALUs. A heterogeneous architecture will exhibit multiple styles of this concurrency.

  C provides only basic arithmetic and logical operators and standardized data types. Architects must extend the C language with compiler intrinsics to give programmers visibility into custom data types and application-specific function units.

The C language fails on all counts as an abstraction for heterogeneous multiprocessors. Programmers must manually translate high-level application concepts into a set of interacting C programs. As previously described, this is slow and error prone. However, when viewed from an architectural perspective the problem is even worse. The low-level language that programmers are targeting does not even reflect the true capabilities of the architecture. In the process of writing C programs, designers must discard the data-level and datatype-level concurrency that is inherent in the application. Instead they write a description in a language that expresses neither the application's concurrency nor the architecture's concurrency.

For a given application domain and architecture, it is possible to create a good programming model. Such an abstraction will effectively capture the architecture's concurrency, and provide an intuitive way to implement the concepts that are important to the application domain. For network processing applications and the IXP1200 architecture, NP-Click is an example [106]. NP-Click is a library-based approach that sits between the IXP C programming model and the Click language. This is the location of the maze in Figure 1.7. NP-Click's goal is to hide most of the complexities of the IXP C programming model and export a more concise computational abstraction that better matches the one found in a Click application model. Click elements are implemented by the NP-Click library using the appropriate IXP C intrinsics. The Click communication abstraction is implemented using the IXP's on-chip network.

To create NP-Click, Shah et al. had to thoroughly explore the intricacies of the IXP1200 architecture and determine which capabilities were important to export to programmers whose focus is network processing applications. This leads to the downside to the approach. It is tightly connected to one particular architecture and one particular application domain. It is not flexible enough to handle architecture design space exploration, or even different architectures in the same family such as the IXP2800 [27]. Architectural changes in the IXP2800 required the NP-Click library to be rewritten [97]. This is not as simple as porting an application in the traditional sense, for example from Windows to Macintosh. The reorganization of the on-chip network in new network processors requires NP-Click to use entirely different methodologies and algorithms to implement communications.

In summary, architects need to recognize the shortcomings of the C language and provide a better abstraction of the capabilities of their machines. The critical importance of concurrency motivates this need.

Poor Support for Application Requirements

The second half of this criticism suggests that modern architectures provide features that are not actually beneficial to the application domains they are designed for. Architects build architectures based on their past experiences. Architectures are not developed with the help of a feedback loop that incorporates input from application domain experts and the results of previous deployment attempts.

The diversity of architectures in the network processing space alone shows that there is a large difference in opinion on what makes a good architecture for network processing. Figure 1.8 illustrates this diversity [105]. This scatter plot organizes architectures by process-level concurrency on the Y axis (number of PEs) and data-level concurrency on the X axis (issue width per PE). Most of these architectures were commercial failures. This suggests that architects are often wrong in their thinking about what makes a good architecture (and it also suggests that having a good architecture is no guarantee of commercial success).

An architecture may not have enough features for the application domain, or it may be loaded down with expensive features that are never actually needed. Within a PE, the control logic may be too generic or too specialized. The datapath function units may be too generic or too specialized. The data types the machine operates on may be too big, too small, or completely wrong for the application.

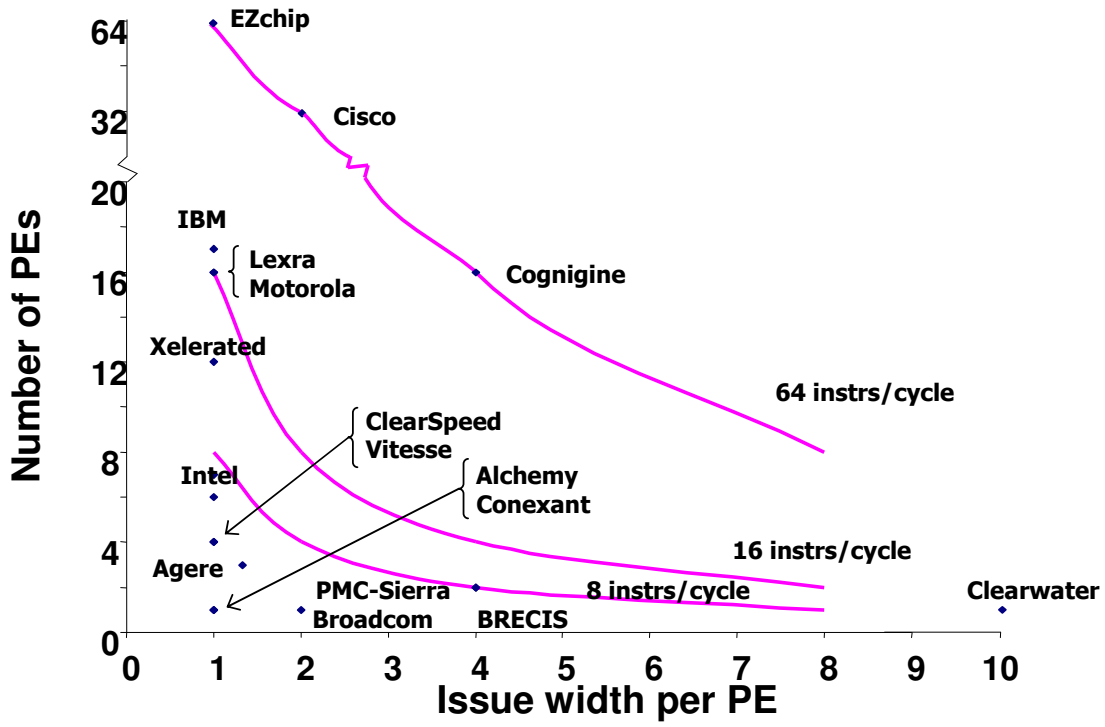


Figure 1.8: Diversity of Network Processor Architectures [105]

The multiprocessor architecture may be too simple or too complex for the application domain. It could have the wrong number of PEs, hardware accelerators, or memories. There may be insufficient or too much on-chip bandwidth. The connectivity between PEs may be poor or overwhelming. Finally, the architecture could be full of idiosyncrasies that become unexpected bottlenecks, such as the control status registers and transmit FIFO in the IXP1200.

These questions are supposed to be answered by effective architectural design space exploration that is tightly integrated with representative benchmarking [44]. If the architecture's heterogeneous concurrency does not support that of the application, programmers will naturally find the system difficult to program.

1.4 The Concurrency Implementation Gap

It does not matter where the finger points. These criticisms reveal a *systemic* ad hoc treatment of concurrency for systems in which concurrency is a first-class concern. This problem is the *concurrency implementation gap* (Figure 1.9). Concurrent applications and heterogeneous multiprocessor architectures each have their own styles of concurrency. These styles are not equivalent.

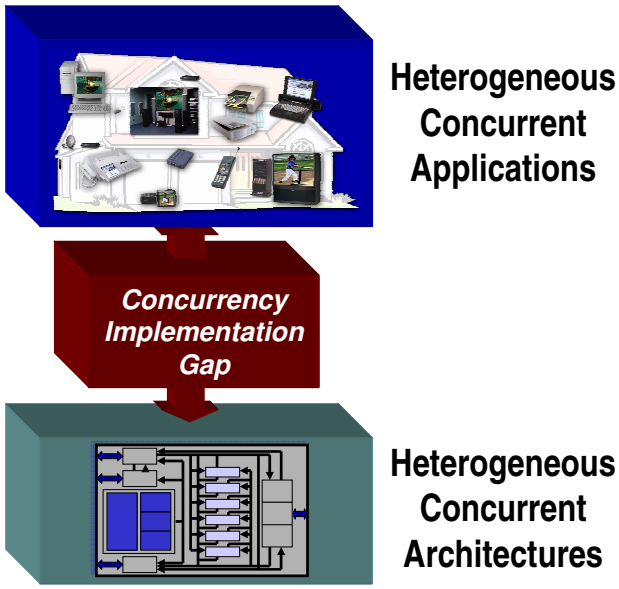


Figure 1.9: The Concurrency Implementation Gap

Today, architects lack a formal mechanism that is able to communicate the capabilities of concurrent architectures to programmers. Likewise, programmers are not successful in preserving a formal model of application concurrency throughout the implementation process. When it comes time to deploy an application on an architecture, designers must find a mapping between the application's concurrency and the architecture's concurrency. Without correct models of the application's requirements and the architecture's capabilities, deployment is nearly impossible. This can be summarized by saying that the concurrency implementation gap is hard to cross.

When programmers fail to preserve application concurrency throughout the implementation process, architects are never made aware of the application's true requirements. Therefore they continue to be prone to designing architectures that exhibit a serious mismatch with the concurrency requirements of the application domain. This can be summarized by saying that the concurrency implementation gap is wide.

Ad hoc approaches to concurrency are slow and error prone. Success is defined as producing one functional implementation within the available design time. There is no time to explore the design space thoroughly. As a consequence, architects cannot prove that their designs provide features that actually benefit the application domain because the architecture is not designed in a feedback loop that considers application requirements and input from programmers. At the same time, programmers cannot know that they are exploiting the architecture to the fullest because they have not had time to explore alternative implementation strategies.

By inhibiting an iterative design process, the concurrency implementation gap stops designers from using the only technique that can make it go away: design space exploration. Therefore, in addition to being wide and difficult to cross, the gap is also self-sustaining. Proper methodologies for concurrency are required to make the gap easier to cross. This will enable design space exploration, which will make the gap smaller. Until then the gap will continue to ruin the promise of programmable multiprocessors.

1.5 Solving the Concurrency Implementation Gap

To overcome the concurrency implementation gap and realize the potential of programmable multiprocessors, one must solve this challenge: How to model and compare application concurrency and architecture concurrency? Then one can replace slow, cumbersome and error-prone techniques for system deployment with disciplined methodologies.

This dissertation describes a novel methodology based on multiple abstractions for solving this challenge. This is the missing piece required to drive the current design discontinuity and enable heterogeneous programmable multiprocessors as a replacement for traditional system design. The next chapter outlines this new methodology.

Then, Chapter 3 compares the multiple abstraction approach to related work. The following chapters go into details about each of the three abstractions in the multiple abstraction approach. Chapter 4 describes an architectural abstraction for capturing the concurrency of heterogeneous multiprocessor architectures. Chapter 5 describes an application-level abstraction for heterogeneous concurrent applications. In Chapter 6, an abstraction and methodology are introduced for mapping application concurrency onto architecture concurrency and producing implementations. The multiple abstraction approach is demonstrated with a capstone design example in Chapter 7. Lastly, the contributions of this dissertation are summarized in Chapter 8.

Chapter 2

A Multiple Abstraction Approach

Solving the concurrency implementation gap requires a new design methodology that treats concurrency as a first-class concern. The methodology must consider both the heterogeneous concurrency found in embedded applications and the heterogeneous concurrency found in programmable platform architectures. Most importantly, it must consider how programmers and architects exchange information about concurrency. Comparing application requirements and architectural capabilities is not only critical for making high-performance implementations, it is also vital knowledge that drives design space exploration. This chapter introduces a novel multiple abstraction methodology that meets these requirements.

2.1 Three Core Requirements for Success

The central hypothesis of this dissertation is that there are three core requirements a design methodology must take into account in order to solve the concurrency implementation gap. If these requirements are met, designers will be able to quickly and correctly deploy concurrent applications on multiprocessor architectures. Iterative design will be possible because of this improved productivity. Consequently, design space exploration will find system solutions that meet performance requirements at lower cost and with less risk than systems designed with traditional RTL methodologies.

2.1.1 Core Requirement 1: Use a Formal High-Level Abstraction To Model Concurrent Applications

The first core requirement is to use a formal high-level abstraction to model concurrent applications. A formal application abstraction is a language that provides constructs that allow programmers to express concurrency efficiently and correctly. The expression of concurrency is declarative rather than imperative. That is, the abstraction allows designers to express concurrency without explicitly stating how the concurrency will be implemented.

For example, consider the differences between describing a network processing application in the Click language versus the C language. In Click, an application is modeled by drawing a graph using library elements that perform individual header processing computations. Designers do not have to divide the application into processes or specify how processes are scheduled. Instead, Click can automatically partition the application by analyzing the graph. This is possible because Click provides an abstraction for process-level concurrency. There is a pre-defined semantics, or set of rules, that dictate how the graph of elements is supposed to behave.

Click provides abstractions for lower-level granularities of concurrency as well. Designers do not have to specify the details of a packet header data type or how the elements perform computation on this data type. The application's datatype-level concurrency is also encapsulated within the elements.

In C, the designer would have to specify all of this concurrency explicitly. The application would have to be partitioned into processes and scheduled manually. The packet header data type and the corresponding mathematical operations would have to be built up manually using the primitives available in the C language.

Modeling concurrency explicitly in a low-level language like C is slow and error prone. Designers must form a mental model of the application's concurrency and then translate that model into a set of interacting C programs. Concurrency is difficult to visualize and reason about. The fact that embedded applications have multiple styles of concurrency on several levels of granularity makes this especially difficult. High-level abstractions are necessary to help designers cope with this complexity.

A Natural Representation

High-level application abstractions provide several other benefits as well. First, they provide a natural representation that domain experts can use to describe application features in an intuitive

language. The abstraction brings the application domain's primary concerns to the forefront. Continuing with the previous example, the Click abstraction allows programmers to focus on concepts such as the spatial and temporal processing of packets. Elements like *LookupIPRoute* make decisions on how packets are routed and *Queue* elements buffer packets. If designers want to modify the statistical behavior of the network processing application, they can do that by modifying the placement of Queues and other elements. This application-level design space exploration is done independently of the final implementation. This greatly benefits productivity.

A Precise Representation

Second, a high-level application abstraction allows programmers to make a precise statement about the application's requirements for concurrency. The declarative approach expresses exactly what the application needs without introducing artificial requirements. When concurrency is described in an imperative fashion, designers are tying the description to a particular style of implementation. One cannot separate the aspects of the model that are actual application requirements from those that are artifacts of the implementation strategy. For example, it is not possible to automatically inspect an arbitrary C program and discover that there are parallel processes that communicate using Click's *push* and *pull* semantics for process-level concurrency. The application will instead appear to have the same requirements as every other C program: a program counter, a stack, a heap, and integer arithmetic on 32-bit words. This is not a precise model of a network processing application's true requirements.

Exploring Alternative Implementations

Third, separating the application specification from the implementation allows designers to experiment with different implementation strategies starting from the same application model. This is important because it permits design space exploration to occur while preserving the ability to verify that the high-level behaviors of the application are kept unchanged.

The Flexibility Fallacy

Often, the creators of a language tout its flexibility as a primary benefit. They claim that programmers can do exactly what they want in the language, no matter what it is. This line of argumentation derides domain-specific languages like Click for placing too many restrictions on the designer.

It is a fallacy to treat flexibility as a primary concern. Edsger Dijkstra wrote:

> "Another lesson we should have learned from the recent past is that the development of "richer" or "more powerful" programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally. I see a great future for very systematic and very modest programming languages." [33]

In discussing the use of Models of Computation (MoCs) as design abstractions, Jantsch and Sander wrote (emphasis in the original):

> "As a general guideline we can state *that productivity of tools and designers is highest if the least expressive MoC is used that still can naturally be applied to the problem.*" [65].

High-level application abstractions make design easier by simultaneously assisting and restricting programmers. The abstraction assists programmers by relieving the burden of describing implementation details. By using implicit mechanisms for describing concurrency, programmers can build more complex systems faster and with fewer errors.

An abstraction restricts programmers by forcing them to follow a set of pre-defined rules. One cannot describe any process-level, data-level, or datatype-level concurrency. One can only describe particular styles of concurrency that have been selected because of their suitability for a particular application domain.

The assist and restrict characteristics are inseparable. To enjoy the benefits of an abstraction, programmers must be willing to give up some measure of flexibility. A language that is general enough for all applications will not be able to make a natural or a precise representation of any particular application domain.

For embedded applications with heterogeneous concurrency, some measure of flexibility is still necessary. Instead of choosing a single weaker application abstraction, this dissertation argues in favor of using multiple high-level abstractions for the different facets of the application. This will allow designers to create proper models of individual application facets as well as the interactions between the different facets.

Application Abstractions vs. Programming Models

Last but not least, it is important to make firm distinction between the terms "application abstraction" and "programming model" as used in this dissertation. Despite the word "programming"

in the name, a traditional programming model is actually an architectural abstraction. Its goal is to balance visibility into features of the architecture that are necessary to obtain high performance implementations, while hiding the unnecessary details that sap productivity.

An application abstraction seeks to model the native characteristics of an application in a way that is independent of any target architecture. It focuses on application concepts, not architecture concepts.

2.1.2 Core Requirement 2: Focus on Simple Architectures

The second core requirement is to focus on simple architectures as target implementation platforms. For many, the obvious choice for a programmable element to use in heterogeneous multiprocessors is a standard RISC processor, perhaps with instruction extensions. RISC processors are good at executing sequential programs with jumps. Their control logic is designed expressly for this purpose. Embedded applications are not always sequential programs with jumps. Instead, embedded applications have process-level, data-level, and datatype-level concurrency. RISC processors are often not the best choice when concurrency is considered.

Instruction extensions, such as those available in the Tensilica Xtensa processor [63], mainly target datatype-level concurrency. Customization of the processor's data-level concurrency is possible only to a lesser extent. The Xtensa approach starts with a generic RISC processor. Architects add additional datapath function units that perform custom arithmetic and logical operations on custom data types. This added functionality is exported to programmers through compiler intrinsics that can be called from C programs.

Tensilica's results show that there is significant performance to be gained by matching the application's datatype-level concurrency. This dissertation argues that designers can achieve similar performance breakthroughs by matching the application's process-level and data-level concurrency as well. However, in order to do this, one must look beyond traditional RISC design patterns and consider unusual architectures.

To match an application's process-level concurrency, one must consider changing the RISC increment-PC-or-jump control scheme. Architects may create a multiprocessor where the PEs have a more tightly-coupled global scheme of control. For example, the processes defined in a Click application model execute in response to push and pull communication events received from each other and from network interface peripherals. This reactive style of process-level concurrency can be emulated by interacting sequential programs, but a novel style of control logic could implement

it more cheaply and efficiently.

In many cases, it is not even necessary for a processing element to be Turing complete. A traditional hardware accelerator, which is hard-coded to perform only one computation, is an extreme example of this case. There are many interesting architectures along this axis that are still considered programmable, but only have the power to solve a limited set of computational problems. Examples will be given in Chapter 7.

To match an application's data-level concurrency, one must consider changing the RISC datapath topology. An application-specific processing element may have an irregular pipeline structure instead of a rigorous five-stage structure. There may be multiple register files and memory interfaces scattered throughout the pipeline. Complex function units that span multiple pipeline stages may be interspersed with smaller function units. The labels "fetch" and "execute" no longer strictly apply to these irregular pipelines. In the same stage, there may be simultaneous computations, memory lookups, and operand fetches for later stages.

These architectural patterns go beyond making extensions to an existing processor template. They also suggest leaving out features that most architects never question. Simple architectures, as defined in this dissertation, provide only the hardware complexity that the system demands. If absolute performance is required, designers should choose processing elements with application-specific features. If reusability is important, designers can seek a balance between general-purpose features and application-specific features. These design decisions are necessary to find an architecture that makes the concurrency implementation gap small. To satisfy this core requirement, a design methodology must consider simple architectures outside of the RISC box.

2.1.3 Core Requirement 3: Use an Intermediate Abstraction to Map Applications onto Architectures

The final core requirement is to use an intermediate abstraction to map applications onto architectures. The major difficulty in crossing the concurrency implementation gap is comparing an application's requirements for concurrency with an architecture's concurrency capabilities, and then producing an implementation based on that comparison. Currently, this is done mentally as programmers convert their intuition for how an application's concurrency should be implemented into code written in a low-level programming model.

This ad hoc approach obfuscates the concurrency implementation gap. Application concepts, architectural concepts, and implementation concepts are hopelessly blended together in the form

of low-level programming language code. If the result fails to meet expectations, it is never clear what part of the system is to blame. The original application specification may be faulty, or the architecture could be insufficiently powerful. The reason could be that the implementation process itself has introduced inefficiencies. Whatever the problem is, the solution will involve rewriting large amounts of code.

The role of an intermediate mapping abstraction is to turn the process of crossing the concurrency implementation gap into a distinct step in the design flow. It provides a formal mechanism for comparing application concurrency and architecture concurrency. This elevates concurrency to a first-class concern.

In practice, an intermediate mapping abstraction is a language that designers use to declare associations between application computations and architectural computational resources. Application computations are declared using a formal application abstraction as described above. Architectural resources are described using an architectural abstraction. A mapping association states that a particular application computation is to run on a particular architectural resource. For example, a process may be assigned to a programmable processing element, or a communication link may be assigned to a particular network-on-chip path.

These assignments can then be analyzed automatically to determine how well the application and the architecture match. A compiler can turn an assigned process into executable code for a PE. It can tell designers how fast the PE can run the process, and how much of the PEs cycles and instruction memory are used up. Analysis of communication assignments returns information on throughput, latency, and utilization of the on-chip network.

In reality, simple mapping assignments are not sufficient because of mismatches between the application and the architecture. Compilers fail because a processing element may not have the function units necessary to implement the data-level and datatype-level concurrency described within an application process. The communication semantics of an on-chip network may not match the communication semantics required by the application. During implementation, designers make compromises in order to match up dissimilar applications and architectures. Networking protocols are used to mediate communication events. Math libraries are used to emulate unavailable function units. This is extra computation that the architecture must do, in addition to the computations explicitly required by the application, in order to implement the application correctly.

The mapping abstraction must also allow designers to declare these compromises explicitly. The idea is to maintain a distinction between the computation that is required by the application and the computation that is a result of architecture-specific implementation decisions. Then, when

designers study the results of an implementation, they can determine where the performance goes. Is there a problem with the application itself, the architecture, or the implementation decisions?

A mapping abstraction should be a different language from those used to model the application and the architecture. This is because the concepts that are important to the mapping process are different from those that are important to application design or architecture design. The power of an abstraction is reduced if one attempts to pack too many dissimilar concepts into the same language.

In addition to clarity and cleanliness of purpose, the concept of having different abstractions for the application, architecture, and mapping enforces an important separation of concerns. This makes it possible to independently change different facets of the system. Architects can make modifications to the architecture without forcing application designers to rewrite large amounts of C code. Architectural changes can be automatically propagated into the mapping abstraction. Then, designers can analyze what parts of a previous mapping are consistent or inconsistent with the new architecture. This makes it clear what implementation decisions can be retained and which need to be re-thought in order to complete the design. A similar process works for application changes.

This is a powerful way to perform design space exploration. The mapping abstraction makes it clear how wide the concurrency implementation gap is. When designers apply their ideas to improve performance, they can see exactly how their changes make the gap smaller.

2.2 Cairn: A Design Flow Based on the Core Requirements

The three core requirements call for a design methodology that provides different abstractions and languages for the different facets of the system design problem. It is important that these abstractions be as formal as possible to simultaneously restrict and assist the designer. Strict abstractions are also amenable to automatic analysis, which is important for propagating information between the abstractions. Furthermore, the core requirements suggest neither a top-down nor a bottom-up methodology, but rather that a combination of top-down and bottom-up techniques be used simultaneously. This means that designers work on the application model and the architecture model simultaneously, and that these models co-evolve during design space exploration.

This dissertation proposes a multiple abstraction design methodology based on three main abstractions, as shown in Figure 2.1. This approach is called *Cairn*[1]. An application abstraction allows programmers to formally model heterogeneous concurrent applications. This satisfies the first core requirement. An architecture abstraction is used to model heterogeneous multiprocessor architec-

[1] A cairn is a pyramid-shaped pile of stones used to mark hiking trails and burial sites.

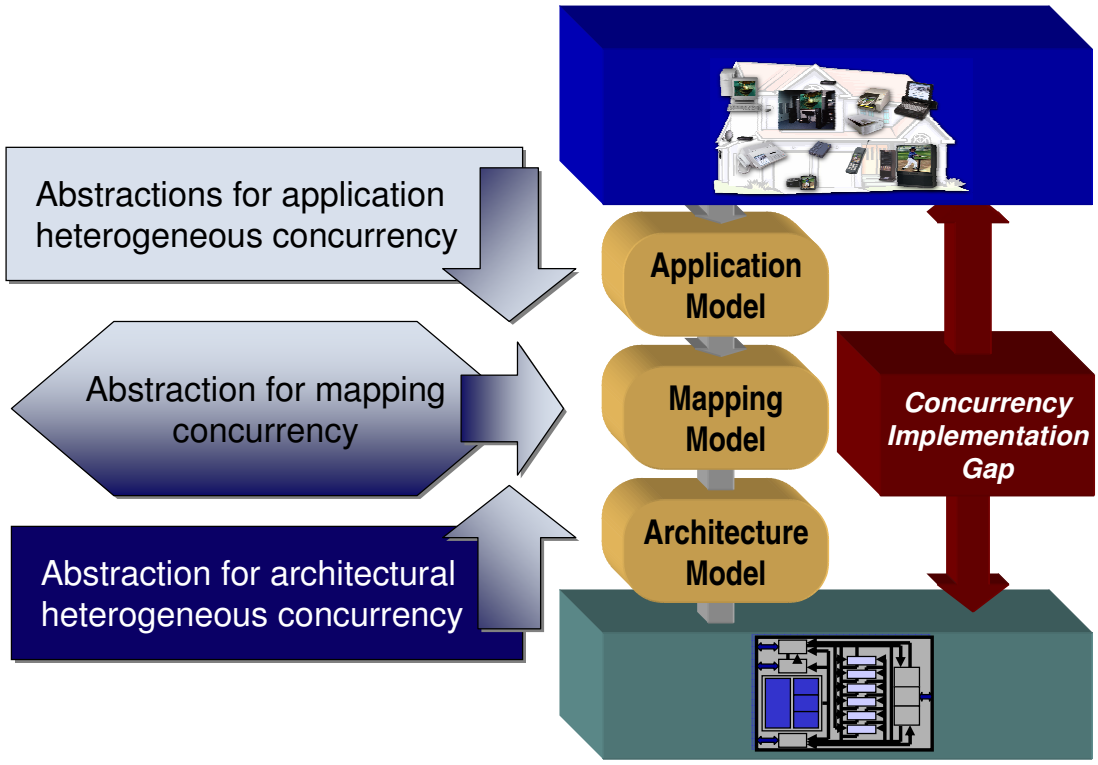


Figure 2.1: Three Abstractions for Crossing the Concurrency Implementation Gap

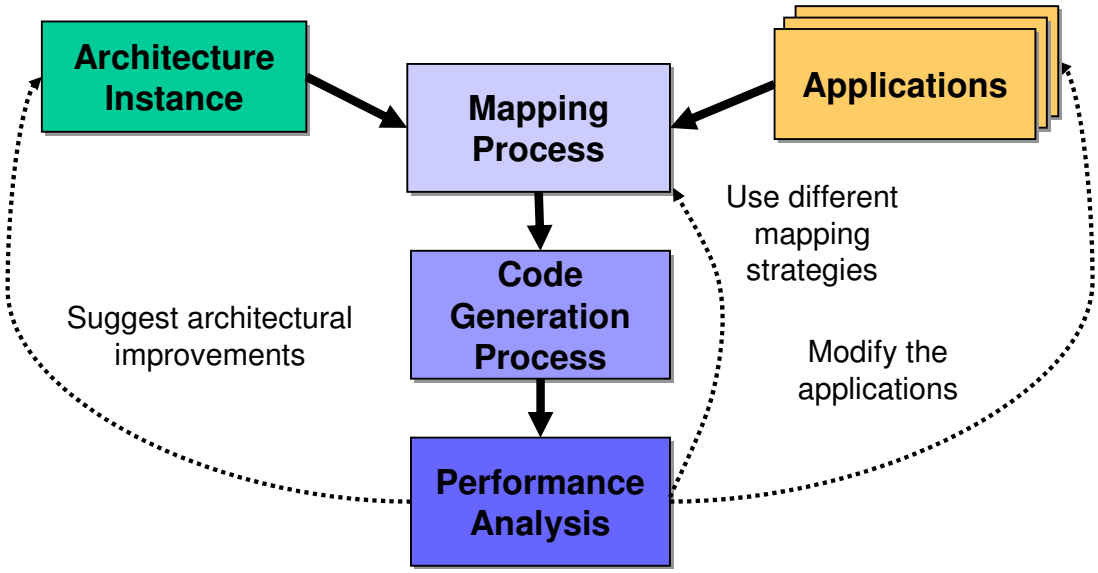


Figure 2.2: Y-Chart Design Flow

tures and export their concurrency capabilities. The term “architecture abstraction” is used in lieu of “programming model” to emphasize that the architecture abstraction focuses on concepts that are important to the hardware. Last, a separate mapping abstraction is used to compare application concurrency to architecture concurrency. This satisfies the third core requirement.

These three abstractions are combined to form a Y-chart design flow as shown in Figure 2.2. This design flow is similar to those proposed by Paulin et al. for the FlexWare project [92], by Kienhuis [70], and by Balarin et al. for the POLIS project [7]. Architects build a programmable multiprocessor and programmers construct a family of applications in parallel. The applications are mapped onto the architecture using the mapping abstraction in a distinct step of the methodology. A complete mapping is sent through a code generation process that transforms the mapped application into executable machine code for the programmable elements in the architecture.

This implementation can be considered complete, or it can be analyzed to drive the design space exploration process. The exploration options are marked by dotted lines in Figure 2.2. The performance analysis may suggest architectural improvements, application improvements, or that changes should be made to the mapping. All of these changes can be tried independently or in combination.

2.3 A “Tall, Skinny” Approach

This dissertation adopts a depth-first approach to solving the concurrency implementation gap. It does not attempt to solve every aspect of concurrent application design, multiprocessor architecture design, and mapping. Instead, the focus is placed on primary concerns within each of the three design areas. The goal is to choose good abstractions for each area and combine these abstractions into a workable design flow that meets the three core requirements.

To achieve this, the range of applications and architectures that will be considered is restricted. First, a particular family of target architectures with compelling characteristics is selected. Sub-RISC processors are simple programmable machines that are easy to design and allow architects to experiment with design patterns outside of the RISC paradigm [119]. This makes it possible to build architectures that support application-specific process-level, data-level, and datatype-level concurrency. Tools exist to quickly modify Sub-RISC designs during design space exploration and automatically export changes through an architectural abstraction. A poor choice of architectural building blocks will lead to a large concurrency implementation gap and mediocre solutions, reflecting poorly on the methodology as a whole. The scope of architectures considered in this work

is therefore limited to multiprocessor compositions of Sub-RISC processing elements.

Second, one application domain that exhibits heterogeneous concurrency is chosen. Network processing is a domain where exploiting concurrency is necessary to meet performance requirements. It is a domain where multiprocessor architectures have been tried in the past, but it is unclear what architectural features are actually beneficial to the domain. Furthermore, it is a domain where programmers have been stymied by the concurrency implementation gap while trying to target commercial multiprocessor architectures. This is a nontrivial application domain that will prove that the Cairn multiple abstraction methodology is capable of handling complex concurrency issues.

The main body of this dissertation, Chapters 4, 5, and 6, go into detail about Cairn's architecture abstraction, application abstraction, and mapping abstraction respectively. A design example using the complete methodology is given in Chapter 7. This demonstrates that by following the three core requirements, one obtains a design methodology that produces superior results with high designer productivity. But first, related work is considered in the next chapter. The Cairn multiple abstraction approach is compared to other methodologies that attempt to deal with heterogeneous concurrency. It is shown how Cairn combines the best features and avoids the pitfalls of these related works.

Chapter 3

Related Work

This chapter compares the Cairn multiple abstraction design methodology to related work that deals with concurrent applications and architectures. Three separate abstractions may seem like heavy machinery that will burden designers, but this comparison will reveal that each abstraction plays an essential role in avoiding the shortcomings of alternative approaches. The separation of concerns is also crucial for combining the best features of related projects without introducing conflicts or diluting the power of the individual concepts.

The first section lists the points of comparison for discussing related works. A tendency that must be avoided is to focus solely on a methodology's design flow, or the steps that designers follow when using the methodology. More important are the techniques that a methodology employs to help designers deal with the concurrency implementation gap. That is, what abstractions does the methodology provide to help designers model complex concurrency?

To show this aspect of related methodologies, a new type of chart is introduced called the *design abstraction chart*. This chart complements a regular flow chart. It shows what abstractions are used within a methodology. Section 3.2 describes how to interpret design abstraction charts. Finally, Section 3.3 uses design abstraction charts to describe the pros and cons of related design methodologies.

3.1 Points of Comparison for Related Design Methodologies

To compare related design methodologies, the following criteria are important:

- *Breadth of Applications Considered:* What applications does the methodology consider? A

methodology may support a very broad class of applications, or it may focus on a specific application domain. Some methodologies are meant to be used only for portions of applications, for example to produce implementations of small kernels. Others support heterogeneous applications that have a variety of computational requirements.

- *Target Architectures Considered:* What kind of implementations does the methodology produce? Does it create a hardware architecture, or does it produce software for a programmable machine? If the result is software, what kind of architecture is the software supposed to run on? A methodology may target an existing commercial architecture, or one of the user's own design. If the methodology produces its own architecture, what is the nature of this machine? Is it a hard-wired circuit that performs exactly one application, or is it a programmable machine that can be reused for other applications besides the one initially given?

- *Direction of the Design Flow:* This criterion covers how designers use a methodology. In a top-down methodology, designers start with an abstract specification of an application, and gradually refine this specification into a physically realizable implementation. In a bottom-up methodology, designers start by creating detailed models of architectural components. These components are then grouped together to form larger components and eventually an entire system. "Meet-in-the-middle" methodologies combine aspects of top-down and bottom-up design. Designers construct application models and architectural models in parallel. In a separate mapping step, application computations are matched up with architectural components.

- *Models that the Designers Must Create:* Different methodologies call for designers to create different models of the system during the design process. This criterion covers the set of models that must be made, what details of the system are described in each model, and the relationship between the models. For example, in a refinement-based methodology designers start with an abstract specification and create multiple intermediate models that contain progressively more implementation details. Manual processes that involve making design decisions, and sometimes automatic processes, are used to go from one model to the next. In a behavioral synthesis methodology, designers may create just one abstract specification and use an automatic process to get a concrete implementation. Both of these methodologies are top-down, but they are quite different in how they work and what kind of models they require designers to create.

- *Abstractions that the Designers Use:* Finally, what abstractions does the methodology pro-

vide to help designers create these models? Abstractions are necessary to help designers model complex concurrency efficiently and correctly. Does the methodology provide different abstractions for different modeling tasks, or is there a single universal abstraction? A good abstraction provides a natural, intuitive way to capture concepts that designers are interested in. It improves designer productivity by hiding implementation details.

To judge a methodology against these criteria, it is not enough to simply look at a flow chart. A flow chart only describes the steps designers follow when using a methodology. It does not explain how the methodology helps designers deal with the concurrency implementation gap. To do this, it is useful to visualize and organize the different models that are made while using the methodology and the different abstractions used to create them. A design abstraction chart, described in the next section, organizes this information in the context of the concurrency implementation gap.

3.2 Design Abstraction Charts

In previous figures, such as Figure 1.9 and Figure 2.1, the concurrency implementation gap is represented as a vertical space between applications and architectures. This space represents a disparity between concepts at the application level and architecture level. Figure 2.1 fills in this space with blocks that represent models of the system at different levels of abstraction. Separate models and abstractions break up a large concurrency implementation gap into smaller pieces that are easier to solve.

The idea behind design abstraction charts is that this diagram of system models and design abstractions is useful for comparing related design methodologies. The goal of this section is to give these diagrams meaningful semantics by defining the visual grammar more formally. There will not be so much formalism as to enable rigorous mathematical proofs, but just enough for the reader to develop an intuition about how a design methodology tries to solve the concurrency implementation gap.

The important concepts in Figure 2.1 are that blocks represent system models and arrows represent relationships between models. Vertical position has meaning as well. Blocks at the top of the diagram represent application models. These are high-level specifications that describe function but not form. They describe in an abstract way what computations designers would like to be able to perform, but they lack concrete implementation details. They do not describe how these computations are to implemented on a tangible architectural platform. Blocks at the bottom of the

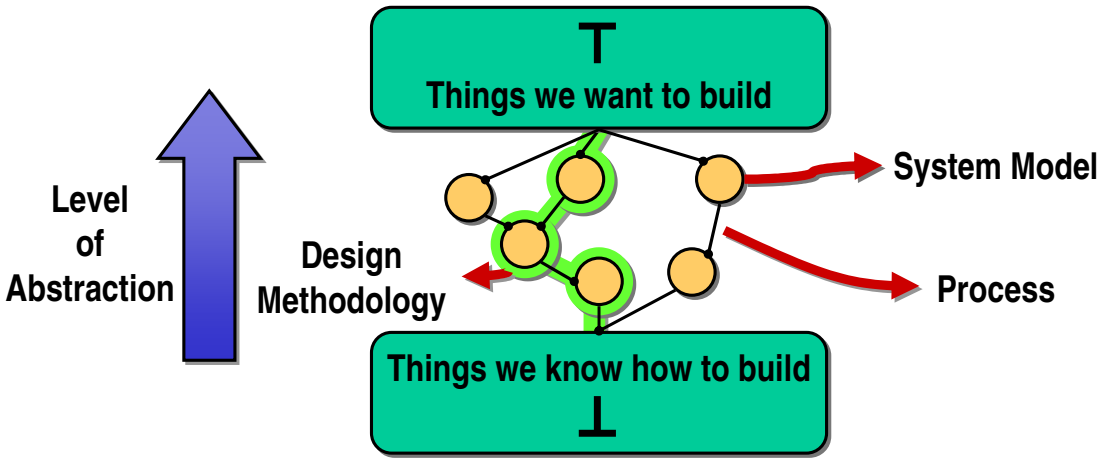


Figure 3.1: Design Abstraction Chart

diagram represent physically realizable architectures, where all of the implementation details are fully specified.

An edge between a higher model and a lower model indicates that the models are related. The higher model hides one or more implementation details that are present in the lower model. Alternatively, one can say that the lower model refines something that is expressed abstractly in the higher model.

These concepts are analogous to those found in mathematical lattices [114]. A lattice is a graph with nodes and directed edges, and the directed edges indicate a partial ordering of the nodes. "Level of implementation details" is the analogous concept to a partial order.

A design abstraction chart archetype is shown in Figure 3.1. Nodes in this chart represent system models. Directed edges, shown graphically as $A \multimap B$, indicate that model A contains fewer implementation details than model B. An edge indicates that there exists a process, either manual or automatic, that designers can use to go from one model to another. An edge can have a label that indicates the concept or design principle that enables this process or differentiates the two models.

The direction of the edge is not related to the direction of the process. Some processes take an abstract model and fill in concrete details, while other processes export abstractions of models. Design flow charts provide information about the order in which design processes are applied; design abstraction charts show how these processes help designers cross the concurrency implementation gap.

Analogous to a mathematical lattice, the design abstraction chart is bounded by "top" and "bottom" elements. The top element (written as $\top$) represents completely abstract back-of-the-envelope

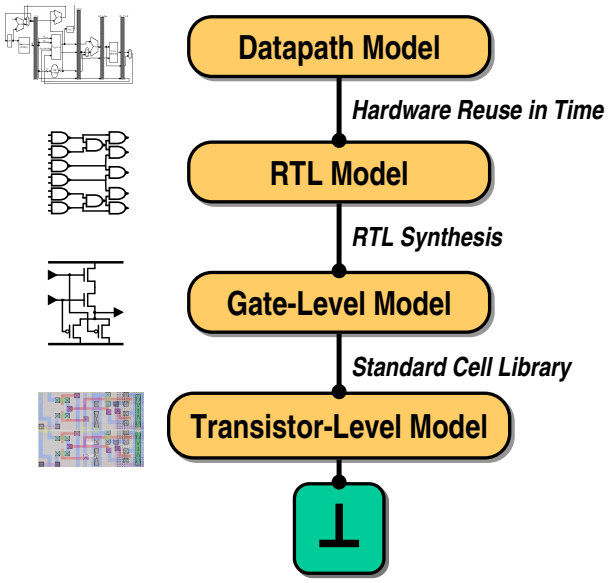


Figure 3.2: Design Discontinuities Drawn as a Design Abstraction Chart

specifications. These are “things we want to build”, such as proposals, specifications, and problems without solutions yet. The bottom element (written as $\perp$) represents concrete implementable specifications, or “things we know how to build”. Connecting a node to $\perp$ states that the path from that model to a physical implementation is taken as understood.

A design methodology can be represented as a connected path from $\top$ to $\perp$. This path indicates that there exists a set of models and abstractions that can be used to implement some set of application specifications in some concrete form. Parallel paths represent alternative design methodologies, perhaps considering different application domains and architectural targets.

3.2.1 Design Abstraction Chart Example

A design abstraction chart can be used to describe the sequence of major design discontinuities that were presented in Chapter 1. This is shown in Figure 3.2. In transistor-level design, designers create very detailed system models using geometric layout tools. These tools give designers a very low-level abstraction of the system: a view of the actual masks that are used to manufacture a chip. This is something that can be taken directly to implementation, therefore it is connected directly to $\perp$.

Gate-level design adds another layer on top of this. Designers create more abstract models that describe compositions of logic gates instead of individual transistors. The gate-level abstraction

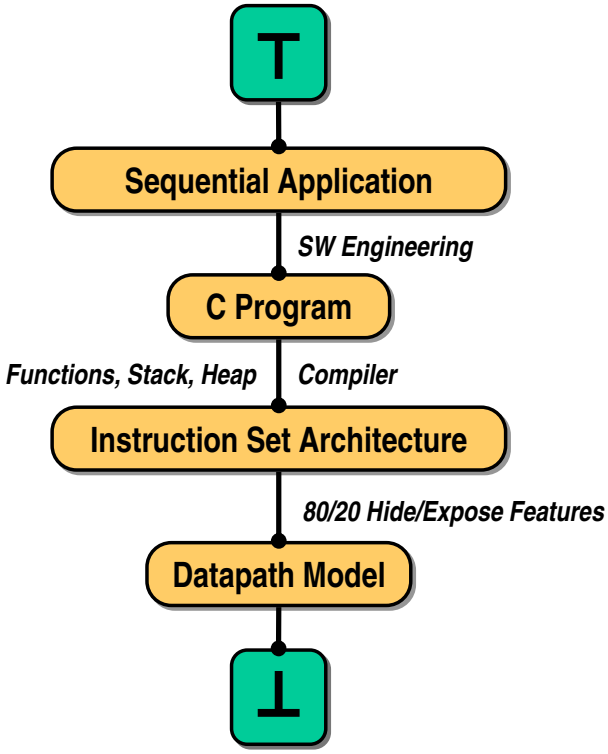


Figure 3.3: Design Abstraction Chart for Programming Uniprocessors in C

simultaneously assists and restricts designers in that it allows them to work at a coarser level of granularity while hiding the details of how gates are implemented using transistors. There is an automatic process based on standard cell libraries and place-and-route algorithms that converts a gate-level model into a transistor-level model.

Register-transfer models written in languages such as Verilog and VHDL again assist and restrict designers. Work is done at the level of compositions of state machines instead of at the level of individual gates. RTL synthesis algorithms make up the process that goes from RTL models toward implementation.

An edge between models need not be an automatic transformation process. It can be a manual technique or a self-imposed design discipline such as an architectural design pattern [31]. The *hardware reuse in time* pattern is a restriction on top of arbitrary RTL designs that leads to datapaths. In a datapath model, designers abstain from creating arbitrary compositions of state machines and instead create particular compositions of register files, multiplexers, and arithmetic function units.

Figure 3.3 continues this example. The path from datapaths to $\perp$ is condensed since it is now well understood. An instruction set architecture (ISA) is a model that captures some of the

functionality of a datapath while hiding a majority of the details. Following the Pareto principle, a good balance is to export the 20% of the features that are necessary to obtain 80% of the possible performance of the architecture. An ISA models a datapath as a machine that executes sequential lists of instructions. It hides implementation details such as the datapath structure and the amount of time it takes to execute each instruction.

Now coming from the opposite direction, there exists a domain of predominately sequential desktop computing applications. This is a subset of "things we want to build" and is thus connected to $\top$. The abstraction of sequential computation provided by the C language is a good match for both this application domain and the ISA abstraction. Designers use software engineering to turn an abstract specification of a sequential application into a more concrete C program.

C also makes further abstractions on top of an ISA. The instructions for jumping within a program are hidden behind a model of process-level concurrency based on function calls. Another concept is that the memory is divided into a stack and a heap. The stack is hidden from programmers, while the heap is accessed through a dynamic memory allocation API. A C compiler is an automatic process for going from a C-level model to an ISA-level model. The ISA-level model is simply an assembly language program.

A C program is not a completely concrete model because it does not specify how computations are to be implemented in hardware. Execution on a datapath processor is only one possible implementation strategy. Another option is to use a behavioral synthesis approach such as CatapultC [82] or Cynthesizer [103] to build an RTL circuit that implements the functionality described in the C program. These methodologies take models written in C down different paths in the design abstraction chart toward implementation. They do not use C as an abstraction of an ISA, but instead interpret C programs more directly as descriptions of sequential computations. These approaches will be discussed further in Section 3.3.2.

3.2.2 Visualizing the Concurrency Implementation Gap with a Design Abstraction Chart

In the previous example, the concurrency requirements of sequential applications and the concurrency capabilities of sequential datapaths are a good match. Therefore there is no significant concurrency implementation gap. Moving forward to modern systems with heterogeneous concurrency at both the application and architecture level, a design abstraction chart will show the concurrency implementation gap clearly.

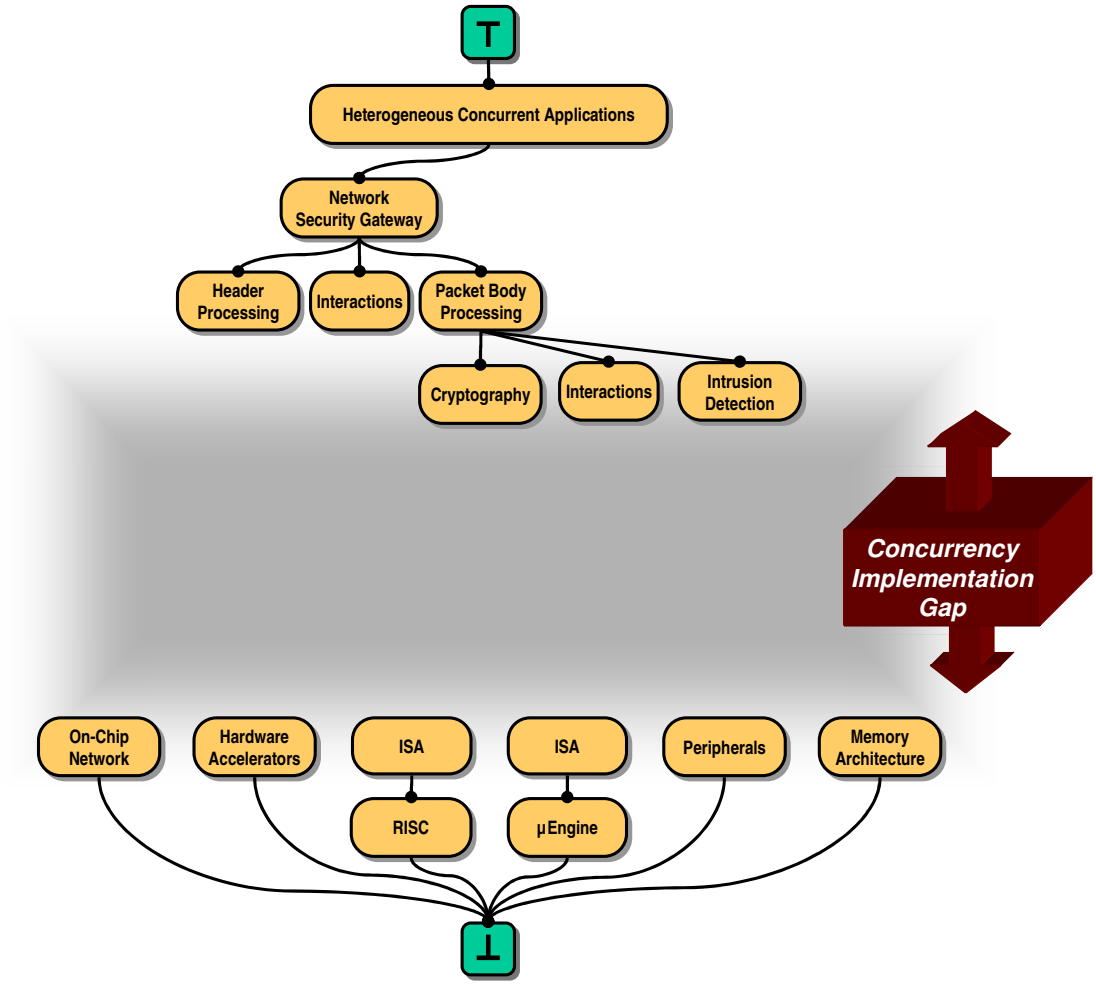


Figure 3.4: Design Abstraction Chart for Concurrent Applications and Heterogeneous Multiprocessors

Figure 3.4 demonstrates this case. On the architecture side, there is now a heterogeneous multiprocessor architecture. There is no longer a single processing element that can be abstracted with a single ISA. Instead, there is a variety of processing elements that have different ISAs. In addition, there are many elements that cannot be adequately exported with an ISA abstraction, such as hardware accelerators, peripherals, the on-chip network, and the memory architecture.

On the application side, sequential desktop computing applications are replaced with heterogeneous concurrent applications. These applications have multifaceted concurrency requirements. The network security gateway from Chapter 1 can be broken down into several parts that are modeled separately. There is a header processing part and a packet body processing part. Packet body processing involves both cryptography algorithms and intrusion detection algorithms. Furthermore, there are complex interactions between these application facets that are critical to the performance and functionality of the application. These nodes in the chart represent top-down refinements of the abstract application specification. They are still at the level of an English language specification.

In this example, there are significant differences between the concurrency requirements expressed at the application level and the concurrency capabilities of the architectural components. There is no single process that spans the gray area between these things. This is the concurrency implementation gap.

A casual approach to crossing the concurrency implementation gap might look something like Figure 3.5. In this case, the heterogeneous multiprocessor architecture is considered to simply be the sum of its parts. This is the "system-in-the-large" approach [91]. A different programming abstraction is provided for each of the parts, and no special attention is paid to the interactions between the parts. Datapaths such as RISC cores and network processor microengines are programmed with C. Application-specific hardware accelerators are accessed from within C programs using compiler intrinsics (e.g. in the case of the IXP programming environment). Programmers are expected to use assembly language to manage the on-chip network and the peripherals.

Likewise, the application is modeled as the sum of its parts. Programmers use a variety of ad hoc methods to model the different parts of the application, and then to divide the computations across the architectural elements. This is the "mess of manual processes" shown in the figure. Programmers are on their own to ensure that their partitioning correctly implements the functionality required by the application specification and that all of the interactions between the application facets are handled properly.

This methodology exemplifies the systemic ad hoc approach to concurrency that is impeding the deployment of heterogeneous multiprocessors. It breaks all three of the core requirements. First,

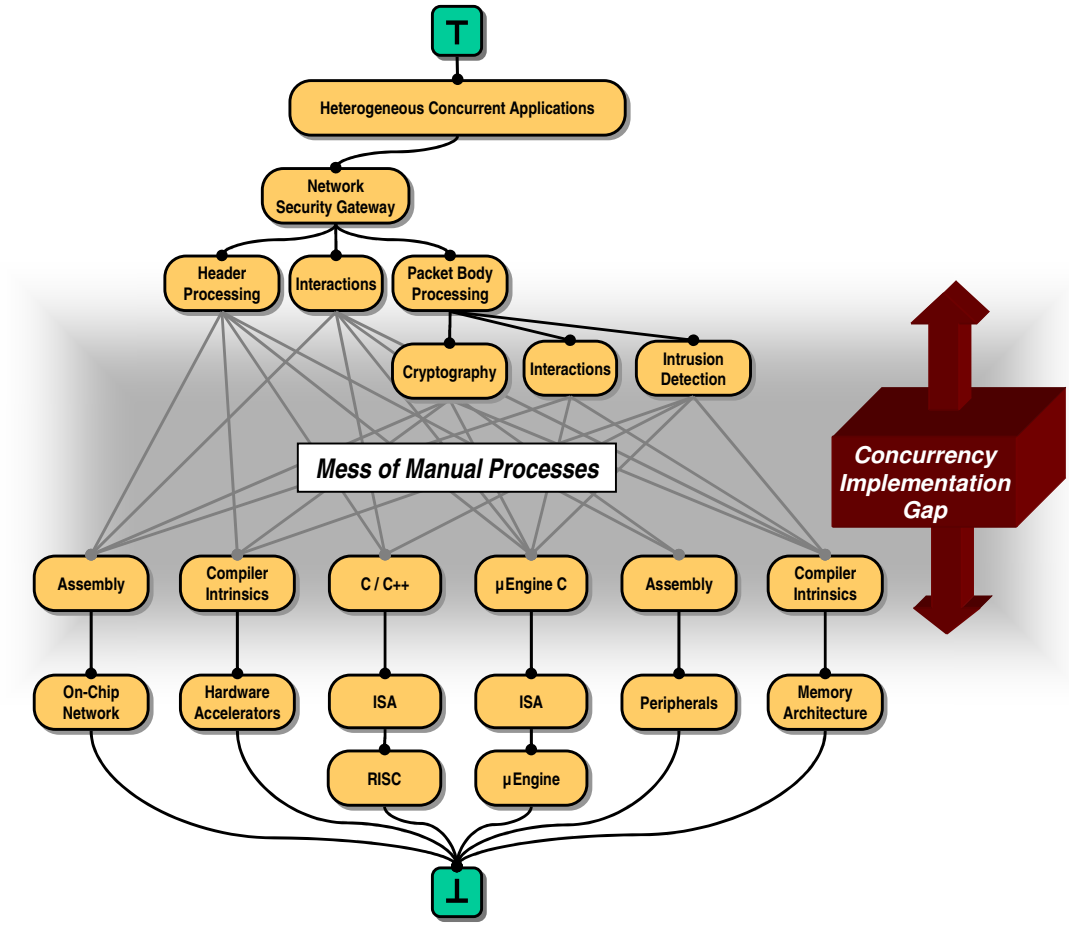


Figure 3.5: The System-in-the-Large Approach

there is no coherent model of the application's requirements. Programmers are going directly from an abstract specification to a partitioned implementation in low-level programming languages. Thus, the application exists only as a disjoint set of solutions to subproblems. The interactions between the application subproblems are not modeled explicitly. Instead, they are spread across and hidden within the separate subproblem solutions.

Second, this approach is not focused on simple architectures. It only considers traditional architectural elements that have well-known programming models. RISC processors are fine for general-purpose sequential applications, but they do not match the application-specific process-level, data-level and datatype-level concurrency found in heterogeneous applications. Simple architectures will try to better match application concurrency, and will not necessarily fit under an ISA and C abstraction. The system-in-the-large approach thus misses out on a compelling portion of the design space.

Third, there is no coherent model of the mapping of the application to the architecture. Designers used an ad hoc approach to divide the application across the architectural resources. The factors behind these decisions are not well documented. At best, there is a diagram on a whiteboard. At worst, there are some comments scattered throughout a collection of source code in different programming languages. This makes it impossible to understand the effect that the partitioning has on the function and performance of the system. Architectural artifacts and implementation artifacts are interspersed with the application code. A given piece of code may be solving an actual implementation problem, or it may be a workaround for a problem that is a result of a mapping decision. It is difficult to experiment with alternative mapping strategies because even small changes will require rewriting lots of code.

By breaking the three core requirements, the system-in-the-large approach fails to solve the concurrency implementation gap. Design is slow and not amenable to design space exploration. The benefits of programmable systems are not realized.

3.3 Related Work

The Cairn multiple abstraction approach is not the first attempt at solving the concurrency implementation gap. Early projects attempted to manage heterogeneous concurrency by providing only one or two new abstractions. By studying the design abstraction charts for these methodologies, one can see what they do right and where they fall short. Combining the best parts of these earlier approaches leads naturally to a multiple abstraction approach.

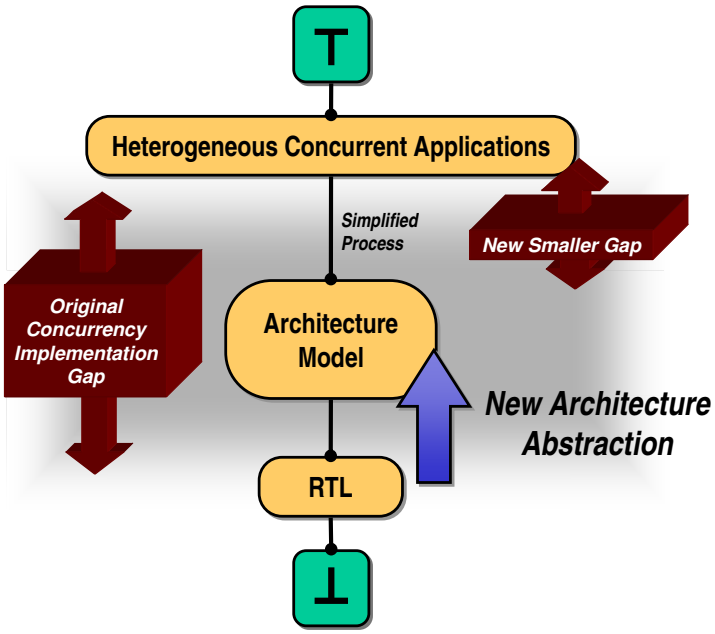


Figure 3.6: Bottom-Up Methodologies that Introduce a New Architecture Abstraction

This section divides related work into three categories base on the direction of the design flow. The first category of projects uses bottom-up design principles to solve the gap. Here, the strategy is to provide a new architectural abstraction on top of standard RTL languages. This new abstraction is meant to help architects create models of the target architecture at a higher level of abstraction and thus shrink the concurrency implementation gap between the architectural platform and the given application domain. The general form of these projects is shown in Figure 3.6.

The second category contains top-down methodologies. These projects introduce a new application abstraction that is meant to help system designers model more complex concurrent applications. Approaches in this category often include a process that is used to refine the abstract application specification toward implementation. The idea is that a more formal application model will be amenable to automated techniques for refinement and thus there will be a smaller gap to cross. These processes frequently encapsulate implicit design decisions and rely on a limited target architecture scope in order to function. The general form of these projects is shown in Figure 3.7.

The third and final category of projects use two or more novel abstractions to attack the gap from the top and bottom at the same time. These meet-in-the-middle methodologies are those most similar to the Cairn methodology presented in this dissertation. The structure of these projects is shown in Figure 3.8. The following sections will explore specific projects from each of these three categories in detail.

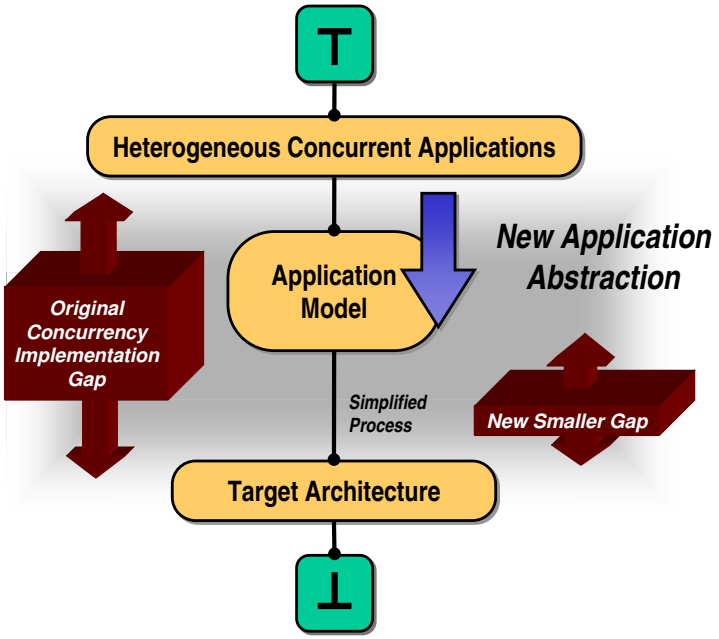


Figure 3.7: Top-Down Methodologies that Introduce a New Application Abstraction

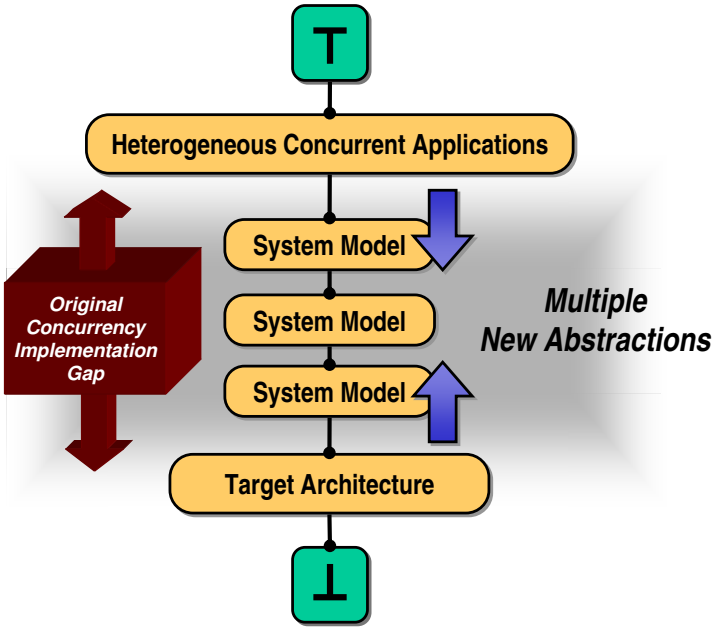


Figure 3.8: Meet-In-The-Middle Methodologies that Introduce Multiple New Abstractions

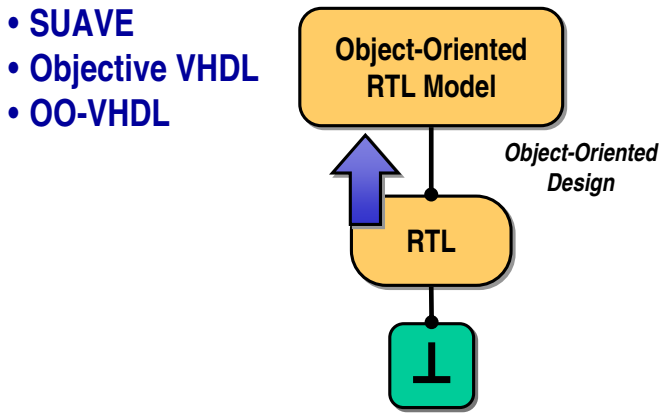


Figure 3.9: Extending Traditional RTL Languages

3.3.1 Bottom-Up Methodologies

Methodologies in this category introduce new abstractions for architecture-level concurrency on top of existing hardware description languages.

Object-Oriented RTL Languages

One philosophy suggests that RTL languages can be improved by adding better abstractions for datatype-level concurrency. Projects such as OO-VHDL [116], Objective VHDL [99], and SUAVE [3] attempt this by adding object-oriented features to RTL languages (Figure 3.9). Ashenden et al. [3] give a survey of other languages in this category. Traditional object-oriented languages help software programmers manage datatype-level concurrency by allowing them to define and reuse abstract data types. The theory is that by adding these same features to hardware description languages, architects will be able to enjoy the same benefits.

Unfortunately, the design patterns that work well for sequential code reuse are difficult to apply to hardware design problems. The root of this is a conflict in the semantics of software variables and hardware signals. A software object is an abstraction of one or more memory locations that store single values of given types. On the other hand, a signal can have multiple drivers. Driving a signal can activate processes that are sensitive to the value of the signal. Mixing these two semantics becomes unwieldy.

A second problem with these approaches is that focusing on datatype-level concurrency alone is not sufficient to solve large design problems. Objective VHDL succumbs to the flexibility fallacy and ignores process-level concurrency:

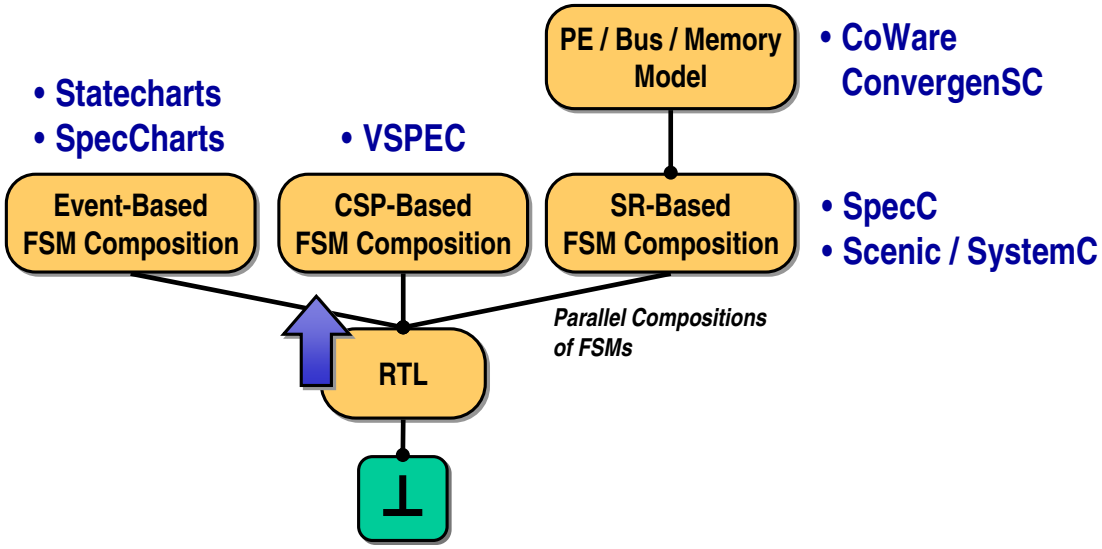


Figure 3.10: Abstractions for Parallel Compositions of FSMs

> "... Objective VHDL provides no pre-defined mechanism for message passing among entity classes because fixing a message passing mechanism would restrict the communication flexibility too much." [99]

Compositions of Finite State Machines

Other projects take the position that architecture design is fundamentally about the composition of finite state machines, and provide new process-level concurrency abstractions for doing this. Statecharts [34] and SpecCharts [89] are examples in this category. These languages allow architects to build complex compositions of parallel state machines using an event-driven model of computation. Processes are provided that transform the specifications into HDL code.

Figure 3.10 shows these projects and others that use more complex models of computation to describe the process-level concurrency between state machines. VSPEC [2] uses a Communicating Sequential Processes [52] model of computation to compose FSMs. Architects can write declarative functional constraints alongside their models and use a process algebra to prove that the constraints are met.

SpecC [39] and SystemC [45] (with roots in Synopsys' Scenic project [79]) compose FSMs in a synchronous-reactive model of computation. Both of these languages feature a *wait* statement that causes a process to synchronize to a communication event. CoWare ConvergenSC is a commercial product that provides coarser-grained components such as processors, buses, and memories on top of SystemC [58].

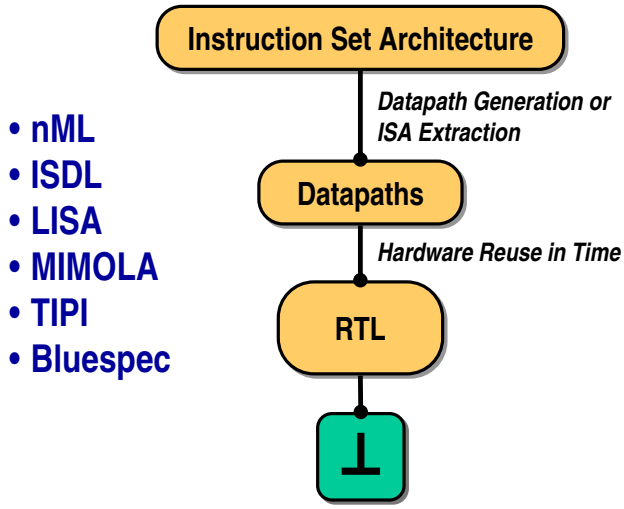


Figure 3.11: Architecture Description Languages

Architecture Description Languages

Architecture Description Languages (ADLs, Figure 3.11) advocate the use of a coarser-grained basic block than finite state machines. Instead of making arbitrary compositions of finite state machines, architects apply the hardware-reuse-in-time architectural design pattern to make programmable datapaths. By restricting themselves to this particular architectural class, designers get the benefit of an ISA abstraction that exports the capabilities of the architecture. Some ADLs provide a process that extracts an ISA model from a datapath model, while others extract a datapath model from an ISA model. Examples will be discussed in more detail in Chapter 4.

ADLs focus on data-level and datatype-level concurrency. Architects can pick the data types and computations they wish the machine to provide, and there is frequently the ability to specify how the machine can perform several computations in parallel. Process-level concurrency is often assumed to be a traditional sequential program with jumps. Thus some aspects of the machine's control logic, such as program counters and branch units, can not be customized to support the process-level concurrency of particular application domains. The ability to match all levels of concurrency is important in order to satisfy core requirement number two: consider simple architectures.

Pitfalls of Bottom-Up Methodologies

By itself, a new architecture abstraction is not sufficient to solve the concurrency implementation gap. The focus is on the design and modeling of a manufacturable architecture, not on the

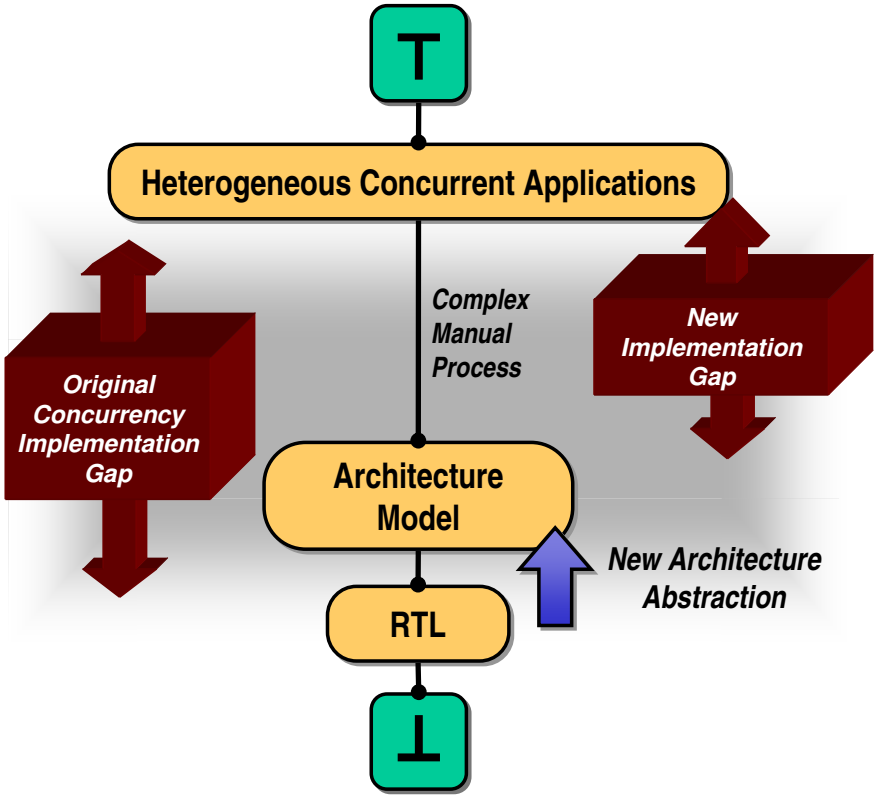


Figure 3.12: Pitfalls of Bottom-Up Methodologies

deployment of the architecture. Only the ADLs restrict designers to create programmable architectures. Unfortunately, even those tools leave off at the ISA abstraction where the capabilities of the architecture are exported to programmers. The process for getting from a heterogeneous concurrent application to the ISA is left up to the designers.

This situation is summarized by the design abstraction chart in Figure 3.12. A new architecture abstraction does make the concurrency implementation gap smaller because architects benefit from increased productivity in modeling machines that implement concurrency. However, the concepts expressed in these models are still heavily weighted toward implementation details. Architects are concerned with how machines store data, perform computations on data, and the cycle-by-cycle behavior of the machine. The objects of interest are register files, multiplexers, function units and control logic.

An application domain expert has different concerns and will have difficulty getting started with such a tool. In network processing, for example, programmers are concerned with the spatial and temporal processing of packets. The objects of interest are packet headers, queues, and routing

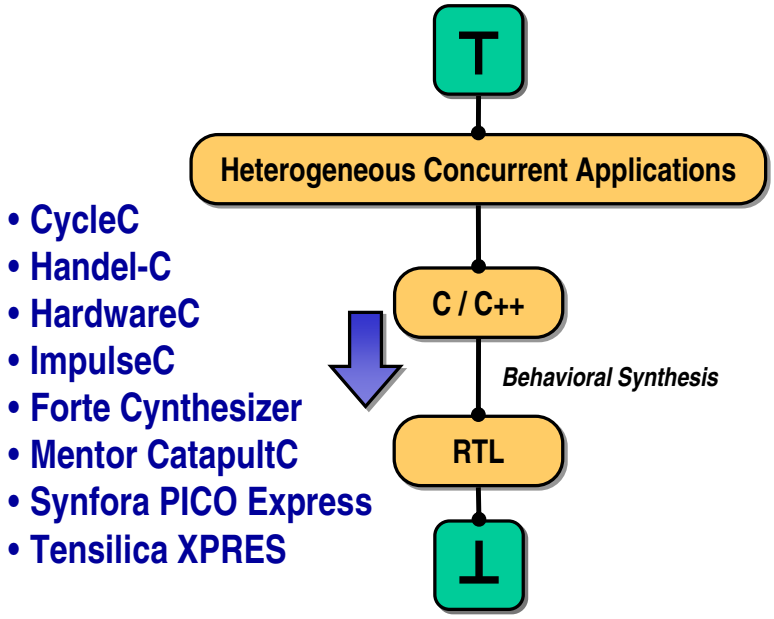


Figure 3.13: C Language Behavioral Synthesis

algorithms. These application concepts can be described using the new architectural abstraction, but this requires a complex manual design process. The domain expert must decide how to store packet header data in register files, what function units are necessary, how multiple processing elements will interact, how many memory interfaces to include, how big of an instruction memory to provide, what kind of pipeline structure should be used, and so on. These are critical implementation decisions that must be made before designers can even begin creating a model. The challenge is that none of these decisions fall within the realm of expertise of the network processing domain expert. Thus there is still a significant concurrency implementation gap that these methodologies fail to solve.

3.3.2 Top-Down Methodologies

The next category of related work takes the opposite approach. A new application-level abstraction is introduced to capture the requirements of concurrent applications, and a process is proposed for taking models designed with this abstraction toward implementation.

Behavioral Synthesis

One philosophy is that C is the appropriate language for application design because it is universally familiar, trivial to simulate, and has an enormous body of algorithms ready for reuse. The projects illustrated in Figure 3.13 start with a model written in a dialect of C or C++ and target implementation with a behavioral synthesis process. Since C is not capable of modeling concurrency, these tools often make significant efforts toward extracting implicit concurrency from regular code. Hints from the programmer are usually required and this leads to various small extensions to the basic C language.

HardwareC [85] adds the concept of processes that can communicate through shared memory or via a synchronous message passing system. The language also offers Boolean bit types and statically-sized bit vector types for describing datatype-level concurrency. Annotations identify a block of code as containing explicit data-level concurrency, or users can let the tool extract implicit data-level concurrency based on data dependency analysis. Hercules is the corresponding behavioral synthesis tool that transforms HardwareC specifications into datapaths [84]. Techniques such as loop unrolling, constant and variable propagation, common subexpression elimination, dead code elimination, conditional collapsing, and dataflow analysis are applied to perform this process.

The Compilogic C2Verilog Compiler (later C-Level Design System Compiler) creates an RTL datapath that matches a pure C program [111]. There is essentially a one-to-one correspondence between constructs in the C code and blocks in the architecture. Users of this tool discovered that it did a poor job of applying the hardware-reuse-in-time architectural design pattern to the datapaths it created. Datapaths with hundreds of levels of logic and control state machines with thousands of states were common. To remedy this, the CycleC cycle-accurate programming style was announced at the Design Automation Conference in 2000. This proprietary C dialect allows designers to add structural cycle boundaries to their C application models to help guide the tool toward better implementations. This technology was acquired by Synopsys in November of 2001 and shelved.

Recent years have witnessed a resurgence in popularity of C-based approaches. Forte Cynthesizer [103], Mentor CatapultC [82], and Synfora PICO Express [59] are examples of commercial tools that start with ANSI C and target RTL synthesis. Application designers use C directly and perform functional testing and debugging on their development workstations. The supplied behavioral synthesis tools produce a datapath, control logic, and memories that provide equivalent functionality. Designer assistance is required to help produce efficient datapaths. These tools ex-

tract control-dataflow graphs from the C source and allow the user to visualize the effect of loop unrolling and pipelining transformations on the generated architecture. This allows exploration of different area/performance trade-offs.

Impulse Accelerated Technologies CoDeveloper [54], the Handel-C DK Design Suite [56, 57] from Celoxica, and the Tensilica XPRES Compiler [61] focus on accelerating kernels of C programs. ImpulseC makes FPGA accelerators for C processes that appear as IP blocks in Xilinx's Platform Studio. Designers can then compose these blocks with embedded processor cores that run the remainder of the C code natively. In Celoxica's tool, application kernels described in Handel-C interact with the rest of the application code through streaming I/O functions. Verification and performance analysis is done using co-simulation of the Handel-C accelerators and the C software. The XPRES Compiler focuses on implicit data-level concurrency in the C code. Commonly used sequences of arithmetic operations are merged into a single instruction that is implemented as an Xtensa processor instruction extension. The tool can also create so-called FLIX instructions which can be issued in parallel on a multiple-issue machine. Support is provided for exploring Pareto-optimal solutions that balance cycle count reduction against total hardware area.

C language behavioral synthesis approaches have the same general shortcomings, which are shown in Figure 3.14. First, they have narrow architectural scope. The tools only target the specific architectures that they generate, and not an architecture provided by the designer. On top of this, designers have limited ability to influence the architecture that is created. This is because the tools encapsulate many design decisions to support the goal of automation.

Second, since C is poor at modeling concurrency, the tools must infer concurrency from the code. The opportunities for this are usually limited to instruction-level parallelism and pipelining. ILP reaches its maximum potential at approximately 5 simultaneous instructions, and pipelining only works well for structured loops and not for irregular computation [35]. Therefore, the performance of systems designed with these methodologies will be suboptimal because the input specification does not contain a great deal of concurrency to exploit in hardware.

Finally, there is still a large concurrency implementation gap between heterogeneous concurrent applications and the C language. Programmers must use ad hoc methods to partition their application into a set of interacting C programs. The consequences of this were detailed in Section 1.3.2.

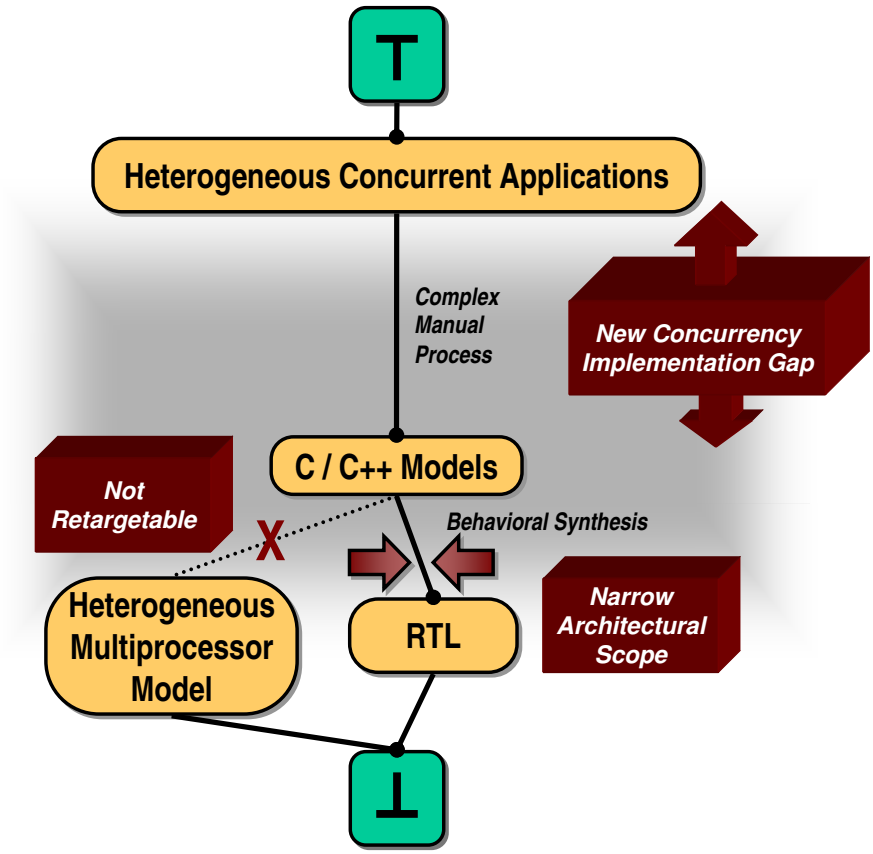


Figure 3.14: Shortcomings of C Language Behavioral Synthesis

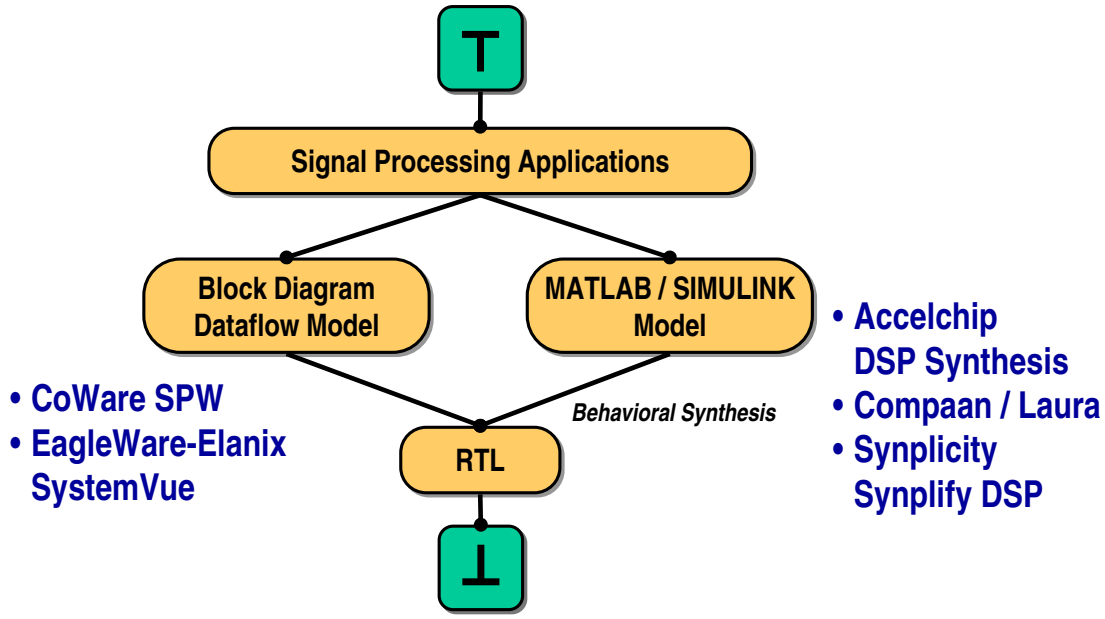


Figure 3.15: Domain Specific Languages for DSP Applications

Domain-Specific Languages

There is a separate category of projects that admit that C is a poor abstraction for capturing concurrency and instead advocate the use of domain specific languages (DSLs). In the signal processing domain, block diagram dataflow models are known to be intuitive for DSP experts and are capable of capturing the streaming nature of DSP applications. Projects that provide a behavioral synthesis process to refine from a dataflow model to an RTL implementation are shown in Figure 3.15.

CoWare SPW [58] and EagleWare-Elanix SystemVue [24] provide their own graphical editors and actor libraries for building dataflow application models. Accelchip [55], Compaan/Laura [124, 115], and Synplify DSP [60] use MATLAB and/or SIMULINK as input languages. In Accelchip's tool, Accelware hardware IP blocks implement fixed-point versions of MATLAB toolbox functions. Users must manually edit their MATLAB source code to replace instances of toolbox functions with Accelware functions. The RTL output integrates with Xilinx System Generator to build an FPGA-based system that is a combination of sequential-program processor cores and Accelware IP blocks. Accelware components focus on implementing the data-level and datatype-level concurrency specified in the MATLAB input model.

Synplify DSP uses behavioral synthesis to implement a discrete-time SIMULINK application in RTL. Designers provide synthesis constraints including the target FPGA architecture, the desired sample rate, and the speed requirement. The tool performs retiming, resource allocation, resource scheduling, and multichannelization.

The Compaan/Laura approach starts with applications written in a subset of MATLAB that permits only static nested loops. The Compaan tool analyzes this code and turns it into a process network. The Laura tool then converts the process network into a network of hardware processors described in VHDL. This synthesis step creates inter-processor communication links in hardware that match the semantics of the communication links between processes in the process network. Four varieties of links based on FIFOs are identified and implemented using Xilinx FIFOs and dual-port Block RAMs.

In the network processing domain, Click is the de facto domain-specific language [72]. Figure 3.16 shows the various ways to refine a Click model toward implementation. The standard MIT Click toolkit provides a C++ library of objects that implement Click elements, and a scheduler that implements the process-level concurrency described in a Click model. The result is a C++ program intended to run on a uniprocessor Linux PC. For a multiprocessor Linux machine, the same approach to implementing process-level concurrency is not effective because it does not consider the

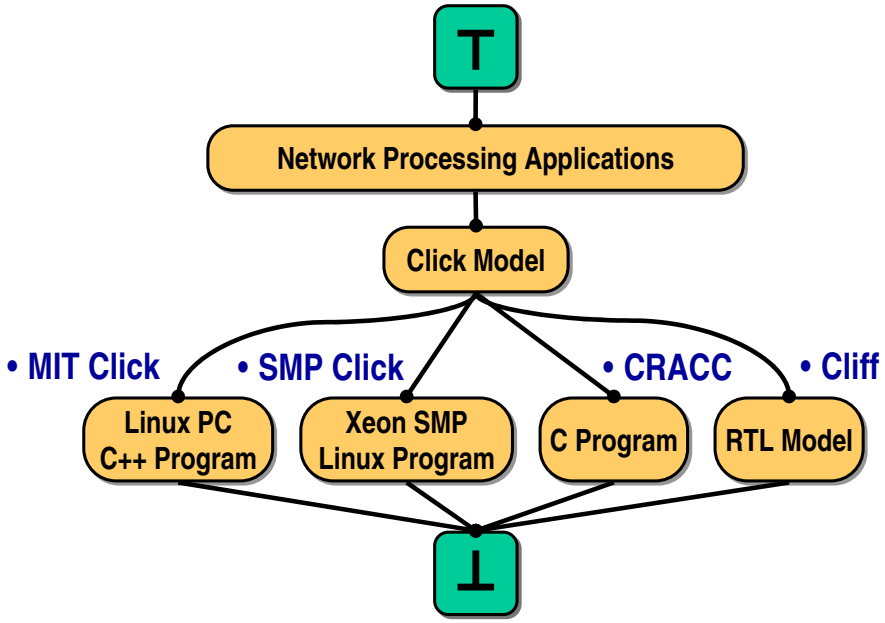


Figure 3.16: Domain Specific Languages for Network Processing

performance impact of synchronization and communication between processors. The SMP-Click approach was introduced to tackle this issue [21].

CRACC stands for "Click Rapidly Adapted to C Code" [104]. This approach uses an object-oriented programming style on top of regular C code to implement Click processes on small embedded processors in a multiprocessor context. The target platform of interest is an Infineon DSLAM backplane and multiple line card assembly.

The Cliff approach provides a library of RTL cores that implement Click elements. A synthesis process turns a Click model into an FPGA implementation by replacing elements with cores in a one-to-one fashion. It then connects the cores with point-to-point handshaking communication links that match the Click push and pull communication semantics.

Pitfalls of Top-Down Methodologies

Like the bottom-up methodologies, these top-down methodologies are also insufficient by themselves for solving the concurrency implementation gap. Figure 3.17 summarizes the important pitfalls. First, a single DSL cannot model all of the facets of a heterogeneous application. Click is excellent for header processing, but a routing application is more than just header processing. A router must also process packet bodies and respond to dynamic routing updates. Different DSLs are needed to model the different parts of a full application, and it is critical to model the interactions be-

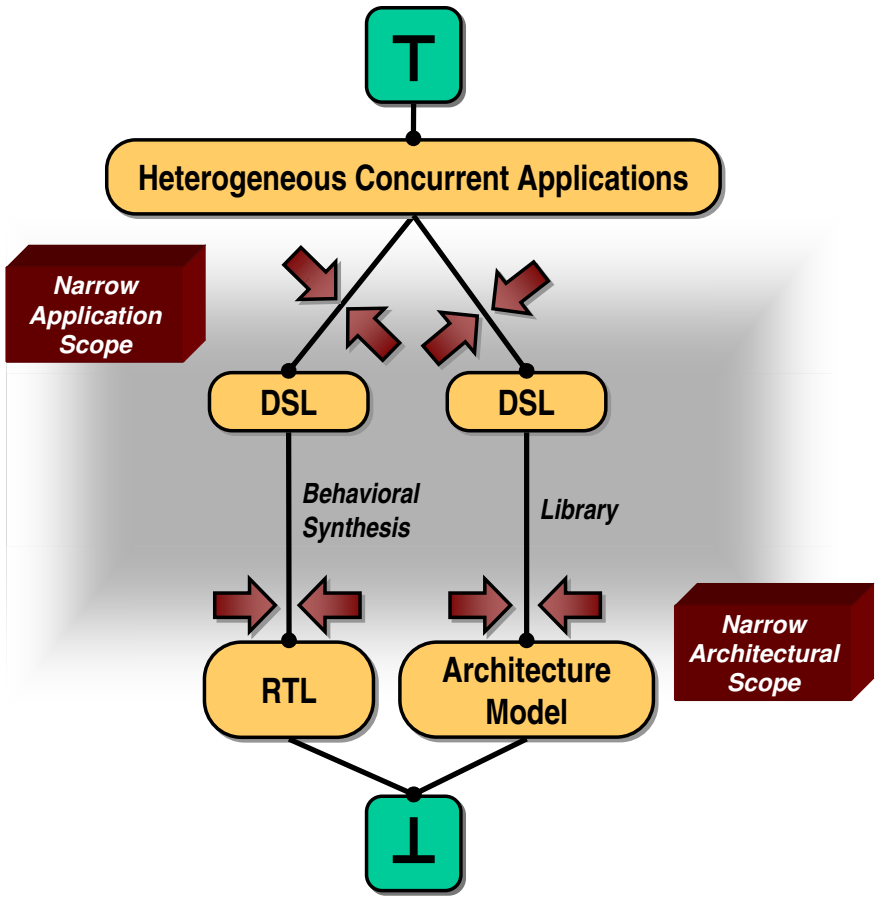


Figure 3.17: Shortcomings of DSL Implementation Methodologies

tween the different parts in addition to the contents of the parts. Existing DSL-based methodologies do not consider multiple DSLs and interactions between DSLs.

Second, the behavioral synthesis processes that are commonly used to refine DSLs suffer from a narrow architectural scope. Designers cannot target an arbitrary architecture, but are instead forced to accept the architecture that these tools generate. In order for these tools to work in an automated fashion, they must hide numerous design decisions from the user. Thus architects can exert little influence on the end result. If performance constraints cannot be met there are few options for improvement. There is no separate model of the architecture that can be designed and explored independently of the application model.

If the DSL's implementation process uses a software library to target a specific architecture, a similar argument applies. The library is designed with a particular machine in mind. To target a different architecture, the library must be redesigned.

3.3.3 Meet-in-the-Middle Methodologies

From studying the pitfalls of the top-down and bottom-up methodologies, it is clear that more than one design abstraction is necessary to solve the concurrency implementation gap. This third category of related work does exactly that. This section discusses six multiple abstraction approaches: ForSyDe, NP-Click, Gabriel, Artemis, CoFluent Studio, and Metropolis. ForSyDe introduces multiple application abstractions for modeling heterogeneous applications. The latter five methodologies introduce separate abstractions for applications and architectures.

ForSyDe

ForSyDe stands for "Formal System Design" [102]. The goal of this methodology is to model applications at a high level of abstraction and use model transformations to refine the application toward an implementation. A model transformation may or may not be semantics-preserving. A semantics-preserving transform does not effect the functionality of the application. A transform that changes the application functionality is called a design decision transform. This transform encapsulates a choice that designers have to make as to how to implement an application concept.

Figure 3.18 shows a design flow chart for the ForSyDe methodology. Designers begin by writing an abstract application model. Transforms from a pre-existing library are then selectively applied to the model. This is an iterative process where design decisions and optimizations are made to refine the application model into an implementation model. Finally, the implementation model is

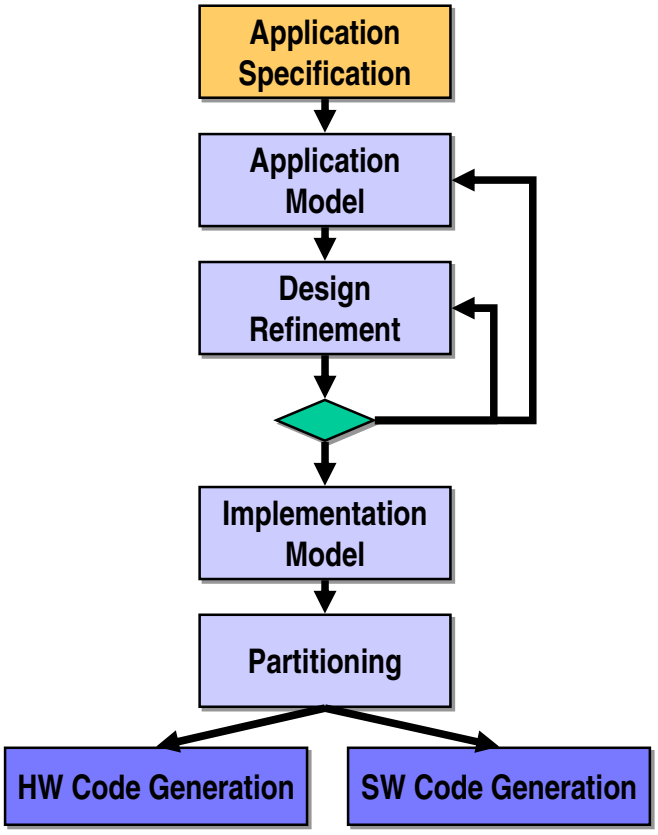


Figure 3.18: ForSyDe Design Flow Chart

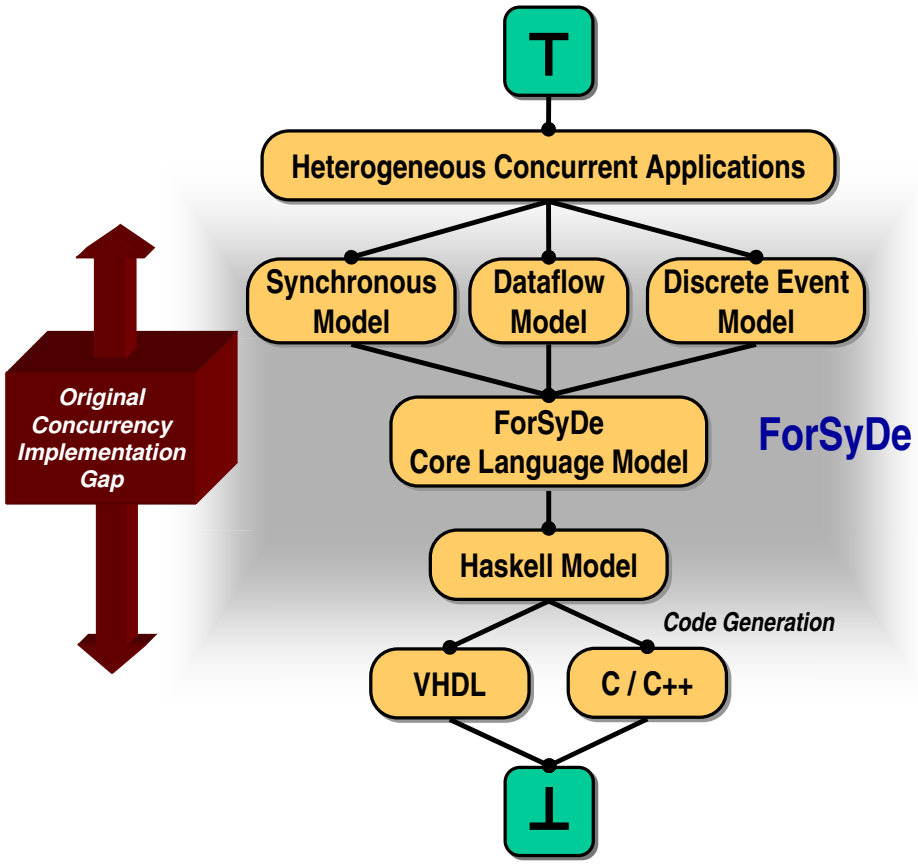


Figure 3.19: ForSyDe Design Abstraction Chart

manually partitioned into hardware and software components. Code generators output synthesizable VHDL for the hardware components and C for the software components [80].

Figure 3.19 shows ForSyDe's design abstraction chart. ForSyDe is similar to the top-down design methodologies described in the previous section, with the exception that there are multiple models of computation provided for modeling applications. This allows ForSyDe to avoid a major pitfall of the other approaches in that one model of computation is not sufficient to model heterogeneous applications. Designers can instead mix synchronous models (similar to synchronous-reactive languages such as Esterel) with dataflow models and discrete event models to capture multifaceted applications.

The ForSyDe Core Language is a unifying semantic domain that supports these different models of computation. At this abstraction level the application is treated as a network of processes that communicate using signals. Communication is based on the *perfect synchrony hypothesis*. This hypothesis states that process outputs are synchronized with process inputs and the reaction of the

system takes no observable time. The application abstractions that designers use (synchronous, dataflow, and discrete event) are each different restrictions on the set of behaviors that can be modeled in the core language.

The core language itself is built on top of the purely functional language Haskell. Haskell is chosen instead of a general-purpose language like C++ or Java because it is side-effect free, deterministic, and strongly typed. These qualities help ForSyDe enforce the restrictions defined by the core language abstraction. With C++ or Java, designers could cheat the system by using language features such as ambiguous pointers that violate the semantics of the core language. A formal language like Haskell thus helps satisfy one of the major criteria of a good application abstraction, namely the ability to make a precise representation of the application's requirements.

A downside to the ForSyDe approach is that it does not consider mapping the application to a heterogeneous multiprocessor architecture. Processes can be synthesized as either hardware or software components, but there are no abstractions provided for modeling an architecture independently of the application or for modeling the mapping of the application across hardware and software.

Designers must use ad hoc methods to accomplish these goals. Therefore, ForSyDe has problems similar to those of the behavioral synthesis approaches described previously. The final system design is the result of a top-down refinement process starting from a specific high-level application model. If designers want to explore the architectural design space, they cannot make modifications to the architecture directly. Instead, they must backtrack through the refinement steps and make different decisions that hopefully result in the desired architectural changes. Experimenting with several different applications on the same architecture is equally difficult. The refinement process must be repeated for each application. Each time, designers must carefully make design decisions so that the result has the same hardware architecture as the original platform.

Methodologies based on the Y-chart (Figure 3.20) solve these problems by allowing applications and architectures to be developed independently and in parallel. Different abstractions are used for modeling applications and architectures. Designers can map the same application onto several different architectures, or several applications onto the same architecture. The remaining projects discussed in this section provide abstractions that support a Y-chart design flow.

NP-Click

NP-Click is a methodology for implementing network processing applications on network processor architectures [106]. The design abstraction chart for this approach is shown in Figure 3.21.

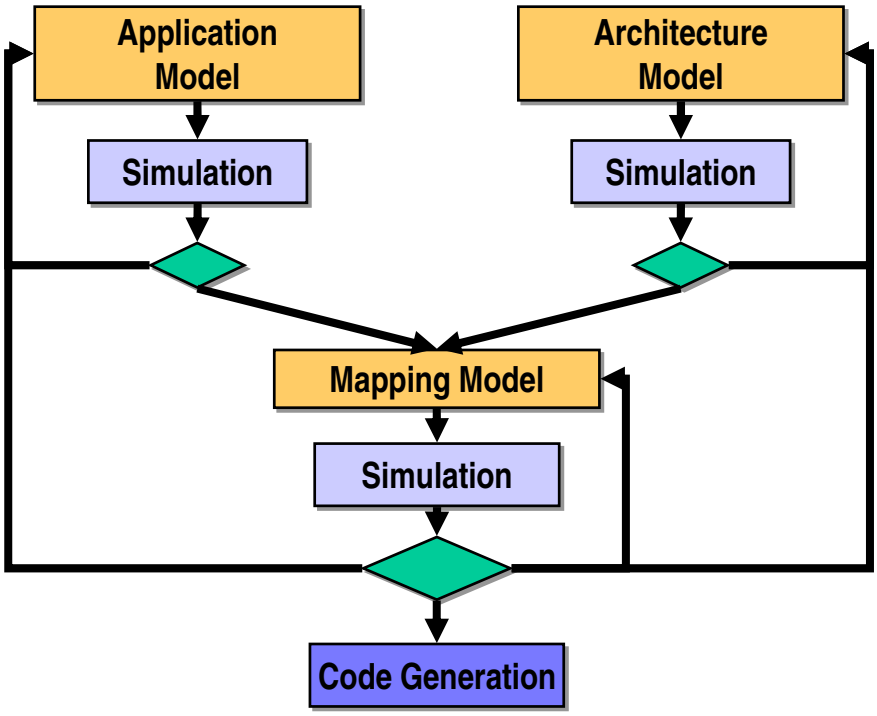


Figure 3.20: Design Flow Chart for Y-Chart Methodologies

There are three abstractions used.

First, domain experts model network processing applications using the Click language. At this point it is important to reiterate the difference between the Click language and the Click toolkit. The Click language is the syntax and grammar that programmers use to describe packet processing applications. There is both a text-based language and a graphical block diagram representation described in the original Click publications [72]. The Click toolkit is a package of downloadable software that contains the tools necessary to compile applications written in the Click language and run them on Linux-based computers. One can view NP-Click as an alternative toolkit for the Click language that targets implementation on network processors instead of Linux computers. The language abstraction is the same, but the implementation flow and the supporting tools are different.

Second, NP-Click provides an abstract model of the target network processor that provides visibility into the features that are necessary to obtain performance while hiding extra details. In this abstraction, the network processor's programmable elements (PEs) are viewed as computational resources that can implement the computations found inside Click elements. Each PE has a finite amount of instruction memory which limits the number of elements that it can implement. Furthermore, each PE is characterized by the amount of time it takes to execute each kind of Click element.

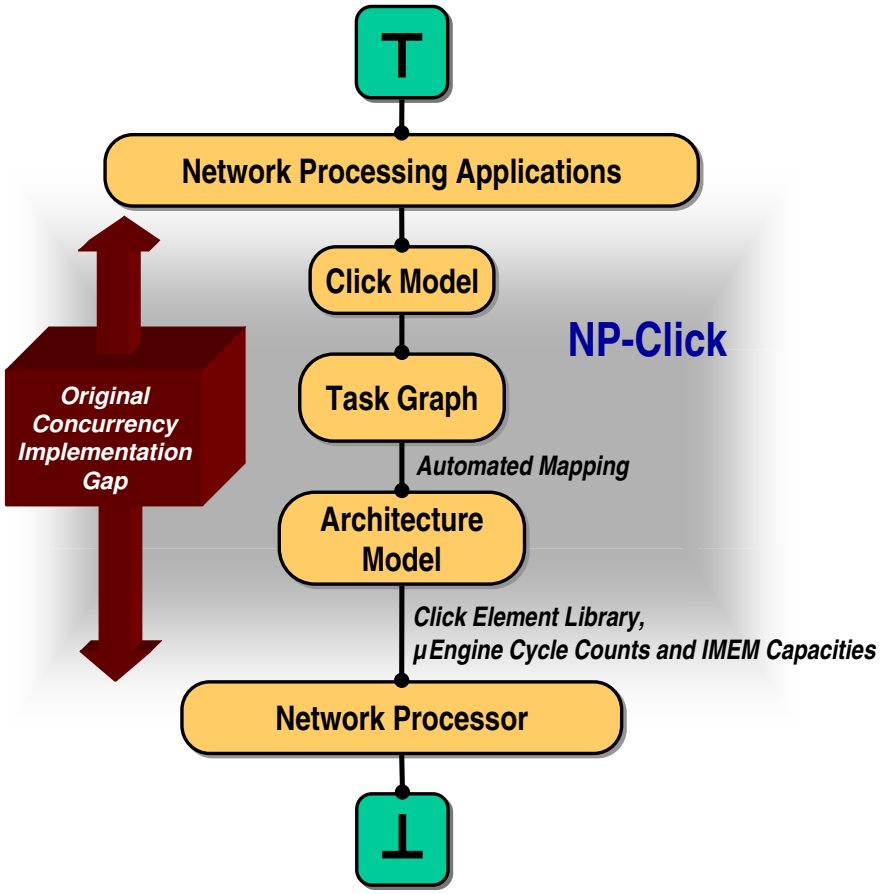


Figure 3.21: NP-Click Design Abstraction Chart

This abstraction of the architecture is sufficient to obtain high-quality implementations.

Along with an abstract model of the architecture, NP-Click provides a pre-written and pre-characterized element library. Developing this library for a given network processor architecture is a manual task that involves porting Click elements into the native programming language of the PEs. The push and pull interfaces for the elements must also be implemented using the on-chip network communication capabilities of the architecture. Currently, this work has been done for only the IXP1200 and IXP2800 network processor architectures [107, 97].

NP-Click uses an intermediate task graph to map Click elements onto processing elements. A task graph is a refinement of the process-level concurrency described in the Click model into distinct processes that are bounded by queues. Each process is a chain of one or more Click elements. This formulation is obtained from the Click model by an automatic analysis process.

The downward link in the design abstraction chart from task graph to architecture model is also an automated process. This mapping process assigns processes from the task graph onto PEs in the architecture [98]. This is formulated as an optimization problem that tries to maximize the packet throughput of the application while staying within the instruction memory constraints of the individual PEs. The characterized execution times of the Click elements on the PEs are provided as inputs, and the problem is solved using 0-1 integer linear programming.

The NP-Click methodology can produce implementations that are within 5% of the performance of hand-coded designs with much higher designer productivity [106]. This shows that the automated design space exploration is successful and the use of multiple abstractions does not detract significantly from the quality of results. A major benefit of NP-Click's separate application and architecture abstractions is that the same application can be implemented on different network processor architectures. This is in contrast to a top-down synthesis approach such as Cliff, where designers cannot supply their own architecture.

One downside to NP-Click is that the architectural abstraction cannot be automatically extracted from the network processor architecture itself. Therefore, while it is theoretically possible to perform iterative architectural design space exploration with NP-Click, it is not practical because developing a new architectural abstraction takes a great deal of time.

Another downside is that the NP-Click application abstraction is only useful for modeling the header processing component of network processing applications. Designers will need more than just the Click language to model heterogeneous applications.

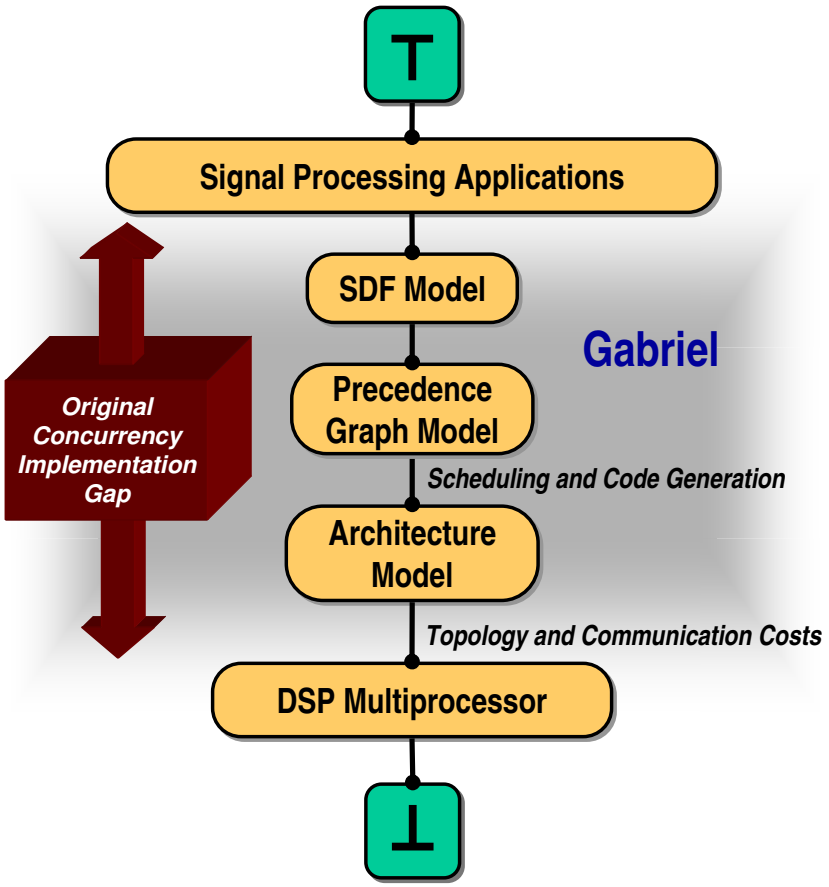


Figure 3.22: Gabriel Design Abstraction Chart

Gabriel

The Gabriel methodology implements signal processing applications on multiprocessor architectures composed of digital signal processor PEs [12]. The design abstraction chart for this approach is shown in Figure 3.22. Like the NP-Click approach, three abstractions are used.

Signal processing applications are modeled using an application abstraction based on the Synchronous Dataflow model of computation. Gabriel provides a graphical block diagram editor for creating these models. Computational elements, called *stars*, consume data tokens on their input ports and produce output tokens on their output ports. A composition of stars is called a *galaxy*. Galaxies can be further composed to make a hierarchical application model called a *universe*. Gabriel provides a library of stars that implement common signal processing computations.

Gabriel's architecture abstraction is designed to capture the important features of a variety of multiprocessor DSP architectures. Architects model an architecture by describing the number

and type of DSP PEs, the memory maps for each PE, and the shared memory topology. Architects also provide *send* and *receive* functions that implement synchronous interprocessor communication. When an application is mapped to the architecture, these functions will be used to move application data tokens between stars on different PEs.

Like the NP-Click element library, the Gabriel star library is written for a specific DSP architecture. If one wishes to build an architecture with a novel type of PE, the star library must be ported to this PE. Unlike NP-Click, Gabriel stars are not written directly in the native programming language of the target PEs. Instead, they are LISP code generators that execute to produce code in the native programming language.

Gabriel uses an acyclic precedence expansion graph (APEG) as an intermediate mapping model. This formulation is derived automatically from a Synchronous Dataflow application model by finding a static schedule for the dataflow graph. Each node in the APEG represents a single invocation of a star, and the edges represent the flow of tokens between invocations of stars. A single star in the application model may appear multiple times in the APEG based on the static schedule.

An automated mapping algorithm maps computations from the APEG onto the processing elements of the target architecture. This is an optimization problem that tries to minimize the time it takes the architecture to compute a complete application schedule, while taking into account interprocessor communication and synchronization costs. A variety of heuristics are used to solve this problem.

Separate application, architecture, and mapping abstractions allow users of Gabriel to independently explore the application, architecture, and mapping design spaces. The same application can be quickly evaluated on several different architectures. Different mapping strategies can be tried without modifying the application or the architecture. A downside to the Gabriel approach is the limited application scope. The Synchronous Dataflow application abstraction is ideal for certain signal processing applications, but heterogeneous applications will require a richer set of application abstractions.

Artemis

Artemis stands for “Architectures and Methods for Embedded Media Systems” [96]. A major theme in this work is to implement a range of applications with varying demands on highly programmable architectures. To explore the design space of these architecture effectively, Artemis seeks to provide simulation tools that work at multiple levels of detail. Early in the design process,

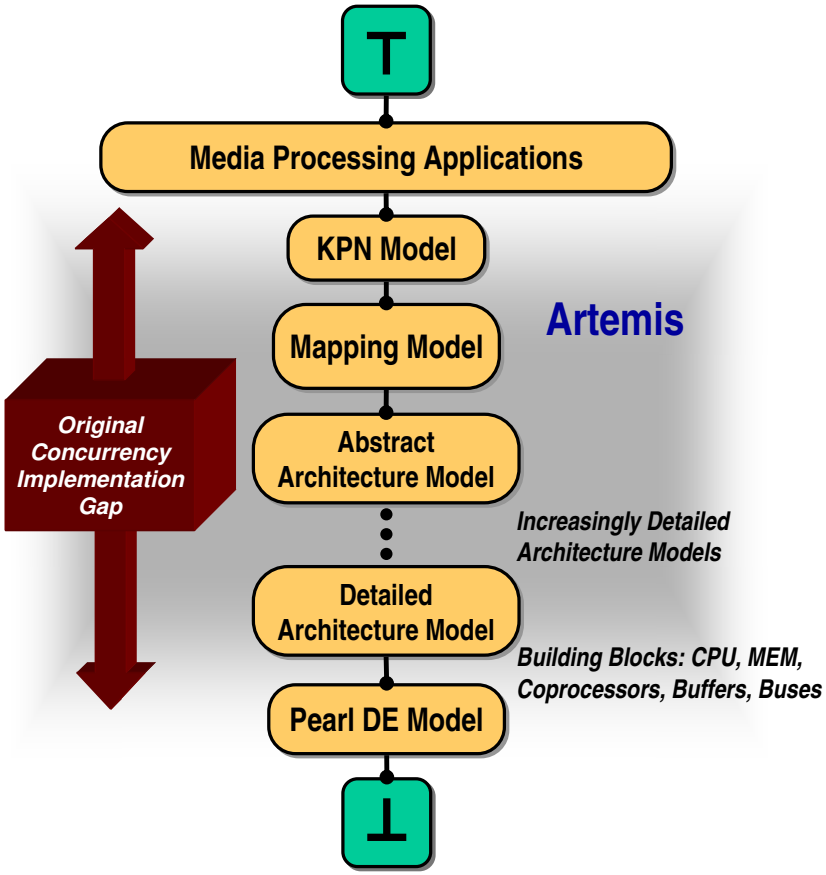


Figure 3.23: Artemis Design Abstraction Chart

back-of-the-envelope calculations and abstract executable models help users make coarse design decisions with small amounts of effort. Later, more expensive cycle-accurate models and synthesizable RTL models help users make fine design decisions leading to a final system implementation.

The Artemis application abstraction is based on the Kahn Process Networks model of computation. This is effective at capturing the process-level concurrency of streaming media applications. Processes are written in C or C++ and communicate over theoretically unbounded FIFO channels.

The Artemis architecture abstraction models a multiprocessor as a graph of processors, coprocessors, memories, buffers, buses, and other coarse-grained components. An edge in this graph represents the set of wires that connect two components together. The abstraction hides the details of these connections.

There can be several versions of each architectural component that model the behavior of that component at different levels of detail. At a low level of detail, a processor block may model only the timing and not the functionality of a processor. This makes simulation faster, and consequently design space exploration is less expensive. These building blocks are created using the Pearl discrete-event simulation and modeling language [88].

In order to map applications onto Artemis architectures, users assign Kahn processes and FIFO channels to architectural components such as processor cores and buses. This is done in an explicit mapping step that uses an independent mapping abstraction. Users can explore the mapping design space by making different assignments of processes and channels to architectural resources. Unlike Gabriel, mapping in Artemis is a manual process.

The mapping abstraction is based on simulation event traces. When an application process interacts with a Kahn channel a simulation event is recorded. The assignments given in the mapping abstraction direct these simulation events to the appropriate architectural components. The architectural components perform trace-driven simulation to generate timing results. There can be many application components mapped to the same architectural component. In this case, the architecture simulator must schedule the events from these multiple sources.

One downside to the Artemis approach is that application processes are written in C. The Kahn Process Networks model of computation is a good abstraction for process-level concurrency, but C falls short at modeling the data-level and datatype-level concurrency within processes. The use of C also makes implications on the minimum complexity of PE architecture that can be supported. PEs must have program counters, instruction memories, and local memory for a stack and a heap. For some media applications this level of hardware complexity is not always necessary.

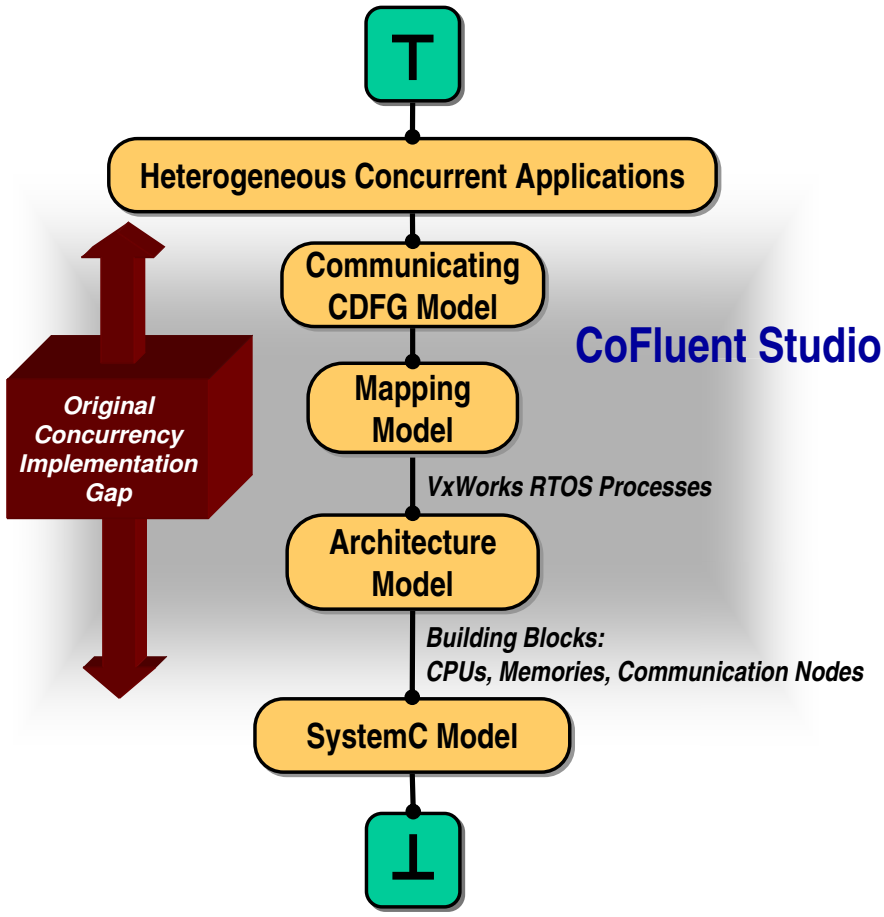


Figure 3.24: CoFluent Studio Design Abstraction Chart

CoFluent Studio

CoFluent Studio is a commercial offering based on research done at the University of Nantes in France [23]. This tool supports a Y-chart design methodology based on three abstractions, as shown in Figure 3.24.

The CoFluent application abstraction is more general than those of Artemis, Gabriel, and NP-Click. There is no specific application domain cited as the intended target market. Application models are systems of communicating concurrent processes. Programmers can use a mix of communication mechanisms such as FIFOs, shared variables, and signals to connect the processes. Therefore CoFluent is not as restrictive as the other projects whose application abstractions are based on more formal models of computation. At the same time it does not provide an abstraction for process-level concurrency that is as strong as those found in the other projects.

Compositions of processes and communication links are modeled using a graphical block dia-

gram editor. Individual application processes are control-dataflow graphs (CDFGs) which are also modeled graphically. The library of blocks provided for this purpose includes control structures such as if-then-else loops. The leaves of the CDFGs are C code fragments.

The CoFluent architecture abstraction is similar to that of the Artemis project. An architecture is modeled as a graph with nodes that represent processors, shared memories, signals, and communication resources such as buses, networks, and point-to-point links. Processors are viewed as resources that run the VxWorks real-time operating system. The architectural building blocks are implemented using SystemC.

In a distinct mapping step, application CDFGs are mapped onto processors and implemented as VxWorks processes. When a CDFG uses a communication mechanism such as a FIFO, shared variable, or signal, this action is implemented using VxWorks system calls. When multiple application processes are mapped to the same processor, the operating system is responsible for scheduling the parallel computations. The mapping abstraction allows designers to make mapping assignments using a drag-and-drop interface. Designers can explore different mapping strategies without having to change either the application model or the architecture model. Similarly, the same application can be tested on different architectures.

To evaluate a mapping, CoFluent Studio converts mapped application CDFGs into instrumented C processes that collect event traces when communication resources are accessed. This data is combined with the SystemC simulation of the actual architecture to obtain detailed timing information about the system. Designers can explore a graphical timeline of the activity of each architectural element to determine if the application is making full use of the architecture's capabilities.

CoFluent Studio is an interesting contrast to the behavioral synthesis approaches that predominate other commercial tools. Designers can actually target their own architecture with this tool. It remains to be seen, however, if the communicating CDFG application abstraction is useful and intuitive for any real-world concurrent applications.

Metropolis

The Metropolis project seeks to provide a methodology for implementing a wide variety of heterogeneous concurrent applications on platform architectures. The core of this approach is the Metropolis *metamodel*, which is a single language for describing a system at various levels of detail. This includes abstract application-level models, concrete architectural models, and various

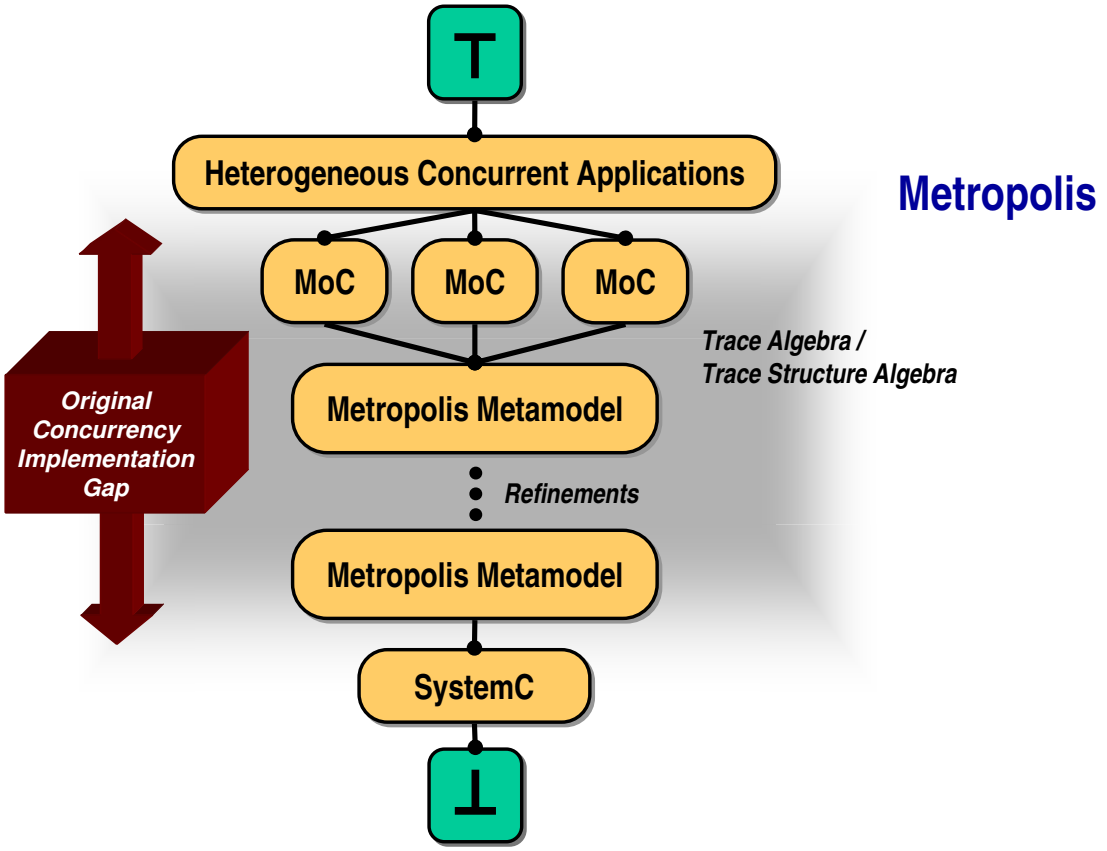


Figure 3.25: Metropolis Design Abstraction Chart

levels in between. The choice to use a single language for different design problems is intended to help ease communication between different design teams. Each team uses the metamodel in a different context, and therefore it appears multiple times in the Metropolis design abstraction chart (Figure 3.25).

The metamodel is primarily an abstraction for describing process-level concurrency. There are two basic objects that appear in a metamodel model: *processes*, and *media* (singular: *medium*). A process models a sequential computation that accesses communication resources. A medium is an object that provides an implementation of communication resources. Processes communicate with each other through media. The implementation of media is completely open-ended. This gives designers a great deal of flexibility in creating different process-level communication semantics.

To specify process-level control semantics, designers use *await* statements within processes. This construct wraps a block of sequential code that accesses one or more media. The await construct declares a guard that is dependent on the states of the given media. The process blocks until the guard evaluates to be true. Then the sequential code within the await construct executes atomically.

It is possible to write a parallel await statement that has multiple branches with different guard conditions and sequential code blocks. When a process encounters such a statement, a single branch that has a guard that evaluates to true is allowed to execute. If more than one guard is true, then the branch to execute is chosen nondeterministically. Await statements and media can be used together to implement a wide variety of styles of process-level concurrency.

As stated, the metamodel is meant to be used to describe applications, architectures and mappings. To model an application, programmers are not meant to use the metamodel by itself because it is too complicated to maintain a consistent and meaningful set of communication and control semantics. Instead, programmers use a library of pre-written processes and media that implement semantics closer to traditional models of computation. An example is the work with the YAPI [28] API presented in [9]. This paper describes a set of processes and media that implement an extended version of Kahn Process Networks. Programmers inherit from these base objects to build their own YAPI applications.

To define and prove properties about heterogeneous compositions of models of computation, Metropolis provides a theoretical framework based on trace algebras and trace structure algebras [16]. Designers are also meant to write abstract performance and cost constraints alongside their application models. A mathematical logic-of-constraints formulation is used to check that these constraints are met throughout the design process [8].

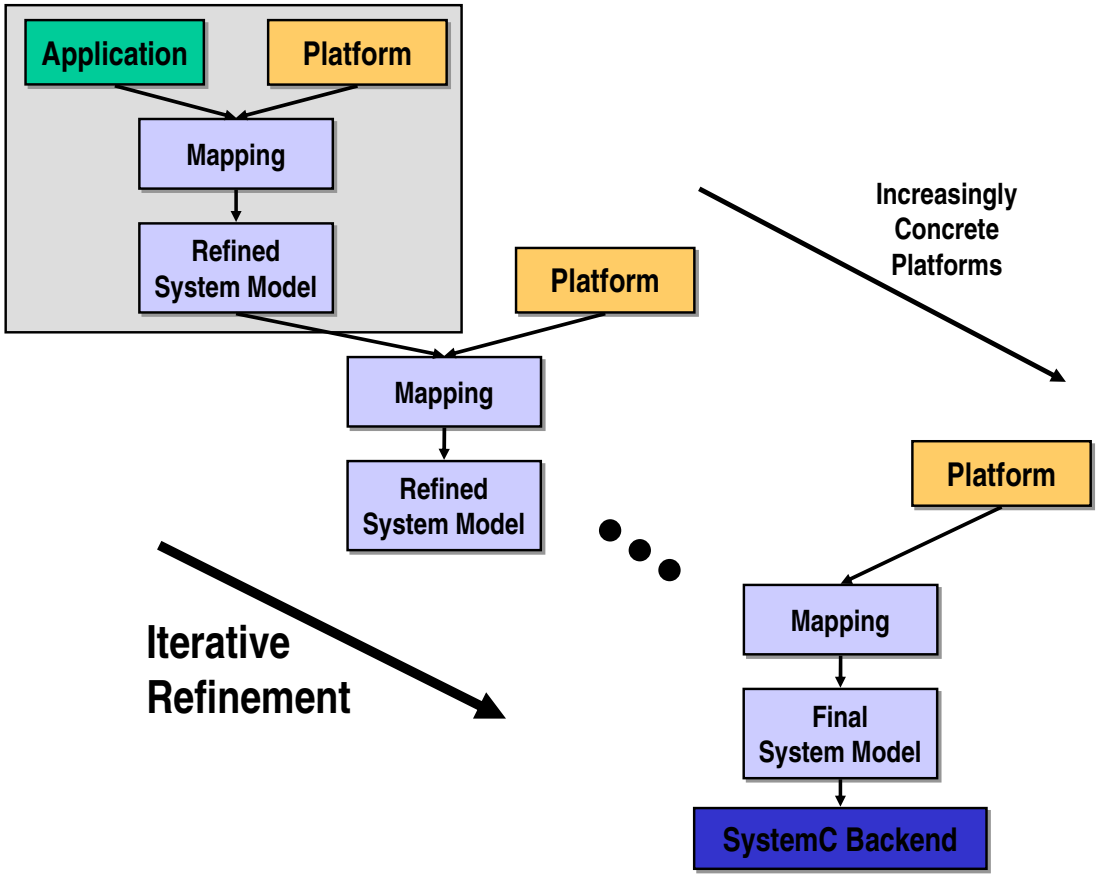


Figure 3.26: Metropolis Design Flow Chart

Architectures are modeled in Metropolis by writing media for coarse architectural components (CPUs, buses, memories, *et cetera*). These media implement *services* used by the processes and media found in the application model. Mappings between application components and architecture components are done using a trace-driven co-simulation approach similar to Artemis. When an application process or media executes an event (such as the beginning or end of a computation or a communication), a call is made to the corresponding architecture media object. The architecture media keeps track of any costs associated with the event (such as time and energy) by updating global *quantity managers*. Quantity managers collect statistics about the execution of the system as a whole.

One way that Metropolis differs from other Y-chart design methodologies is its emphasis on top-down design refinement. In the previously discussed works, a single iteration through the Y-chart (Figure 3.20) produces a functionally correct, fully-implementable system. This system might not meet all of its performance requirements, but it is otherwise complete. Designers use the feed-

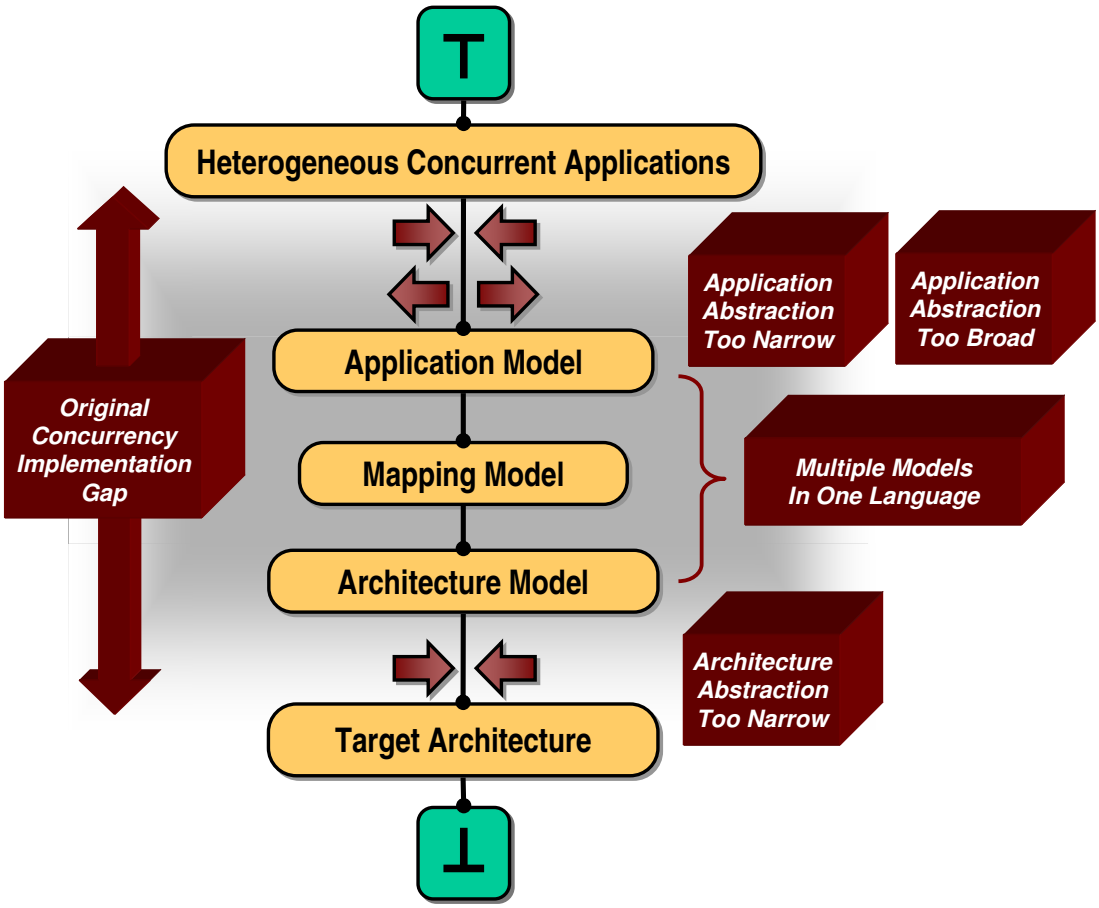


Figure 3.27: Potential Pitfalls of Multiple Abstraction Approaches

back paths to perform design space exploration to meet performance goals. In Metropolis, the output of each iteration through the Y-chart is not necessarily a complete system. Instead, designers build a series of increasingly concrete systems called *platforms*. Intermediate platforms lack some of the implementation details necessary to create a tangible implementation. Designers gradually narrow the design space by filling in these details step-by-step. This variation on the Y-chart is shown in Figure 3.26.

Pitfalls of Meet-in-the-Middle Methodologies

Figure 3.27 summarizes the lessons learned from previous approaches that use multiple abstractions to cross the concurrency implementation gap.

First, it is important to choose an application abstraction that is not too narrow. Methodologies starting from MATLAB, Click, Synchronous Dataflow models, and Kahn Process Networks

have been presented. These abstractions satisfy many of the criteria of the first core requirement: use a formal high-level abstraction to model concurrent applications. A narrow, restrictive model of computation improves designer productivity and makes a precise statement about application requirements. What these methodologies lack is a mechanism to combine different application abstractions for the different facets of heterogeneous applications.

The wrong way to deal with application heterogeneity is to use a less restrictive, lowest-common-denominator abstraction. This is evidenced by methodologies that start with dialects of C or C++. These languages do not help designers model complex concurrency. Therefore the second lesson is not to choose an application abstraction that is too broad.

Third, it is important to use different abstractions for different parts of the design problem. The Metropolis methodology provides one syntax and semantics for applications, architectures, and mappings. Mixing multiple orthogonal concepts into one design language dilutes the power of the language. Designers cannot focus on the different concepts and concerns that are important for the different parts of the design problem. A proliferation of small, focused languages should not be a concern. A lightweight language that excels at describing one important concept is easier to learn and use than one heavy language that is meant to do everything.

Last but not least, it is important to choose a broad range of target architectures. Behavioral synthesis approaches prevent designers from performing architectural design space exploration because they can only target one architecture: the architecture that they create. Approaches that focus on traditional programmable elements such as RISC and DSP cores cannot meet core requirement number two: consider simple architectures. Traditional PEs lack the customizability to match application-specific process-level, data-level, and datatype-level concurrency.

3.3.4 The Cairn Design Methodology

With these lessons in mind, one can now draw a detailed design abstraction chart for the Cairn methodology that will become a road map for the remainder of this dissertation. This is given in Figure 3.28.

Cairn seeks to combine the best features of the related approaches while avoiding the pitfalls. First, there is an architecture abstraction for a broad class of simple and compelling Sub-RISC architectures. This abstraction can be automatically exported from an architecture model. Designers can perform architectural design space exploration without the lengthy tasks of porting a library of computational elements and writing a new compiler. This part of the Cairn methodology is described

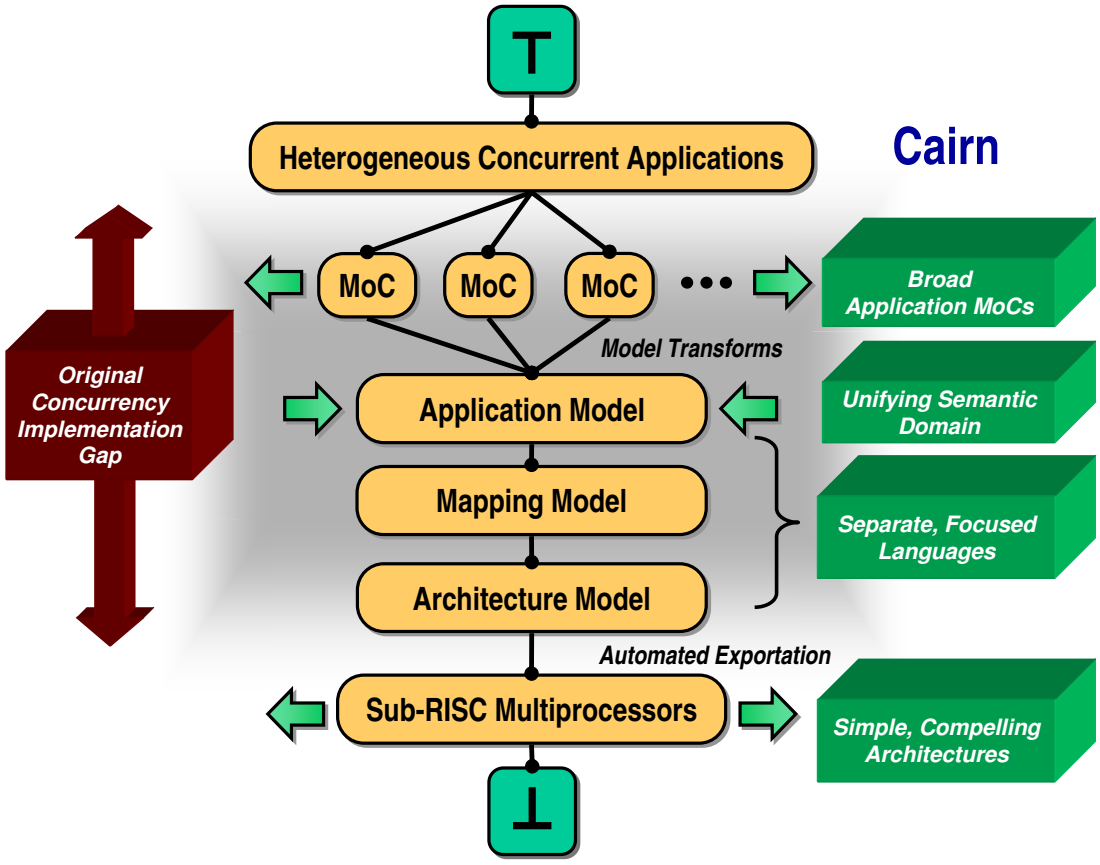


Figure 3.28: Cairn Design Abstraction Chart

in detail in the next chapter.

Heterogeneous concurrent applications are modeled in Cairn using multiple models of computation. Model transforms are used to automatically convert application models into a unifying semantic domain. Models of computation are an intuitive formalism for domain experts, while the underlying application abstraction captures the application's concurrency requirements precisely. The abstractions that programmers use are different from those that architects use because these two groups are focused on different design problems. Application modeling and model transforms are described in Chapter 5.

The Cairn methodology has a distinct step where applications are mapped onto architectures. This step uses a third independent language that is used to match application concurrency and architecture concurrency. Application computations are kept separate from computations that are used to resolve the mismatches between the application and the architecture. Thus, designers can isolate the performance impact of their mapping strategy and explore the mapping design space independently of the application and architecture design spaces. Mapping and the path to implementation are described in Chapter 6.

The following three chapters detail the technical contributions of this dissertation. Afterward, a detailed design example with implementation results will be given in Chapter 7. Chapter 8 summarizes these contributions and offers some concluding remarks.

Chapter 4

The Next Architectural Abstraction

The detailed exposition of the Cairn methodology begins at the bottom of the design abstraction chart (Figure 4.1). At this endpoint, designers build multiprocessor architecture models. An architectural abstraction simplifies this design process and provides a mechanism for characterizing the concurrency capabilities of the architecture.

This chapter considers two important decisions: What kind of programmable basic blocks should be considered for building multiprocessor systems? Then, what abstraction should be used to export the capabilities of a heterogeneous composition of these basic blocks?

The choice of target architectures must be considered carefully. This endpoint must be broad enough to cover novel, compelling architectures that demonstrate superior results compared to other methodologies. This is balanced by a need to limit the scope of the family of architectures in order to maintain simplicity in the architectural abstraction and to bound complexity in the complete design flow. A poor choice of architectures will lead to disappointing results, even if the methodology otherwise follows all of the three core requirements.

The role of the architectural abstraction is to capture the architecture's capabilities for heterogeneous concurrency and export these capabilities upward in the design abstraction chart to programmers. Capturing the capabilities of individual processing elements is only half of the task. The architectural abstraction must also cover the interactions between the processing elements. Understanding the communications and control between the elements is essential to taking advantage of the architecture's process-level concurrency.

In addition to answering these questions, this chapter also covers the procedural aspects of building architectures using the Y-chart design flow. This is shown in Figure 4.2. The techniques that designers use to build and simulate architectural models are explained. The processes that lead

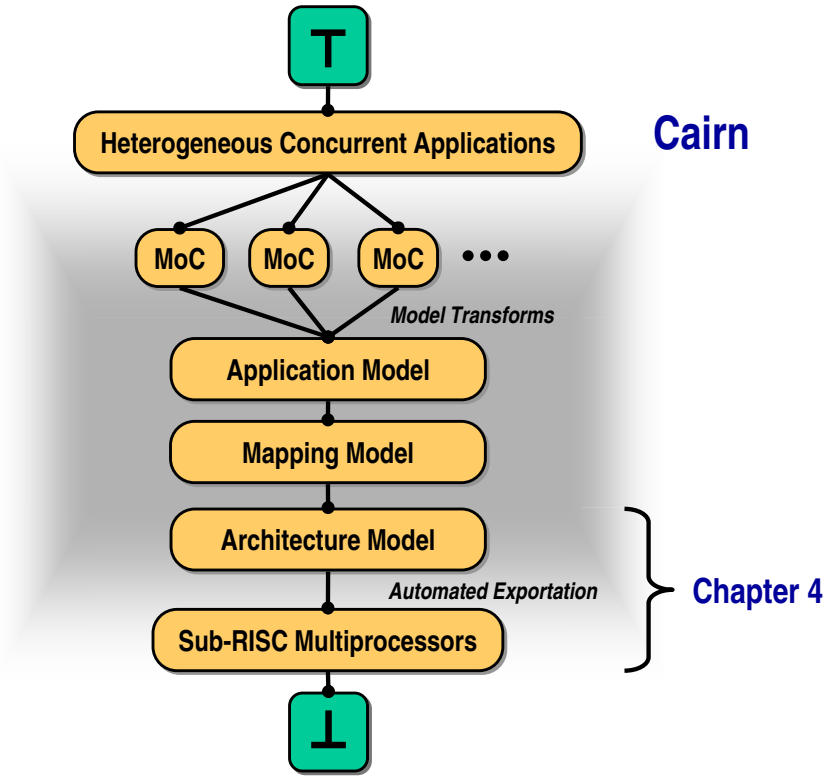


Figure 4.1: Architecture Side of the Cairn Design Abstraction Chart

downward in the design abstraction chart from an architecture model to a realizable implementation are described.

The next section explores the various criteria that a programmable basic block should meet. Then, the potential candidates are organized and compared. Sub-RISC processors are identified as an architectural family with compelling qualities. The final sections describe in detail the methodologies for designing individual Sub-RISC processing elements and for making multiprocessor compositions. An architectural abstraction for heterogeneous Sub-RISC multiprocessors is presented. This abstraction is one of the three major abstractions of the Cairn methodology and is a key contribution of this dissertation.

4.1 Programmable Basic Block Characteristics

The benefits of programmable multiprocessors were previously described in Section 1.2. In order to live up to these promises, the programmable elements that are used to build these systems must meet the following criteria:

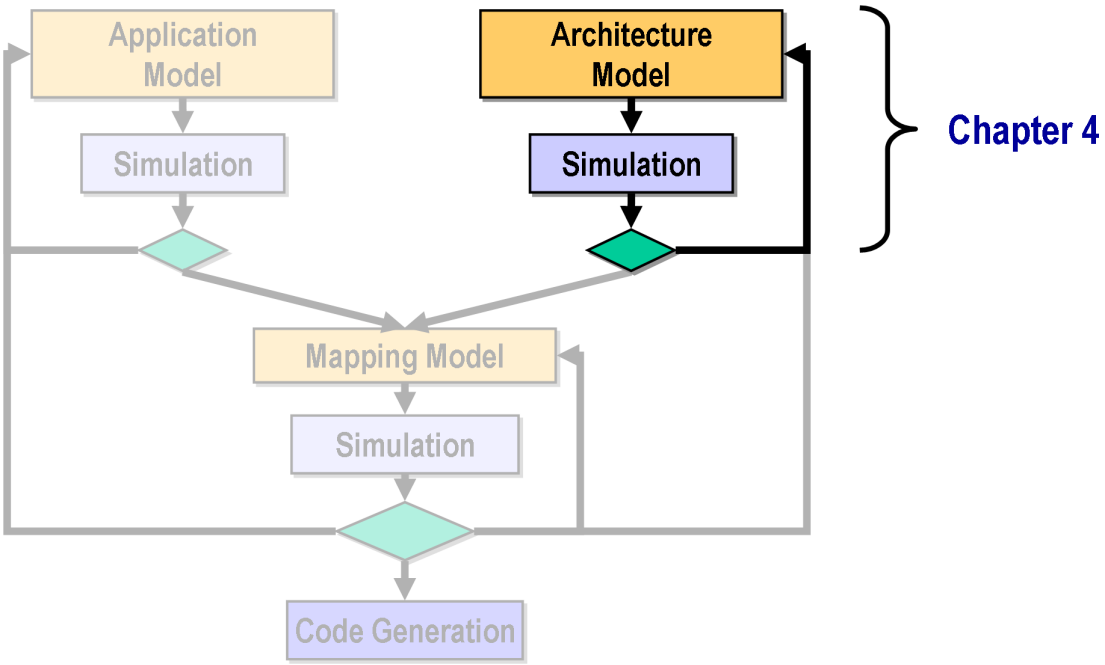


Figure 4.2: Architecture Side of the Y-Chart Design Flow

- *Improving Designer Productivity:* RTL methodologies give architects fine control over the details of a design. This is good for building high-performance systems, but it does not scale. The next architectural abstraction will accept coarser granularity in exchange for a more productive design process.

- *Physical Constraints:* The next basic block must solve the predictability challenges that are troubling RTL ASIC design methodologies. This can be accomplished by providing an abstraction that serves as a boundary between circuit-level design issues and system-level design issues. Physical design issues such as reliability, timing and power are constrained to the scope of individual blocks. System-level compositions of these blocks can be assembled with an abstraction that hides these details. It must be possible to characterize the performance of a basic block at a high level, so that the block's behavior as a component in a system can be easily understood.

- *Customizability:* The basic block should be customizable to meet the needs of a particular application or application domain. Therefore it should not be a static entity, but rather a template that defines a family of possible blocks. Features that support the application's process-level, data-level, and datatype-level concurrency are desirable.

Datatype-level customizations include support for performing computations on application-specific data types. An example from the signal processing domain is fixed-point numbers. A custom data type need not be a scalar value. In the packet processing domain, an entire packet header can be considered a data type.

Data-level customizations give an architecture the power to work on multiple pieces of data simultaneously. This is accomplished with vector-type operations, multiple register files, and multiple memory interfaces.

Process-level customizations lead to architectures that can support multiple parallel tasks, such as simultaneous multi-threading (SMT) machines.

- *Degrees of Reusability:* In some cases, architects are willing to make a design application-specific in exchange for higher performance. In other cases, the ability to work across a broad application domain is more important than raw performance. The next basic block should permit architects to find the sweet spot along this spectrum by exploring trade-offs between application-specific features and general-purpose features.

- *Explorability:* Highly customizable basic blocks have large design spaces. One danger is that the basic block may be so configurable as to overwhelm designers with implementation choices. A good basic block will present designers with only those options that lead to compelling performance results, while hiding architectural minutiae.

  Furthermore, it must be possible to experiment with these design choices easily. A design change to one element in a multiprocessor must not instigate a chain reaction that extends throughout the design methodology. A seemingly harmless change may require changes to neighboring elements, modifications to the mapping, and possibly even changes to the application software. This can make experimentation very expensive. By lowering the cost of performing experimental iterations, designers will be able to perform more iterations and find better system solutions.

There are additional criteria directly inspired by the three core requirements:

- *Architectural Simplicity:* A programmable datapath basic block must be simpler than today's common embedded microprocessors such as the XScale or the ARM. General-purpose architectures have reached a critical mass of complexity where further incremental improvements in performance come at a huge hardware cost. Agarwal et al. report that the techniques used to

enable the rapid scaling of microprocessor performance in the past, namely increasing clock rate and instructions per cycle (IPC), are not capable of sustained performance growth in the future [1]. A different approach to microarchitecture design is required. This dissertation argues that architectural features that match the concurrency requirements of applications are an answer to this challenge.

A design methodology based on iterative design space exploration implies that the programmable basic blocks will be redesigned several times over the course of a project. For this to be feasible, the complexity of the basic blocks should be limited to small, easily designed and verified components.

Power has also become a dominant design constraint for future desktop systems [13, 14, 87]. Embedded systems are even more power-constrained than desktop systems. This darkens the outlook on using traditional processors as basic blocks in embedded systems.

Small, simple processors are also favorable from a manufacturability standpoint. A complex processor that occupies a large amount of area can be ruined by a single manufacturing defect. Small processors make redundancy inexpensive. If a handful of processing elements in a multiprocessor are defective, the remaining elements can still be used and the chip can be sold in a lower performance class.

These arguments favor a minimalist approach. Complexity should only be introduced and paid for when the application requires it. If the basic blocks are complicated, then even simple problems will lead to complex solutions. For a customizable basic block, this mandates the ability to omit unnecessary features as well as the ability to add application-specific features.

- *Composability:* Basic blocks must be able to interface cleanly with other basic blocks. Otherwise, they will not be useful as a component for building composite systems.

  Creating a composition of basic blocks builds the process-level concurrency of the system. Here, the important concerns are how the basic blocks communicate data and execute relative to each other. An ideal basic block will separate the concerns of communication and control between blocks from the computation that occurs within a block. This makes it possible to reuse the same computational elements in different multiprocessor contexts.

  To accomplish this, and to satisfy the architectural simplicity core requirement, a basic block should export an interface with more primitive communication and control capabilities than those found in traditional processors. Common architectural design patterns such as symmet-

ric multiprocessing with cache coherency are expensive and provide a general-purpose rather than an application-specific process-level concurrency scheme. Blocks with primitive interfaces can be composed to build more complex interprocessor communication schemes when and if the application demands it.

Two additional aspects of composability are higher-order modeling and data polymorphism. Higher-order modeling refers to the ability of a block to parameterize its behavior based on its contextual usage. Consider a processor that can be connected to an arbitrary number of on-chip network links. Based on the connectivity, the processor can customize itself at design time to produce the correct datapath hardware and instructions for accessing all of the communication resources. This capability promotes reuse and improves productivity.

Data polymorphism refers to the ability of a block to be datatype-agnostic. This borrows from the concept of generics in programming languages. Consider a digital signal processor that can work on arbitrary fixed-point data types. This block could be composed into different systems that compute at different accuracies without requiring internal changes to be made by hand.

- *Exportability:* Architects must be able to communicate the capabilities of the basic block to programmers so that they can deploy applications on it. Developers require an architectural abstraction that finds the right balance between visibility into architectural features that are necessary for performance, and opacity toward less-important details. At the multiprocessor level, the architectural abstraction must go one step further and characterize the architecture's process-level concurrency. This includes the ways that the basic blocks communicate and execute relative to each other.

  A clean abstraction of architectural capabilities makes a design flow resilient to architectural design space exploration. It also sets the foundation for a disciplined mapping methodology where application requirements are compared to architectural capabilities.

  Ideally, an automated process should be used to extract the architectural abstraction from the architecture model. Otherwise there will be separate, manually maintained models of these two things. Maintaining consistency between these models is an enormous verification burden on the designer. This is a pitfall that must be avoided.

  A second pitfall to avoid is an exportation process that leads from heterogeneous blocks to an unruly zoo of distinct architectural abstractions. This is a lesson learned from the failures

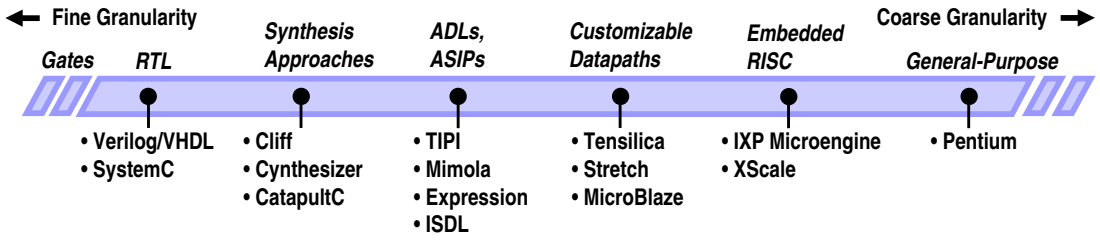


Figure 4.3: Spectrum of Basic Block Candidates

of the system-in-the-large approach (Section 3.2.2). Even though the blocks in a heterogeneous multiprocessor are different, it is desirable to have a common formalism for comparing and contrasting their capabilities. Thus, the exportation process should be applicable to all members of the architectural family.

4.2 Programmable Basic Block Candidates

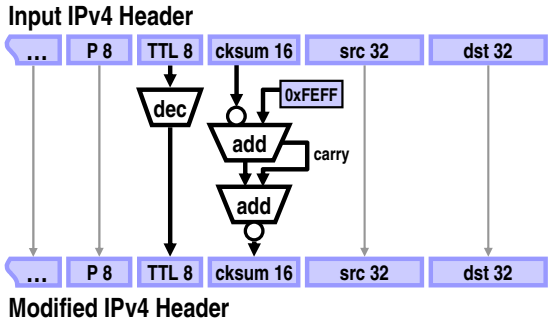
RISC processors are familiar to both hardware designers and software developers, so they appear to be an obvious choice. However, they are only one of many different kinds of programmable elements. Figure 4.3 arranges the important candidates along a spectrum according to level of granularity. Existing RTL design methodologies are included as a point of reference on the left. The interesting basic blocks are those with a coarser granularity than state machines.

At the opposite end of the spectrum, there are large general-purpose processors such as the Pentium. This type of processing element has been used in multiprocessor compositions such as the Beowulf system [101]. A basic block such as this is very expensive in terms of area and power, and it does not include any application-specific features that benefit any particular application domain.

The ideal basic block will lie somewhere in between these two extremes. In this section, each candidate is explored in detail and the merits and shortcomings of each are discussed relative to the criteria given above.

4.2.1 RISC PEs

The trouble with using a RISC processor as a basic block can be summarized in one seemingly contradictory statement: They are simultaneously too simple and too complex for concurrent embedded applications. The “too simple” argument follows from the customizability criterion given previously. A traditional RISC processor such as a Xilinx MicroBlaze is a general-purpose entity. It provides features that work for all applications but are exceptional for no application. There are



(a) Custom Logic

```
⋮
mod.TTL = input.TTL – 1;
uint32 tmp = ~input.cksum & 0xFFFF;
tmp += 0xFEFF;
if ((tmp >> 16) != 0) tmp++;
mod.cksum = ~tmp & 0xFFFF;
⋮
```

(b) Pseudocode for a RISC PE

Figure 4.4: Implementing Datatype-Level Concurrency

few, if any, opportunities to customize the architecture or to explore trade-offs between application-specific features and general-purpose features.

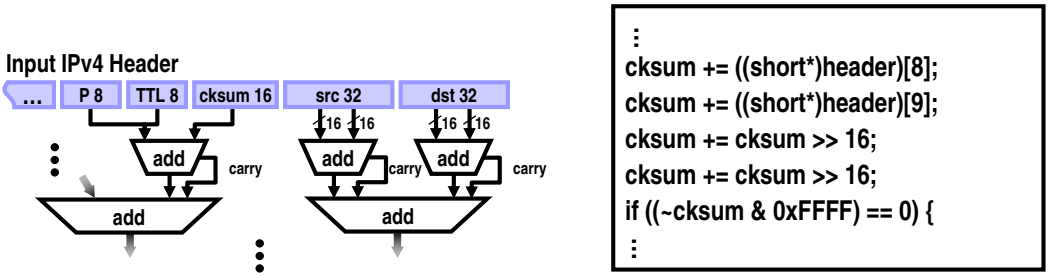
The "too complex" argument begins at the architectural simplicity criterion. A PE that provides a comprehensive set of general-purpose features will inevitably include features that will never be used by certain applications. Therefore, RISC PEs will sometimes be unnecessarily complicated for the problem at hand.

These arguments apply to each of the three levels of concurrency described in Section 1.3.2: process-level, data-level and datatype-level. As an example, consider the difference between deploying a network processing application on a RISC PE versus using an approach like Cliff [73]. Cliff is a structural synthesis approach that builds a custom FPGA design starting from a Click application model. As such, it has the opportunity to build a circuit that exactly matches the computations described in the application. A RISC PE, on the other hand, must make do with the available instruction set. This comparison will reveal that there is a large design space gap between what is possible with an application-specific datapath versus what is necessary with a RISC PE.

Datatype-Level Concurrency

Recall that datatype-level concurrency refers to the way that data is represented as a bit vector and to the arithmetic and logical computations that are carried out on that data. In a Click application, the fundamental data type is a packet header. Click *elements* describe bit-level computations on these headers.

The *DecIPTTL* element will serve as an example. This element decrements the IP time-to-live header field (an 8-bit value) and incrementally adjusts the checksum field to match this change using



(a) Custom Logic (b) Pseudocode for a RISC PE

Figure 4.5: Implementing Data-Level Concurrency

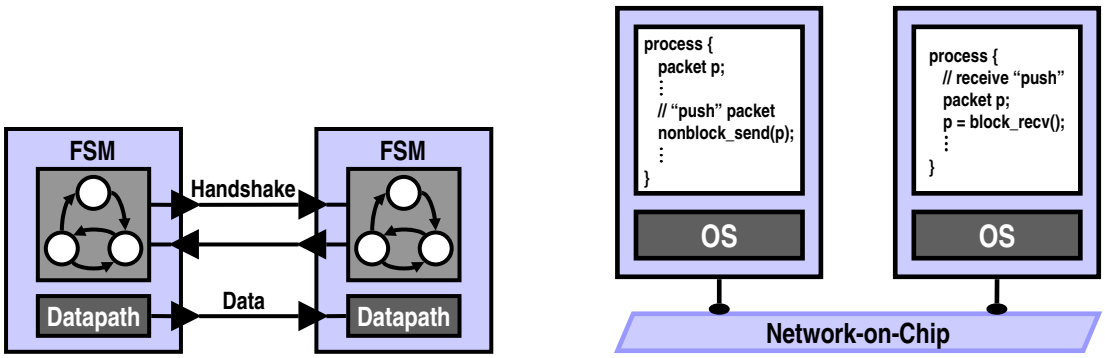
1's complement arithmetic. This element is a fundamental part of IP forwarding applications and is executed on every packet that goes through a router.

Cliff can produce an FPGA design that exactly matches the data types and the 1's complement arithmetic expressed in the *DecIPTTL* element. This can be viewed as a miniature datapath as shown in Figure 4.4(a). A RISC PE, on the other hand, lacks instructions for 1's complement arithmetic. This datatype-level concurrency must be emulated using logical shifts, bit masks, and 2's complement arithmetic as shown in Figure 4.4(b). The RISC PE is sufficient for implementing the *DecIPTTL* computations, but it is neither the cheapest nor the fastest way to implement them.

Data-Level Concurrency

The *CheckIPHeader* element demonstrates a mismatch in data-level concurrency. This element treats an IPv4 header as an array of ten 16-bit fields and performs a 16-bit 1's complement checksum operation. In the Cliff design shown in Figure 4.5(a), the datapath is wide enough for the entire header. All ten fields can be immediately passed into an adder tree.

A typical RISC PE will not have enough parallel adders or enough memory ports to process 160 bits of data at once. The pseudocode shown in Figure 4.5(b) specifies the checksum calculation sequentially. If the PE is a multiple issue machine, it is up to the compiler to try to reverse-engineer the data-level concurrency from the sequential code. A PE with dynamic data-level concurrency (e.g. a superscalar machine) is an even more expensive solution: It has extra hardware to calculate dynamically what is known a priori.



(a) Custom Logic (b) RISC PEs

Figure 4.6: Implementing Process-Level Concurrency

Process-Level Concurrency

Click's *push* and *pull* connections are an abstraction for a control-follows-data style of process-level concurrency. In push communications, the upstream element performs computation on a packet header and then passes the flow of control to the downstream element. This is different from a function call because the upstream element does not wait for a return value and it does not continue. In pull communications, the downstream element initiates the transfer of control. Control is passed to the upstream element, and the downstream element blocks until the upstream element provides a packet header.

Cliff can build control structures at the multiprocessor level that mirror these semantics exactly. Each Click element in the application is implemented as a miniature datapath as described above. These datapaths are controlled by finite state machines that communicate using a three-way handshake. This is shown in Figure 4.6(a). Push and pull communications events are implemented using these inter-element control signals. This represents a tight coupling between the control logic of elements at the multiprocessor level.

A RISC PE has a more self-sufficient model of control where the normal behavior is to forge ahead though a sequential program. Aside from interrupts, there is no mechanism for multiple PEs to directly influence each other's flow of control. A possible RISC implementation is shown in Figure 4.6(b). Here, there are two RISC PEs that are connected to an on-chip network. The Click computations are implemented as processes, and an operating system provides access to the communication hardware resources through a message-passing interface. The processes use blocking and non-blocking send and receive functions to emulate Click's push and pull semantics in soft-

ware. Since the PEs are loosely-coupled in their control logic, it is more difficult and expensive to implement certain styles of process-level concurrency.

There is quite a large design gap between RISC PEs and what is possible with application-specific datapaths. In each of the three granularities of concurrency, this implementation gap has a noticeable effect on performance. Tensilica reports that there is significant performance to be gained by adding support for the application's datatype-level concurrency to a RISC PE's instruction set [62]. To work around a mismatch in data-level concurrency, Sourdis et al. [113] outfit PEs with header field extraction engines to efficiently move data into register files. For process-level concurrency, SMP-Click finds that the synchronization and communication between PEs is a major performance bottleneck [21]. The mismatch between the application and the architecture is so strong in this case that the authors resort to a mapping strategy that minimizes inter-PE communication.

For these reasons, it is maintained that a finer-grained programmable component is a better choice than RISC PEs. More potentials for application-specific concurrency, especially process-level concurrency, are desirable. Whenever possible, one also wishes to remove general-purpose features that are not used by the target applications. This will lead to higher performance at lower cost.

4.2.2 Customizable Datapaths

The next finer-grained PE candidate is a configurable datapath such as the Tensilica Xtensa or the Stretch architecture. These PEs permit architects to add custom instructions to a RISC core to implement application-specific arithmetic and logic computations. This provides opportunities to match datatype-level concurrency. However, the RISC core itself is mostly fixed and it is not possible to trim away certain unwanted general-purpose functions.

These PEs are also inadequate in terms of process-level and data-level concurrency. Traditional sequential control logic consisting of a program counter and an instruction memory is assumed. The number of register files, the number of register file read ports, and the number of memory ports are only customizable to a limited extent.

4.2.3 Synthesis Approaches

The main difficulty with synthesis approaches such as Cliff is that they do not create a programmable system. If designers need to make changes to the Cliff application model in the future, the physical hardware must also change. FPGAs allow for this kind of reconfiguration, but at a high

cost in terms of area, power, and achievable clock frequency. A system that can be updated to match the new application by only changing the software will be less expensive. Synthesis approaches overshoot the goal in terms of customization, sacrificing reusability.

A second problem is that synthesis provides limited opportunities for design space exploration. In order to go from a high-level input language to an RTL implementation automatically, the tool has to make a large number of design decisions. Most of these are not visible or controllable by the architect. If the tool's design cannot meet performance goals, architects are left with undesirable choices such as editing the generated RTL by hand.

A tool like Cliff is in a better starting position for making good architectural choices because it knows a great deal about the application domain. Commercial tools such as Forte Cynthesizer and Mentor Catapult C start with general-purpose input languages and do not have this application knowledge. It is unrealistic to think that these tools will always make the right design decisions for any particular application. The lack of an effective path for exploring improvements degrades designer productivity.

4.2.4 Architecture Description Languages

The processing elements produced by Architecture Description Language (ADL) methodologies are good candidates for basic blocks. They give architects more programmability than synthesis approaches and provide more opportunities for customization than RISC PEs. In a nutshell, an ADL PE is a datapath whose capabilities can be exported as an instruction set. The instruction set defines the capabilities of the PE that can be utilized by a software program.

Like the customizable datapaths discussed earlier, ADLs permit architects to build custom instructions for application-specific arithmetic and logic operations. However, the customizability does not end at the level of making extensions to a core architecture. Architects can design the entire architecture and can therefore decide what features to leave out as well as what to include. One can design a simple architecture with a few instructions, or a complex architecture with hundreds of instructions. The instructions can be tuned for a particular application, or for a domain of applications, or they can be general-purpose. Traditional architectural styles, such as five-stage pipelines with a single register file and memory port, can be broken in favor of schemes that support application-specific data-level concurrency. This is attractive because it leads to PEs that are both "simpler" and "more complex" than RISC PEs. Thus, ADL PEs satisfy the customizability, reusability, and simplicity criteria given at the beginning of the chapter.

ADLs also greatly improve designer productivity. In order to design a programmable element using an RTL language, architects must specify both the datapath and the control logic manually. The hardware implementation must be kept consistent with the description of the instruction set and the software development tools. This includes instruction encodings, instruction semantics, integration with an assembler, a simulator, and many other things. Unfortunately, RTL languages do not have an abstraction for describing these details. In practice, instruction sets are usually English language specifications. The problem of maintaining consistency is a verification nightmare. This impedes the ability to explore architectural options.

ADLs solve this problem by using a three-part approach. First, they provide formalisms for specifying instruction sets. This counters the lack of precision and the analysis difficulties of English language specifications.

Second, they enforce particular architectural design patterns. Architects can no longer freely make arbitrary collections of finite state machines. Instead, design is performed at the level of components such as register files, multiplexers, and function units. The focus is on designing the instruction-level behavior of the PE.

Third, there is a mechanism for ensuring consistency between the hardware structure and the instruction set. ADLs can be roughly grouped into two categories based on how they attack this last problem.

Generating the Architecture from the Instruction Set

In ADLs such as ISDL [47] and nML [37], designers model an instruction set and then a correct-by-construction synthesis tool generates a matching architecture. The appropriate software development tools are also created. This guarantees consistency. However, this approach has drawbacks similar to the synthesis approaches described earlier. The design tools make decisions about the implementation of the architecture that are not under the control of the architect.

To remedy this, ADLs such as LISA [53] and EXPRESSION [48] were developed. Here, designers create a single unified description of the ISA, the datapath structure, and a mapping between the two. This gives architects more control over the details of the implementation. However, designers must specify an instruction set and an instruction encoding by hand.

A downside that affects all ADLs in this category is that the architect is expected to conform to a traditional architectural style. Like the customizable datapaths described previously, there is assumed to be a sequential control scheme with a program counter and an instruction memory. More

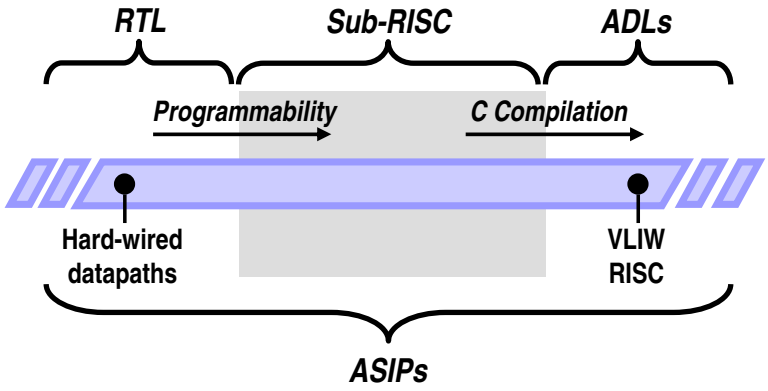


Figure 4.7: Spectrum of ASIP Basic Blocks

flexibility is required in this area in order to construct application-specific process-level concurrency schemes.

Extracting the Instruction Set from the Architecture

The opposite approach is to have the designer build the architecture, and then automatically extract the instruction set and the software development tools. MIMOLA is an example of an ADL in this category [78]. The theory is that for unusual architectures with complex forwarding paths, multiple memories, multiple register files, and specialized function units, it can be more difficult to describe the instruction set than it is to just build a structural model of the architecture.

This method is clearly advantageous for architectures that question traditional RISC design patterns. However, there are still issues with application-specific process-level concurrency. MIMOLA requires a traditional control scheme in order to produce a compiler. Designers must manually identify key architectural features such as the program counter and the instruction memory. Thus the deployment methodology for these PEs is only applicable to a subset of the architectures that can be designed.

4.2.5 TIPI: Tiny Instruction Set Processors and Interconnect

To build PEs with application-specific control logic, one must consider an area of the design space that is not adequately covered by existing ADLs. This space is shown in Figure 4.7. Many of the basic blocks considered in this chapter, from the hard-wired datapaths created by synthesis approaches to VLIW- or RISC-like customizable datapaths, can be called Application-Specific Instruction-set Processors (ASIPs). For some of the finest-grained hard-wired designs on the left

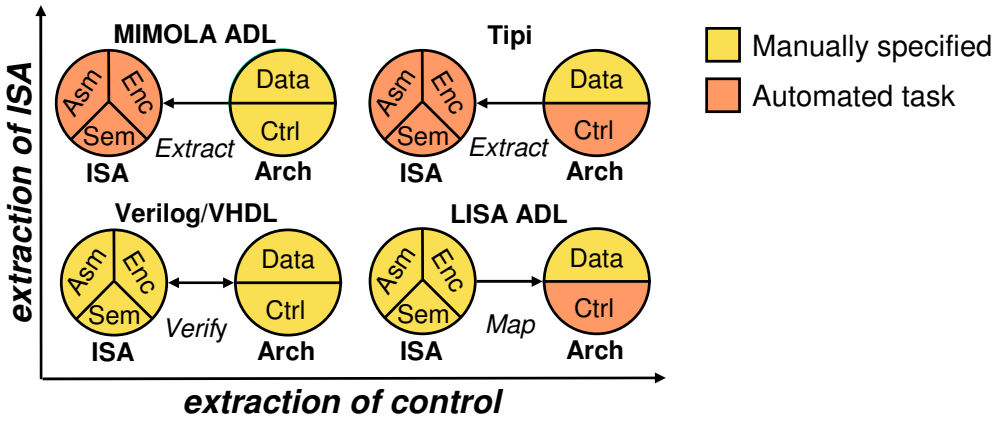


Figure 4.8: ADL Design Approaches

in the figure, RTL languages offer sufficient designer productivity. However, as one starts to add programmability to the architectures, designers quickly become overwhelmed in the specification of details of the control logic for the architecture. RTL languages are no longer sufficient.

ADLs make it easier to describe programmable machines, but their focus is on architectures that are amenable to traditional compilation techniques (e.g. sequential C code). It is clear that C is not a good language for modeling most of the concurrency found in embedded applications. Therefore it is perfectly natural to consider architectures that are different from traditional machines that run C programs.

When one considers application-specific process-level concurrency, and multiprocessor machines with tightly-coupled control logic between PEs, it is clear that there is a gap in the spectrum between RTL languages and ADLs. This intermediate area is called the Sub-RISC design space. Sub-RISC architectures are programmable architectures that focus on concurrency instead of sequential C programs. These are architectures that are too complex to design with RTL languages in terms of productivity, but are also not targeted by existing ADLs.

The TIPI system targets exactly this design space. TIPI stands for “Tiny Instruction-set Processors and Interconnect” [119]. This approach builds upon the ADLs that extract the instruction set from the architecture.

Figure 4.8 compares TIPI to the other categories of ADLs. The traditional RTL approach is included as a point of reference. With languages such as Verilog and VHDL, designers must specify the ISA, the datapath and the control logic of a PE manually. Verifying consistency between the architecture and the ISA is a challenging manual task.

Moving along the *x* axis, the LISA approach adds automatic extraction of the control logic.

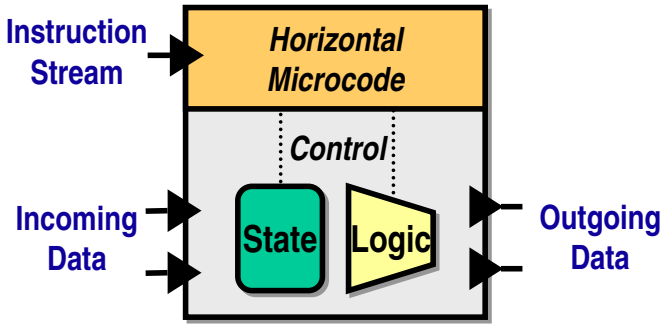


Figure 4.9: TIPI Control Abstraction

Designers only specify the datapath structure and the ISA. The *y* axis is for automatic extraction of the ISA. In MIMOLA, architects design both the datapath and the control logic. The ISA is generated by the tools.

TIPI combines automatic ISA extraction and automatic extraction of control. Designers create only the datapath and the tools produce the corresponding control logic and ISA.

TIPI uses the concept of a *horizontally microcoded, statically scheduled* machine as the core architectural abstraction. Designers build structural models of datapaths, but leave control signals (such as register file address inputs and multiplexer select signals) disconnected. These signals are automatically driven by a horizontal microcode unit that decodes an instruction stream provided by a source external to the PE. This is shown in Figure 4.9. This abstraction applies to the whole range of architectures that can be designed.

TIPI programmable basic blocks meet all of the goals given at the beginning of this chapter:

- *Improved Designer Productivity:* Instead of making compositions of state machines with RTL languages, architects work at the level of structural datapaths composed of register files, memories, multiplexers and function units. Complex control logic may be omitted. An automatic ISA extraction process discovers the capabilities of the datapath and produces the necessary control logic, eliminating the need for consistency verification.

- *Predictability:* An automatically extracted instruction set captures the behaviors of a PE at the cycle level. This allows the system-level performance of the PE to be characterized in terms of the number of cycles it takes to perform a given computation. Circuit-level issues, such as power and timing, are encapsulated within the design of the PE.

- *Customizability:* TIPI PEs can be customized to support all three granularities of concurrency. Architects can add function units for application-specific arithmetic and logic computations.

Beyond this, architects can create unusual pipelines with varying numbers of register files and memory ports for better data-level concurrency. Most importantly, TIPI PEs permit the construction of multiprocessor systems that support application-specific process-level concurrency. The simplified control abstraction shown in Figure 4.9 is a building block for making multiprocessors with tightly-coupled control logic.

- *Degrees of Reusability:* Architects can explore a range of architectural alternatives, from datapaths that provide a large number of general-purpose features to datapaths that focus on application-specific performance.

- *Explorability:* Designers can explore the architectural design space by making changes to the structural datapath model. The ISA extraction process automatically determines what new instructions have been added, what old instructions have been removed, and what instructions are unchanged. It is not necessary to manually verify consistency between the architecture and the ISA.

  A correct-by-construction code generation approach produces synthesizable RTL implementations and simulators. Architects can therefore quickly measure the quantitative performance impact of their changes. This analysis leads to ideas on what changes can be made to improve performance. Since these experiments can be done quickly, a larger design space can be evaluated and the end result will be better than that of methodologies with slow exploration techniques.

- *Architectural Simplicity:* Deciding what to leave out of an architecture is just as important as deciding what to add. A minimal TIPI PE can have as few as two or three instructions. Traditional control elements, such as program counters and instruction memories, are not assumed to exist. Architects only pay for complexity when the application demands complexity.

- *Composability:* The abstraction of a PE that receives instructions and data from an external source is the basis for making heterogeneous compositions of PEs. This abstraction is independent of the internal workings of the PEs, and is therefore applicable to all of the different architectures that can be described.

- *Exportability:* The automatically-extracted instruction set describes the possible cycle-by-cycle behaviors of the design. This exposes the PE's capabilities for data-level and datatype-level concurrency.

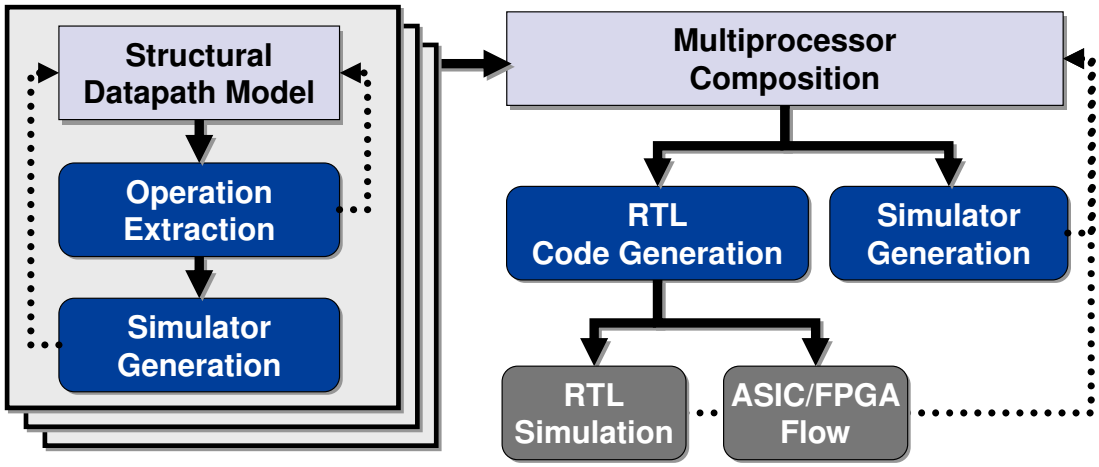


Figure 4.10: TIPI Design Flow

4.3 Designing Sub-RISC Processing Elements

This section covers the design of individual TIPI processing elements. This is shown in the left half of Figure 4.10. In the following section, individual PEs will be combined into multiprocessor systems as shown in the right half of the figure.

The TIPI architecture design flow is an iterative methodology. Designers begin by building a structural model of a PE datapath. This process is detailed in the following section. The capabilities of this datapath are characterized by running an automatic operation extraction algorithm (Section 4.3.2). At this point, architects can iterate over the design and improve the architecture until it implements the desired functions, as shown by the dotted line feedback path. TIPI also produces a cycle-accurate, bit-true simulator for the PE, so timing simulation can be included in the design loop (Section 4.3.4).

4.3.1 Building Datapath Models

TIPI PEs are designed by creating structural datapath models. Architects assemble *atoms* from a library that contains common things such as register files, multiplexers, memories, pipeline registers, and arithmetic and logical function units.

Figure 4.11 is an example of a simple architecture. This datapath has two register files. One is connected to a decrement function unit and the other is connected to an equality comparison unit and a checksum update unit.

Only a few extra details must be filled in by the designer to turn this simple example into a

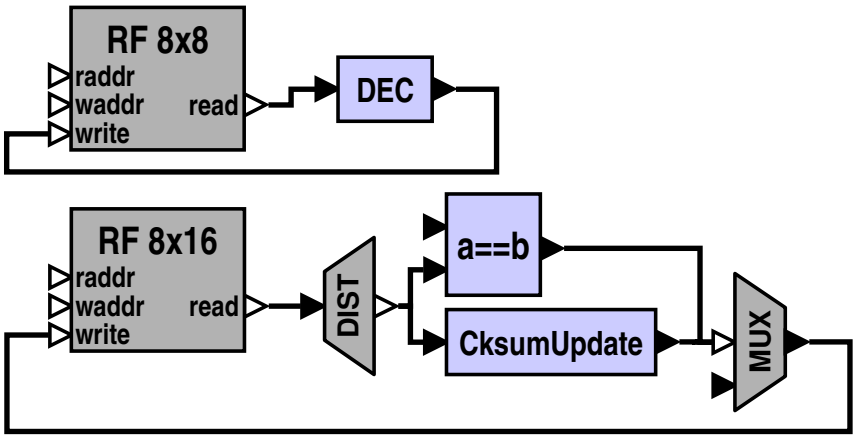


Figure 4.11: Example TIPI Architecture Model

complete datapath model. The width and depth of the register files must be specified by setting parameters on those atoms. In this case, the first register file is 8 bits wide and has 8 entries, while the second register file is 16 bits wide and has 8 entries.

Most TIPI atoms are type polymorphic in terms of bit width, which simplifies the design process. For this datapath, the width and depth of the register files is sufficient information for a type resolution algorithm to determine the bit widths of all of the wires in the design. Type conflicts are a possibility. These are considered syntax errors and must be corrected manually.

TIPI atoms can also be context-sensitive higher-order components. For example, the multiplexer in this design has an input port that can accept a variable number of connections. This is called a *multiport*. The atom is an N-way mux, where N is determined by how the multiplexer is used in the datapath model. The register files also feature multiports. Based on the connections, one can specify register files with N write ports and M read ports. In this example each register file has only one read port and one write port.

The most important thing about this datapath model is that it is acceptable to leave many of the ports in the design unconnected. The read and write address ports on both register files, one input to the comparison function unit, and the select signal to the multiplexer are all unspecified in this design.

This is a major concept that distinguishes TIPI from other ADLs. A TIPI datapath is a nondeterministic description of computations that are atomic with respect to architectural state and outputs. The TIPI control abstraction states that the unconnected control signals are implicitly connected to a horizontal microcode control unit. Their values will be controlled by statically scheduled software on a cycle-by-cycle basis. The register file read and write addresses and multiplexer se-

lect signals become part of the instruction word. Unspecified data inputs, such as the input to the comparison function unit, become immediate fields in the instruction word.

Also missing are traditional control logic elements such as a program counter and instruction memory. Instead, this design receives instruction words from an external source as shown in Figure 4.9.

Obviously, there are restrictions on what combinations of control bits represent valid and useful settings for the datapath. The next step in the design process is to analyze the datapath and determine what these combinations are.

4.3.2 Operation Extraction

TIPI provides an automated process that analyzes a datapath design and finds the set of valid execution paths that can be configured statically by software. The resulting *operations* are a formal way of representing the programmability of the datapath.

An operation is a generalization of a RISC instruction. Operation extraction is analogous to ISA extraction in other ADLs. Instead of describing a register-to-register behavior, an operation describes a single-cycle state-to-state behavior. This exposes a pipeline-level view of the datapath to the software. A fine-grained abstraction of the architecture such as this is consistent with two goals: First, to build PEs that support application-specific process-level, data-level and datatype-level concurrency, and second to export these capabilities to the programmer.

Operation extraction works by converting a structural datapath model into a system of constraints, and then finding satisfying solutions to the constraint problem. Within a cycle, each wire in the design can have a data value be *present* or *not present*. Each atom has *firing rule constraints* that specify valid combinations of present and not present signals on its input and output ports. A valid datapath configuration occurs when every atom's constraints are simultaneously satisfied.

Examples of atom constraints are given in Figure 4.12. Figure 4.12(a) is the definition of a simple addition function unit atom. The beginning of this definition declares input and output ports. The atom has one explicit firing rule (called "fire") that performs the addition action.

Firing rules are written in a feed-forward dataflow language based on Verilog. This allows architects to specify arbitrary datatype-level arithmetic and logic computations. From this expression, TIPI infers the required presence or non-presence of values on the atom's ports. In this case, the "fire" rule requires values to be present on both *inputA* and *inputB*. This then implies that a value will be present on *out*.

```
atom Add(out, inputA, inputB) {
   output out;
   input inputA;
   input inputB;
   rule fire() { out = inputA + inputB; }
}
```

(a) Add Atom

```
atom Mux(out, din, select) {
   output out;
   input select;
   input <out.type> @din;
   foreach (i) {
      rule fire(select) {out = din[$i];}
   }
}
```

(b) Multiplexer Atom

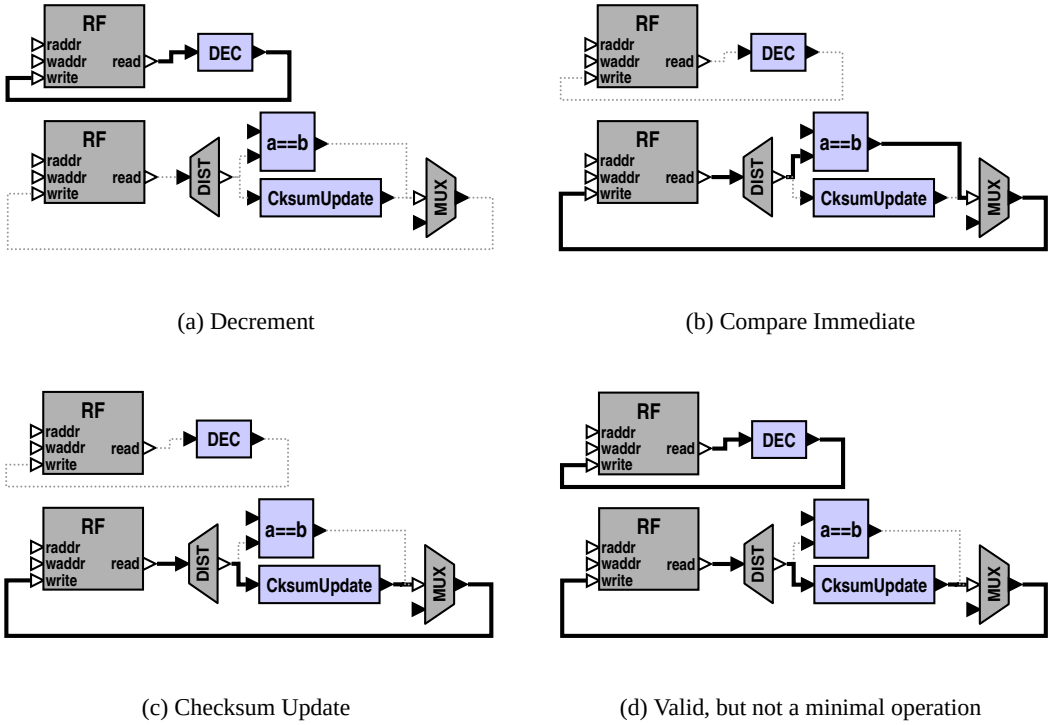
Figure 4.12: TIPI Atom Definitions

Atoms also have an implicit no-fire rule that states that if no values are present on any input port, then no values will be produced on any output port. The addition atom therefore has two possible behaviors within a cycle: it can perform an add or it can remain idle. It is not allowed for a value to be present on only one of the two input ports, or for a value to be present on the output port but not the input ports. These situations represent invalid datapath configurations that the operation extraction algorithm will discard.

A more complex atom is shown in Figure 4.12(b). The multiplexer is a higher-order component whose firing rules are generated based on its context within the datapath model. The *foreach* statement is used to generate N firing rules for an N-way mux, where N is determined from the number of connections made to the input multiport. The multiplexer has $N + 1$ valid utilizations: one for each input-to-output assignment plus the no-fire rule. Each assignment rule states that a value is present on one input, that the output is present, and that the remaining inputs are not present. It is forbidden for the datapath to be configured in such a way as to have values present on more than one mux input. This would create a situation where the datapath computes a value only to have it discarded. A benefit of the TIPI approach is that these wasteful configurations are automatically avoided.

Firing rules from all of the atoms in a datapath are combined to form a system of constraints. An algorithm based on iterative Boolean satisfiability is used to find configurations where all of the atoms' rules are satisfied [119].

In a typical datapath, there are an enormous number of solutions that represent valid cycle-level configurations. To represent the programmability of the PE compactly, TIPI defines an operation



(a) Decrement

(b) Compare Immediate

(c) Checksum Update

(d) Valid, but not a minimal operation

Figure 4.13: Extracted Operations

	Decrement	CompareImm	ChecksumUpdate
Decrement			
CompareImm			×
ChecksumUpdate		×	

Figure 4.14: Conflict Table

to be a *minimal* solution to this constraint system [120]. Briefly, a minimal operation is a datapath configuration that is not simply a combination of smaller operations.

Continuing with the example of Figure 4.11, Figure 4.13 shows the set of extracted operations. There are three minimal operations for this datapath: decrement, compare immediate, and checksum update. A *no-op* operation also exists in which the datapath remains idle for a cycle, but this is not shown in the figure. The valid datapath configuration shown in Figure 4.13(d) is not a minimal operation because it is simply the union of the decrement and checksum update operations.

TIPI captures these non-minimal solutions by creating a *conflict table* for the PE. This is given in Figure 4.14. The conflict table shows that it is possible for software to issue the decrement and checksum update operations simultaneously in one cycle without violating any of the PE's

Cycle	*Slot 0*	*Slot 1*	*Slot 2 . . .*
0	Decrement	Compare	
1	CksumUpdate		
. . .			

Figure 4.15: Macro Operation

constraints. On the other hand, it is not possible to issue the compare immediate and checksum update operations simultaneously. The multiplexer can only forward the result of one of these function units to be written into the register file. Together, the operation set and the conflict table are a compact way of describing the valid ways in which software can use the datapath.

4.3.3 Macro Operations

In many cases there will be combinations of operations that are used frequently in software. For example, in a pipelined machine there will be separate TIPI operations for the consecutive stages of a computation because each stage is a separate single-cycle state-to-state behavior. However, the operations are not useful individually and should always appear in sequence. For this reason TIPI allows architects to define *macro operations* and include these in the set of basic behaviors that the datapath implements.

A macro operation is a spatial and temporal combination of minimal operations. It is defined by creating a table as shown in Figure 4.15. Each row in the table represents a clock cycle, and the columns show what operations are issued in that clock cycle. A macro operation defined over several cycles is a temporal combination of operations. When several operations are issued in one cycle, it is known as a spatial combination of operations.

In this example, the macro operation extends over two cycles. In the first cycle, the decrement and compare immediate operations are performed simultaneously. In the second cycle, the checksum update operation is issued. A macro operation can be arbitrarily long, and the spatial combinations can be arbitrarily wide, as long as the operations in each cycle satisfy the constraints given by the conflict table.

4.3.4 Single PE Simulation

After operation extraction, TIPI can produce a cycle-accurate bit-true simulator for the PE. This is a C++ program that compiles and runs on the host machine. The input to the simulator is

a list of data words, operations, and macro operations that are to be fed to the PE, along with the timing information that says on what cycles the inputs appear. The simulator models the cycle-by-cycle functional behavior of the datapath. The GNU GMP library is used to model the arbitrary bit widths in the datapath [42].

Performance results for single-PE simulation are given in [121]. On average, TIPI simulators are $10\times$ faster than the equivalent cycle-accurate Verilog simulations using a commercial simulator. If the program trace to be simulated is known a priori, TIPI also provides the option to create a statically scheduled simulator. With this technique a speedup of $100\times$ over the equivalent Verilog simulation is attainable. Thus, simulation is not a bottleneck in the iterative design approach even when large programs are used. This is a great benefit to design space exploration.

Automatic simulator generation also solves an important consistency verification problem that impacts other design methodologies. If architects must build simulators by hand, a great deal of effort must be spent ensuring that the simulators are all consistent with the original high-level datapath model. Changes to the original model must be correctly propagated to all of the simulators. This problem is avoided by generating simulators directly from the high-level model in a correct-by-construction fashion.

After designing and simulating individual Sub-RISC processing elements, designers are ready to make multiprocessor compositions of these elements. The methodology and abstractions used for this process are described in the next section.

4.4 Designing Sub-RISC Multiprocessors

Building a multiprocessor architecture is not as simple as merely instantiating a handful of processing elements. Architects must carefully design a system of communications and control that encompasses all of the processing elements. The goal is to construct an architecture that has process-level concurrency capabilities that benefit the intended application domain. Architects must be able to communicate these capabilities to programmers so that applications can take full advantage of the architecture.

On top of this, architects must be able to perform effective design space exploration. After the desired applications are implemented on an architecture and performance results are obtained, it is often necessary to go back and make improvements to the architecture. If it is difficult to make modifications to individual PEs or to the communications and control logic between PEs, then the Y-chart iterative design methodology will break down.

To solve these challenges, Cairn provides a multiprocessor architectural abstraction that simplifies the design process. This abstraction does for multiprocessors what TIPI does for single processors.

TIPI makes it easy to design individual PEs by providing an abstraction that simultaneously assists and restricts architects. Designers do not make arbitrary collections of state machines. Instead, they assemble datapath atoms from a library. The interactions between atoms are defined by a set of formal rules. The benefit of this abstraction is that the valid cycle-by-cycle behaviors of the resulting design can be automatically extracted.

Cairn's multiprocessor architectural abstraction seeks to formalize the interactions between datapaths to achieve similar benefits. Multiprocessors are created by assembling pre-constructed and pre-tested datapaths. The interactions between these datapaths will be defined by a set of formal rules. Designers will find it easier to construct and experiment with multiprocessor architectures, and there will be an abstraction that captures the process-level concurrency of the system.

Like the single-PE design flow, the multiprocessor flow is iterative. This is shown in the right half of Figure 4.10. From a multiprocessor architecture model, there are two code generation processes for producing implementations. One option is to produce a multiprocessor simulator that runs on the host machine. Designers study simulation results to generate ideas on how to improve the system, and then perform architectural design space exploration. The other option is to output synthesizable RTL Verilog. This code is ready to be processed by third-party synthesis tools to create an FPGA or ASIC implementation.

The multiprocessor design environment and simulator generators speed up iterative design by making it easy to perform structural changes and evaluate the consequences. The processes and abstractions used here are extensions to TIPI and are novel contributions of the Cairn project.

This section first describes the criteria for a multiprocessor architectural abstraction. Then the abstraction used in the Cairn design methodology is defined. Finally, the processes for producing implementations of multiprocessor systems are described.

4.4.1 Multiprocessor Architectural Abstraction Criteria

A multiprocessor architectural abstraction must meet the following criteria:

- *Express Architectural Concurrency:* The first concern is to make a formal, precise representation of the process-level concurrency of the architecture. Architectural process-level

concurrency describes that way that processing elements communicate and execute relative to each other. This provides two major benefits.

First, productivity is improved because architects are not burdened with describing the details of the interactions between components in the multiprocessor by hand. The abstraction makes these details implicit in the architectural model. This makes it easier to both design an initial architecture and to make modifications to an architecture during design space exploration.

Second, the abstraction provides a mechanism for exporting the process-level concurrency capabilities of the architecture to programmers. This plans ahead for a disciplined mapping methodology where an application's concurrency requirements are matched up with the architecture's concurrency capabilities.

- *Cover Heterogeneous Architectures:* A multiprocessor architectural abstraction must have sufficient generality to cover the various types of components that appear in heterogeneous systems. It must describe the interactions between all kinds of PEs, memories, on-chip networks and I/O interfaces. A pitfall from the system-in-the-large approach is that there is no coherent model of the process-level concurrency between these heterogeneous components. Designers are forced to consider each component individually. This makes programming difficult.

- *Consider Simple Architectures:* This criterion is motivated by the second core requirement. An architectural abstraction forms the base of a methodology's design abstraction chart. All other abstractions are influenced by the choice of architectural abstraction. The simplest architecture that can be described is also the simplest architecture designers can implement to solve their application problems. If the architectural abstraction assumes a certain level of complexity, then designers must accept that level of complexity throughout the design flow.

  For example, a complex architectural abstraction might state that connected processing elements have cache-coherent shared memory. This is general enough to solve many application problems, but it is expensive and excessive for many others.

  A simple semantics for interactions can be built up to express more complexity, but a complex semantics cannot be subdivided. Designers pay for the complexity regardless of how it is used. An architectural abstraction should therefore appeal to the simplest behaviors that are common to all PEs. It should capture these behaviors precisely.

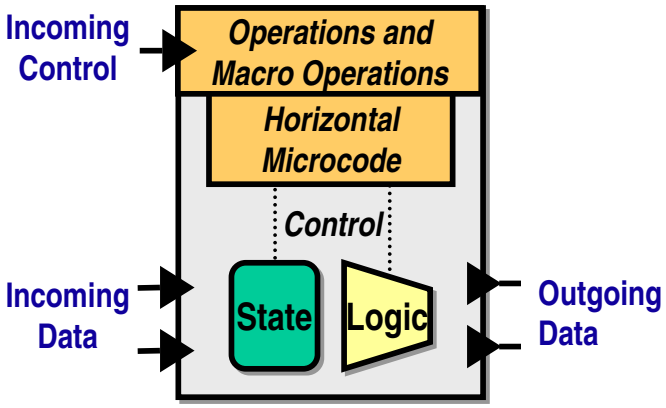


Figure 4.16: External Abstraction of a TIPI PE

4.4.2 The Cairn Multiprocessor Architectural Abstraction

To define an architectural abstraction for TIPI multiprocessors, the first step is to characterize the fundamental behaviors that are shared by all PEs. Figure 4.16 shows how a generic TIPI PE appears to an outside observer. The PE has internal state and logic elements that are controlled by a horizontal microcode unit. Operations and macro operations represent the valid behaviors of the PE. An operation corresponds to a single word of microcode, whereas a macro operation is a sequence of one or more microcode words that are issued to the datapath on sequential clock cycles. During the execution of an operation or macro operation, the datapath may read values from or write values to I/O ports.

In this view of the datapath, the external control and data ports are unconnected. A TIPI PE is not like a RISC processor that fetches and executes instructions from a memory. Instead, it sits idle and performs no computation until control is received from an external source. When a control word appears, the PE executes the corresponding operation or macro operation. This takes a finite and fixed number of cycles. After completion, computation stops until more control words appear. There is no concept of a thread of control that loops, or of a return value that is sent back to the entity that provided the control word.

Because the PE does not fetch its own control words, there is no branching, looping, or conditional execution. These behaviors imply a crossover between data and control within the machine. Values computed in the datapath are used to generate the control words for subsequent cycles. In the TIPI control model, there can be no such crossovers within a PE. If an architect intends to create such a mechanism, it must be described explicitly at the multiprocessor level.

This fundamental behavior places restrictions on the types of computations that a TIPI PE can

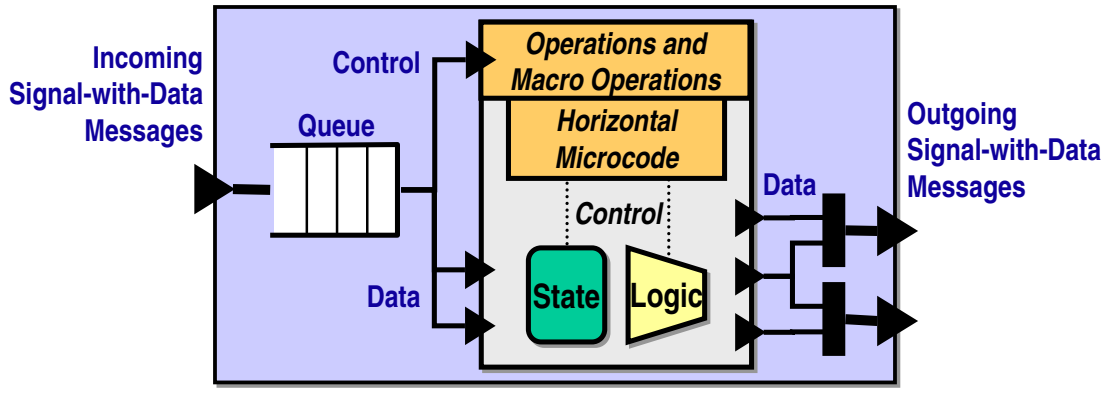


Figure 4.17: TIPI PE Encapsulated as a Signal-with-Data Component

perform. Specifically, recursive functions can not be computed. An operation or macro operation describes the computation of a mathematical projection from the current inputs and architectural state to the outputs and the next architectural state.

This computational model may seem too simple for typical embedded computing applications. A single PE has limited power by design. More complex behaviors are made possible by making multiprocessor compositions of PEs where the PEs provide control and data for each other. This ensures that architects only pay for complexity when the system requires complexity.

The Signal-with-Data Abstraction

Cairn's multiprocessor architectural abstraction provides a semantics for the interaction between PEs based on the fundamental behavior described in the previous section. The basic concept is that PEs can send and receive messages amongst themselves that cause mathematical projections to be computed and architectural state to be updated. A message consists of two parts: a control word indicates what operation or macro operation to perform, and a data component provides values that appear on a datapath's input ports during the execution of the operation or macro operation.

These messages are called *signal-with-data messages*, and the resulting multiprocessor architectural abstraction is called the *signal-with-data abstraction*. A multiprocessor model that uses this abstraction is called a *signal-with-data model*. A signal-with-data model is a directed graph where the nodes encapsulate TIPI PEs and the edges indicate that a source node sends signal-with-data messages to a sink node.

Figure 4.17 shows how a TIPI PE is encapsulated as a component in this model. A signal-with-data component has exactly one input port for incoming signal-with-data messages. Each message

is an atomic unit consisting of a control word and N data words, where N is the number of data input ports on the enclosed TIPI PE. This format is static and is elaborated at design time.

Incoming messages are first pushed into a theoretically unbounded queue. This is used to create a globally asynchronous, locally synchronous (GALS) architectural design pattern. The enclosed PE consumes these messages at its own pace, one at a time and in the order received.

When the enclosed PE is idle and the queue is not empty, the PE will begin processing the signal-with-data message at the head of the queue. The control and data components of the message appear at the PE's input ports for the duration of the execution of the specified operation or macro operation. When this computation is finished, the message is popped off the queue and the PE either becomes idle or begins executing the next message.

Some operations and macro operations do not read from every PE data input port. In this case, the signal-with-data message contains *don't care* values in these fields. The format of the signal-with-data message is the same for all of the operations and macro operations within a PE. In a multiprocessor, each PE may have a different format for incoming signal-with-data messages. Static type checking is used to verify that each source component produces the right message formats for the various sink components that it is connected to.

Although it has only one input port, a signal-with-data component may have an arbitrary number of output ports for producing outgoing signal-with-data messages. These outgoing messages are assembled from the data outputs of the enclosed TIPI PE at the end of an operation or macro operation. In a multi-cycle macro operation, the PE may write to an output port more than one time. In this case only the most recently written value is assembled into an outgoing signal-with-data message. A single PE output may fan out to become a field in more than one message. This situation is demonstrated by the middle output port in Figure 4.17.

The execution of an operation or macro operation, the popping of a message from the queue, the bundling of outputs to create new signal-with-data messages, and the pushing of these messages into the queues of destination components are all treated as a single atomic operation.

A component may generate a signal-with-data message that contains the *noop* operation in the control field. In this case computation stops at the source component and does not continue at the sink component. The outgoing message is not pushed into the queue of the destination component in this case.

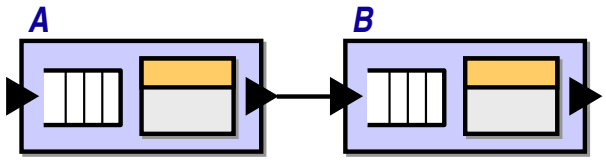


Figure 4.18: Point-to-Point Signal-with-Data Connection

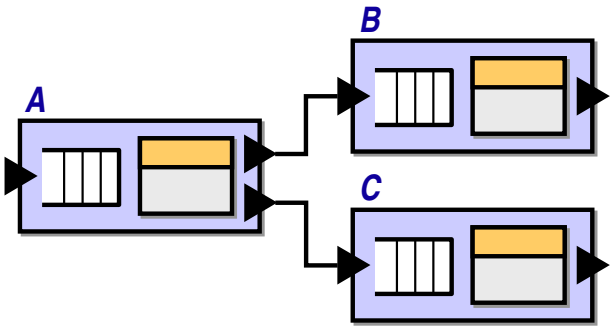


Figure 4.19: One Component Produces Multiple Signal-with-Data Messages

Signal-with-Data Connections

The next figures enumerate the valid ways of connecting signal-with-data components in a signal-with-data model. Figure 4.18 shows the most basic point-to-point connection. In this case components A and B are globally asynchronous, locally synchronous processors where A sends signal-with-data messages to B.

Figure 4.19 shows that one component can produce multiple signal-with-data messages that are sent to different destination components. Computation at component A may invoke different computations on different data at B and/or C.

In Figure 4.20, component A produces a single message that fans out to both B and C. Both destination components will begin executing on the same data. The control word sent to B and C

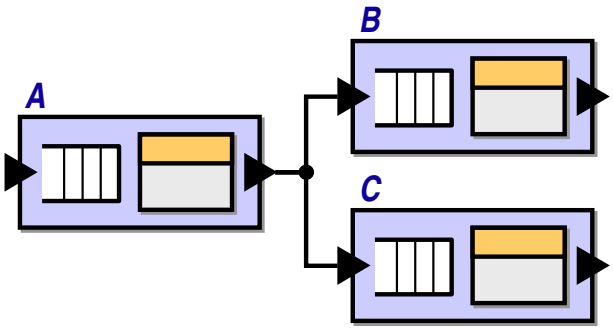


Figure 4.20: Fan-Out of a Signal-with-Data Message

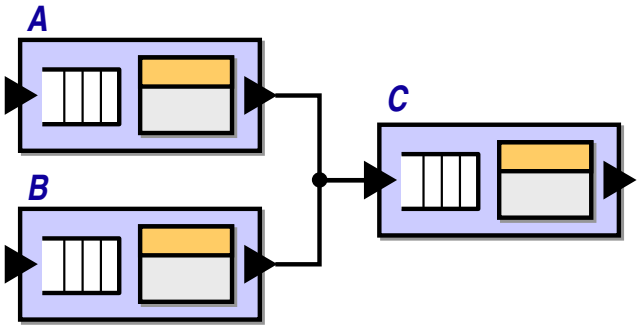


Figure 4.21: Fan-In of a Signal-with-Data Message

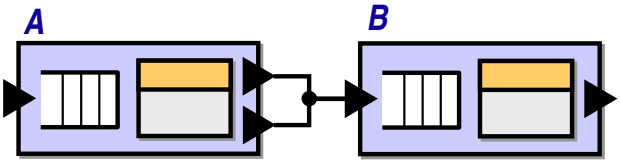


Figure 4.22: Combination of Multiple Messages and Fan-In

is the same, but this word may invoke different computations at B and C.

A signal-with-data component may receive messages from multiple sources. In this case, architects draw multiple connections that fan in to the input port of the sink component. This is shown in Figure 4.21. This type of connection indicates that both A and B can push messages into C's queue. The signal-with-data abstraction does not provide a concept of simultaneity. Component C executes all of the messages it receives. In hardware, a fair arbitrator may be needed to ensure that this requirement is met. Other than fairness, the signal-with-data abstraction places no other restrictions on the arbitration policy.

It is legal to combine fan-out and fan-in of messages. Figure 4.22 shows an example where component A produces two messages that are both sent to component B. Component B executes these computations sequentially in an unspecified order. In another case, component A could produce one message that is fanned out to two messages that are both sent to B. Then component B would execute the same computation on the same data two times in a row.

Cycles are allowed in a signal-with-data model. Figure 4.23 shows an example. This type of multiprocessor may experience deadlock when queue sizes are bounded. It is up to architects to ensure that queue sizes are matched to message production and consumption rates if it is necessary to avoid deadlock in the final implementation.

An important utilization of a cycle is the case where a component sends signal-with-data messages to itself, as shown in Figure 4.24. This is how architects can describe the behavior of a stan-

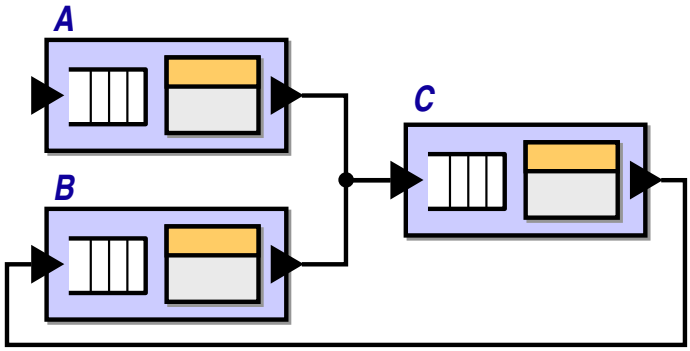


Figure 4.23: Cycles in Signal-with-Data Models

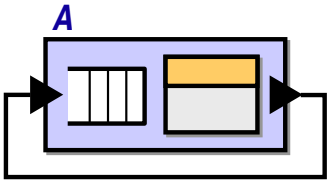


Figure 4.24: A Component Sending a Message to Itself

dard RISC PE using the signal-with-data abstraction. RISC instructions update a program counter in addition to performing an ALU or memory operation. The updated program counter value is used to fetch the next instruction to execute. This is an implicit crossover between data and control.

In the signal-with-data abstraction, the kind of crossover is modeled explicitly using a self-loop in the multiprocessor model. The encapsulated PE may contain a program counter state element and an instruction memory. Each operation or macro operation can update this state in addition to performing other computations. The next state value can be used to index the instruction memory, and the resulting value is treated as a new signal-with-data message to be executed on the following cycle. If the message queue is of size one, then the resulting hardware is exactly the same as what is found in a RISC processor's control logic. The signal-with-data abstraction does not add any overhead.

This example shows how the simple signal-with-data semantics can be built up to implement more complex semantics for process-level concurrency, such as the sequential behavior of RISC processors. Of course, architects can create entirely different schemes for producing new control words other than program counters and instruction memories. This leads to novel schemes for application-specific process-level concurrency. The key is that the signal-with-data control abstraction is simpler and less expensive when RISC-style control is not necessary, and no more expensive than RISC when sequential control is necessary.

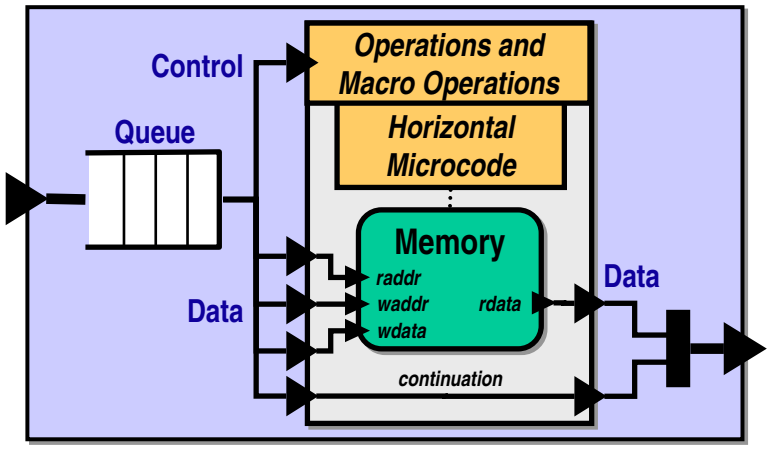


Figure 4.25: On-Chip Memory Modeled as a TIPI PE

Memories, On-Chip Networks, and Peripherals

Heterogeneous multiprocessors contain more than just processors: There are also memories, on-chip networks, and peripherals. In the Cairn framework these components are meant to be modeled as TIPI PEs. Architects build unique datapaths with specialized operation sets that reflect the behaviors of these non-processor components.

An example of an on-chip memory is shown in Figure 4.25. This "datapath" has one component (a state element) and two operations (read and write). Other components in the multiprocessor can read and write to the memory by sending *read* and *write* signal-with-data messages to this component.

The read operation returns a value to the component that sent the *read* signal-with-data message. This behavior is implemented using a mechanism similar to the continuations that are found in programming languages such as Scheme and Standard ML. The calling PE creates a signal-with-data message containing a *read* control word, the desired read address, and an extra control word indicating where computation is to continue after the data has been fetched from memory. The memory component executes this message, fetches the data, and produces a new signal-with-data message using the provided control word and the memory data. This is sent back to the original component, where computation continues using the result of the memory read operation.

On-chip network components and peripherals are modeled using similar specialized datapaths. Instead of providing operations that perform mathematical operations on data, these datapaths provide operations that move data spatially or drive external I/O pins.

The Sub-RISC approach is unique in that it can model these hardware components as data-

paths and export their capabilities as operations and macro operations. With only a few operations, very little control logic is implied. Therefore it is not expensive to encapsulate these objects first as datapaths and second as signal-with-data components. The benefit is that all components of heterogeneous multiprocessors export their capabilities in the same way.

Benefits of the Signal-with-Data Abstraction

Cairn's signal-with-data abstraction meets all of the criteria given in Section 4.4.1. First, the message passing semantics formally represent the process-level concurrency of the system. Processors, memories, on-chip networks and peripherals are all exported as components that interact via signal-with-data messages. The data-level and datatype-level concurrency of each component is exported as a TIPI operation set and conflict table. Later, when applications are mapped onto these architectures, this exportation of the architecture's capabilities will be used to compare application concurrency to architectural concurrency. A disciplined mapping methodology will study any similarities and differences to drive design space exploration toward better system solutions.

A formal architectural abstraction improves productivity because architects do not have to concern themselves with describing the details of wiring together heterogeneous components. Simply drawing an edge in a signal-with-data model implies a defined system of interaction. This makes it easy to construct large multiprocessor models and experiment with architectural changes without manually editing RTL code.

The signal-with-data abstraction works for all of the different components found in heterogeneous multiprocessors and describes their capabilities in a common way. This is an important benefit of the Sub-RISC approach. Components such as memories and on-chip networks can be modeled as special cases of datapaths. Since the Sub-RISC paradigm assumes a simple model of control logic, there is negligible overhead in encapsulating these resources in datapaths.

Finally, the signal-with-data abstraction appeals directly to the second core requirement: consider simple architectures. It precisely captures the fundamental behaviors of Sub-RISC machines. The minimum design remains simple. If designers require more complex styles of process-level concurrency, these can be built by making compositions of basic blocks. Therefore the cost of complexity is only incurred when the applications demand complexity.

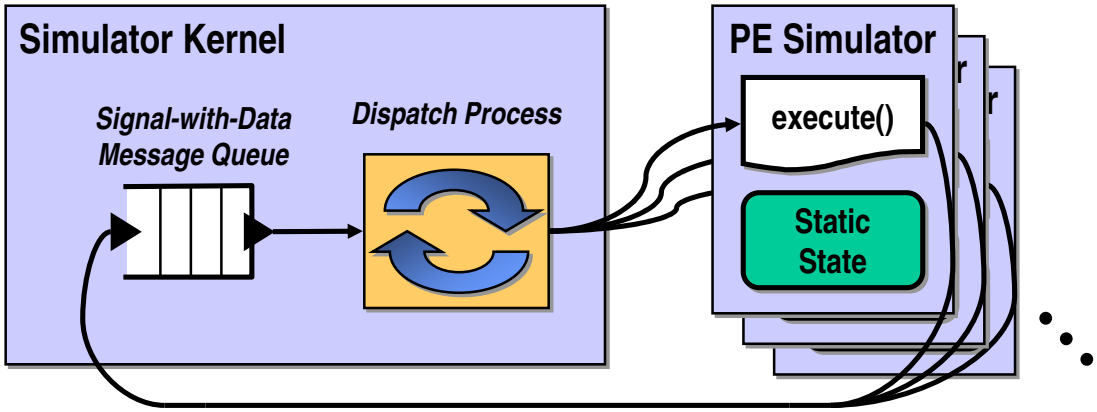


Figure 4.26: Cairn Behavioral Multiprocessor Simulator Block Diagram

4.4.3 Multiprocessor Simulation

The Cairn framework modifies and extends TIPI to generate compiled-code simulators for Sub-RISC multiprocessors. Code generation processes take a multiprocessor system model as an input and produce C++ code. This code compiles and runs on the development workstation to simulate the functionality of the multiprocessor and to collect performance data.

Architects can choose to perform simulation at various levels of detail and accuracy. Two different simulators are discussed in this section. A coarse-grained behavioral simulator models the computations that the multiprocessor performs but does not keep track of timing information. A more detailed cycle-accurate simulator models both behavior and time at the expense of slower simulation.

Behavioral Simulation

Cairn's behavioral simulator models how the processing elements in a heterogeneous multiprocessor create and consume signal-with-data messages. The computations that PEs perform in response to signal-with-data messages are fully simulated with bit-level accuracy. What is not simulated is the amount of time it takes to perform a computation, or the amount of time it takes to send a signal-with-data message from one PE to another. Timing accuracy is exchanged for high simulator performance. This type of simulation is useful to quickly demonstrate the functional correctness of a large system.

A block diagram of the behavioral simulator is shown in Figure 4.26. There are two major components in this simulation system. A set of PE simulation blocks is created for the components

```
class CairnPE {
public:
        CairnPE() { };
        virtual ~CairnPE() { };
        virtual void execute(queue<CairnPE*> *readyQueue) { };
};
```

Figure 4.27: Abstract Base Class for Signal-with-Data Component Simulation

in the multiprocessor. These blocks are C++ classes, and there is exactly one for each component.

Second, there is a simulation kernel that is independent of the multiprocessor architecture. The kernel contains an unbounded queue for storing signal-with-data messages. A looping dispatch process removes messages from the queue and calls into the appropriate PE simulator class object.

The execute function within this class simulates the computation of an operation or macro operation. This code reads inputs from the signal-with-data message, updates architectural state (stored as static data in the class), and creates outgoing signal-with-data messages. Any outgoing messages are pushed into the kernel's message queue.

When the execute function returns, the dispatch process loops and continues at the next message waiting in the queue. This process represents one possible sequentialization of the concurrent behavior of a signal-with-data model. All messages are executed, albeit in an order that may differ from a parallel hardware implementation of the system. If the message queue ever empties, the machine has ceased producing new signal-with-data messages. Computation has halted across all PEs and the simulator exits.

Figure 4.27 shows the C++ code for the base class for the individual PE simulators. A concrete subclass is created for each component in the multiprocessor model. This subclass provides an execute function that simulates all of the operations and macro operations that the component implements. The subclass also provides static variables that represent the state contained within the PE. Static variables are used to ensure that there is one global instance of each state element in the simulator program. Since the simulator kernel is single-threaded, it is not necessary to use synchronization primitives such as semaphores or mutexes to protect these static variables.

Control and data input ports are implemented with normal class instance variables. This data changes with each signal-with-data message. To create a new message, the simulator dynamically creates a new instance of the class for the PE that is the target of the message. The input port values are filled in appropriately. This new object is then pushed into the simulation kernel's message queue for future dispatch.

Pseudocode for a PE simulator subclass is shown in Figure 4.28. The `UINT<A>::T` syntax is

```
class ExamplePE : public CairnPE {
public:
        ExamplePE() { };
        virtual ~ExamplePE() { };

        /* PE input ports */
        UINT<A>::T program;
        UINT<B>::T data1;
        UINT<C>::T data2;

        /* PE state elements */
        static UINT<N>::T regfile[32];

        virtual void execute(queue<CairnPE*> *readyQueue) {
                switch (program) {
                        case macro_operation_0:
                                /* Output ports */
                                UINT<D>::T output1;
                                UINT<E>::T output2;

                                { /* read inputs, read state, update state here */ }

                                /* create outgoing signal-with-data message */
                                if (output1 != noop_operation) {
                                        DestinationPE *pe = new DestinationPE();
                                        pe->program = output1;
                                        pe->data1 = output2;
                                        readyQueue->push(pe);
                                }
                                break;

                        case macro_operation_1:
                                .
                                .
                                .

                        default:
                                cerr << "Invalid signal-with-data message!" << endl;
                }
        };

};
```

Figure 4.28: Individual PE Simulator Pseudocode

```
class CairnSimulator {
public:
        CairnSimulator() { };

        void run() {
                while (!readyQueue.empty()) {
                        CairnPE *pe = readyQueue.front();
                        readyQueue.pop();
                        pe->execute(&readyQueue);
                        delete pe;
                }
        };

private:
        queue<CairnPE*> readyQueue;

};
```

Figure 4.29: Cairn Behavioral Multiprocessor Simulator Kernel

the type name for a bit vector of length A bits. The `UINT` template is a C++ template metaprogram that is evaluated at compile time to match the desired bit vector to a primitive C++ type such as a `short`, `int`, or `long long`. When A is greater than 64, an arbitrary-precision data type object from the GMP math library is used [42]. This speeds up simulation by allowing the host machine's native types to be used whenever possible, and the expensive GMP arithmetic operations to be used only when necessary.

A PE execute function is a switch statement with one case for each operation and macro operation exported by the PE. The first part of each case simulates the computation of the operation or macro operation. TIPI's single-PE simulator generator is used to fill in the contents of this section. This code reads from input ports, and both reads and writes architectural state variables. PE output ports are local variables in this section.

After the operation or macro operation is computed, the simulator creates outgoing signal-with-data messages using the values found on the output ports. As an optimization, control words are checked to see if they are the *noop* operation. If this is the case, then computation does not continue at the destination PE and the simulator avoids creating a message. If not, a new instance is constructed of the destination PE class. The destination PE is known statically from the connections in the signal-with-data multiprocessor architecture model. To send a signal-with-data message to that PE, values must be provided on all of its input ports. This is done by assigning output values of the current PE (contained in local variables of the execute function) to the input port members of the new instance. Finally, the new instance is pushed into the simulator kernel's message queue.

Figure 4.29 shows the simulator kernel dispatch loop. This process continuously executes signal-with-data messages from the queue. The simulation of each message may cause zero or more new messages to be pushed into the queue. When the queue empties, the machine has halted and the simulator exits.

Simulations are started by priming the queue with initial signal-with-data messages. This represents all processing elements receiving a *reset* message when the multiprocessor is booted.

Full Timing Simulation

Detailed simulation with full timing information requires three additional pieces of information. First, the simulator must know how long it takes to execute every operation and macro operation. This data is provided by TIPI when the execute functions for each processing element are generated. An operation always takes exactly one cycle to run. Since macro operations are free of conditionals and loops, they take a fixed and finite number of cycles to run. This is equal to the number of rows found in the table that defines the macro operation (e.g. Figure 4.15).

Second, the simulator must know the clock rates of the PEs. Architects must provide this information as an annotation on each PE in the multiprocessor model. This number is used to convert times measured in clock cycles to times measured in seconds.

Finally, the simulator must know the size of each component's signal-with-data message queue. This also must be provided via annotations in the model. If a source PE wishes to send a signal-with-data message to a destination PE but the destination PE's queue is full, or if an arbitrator is currently withholding access to the queue, the source PE blocks until the message can be pushed.

With these extra pieces of information a discrete-event simulator can be generated for a signal-with-data architecture model. This simulation will be slower than the behavioral simulator described above, but it will still be faster than an RTL simulation using a third-party Verilog or VHDL simulator. This is because the only events simulated in the system are those that relate to pushing and popping signal-with-data messages. An RTL simulator would also create events for the switching activity within each datapath. This is a significant computational burden.

4.4.4 Multiprocessor RTL Code Generation

Cairn also modifies and extends TIPI's RTL Verilog code generators. This enables the generation of synthesizable Verilog models for Sub-RISC multiprocessors that can be taken through third-party FPGA or ASIC design flows.

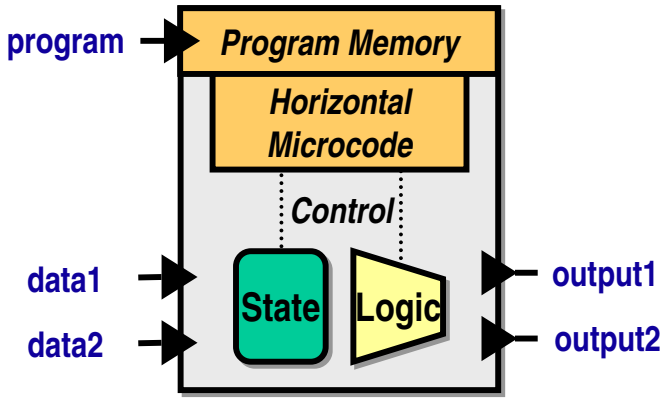


Figure 4.30: Example PE for Verilog Code Generation

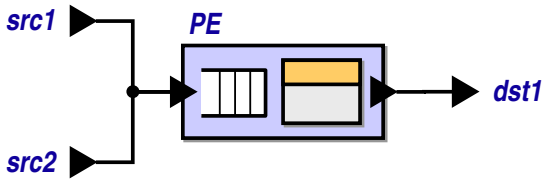


Figure 4.31: Example PE in a Multiprocessor Context

The Verilog code generator produces one Verilog module for each component in the signal-with-data architecture model. A top-level module instantiates one copy of each component module and wires them together. As an example of the structures that are built, consider the PE shown in Figure 4.30. The relevant features of this PE are that it has three input ports (*program*, *data1*, and *data2*) and two output ports (*output1* and *output2*).

Figure 4.31 shows how this example PE is used in the multiprocessor model. The PE receives signal-with-data messages from two sources (*src1* and *src2*) and produces signal-with-data messages for one destination PE (*dst1*).

Figure 4.32 shows the hardware that is generated by the Verilog code generator. The blocks labeled *TIPI Datapath* and *Datapath Microcode* are independent of the PE's multiprocessor context. These structures are generated by the single-PE TIPI Verilog code generator. Cairn's code generator adds the rest of the hardware to wrap the datapath in the queue-like interface implied by the signal-with-data abstraction.

The action of pushing a signal-with-data message from one PE to another is accomplished with a three-way handshake that uses the following ports: *req*, *push*, and *full*. When PE A wishes to send a message to PE B, it drives the message data on its output ports and asserts the *req* signal, requesting access to B's queue. PE B responds using the *full* signal. If *full* is high, PE A must wait

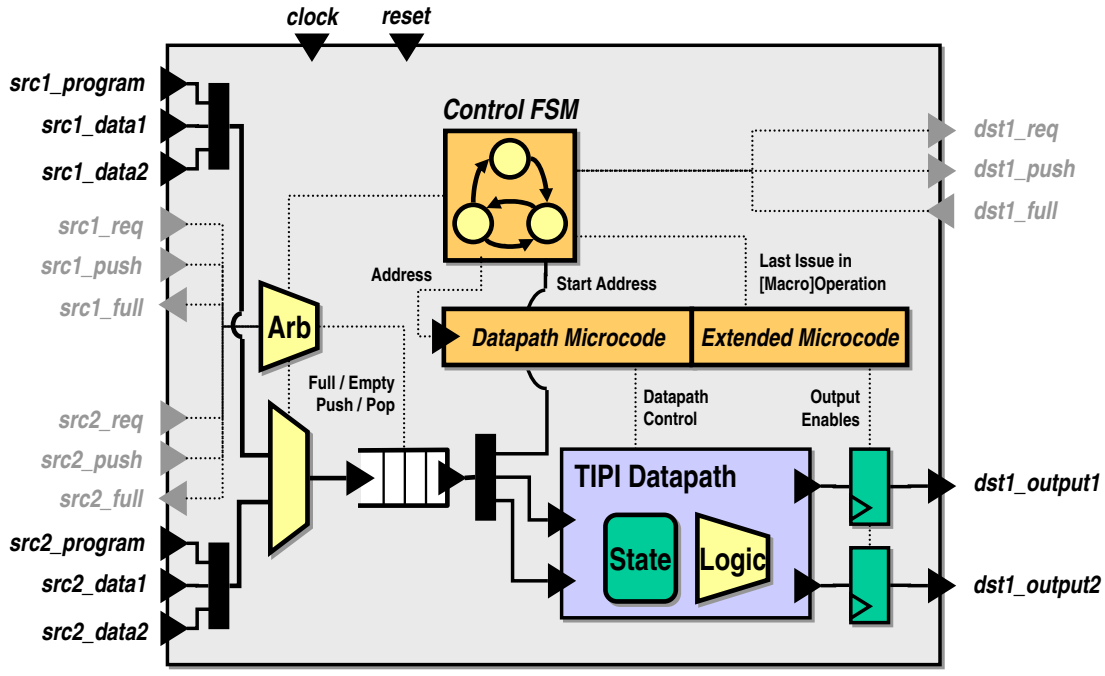


Figure 4.32: Schematic of the Generated Verilog Design

for access to the queue. The queue may either be full, or an arbitrator may be granting access to another PE at the moment. Once *full* goes low, PE A can push the message by asserting the *push* signal. This terminates the handshake.

Cairn's code generator adds one set of handshaking ports for each pair of interacting PEs. In this example, the PE interacts with three others: two sources and one sink. Thus there are three sets of *push*, *req*, and *full* signals. Additionally, a set of input ports is created for receiving signal-with-data messages from each source (*srcn_program*, *srcn_data1*, *srcn_data2*).

A single set of output ports is created for sending signal-with-data messages to any number of destinations. These wires simply fan out if there is more than one destination. Each destination has unique handshaking ports, however.

When a PE receives control from more than one source, Cairn adds a fair arbitrator to the design. The arbitrator knows the current state of the queue (full or not, empty or not) and decides in what order to complete handshakes with source PEs. The signal-with-data abstraction prohibits dropping messages. The arbitrator must therefore ensure that all requested handshakes are eventually completed.

The blocks labeled *Datapath Microcode* and *Extended Microcode* are memories that store the microcode for the datapath's operations and macro operations. A Sub-RISC datapath is programmed

by designing new macro operations and storing the corresponding machine code in these memories. The *Datapath Microcode* block provides control bits for the core TIPI datapath. This circuitry is constructed exactly as in the single-PE case using the regular TIPI tools. The *Extended Microcode* block extends the length of the horizontal microcode words to provide some extra control bits that are necessary for the core TIPI datapath to function in a Cairn multiprocessor context. This includes bits that guide the execution of multi-cycle macro operations and bits that aid in the creation of outgoing signal-with-data messages. The write interface for the microcode memory is not shown in the figure. Also, the clock and reset signals are not connected internally for clarity.

The block labeled *Control FSM* is a datapath-independent state machine that controls the sequencing of operations and macro operations. In the default state, the control FSM idles the core datapath and waits for the local message queue to become non-empty. When this occurs, a signal-with-data message has arrived and control and data are available on the outputs of the queue.

The data portion of the message is sent to the input ports of the core datapath. These values are driven for the duration of the execution of the message. The control portion of the message indicates what operation or macro operation to execute. This value is a pointer into the microcode memory. It indicates the address of the first word of microcode for the operation or macro operation. The control FSM uses this value to address the microcode memory. The resulting control outputs configure the core datapath and one cycle of computation is performed.

Some macro operations take more than one cycle to compute. A bit in the extended portion of the microcode memory tells the control FSM when it has issued the last microcode word in the operation or macro operation. When this is not asserted, the control FSM increments the microcode address by one and performs an additional cycle of computation. This continues until the entire multi-cycle macro operation has been executed.

Unlike a RISC PE, there is no conditional execution or looping. The datapath executes every line of microcode it is given, and the only action that is performed on the microcode address is the increment action.

Cairn creates registers that store the most recently written values of the PE's output ports. When an operation or macro operation writes to an output port, a bit in the extended portion of the microcode enables writing to the appropriate register. This hardware gathers the datapath's outputs so that they may be pushed into a downstream PE as a new signal-with-data message.

After executing the last microcode word for the specified operation or macro operation, the control FSM prepares to send outgoing signal-with-data messages. First, the local message queue is popped. Then, the control FSM uses the *dst1_req*, *dst1_push*, and *dst1_full* signals to handshake

with the downstream PE. The FSM may be in this state for several cycles while waiting for space to become available in the downstream message queue. When the handshake completes, the FSM starts working on the next message in the local message queue.

If there is more than one downstream PE, the control FSM can initiate handshakes with all of them in parallel. This action completes after all downstream PEs complete the handshake.

To wire multiprocessor components together at the top level, the code generator simply follows the edges in the signal-with-data model. Each edge represents a direct connection of the corresponding data, *req*, *push*, and *full* wires.

The resulting Verilog code can be used to create FPGA or ASIC implementations of the multiprocessor architecture. Architects can also perform detailed discrete-event simulations using third-party Verilog simulators.

4.5 Summary

This chapter covered the abstractions and processes found at the bottom of the Cairn design abstraction chart: the abstractions and processes for designing, implementing, and exporting heterogeneous multiprocessor architectures. First, Sub-RISC processing elements were identified as a valuable basic block for building future systems. No other candidate has the same potential for providing hardware support for multiple granularities of concurrency. Also, the lightweight nature of Sub-RISC PEs directly supports the second core requirement: consider simple architectures.

Second, an architectural abstraction was developed for describing multiprocessor compositions of Sub-RISC PEs. This abstraction simultaneously assists and restricts architects. It enforces a particular model of interaction between PEs, freeing designers from having to specify the details of these interactions manually.

The interaction rules provide a mechanism for capturing the process-level concurrency of a multiprocessor and exporting it to programmers. In a nutshell, PEs are sources and sinks of signal-with-data messages. Each message is a combination of control and data that invokes the computation of a mathematical projection and the possible update of architectural state.

In later chapters, this exportation of a PE's capabilities will be used as part of a disciplined mapping methodology that implements application-level concurrency on architectural resources. A key to this approach is the ability to understand and compare application requirements and architectural capabilities. The signal-with-data abstraction plans ahead for this step by making a precise statement about the multiprocessor's process-level, data-level and datatype-level concurrency.

Lastly, two processes for moving downward in the design abstraction chart from an architecture model toward implementation were presented. First, compiled-code simulators and synthesizable RTL designs can be automatically generated. Architects can trade off simulation accuracy for speed to perform either functional testing or performance testing. Second, Cairn's RTL code generator builds correct-by-construction synthesizable designs.

The automatic generation of simulators and implementations greatly improves designer productivity in an iterative design flow. Architects can experiment with structural changes to individual PEs and the multiprocessor architecture alike, with rapid feedback on the results of these experiments. The more design space exploration that takes place, the more confidence designers have in the quality of their designs.

In the next chapter, attention shifts to the top of the design abstraction chart. The abstractions and processes found there help programmers model concurrent applications, with the goal of implementation on the Sub-RISC multiprocessors described here.

Chapter 5

The Next Application Abstraction

This chapter focuses on the abstractions and processes found at the top of the design abstraction chart. Here, programmers use multiple models of computation to describe the different facets of heterogeneous applications and their interactions. An underlying application abstraction captures the concurrency requirements described in the different application facets in a common form. This abstraction carries these requirements downward in the design abstraction chart to a disciplined mapping methodology for implementation on a multiprocessor architecture. These parts of the Cairn methodology are shown in Figure 5.1. The corresponding steps in the Y-chart design flow are shown in Figure 5.2.

Just as in the previous chapter, two important decisions are considered here. First, what application domain should be used as a design driver? Then, what application abstraction should be used to capture the heterogeneous concurrency of these applications?

The intent of the Cairn methodology is to support a diversity of applications rather than one specific application domain. Therefore, a design driver is chosen that exhibits the general problems that arise from heterogeneous concurrency and the concurrency implementation gap. This will demonstrate that the Cairn approach can handle the core challenges of heterogeneous concurrency. Adapting the methodology to support additional application domains requires only extensions to the tools and not a complete methodological overhaul.

A good design driver will require multiple styles of concurrency on several levels of granularity. It should be a multifaceted application where the interactions between different facets are as important as the facets themselves. Additionally, it should be an example where the concurrency implementation gap causes problems for existing methodologies.

After choosing an application domain, this chapter presents an application-level abstraction

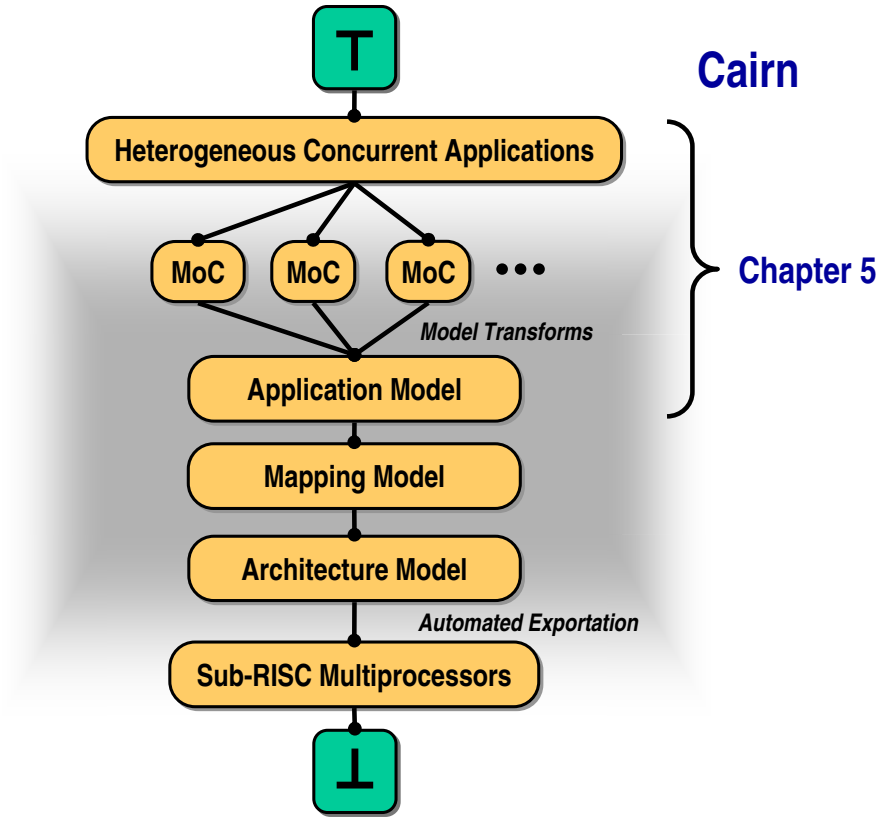


Figure 5.1: Application Side of the Cairn Design Abstraction Chart

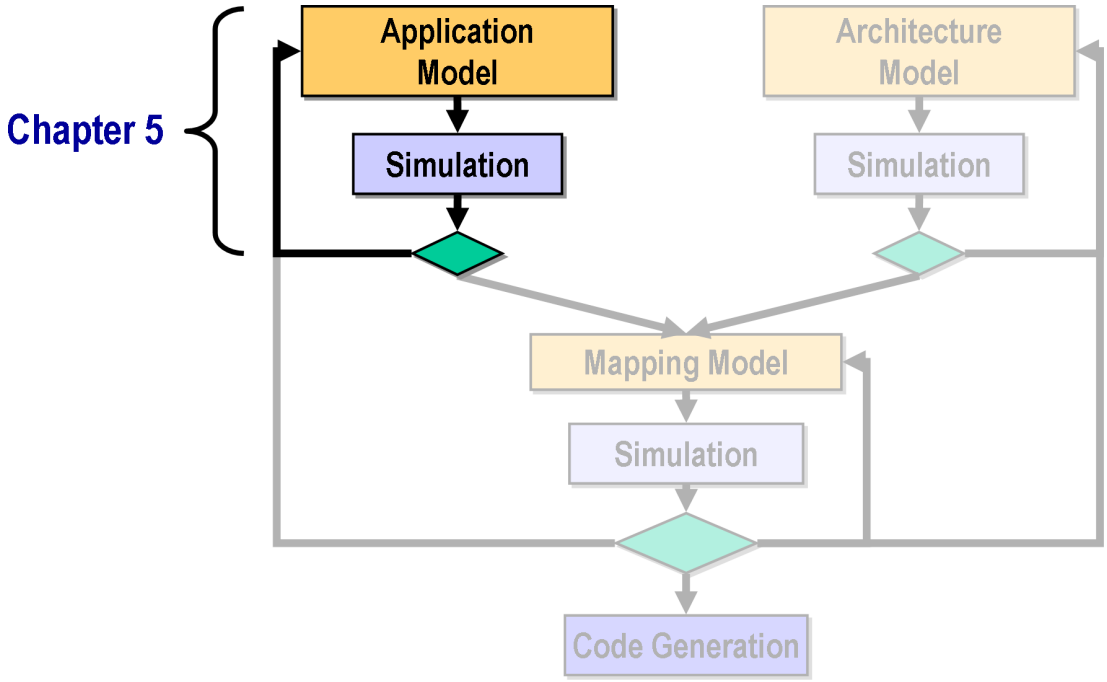


Figure 5.2: Application Side of the Y-Chart Design Flow

that captures the heterogeneous concurrency found in embedded applications. The application abstraction must capture the interactions between different application facets as well as the requirements found within each facet. This knowledge of application concurrency will later be used to compare application requirements to architectural capabilities, leading to functional implementations on multiprocessor architectures.

The next section discusses the characteristics of the network processing application domain. This domain is shown to have sufficient heterogeneity to fully exercise a design methodology. Section 5.2 gives a definition of the term application abstraction and presents the goals and criteria that such an abstraction should meet.

Then, the pitfalls of existing parallel programming languages are organized and compared. Existing languages are not suitable for use as Cairn's application abstraction. The closest match is the Ptolemy II system, which is discussed in depth in Section 5.4. A new application abstraction for multifaceted concurrent applications, based on Ptolemy, is presented in Section 5.5. This abstraction is the second of the three major abstractions of the Cairn methodology, and is a core contribution of this dissertation.

5.1 The Network Processing Application Domain

Network processing rose to prominence as an application domain for programmable multiprocessors in 1999 when Intel, Motorola, and C-Port independently announced network processor products. These network processors offered a combination of software programmability and hardware support for concurrency. This seemed to be the ideal solution for a domain that needs to respond to a continuous influx of new services, protocols, and rapid growth in bandwidth requirements.

Dozens of other network processor architectures followed [105], but all were short-lived. As previously stated, the common sentiment was that these architectures were difficult if not impossible to program. Network processing applications have heterogeneous concurrency, and so do network processor architectures. Existing deployment strategies cannot cope with the concurrency implementation gap in this scenario.

This makes network processing an ideal candidate to demonstrate the Cairn methodology. The applications in this domain require hardware support for concurrency in order to meet performance requirements, but existing architectures do not appear to provide features that truly benefit the application. Cairn's Sub-RISC target architectures have the potential to better match the application's requirements with their "more simple" and "more complex" philosophy.

Likewise, software programmability is necessary to remain flexible to changing application requirements, but existing programming methodologies do not make it easy to rapidly deploy or modify an application. Cairn's disciplined deployment methodology will simplify programming and make it possible to realize the benefits of software programmability.

This section explores the heterogeneous concurrency found in network processing applications. This reveals what concepts programmers must be able to capture using an application abstraction.

Network processing involves performing computations on packet headers and packet bodies. These are separate application facets with distinct concurrency requirements. Header processing tasks focus on manipulating flows of packets and only perform computations on header fields. Packet body processing tasks perform computations on the data contained in packets. Full network processing applications will use a combination of these two facets.

5.1.1 Packet Header Processing

Examples of header processing tasks include routing, quality of service, traffic shaping and overlay networks. These applications are concerned with managing packet flows both temporally

and spatially. Temporal management means changing the time qualities of a flow of packets, such as buffering, queuing, dropping, metering, and shaping. Spatial management means moving packets between different networks, such as in a router or a switch.

For both of these issues, designers are interested in classifying packets according to data found in the header fields, and then applying a policy to the packets according to that classification. All three levels of concurrency are evident here: datatype-level, data-level and process-level. Within each level of granularity, there are several variations.

- *Datatype-level concurrency* appears in the unusual bit widths and data formats found in packet header fields. For example, in an IPv4 header there are fields of length 1, 2, 4, 6, 8, 13, 16 and 32 bits. Common logical operations include setting and clearing individual bits. Arithmetic operations include decrementing an unsigned 8-bit time-to-live field and computing a 1's complement checksum. More complex operations include longest prefix match and multi-field categorization.

- *Data-level concurrency* is found when the application can process multiple packet header fields in parallel. A common operation is to update a packet header field and simultaneously perform an incremental update to a checksum field. When computing a checksum from scratch, the application performs a vector-type reduction add operation.

- *Process-level concurrency* covers the ability to process multiple packets or streams of packets in parallel. In some applications, there are no data dependencies between packets. Each packet receives the same treatment, so all packets currently in the system can be processed simultaneously. Other applications perform stateful processing and have a different style of process-level concurrency. Some packets cannot be processed until previous packets have been examined. Additionally, some networking protocols may be sensitive to packet reordering. While there may be no packet data dependencies at first glance, designers may need to describe a certain style of process-level concurrency for higher-level reasons.

5.1.2 Packet Body Processing

Network intrusion detection and cryptography are examples of tasks that perform computation on the bodies of packets. In network intrusion detection, packet contents are inspected to determine if the packet is designed to exploit a known weakness in an end-user application or service. Attack patterns employed by malware, viruses and trojans are studied and then searched for in packet

flows. This application is a game of cat-and-mouse against crackers, where continuous updates are required to respond to the discovery of new exploits.

Cryptography is an increasingly important application that can help ensure privacy and authentication in communications. Encryption algorithms encode the data in packets to prevent it from being seen by unauthorized parties. Digital signatures and key exchange protocols provide a mechanism for authenticating partners. Each of these applications use datatype-level, data-level, and process-level concurrency, but the exact types of concurrency differ from those found in header processing tasks.

- *Datatype-level Concurrency:* Packet body processing uses more uniform data types than header processing, but the computations performed on these data types are more complex. Data fields are typically viewed as simple arrays of characters. Network intrusion detection applications search for regular expressions within these arrays. This requires character equality comparisons, range comparisons, set membership or non-membership computations, case sensitive or non-sensitive comparisons, and occasionally even fuzzy comparisons. Cryptography requires entirely different arithmetic operations such as XORs, bit shuffles, and modular arithmetic.

- *Data-level Concurrency:* Some pattern matching algorithms process more than one character at a time, and this provides an opportunity for data-level concurrency. Cryptography algorithms have intricate dataflow structures and exhibit a wealth of parallelism.

- *Process-level Concurrency:* Several variations on process-level concurrency are seen in each example application as well. In intrusion detection, a given data stream can be searched for several different patterns at the same time. Alternatively, multiple independent streams can be matched against the same pattern at the same time. A combination of these styles of process-level concurrency can be used as well. In cryptography, multiple packet flows can be encoded or decoded simultaneously.

5.1.3 Interactions between Header and Body Processing

A full-scale network processing application will combine elements of packet header processing and packet body processing. When this occurs, correctly modeling the interactions between the different application facets is as important as correctly modeling the facets themselves.

In a routing application, both packet headers and packet bodies must be forwarded to the proper destination even though the algorithms only inspect the headers. Communicating an entire packet over the on-chip network in a multiprocessor can be expensive, so one option is to split headers from bodies on packet ingress and rejoin them on egress. Packet headers are communicated within the multiprocessor while bodies are offloaded to an external memory until the packet needs to be transmitted. Management of packet bodies is a separate application facet because the concerns of splitting, joining, storing, and fetching packets are different from those in the header processing facet. Interactions between these facets occur immediately outside of the header processing facet.

If the routing application also performs intrusion detection searches, then certain packet bodies might need to be fetched for further processing before transmission. In the SNORT network intrusion detection system [110], packet headers are first examined to identify potentially suspicious flows before expensive pattern matching is performed on the bodies. This adds another point of interaction between the header processing and body processing facets. This interaction point occurs in the middle of the header processing facet, unlike the previous interaction points that are found immediately before and after the header processing facet. This is a more challenging design problem because there are more complex ways that the flow of control moves between the two facets.

Designers must be able to correctly model the concurrency here. Header processing and body processing each have their own process-level concurrency. The interaction between these facets is the source of more process-level concurrency. How do the facets communicate and execute relative to each other? Are there separate processes in each facet, or does a given flow of control move between facets? Can multiple packet bodies be stored or fetched simultaneously? Does header processing block while body processing is done, or can these be done concurrently?

Clearly, network processing is a heterogeneous application domain. The applications are multi-faceted. Within each facet, concurrency is found on multiple levels of granularity. Within each level of granularity there are multiple variations. Different facets have different variations. To top it off, there are cross-cutting issues that affect multiple facets.

Because of this diversity, network processing will make an ideal testbed for the Cairn approach. Obtaining high-performance, low-cost implementations of these applications will show that Cairn is capable of handling the general problems of heterogeneous concurrency.

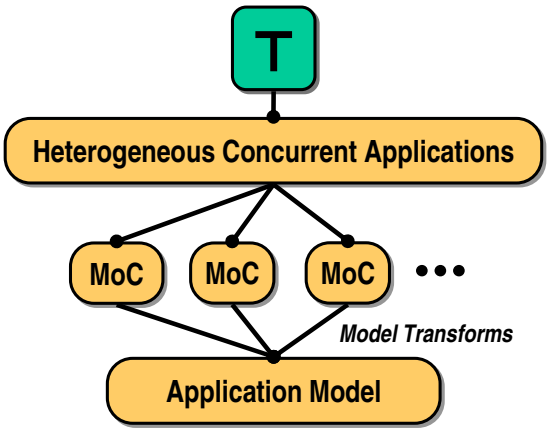


Figure 5.3: Models of Computation Above an Underlying Application Model

5.2 Definition and Goals of an Application Abstraction

The previous examples of heterogeneous concurrency are inherent in the application and are completely architecture-independent. The role of the application abstraction in the Cairn methodology is to provide a formalism that captures these characteristics precisely, without regard to implementation. This sets up an important separation of concerns between application programmers and architects. Domain experts focus on issues that are important to the application domain. Designers can then begin the implementation process with a precise model of the application's requirements. These are the features that, in an ideal world, the architecture should directly support for maximum performance. Later, the application's ideal requirements will be compared against the architecture's actual capabilities to characterize the concurrency implementation gap.

5.2.1 Application Abstraction Characteristics

Based on the complexities found in the application described previously, the following criteria are critical for an application abstraction:

- *Capture the Concurrency Found Across Multiple High-Level Models of Computation:* An application abstraction must provide a formalism for heterogeneous concurrency. Unfortunately, it is not possible to provide the generality required to capture the multiple styles of concurrency found in multi-faceted applications and still provide a high-level abstraction that assists and restricts designers. These are contradictory goals. Domain experts require narrow, targeted abstractions for high productivity.

Therefore, programmers should not design their applications using the application abstraction directly. Instead, they use high-level abstractions based on models of computation to build application facets [75]. Models of computation are found directly above the application model in the design abstraction chart (Figure 5.3). Models of computation enforce certain styles of process-level, data-level, and datatype-level concurrency to simultaneously assist and restrict the programmer. A model of computation expresses a subset of what can be expressed in the underlying application model.

Programmers use different models of computation for different application facets and to model the interactions between facets. The application abstraction provides an formalism that gathers the diverse concurrency requirements together in a common form.

- *Support Multiple Application Domains:* Application domains are distinguished by the combinations of models of computation that they use. The application abstraction should therefore support a diversity of models of computation, and not just those found in the network processing application domain. A sound theory must be provided for the composition of different models of computation to build a multi-faceted application.

- *Plan Ahead for a Disciplined Mapping Methodology:* The third core requirement (Section 2.1.3) states that a disciplined mapping methodology must be used to cross the concurrency implementation gap. In that step, the application's requirements will be compared against the architecture's capabilities to characterize the gap.

  The reason for gathering the application's diverse concurrency requirements together in a common form is to prepare for the mapping step. This ensures that the requirements of individual application facets and the interactions between facets will all be correctly considered in the mapping process.

  A lesson learned from the system-in-the-large approach is that when the application exists as a disjoint set of specifications in different languages, the interactions between facets are difficult to implement correctly. This slows down the deployment process, permitting fewer design space exploration iterations to be performed. A consistent model of the entire application avoids this dilemma.

- *Make a Precise Specification:* An application abstraction must accurately represent the application's requirements, without introducing artefactual requirements. Artefactual requirements appear when programmers cannot express exactly what they want in the language.

For example, consider trying to express datatype-level concurrency in the Java language. Java provides standard 32-bit 2's complement integers. The programmer may desire fixed-point numbers with 12 integer bits and 10 fraction bits. A Java integer can be used to store the fixed-point value and functions can be written to perform fixed-point arithmetic.

However, information about the application's true requirements is lost when this is done. Some of the Java code is an artifact of the programmer's attempt to express computation that cannot be natively expressed in the language. It is not possible to analyze a Java program after the fact and determine that the application really wants fixed-point arithmetic and not 2's complement integer arithmetic. This lack of precision closes the door on an important optimization opportunity. Implementors have to assume that the application requires 32-bit 2's complement arithmetic.

Additional complexities in the Java language make this problem even worse. Java provides a stack, heap, and automatic garbage collection. Java threads use a model of process-level concurrency based on `synchronized` critical sections, `wait` and `notifyAll` function calls. These features may or may not match any of the application facets. In any case, implementors must provide a full Java virtual machine to run the application.

The trade-off here is between succinctness and precision. A fully-featured application abstractions allows designers to express complex programs in a short form. The cost is that the implementation must support all of the features. In the Cairn methodology, programmers will not use the application abstraction directly, so succinctness is not a concern. Models of computation built on top of the application abstraction provide for intuitive programming and deal with complexity. The application abstraction should therefore provide a minimal set of features that can express concurrent computations precisely.

- *Consider Simple Architectures:* The degree to which an application abstraction should be precise is dependent on the characteristics of the family of target architectures. The application abstraction should describe concurrency at a level of detail greater than or equal to what can be implemented by the target architecture.

  In the previous chapter, the second core requirement (Section 2.1.2) motivated Sub-RISC multiprocessors as target architectures. Individual PEs compute mathematical projections and update state using explicit bit-level data types. Data-level concurrency is also described explicitly. Process-level concurrency is based on a simple model of passing the flow of control

from one PE to another using signal-with-data messages.

To match these simple architectures, the application abstraction should be similarly explicit about datatype-level, data-level, and process-level concurrency. For Sub-RISC machines, this means breaking down the application's process-level concurrency into basic tasks that communicate by passing the flow of control between themselves. Each task should express data-level and datatype-level concurrency explicitly.

Architects are interested in knowing these details precisely because Sub-RISC processors can be correspondingly customized to benefit the application. Detailed application models allow designers to make careful characterizations of the concurrency implementation gap, which in turn helps them make intelligent design space exploration choices.

5.2.2 Application Abstractions vs. Programming Models

It is important to reiterate the difference between an application abstraction and a programming model. In Section 2.1.2, a programming model was identified as an architectural abstraction that tries to balance visibility into the architectural features that are necessary for performance while maintaining opacity toward the minutiae that sap productivity. An application abstraction is not concerned with the architecture. Its focus is on capturing the application's inherent requirements.

5.2.3 Application Abstractions vs. Domain Specific Languages

In the Cairn methodology, domain specific languages play the same role as models of computation. Both of these things are tools for modeling specific application facets or combinations of application facets. An application abstraction is a more general abstraction that supports multiple models of computation or domain specific languages.

The difference between a DSL and a model of computation is a matter of packaging. A model of computation is a formalism that describes how components in a system interact. A DSL is a syntax and grammar that makes it easy to use such a formalism.

Sometimes the term "domain specific language" is applied to entire methodologies that combine a language with implementation tools. For example, the entire Click package from MIT is referred to as a domain specific language when in fact the language is just one component of the package. The package also contains a library of elements and software tools that make up an implementation process that goes from the language to a Linux workstation target architecture. It is

important to distinguish the Click language and model of computation from the processes that surround them. The tools that come with the Click package provide one way to take the Click language to implementation on a certain architecture. Other processes and abstractions can be provided to go from the same language to different architectures.

5.3 Pitfalls Of Using Existing Parallel Programming Languages as Application Abstractions

Hundreds of parallel programming languages have been introduced in the literature. All of them were designed to express some form of concurrency, whether it was motivated by a particular application or a particular multiprocessor platform. Any language that provides a mechanism to represent concurrency is a potential candidate for use as an application abstraction. However, existing languages do not measure up well against the criteria given previously.

There are too many projects to discuss each one in detail. Instead, the three most common pitfalls will be presented with examples of languages that exhibit each problem. For detailed surveys of parallel programming languages, the reader is directed to publications from Bal, Steiner and Tanenbaum [6], or Skillicorn and Talia [109].

- *Lack of Support for Multiple Granularities of Concurrency:* The most common problem with existing parallel programming languages is neglect for data-level and datatype-level concurrency. For researchers looking to execute programs on supercomputers composed of RISC-like processing elements it is easy to overlook fine-grained concurrency. The processing elements are not capable of implementing significant data-level or datatype-level concurrency, so it appears that nothing is gained by including it explicitly in the application model.

  Languages that suffer from this problem include those that start with a conventional sequential programming language and add a library or special syntax for threads and communication. The Message Passing Interface (MPI) [86] and Parallel Virtual Machine (PVM) [40] systems provide function calls for synchronous and asynchronous message passing between MIMD-style processes. Each process is written in C, C++, or FORTRAN. Users can express no more intra-process concurrency than what can be expressed in these sequential languages.

  Languages based on a SPMD (Single Program, Multiple Data) model do a better job at expressing data-level concurrency. OpenMP [20] allows users to annotate C, C++ and FORTRAN loops with pragmas to indicate opportunities for data-level concurrency. The run time

system dynamically decides when to execute separate loop iterations in parallel. Titanium provides similar syntax constructs on top of Java [123]. Jade allows users to annotate the variables read and written by a loop iteration so that the run time system can make better dynamic decisions on how to perform loops in parallel [100].

Explicit datatype-level concurrency is typically only found in hardware description languages such as VHDL, Verilog, or Tensilica's TIE language. In these cases, there is no useful abstraction of process-level concurrency.

By failing to consider multiple granularities of concurrency, languages in this category make it impossible to make a precise specification of the application's requirements. This precision is necessary for targeting the Sub-RISC processors considered in this dissertation.

- *Lack of Support for Implementation-Independent Modeling:* Some parallel programming languages are designed to facilitate programming a particular multiprocessor architecture rather than to capture application-level concerns in an implementation-independent fashion. For this reason, these abstractions are better classified as "architecture abstractions in disguise" instead of as application abstractions.

  An example of a programming language in this category is IXP Microengine C [25]. When declaring variables in this language, programmers must include compiler intrinsics that indicate where the variable is to be allocated in the IXP's heterogeneous memory architecture. This is an implementation decision that is intermixed with functional application requirements.

  In such a language, programmers are forced to oscillate between describing application concerns and implementation concerns. This slows down design space exploration. If an implementation is found to have bugs, developers must spend time determining whether the bugs are due to errors in the application functionality or problems in the mapping.

  Maintaining a separation of concerns between application functionality and mapping decisions is necessary to avoid this problem. Developers need the ability to verify that the application is functionally correct before mapping it to an architecture. Then, when performance comes up short or errors are found, the correct Y-chart feedback paths can be taken.

- *Lack of Support for Compositions of Models of Concurrency:* The final and most critical pitfall is poor support for multifaceted applications. Compositions of models of concurrency are

necessary to represent the various concurrency requirements found across different application facets and their interactions.

Often, a language will provide one specific semantics for communication and control which is suitable for some application facets but is not extendible for other application facets. For example, the Click language provides a model of process-level concurrency which is ideal for packet header processing, but is not suitable for packet body processing. Likewise, languages based on models such as Kahn Process Networks [68], Communicating Sequential Processes [52], synchronous reactivity (e.g. Esterel [11]), state machines (e.g. Statecharts [49], or dataflow (e.g. StreamIt [118] or Lustre [17]) are all useful for certain situations. In certain cases, one can even choose a single model of concurrency, such as CSP's rendezvous semantics, and show that the other models can be built on top of it. However, this is often awkward is practice. The simplest thing to do is to use the right tool for each job.

Due to these pitfalls, existing parallel programming languages are not suitable for use as Cairn's application abstraction. Cairn requires an application abstraction that is designed specifically to support multiple styles and granularities of concurrency. One existing project that comes closest to meeting these requirements is the Ptolemy II system. This will be discussed in detail in the next section.

5.4 Ptolemy II

Ptolemy II is a system for modeling heterogeneous concurrent applications using compositions of models of computation [76]. Ptolemy accomplishes this by employing a multiple abstraction approach. The Ptolemy Kernel is a base application abstraction for various models of computation. Each model of computation restricts and assists programmers by allowing them to express a style of concurrency that is a subset of what can be expressed in the underlying Ptolemy Kernel. This structure is shown in the design abstraction chart in Figure 5.4.

The application half of the Cairn methodology is directly inspired by the Ptolemy II system. Cairn's application abstraction plays the same role in the design methodology as the Ptolemy Kernel. In contrast to Ptolemy, Cairn seeks to target implementation on programmable Sub-RISC multiprocessors instead of targeting simulation on Java-based systems. Thus, the two methodologies diverge below the application abstraction in their respective design abstraction charts.

While the Ptolemy Kernel cannot be used directly as Cairn's application abstraction, it still

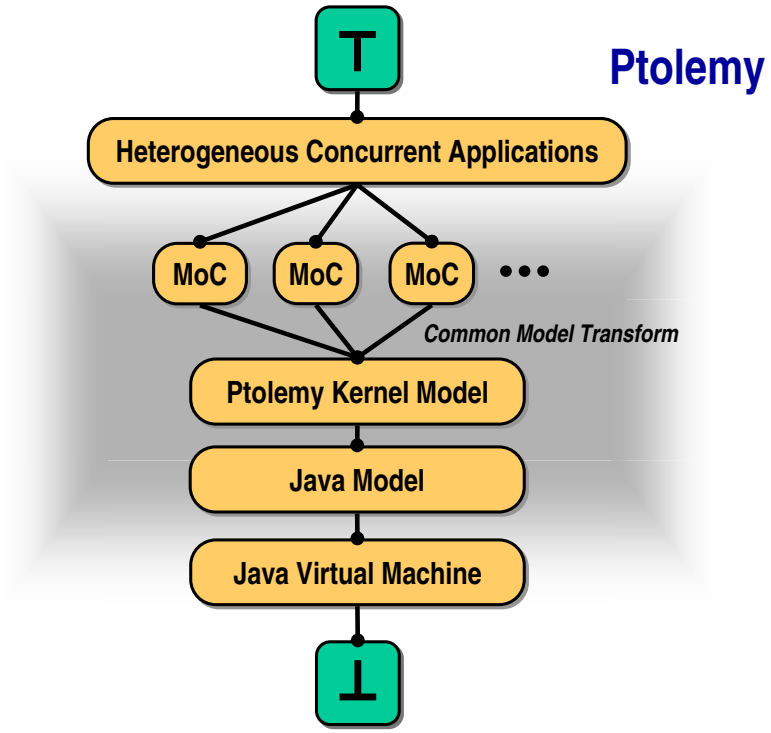


Figure 5.4: Ptolemy II Design Abstraction Chart

provides critical insight into how the Cairn application abstraction should be built. This section describes how the Ptolemy system works and compares the pros and cons of this approach against the application abstraction criteria given in Section 5.2.1. Possible improvements to Ptolemy to make it better match the criteria are given. These lessons guide the creation of a new application abstraction for use in the Cairn methodology, which will be detailed in the following section.

5.4.1 The Ptolemy Kernel

Ptolemy models of computation employ a separation of concerns to provide abstractions to programmers. Computation, which is encapsulated in components called *actors*, is separate from communication (how actors exchange data) and control (how actors execute relative to each other). To create an application model, programmers simply create graphs of actors. Their focus is solely on the computations the application should perform.

Programmers declare what model of computation they wish to use by adding an annotation to the actor graph. The model of computation provides a semantics for communication between actors along edges in the actor graph, as well as a semantics for control that dictates how and when actors

execute. Aside from this annotation, Ptolemy application graphs do not contain any objects that implement the control or communication semantics. These things are purposefully abstracted away by the model of computation.

In the next lower level of Ptolemy's design abstraction chart these abstractions are made explicit. Represented in the Ptolemy Kernel, an application is a graph that consists of three different components: Actors encapsulate computation as before. *Receivers* implement communication channels between actors. *Directors* govern how actors execute. The extra receiver and director components provide concrete implementations of the abstract semantics in the higher-level actor graph.

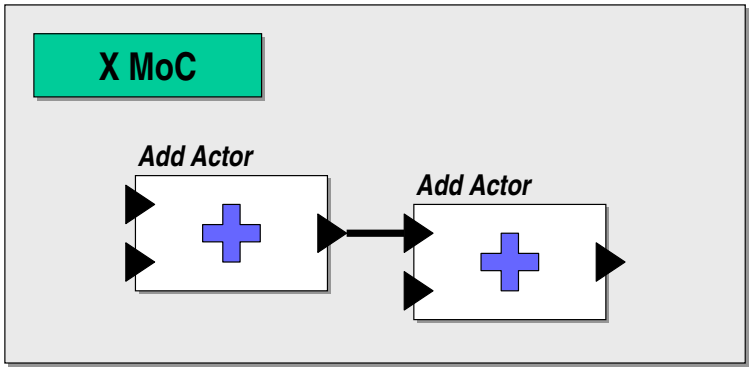
The semantics of communication between components in a Ptolemy Kernel graph is Java function calls. Actors call functions on receivers to transfer data. Directors call functions on actors to invoke computation. The Ptolemy Kernel model *refines* the abstract communication and control semantics of the higher-level actor graph into a concrete Java program.

Ptolemy provides a model transform process that converts an actor graph (with a model of computation annotation) into a Ptolemy Kernel graph that contains receivers and directors in addition to actors. This process is the same for all models of computation. Figure 5.5 demonstrates what is done. Figure 5.5(a) shows the original high-level actor graph and Figure 5.5(b) shows the corresponding Ptolemy Kernel graph. The actors from the original model are carried over into the transformed model unchanged. For every edge in the original actor graph, a receiver object is instantiated in the transformed model. Actors that were connected by the edge are now connected to each other via the receiver object. Finally, a single director object is instantiated and connected to every actor.

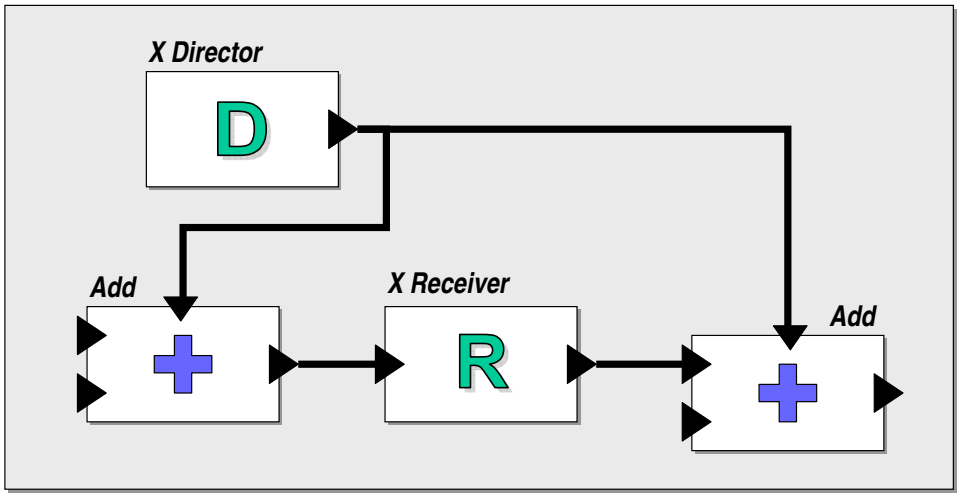
Each model of computation defines its own receiver and director. The model transform chooses the right objects to instantiate by looking at the annotation on the original actor graph.

The receiver implements the communication semantics of the model of computation, while the director controls how and when actors execute. For example, in Communicating Sequential Processes, actors are processes that communicate through mailboxes with rendezvous semantics. The director's job is simply to cause all actors to execute in parallel. The receiver encapsulates storage for a single datum, and exports blocking functions for reading and writing to this storage location.

Different models of computation provide different implementations of the director and receiver. In Kahn Process Networks, actors are processes that communicate through infinite-sized queues. In this case the behavior of the director is the same (it causes all actors to execute in parallel), but the



(a) Original Actor Graph with Model of Computation Annotation



(b) Transformed Ptolemy Kernel Model

Figure 5.5: Ptolemy Model Transform Process

receiver is different. It encapsulates a theoretically unbounded FIFO and exports a non-blocking write function and a blocking read function.

In the Synchronous Dataflow model of computation, the director causes the actors to fire according to a static schedule that is computed at the time of the model transform. The receivers provide storage for a finite number of datums based on the relative production and consumption rates of the connected pair of actors, and the times when the actors are scheduled to execute relative to each other by the director.

Adding a new model of computation to Ptolemy involves designing a new director and receiver. This modular software architecture and the strong separation of concerns of computation, communication, and control are what allows the Ptolemy Kernel to serve as a foundation for arbitrary models of computation.

5.4.2 Pros and Cons of the Ptolemy II Approach

The benefits of the Ptolemy approach that the Cairn application abstraction seeks to preserve are as follows:

- *Actor-Oriented Design:* The encapsulation of computation within actors allows programmers to build concurrent applications quickly and easily. Programmers focus on what computations their application needs to perform. To make a concurrent composition, one simply instantiates and connects actors. A model of computation takes care of the rest.

  Occasionally, programmers need to create new actors. Ptolemy's strong separation of concerns and modular design principles make this simple. Interactions between actors are orthogonal to the behavior defined within an actor. Actor designers do not need to decide how the actor will interact with other actors.

- *Domain Polymorphism:* Since the computations within an actor are separate from the interactions between actors, actors are agnostic to the model of computation in which they are used. This property is called *domain polymorphism*. It allows a single actor to be reused in multiple contexts without necessitating any code changes.

  This is possible because in the Ptolemy Kernel, all actors, directors, and receivers implement consistent Java interfaces. An actor can connect to any type of receiver using the same set of function calls. How the receiver implements these functions decides what the communication semantics will be. Similarly, actors can connect to any type of director using a different set of

function calls. When the Ptolemy model transform process converts a high-level application model into a Ptolemy Kernel model, no changes to the actors themselves are necessary. The transform simply plugs in the correct types of receivers and directors.

- *Data Polymorphism:* Another feature that simplifies the construction of actor-oriented models is data polymorphism. Actors are written to perform computations on abstract data types called *tokens*. This allows the same actor to be reused in different contexts to work on different data types.

  A standard type resolution algorithm is used to elaborate the types within a model at the time the model transform process is applied. Programmers only need to specify type details at select places within their models. The rest of the type information can be inferred automatically.

- *Parameterized Models:* In Ptolemy, both actors and compositions of actors can include configurable parameters. This allows programmers to create models whose behavior can be tuned by changing the values of the parameters rather than redesigning the entire model. For example, a filtering application may expose the filter tap coefficients as adjustable parameters. The model describes a generic filter. The exact behavior of the application depends on the static or dynamic values of these parameters.

  Neuendorffer's work on actor-oriented metaprogramming provides a mathematical theory for the behavior of models that undergo dynamic reconfiguration through the changing of parameters [90]. The Ptalon project seeks to extend the concept of parameterization to the level of full higher-order modeling [18]. In that work, programmers use a composition language to quickly and easily describe complex compositions of actors.

- *Hierarchical Heterogeneity:* Ptolemy allows programmers to design multifaceted applications by making hierarchical models. Different branches of the model are allowed to use different models of computation. This is called *hierarchical heterogeneity*.

  The key to making this work is the concept that a composition of actors, together with a model of computation, forms an encapsulated unit that can be used as an actor in another model. Such an encapsulation is called an *opaque composite actor*.

The Ptolemy approach is not without problems. Users who explore the inner workings of Ptolemy discover that the encapsulations and separations of concerns described in this section are

not always completely implemented. Instead, Ptolemy makes software engineering trade-offs to balance the goal of implementing interesting theoretical concepts with the goal of producing a practical software tool. The downsides to the Ptolemy approach that Cairn seeks to remedy are as follows:

- *Poor Support for Multiple Granularities of Concurrency:* Ptolemy models of computation focus on providing abstractions for process-level concurrency. Computation within a process (i.e. inside of an actor) is described by writing a Java function. This makes it impossible to precisely describe data-level and datatype-level concurrency.

  For simulation on a workstation computer running a Java virtual machine, a higher level of precision is not necessary. The JVM cannot take advantage of information about bit-level computations other than standard integer and floating-point arithmetic, nor can it exploit more instruction-level parallelism than what is available from the underlying processor.

  When Sub-RISC multiprocessors are considered as implementation targets, these details are necessary. Therefore a language other than Java should be used to express computation within actors.

- *Poor Separation of Run-Time and Design-Time Behaviors:* The Ptolemy Kernel separates computation, communication, and control into actor, receiver, and director components respectively. These components are supposed to interact only through formal, well-defined interfaces. The idea is that if these components and interfaces are implemented correctly, then the application will behave in the expected way. Unfortunately, this system architecture is merely a convention on top of the Java language. Java itself cannot force programmers to follow the rules.

  In practice, the goal of making a practical simulation and modeling environment trumps the goal of making a formal model of the application's requirements. Programmers often use arbitrary features of the Java language to bypass the formal architecture of the Ptolemy Kernel. The result is that some of the application's functionality is not precisely captured by the Ptolemy Kernel.

  Consider Ptolemy's Synchronous Dataflow (SDF) director as an example. According to the control semantics of Synchronous Dataflow, the director is supposed to fire all of the actors in the model according to a statically computed schedule. The SDF director contains code to compute this schedule by analyzing the actor graph. It traverses the graph by accessing the underlying Java objects, following references between actors, ports and relations.

This is a form of communication between the director and the actors that is not the same as the semantics of SDF. SDF states that at run time, the director simply sends messages to the actors to cause them to fire. SDF does not say anything about the director walking the graph. In fact, the application does not require run time knowledge of the graph structure at all. The static schedule should be computed at design time and hard-coded into the director.

Since the Ptolemy Kernel is implemented in Java, both the "SDF form" and the "schedule computing form" of director-actor communications look like normal Java function calls. They are not distinguishable. Therefore, implementors must assume that both types of communication are required at run time.

This is another way that Java fails to make a precise representation of the application's requirements. Run-time behaviors that are necessary for implementation are intermixed with design-time behaviors that are necessary to have a practical modeling environment. The result is that artefactual requirements sneak into the implementation.

A better application abstraction will enforce a strong separation of run-time and design-time behaviors. If programmers intend for their application to dynamically recompute SDF schedules, then this behavior should be modeled explicitly using models of computation. This guarantees that complexity will only be incurred in the implementation when the application requires it.

- *Does Not Consider Implementation on Simple Architectures:* The fundamental behavior that the Ptolemy Kernel uses is the function call. Directors call the fire methods of actors, and the thread of control returns to the director when the fire method completes. Actors call `put` and `get` methods on receivers to transmit and receive tokens. Even when these functions return `void`, the full semantics of function calls is necessary to model the application's behavior correctly. The `put` call does not return a value, but the receiver may purposefully block an actor by not returning immediately.

  Function calls are a good match for the process-level concurrency of RISC-like processing elements, but they are not a good match for Sub-RISC elements. RISC processors have program counters and can perform jumps or branches in parallel with normal ALU computations. Sub-RISC PEs are simpler than this. Instead of calling functions and waiting for a return value, Sub-RISC machines pass the flow of control from one PE to another by sending signal-with-data messages. There is no return value associated with this event.

Sub-RISC PEs can mimic the behavior of function calls by sending signal-with-data messages to themselves, but to rely on this as the lowest-level behavior is to ignore an interesting region of the Sub-RISC design space. Sub-RISC PEs can be customized to support styles of process-level concurrency different from ordinary function calls.

Therefore, the Cairn application abstraction should use a primitive other than function calls to model application behaviors. The right primitive is comparable to a Sub-RISC PE's basic behavior: a simple transfer of the flow of control from one computation to the next. This maximizes the opportunities for customization with the goal of creating a high-performance architecture that exactly matches the application's requirements.

5.4.3 Strengthening the Ptolemy II Abstraction

Ongoing works address the shortcomings of Ptolemy II. These projects provide further insights into how the Cairn application abstraction should be structured.

The Cal Actor Description Language

The Cal actor description language changes the way that computation within actors is described [36]. Instead of describing computation using an arbitrary Java function, Cal actors contain explicit firing rules called *actions*. Actions represent the set of computations an actor can perform.

An action describes three things: what tokens the actor consumes, what tokens the actor produces, and how the state of the actor changes. Declaring the consumed tokens is a way of stating a precondition for the firing of an actor. Similarly, declaring the produced tokens forms a postcondition that is true after firing the actor. An additional *guard expression* can condition the firing of an action on the values of the tokens that are to be consumed.

Declaring token consumption and production explicitly allows for the formal analysis of a composition of actors. More information about the behavior of a composition can be determined at design time, opening the possibility for more optimizations. Dataflow applications are one of Cal's primary concerns. Explicit information about token consumption and production enables the calculation of an efficient schedule.

Another use for explicit firing rules is to determine if an actor is compatible with a particular model of computation or in a particular context. For example, an actor may have a single firing rule that requires tokens from all input ports in order to execute. In the Click model of computation, such an actor is only compatible in a *pull* context. In order to perform the action the actor must first

pull packets from every input port. If the actor has been placed in a *push* context, then the firing rule identifies the situation as a semantic error. The Click model of computation does not have a concept of merging multiple *push* threads of control before firing an actor.

One shortcoming of Cal is that the computation defined within an action still does not precisely describe data-level and datatype-level concurrency. Since the description of computation is orthogonal to a firing rule's token production and consumption declarations, this problem could be easily solved by replacing the computation language.

Copernicus

The Copernicus project seeks to optimize Ptolemy applications by identifying and removing code that is part of the design-time behavior of the application from the code that the application requires at run-time [90]. This is done through *a posteriori* analysis of Java bytecodes. This technique is useful for supporting legacy Ptolemy applications, but it is not as effective as starting with an application abstraction that makes a more precise declaration of the application's requirements.

Four optimization techniques are employed. If the exact token types an application uses can be statically determined, then data polymorphic actors can be optimized to remove the layer of indirection supplied by Ptolemy's Token class. This is *type specialization*. Similarly, *domain specialization* replaces the domain polymorphic interfaces between actors, directors and receivers with direct function calls that avoid unnecessary levels of indirection.

Third, *parameter specialization* propagates constant parameter values into the code within actors. Fourth, *structural specialization* replaces code that traverses the model hierarchy with lookups into static data structures. This does not address the problem described earlier about separating run-time and design-time behaviors. Components can still communicate via arbitrary function calls using references that are found in the data structures. The static data structures are simply meant to be faster to access than Ptolemy's normal dynamic data structures.

These techniques result in decreased code size, faster execution time, and smaller memory requirements. The lesson learned from this work is that more complex model transforms than Ptolemy's default model transform are important for obtaining performance in addition to functional correctness when implementing high-level application models.

Formal Metamodeling

Domain specific abstractions are built by hand on top of the Ptolemy Kernel by writing new Java code. Metamodeling is a technique for creating new abstractions in a declarative fashion. The GME [74] and AToM^3 [29] projects are examples of tools that allow this.

A metamodel is a model of a design abstraction. An analogy is the relationship between a grammar and a language. A grammar specifies valid constructions in a language. Likewise, a metamodel specifies valid constructions in a design abstraction.

A metamodel for the Click language would declare that an application model consists of elements, ports, parameters, and relations. It would declare the valid ways to assemble these components. For example, elements *contain* ports which are *connected* via relations. Ports *contain* a parameter that indicates their type: *push*, *pull*, or *agnostic*. Relations can *connect* ports to ports of the same type or to agnostic ports. It is incorrect to connect a push port to a pull port via a relation. These and other declarations form a complete grammar for Click.

GME and AToM^3 provide graphical environments for creating these declarations. Thus a metamodel is itself a graphical model. Note that this usage of the term "metamodel" is different from that used by the Metropolis project (Section 3.3.3).

The utility of metamodeling is that one can build new application-specific abstractions on top of a core application abstraction declaratively instead of writing imperative Java code. Tools and processes for building models using the new application-specific abstraction can be generated automatically from the metamodel. An application abstraction that incorporates these ideas will be easier to extend with additional models of computation and domain specific languages.

5.5 The Cairn Application Abstraction

The Cairn project defines a new application abstraction that addresses the shortcomings of previous work. The architecture of the Ptolemy II system is used as a basis: A Cairn Kernel will serve to capture the heterogeneous concurrency found in multifaceted applications. Domain experts describe their applications using higher-level abstractions built on top of the kernel, such as models of computation and domain-specific languages. Model transform processes will be employed to convert models from the high-level abstractions to the kernel abstraction.

Cairn's application abstraction uses a new actor description language that incorporates some of the elements of the Cal language. Actors are described by writing declarative firing rules. This

allows model transforms to make inferences about the behavior of an actor within a composition. The model transform can determine if the contextual usage of the actor is correct, and if the actor is compatible with the given model of computation. The firing rules also allow model transforms to make scheduling decisions or otherwise analyze the process-level concurrency of the application.

The computation within a firing rule is written using a language that captures data-level and datatype-level concurrency. This plans ahead for implementing actor computations on Sub-RISC processors, solving one of the shortcomings of Cal.

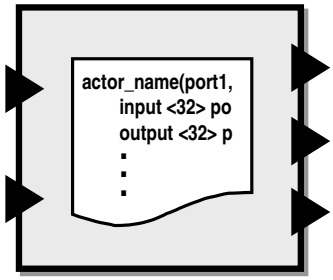
Similarly, the Cairn Kernel abstraction captures process-level concurrency in a way that is suitable for Sub-RISC multiprocessors. Instead of specifying the interaction between components using function calls, the Cairn Kernel uses the explicit transfer of the flow of control as the communication primitive. Control transfers are a match for Sub-RISC's signal-with-data interconnection scheme.

Cairn does not follow Ptolemy's approach of using a common model transform for all models of computation. Instead, different model transforms are used for different models of computation. This provides more opportunities for optimization. The control and communication semantics of a particular model of computation no longer need to be implemented within single director and receiver components. The model transforms can be more generic in what changes they make to the model and what components are added to the model.

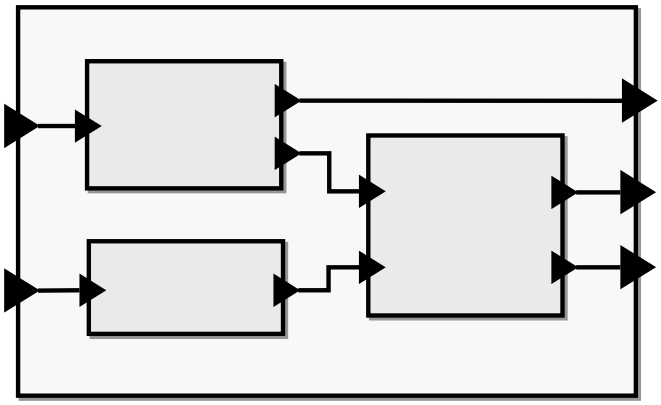
Multiple model transforms also enable support for high-level abstractions with different graphical syntaxes. All Ptolemy models look essentially the same: actors, ports and relations. Domain experts may desire a text language with a specialized syntax, or graphical models with different kinds of components and connections. An example is a state machine diagram, which lacks actors and ports and requires annotations on relations. Another example is Click, which has multiple varieties of ports and contextual restrictions on how many relations can connect to a port. When a separate model transform can be created for each high-level abstraction, Cairn can support whatever modeling and visualization techniques work best for each application domain. This dissertation does not cover the use of formal metamodels as a mechanism for designing high-level abstractions on top of the Cairn Kernel, although there is nothing that precludes this possibility.

5.5.1 The Cairn Actor Description Language

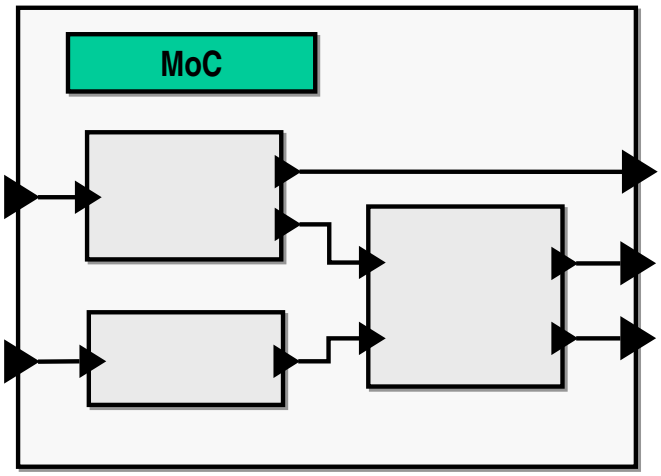
There are three ways that Cairn actors can be defined. The primary mechanism is to write a description using the Cairn actor description language, which is described in this section (Figure 5.6(a)).



(a) ADL-Based Actor



(b) Modular Actor



(c) Opaque Actor

Figure 5.6: Three Ways to Describe Cairn Actors

The second option (Figure 5.6(b)) is to create a composite actor that is defined as a hierarchical composition of existing actors. This is a simple application of modularity. The modular actor can be placed in the actor library and instantiated multiple times in an application model to avoid the repetitive task of creating the same grouping of actors many times.

The third option, shown in Figure 5.6(c), is the equivalent of Ptolemy II's opaque composite actor. This is a hierarchical composition of existing actors that includes a model of computation annotation. Whereas the normal modular actor is simply a grouping of components, the opaque actor has a defined behavior and appears as an atomic component in a composition. Cairn uses opaque actors to model hierarchical heterogeneity.

The Cairn actor description language is intended to be used by engineers who create domain-specific abstractions. These designers create domain-specific or domain-polymorphic actors to populate an actor library that will be used by domain experts to create applications. Normally, domain experts will create modular and opaque actors for themselves, but occasionally they may need to extend the actor library by writing new actors with the actor description language.

A Cairn actor description is a declarative specification that contains four parts: parameters, ports, state, and firing rules. An example of the syntax is shown in Figure 5.7. The description begins with the *actor* keyword and the name of the actor. The four declarative sections are contained within curly braces.

Parameter Declarations

Parameters are exclusively for the design-time configuration of actors. All parameters in a model must be statically elaborated before implementation. A parameter can have a numeric value or a character string value. The *type_expression* following the *parameter* keyword can either be an integer indicating the bit width of the parameter or the *string* keyword. Default value assignments are optional and use Verilog's syntax for writing integers.

Port Declarations

Ports make up an actor's interface for exchanging data with other actors at run time. Input ports receive data from data sources. Output ports transmit data to data sinks. A type expression indicates the width of the port in bits. Declaring the type of a port is optional. Omitting the type specification is a way to make a data polymorphic actor. Unknown types will be determined by a type resolution algorithm at design time.

```
actor actor_name {
        // Parameter declarations
        parameter <type_expression> param1 = default_value;
        parameter <string> port1_click_type = "push";
        parameter <10> param2 = 10'h2a;

        // Port declarations
        input <type_expression> port1;
        input <type_expression> port2;
        output <port1.type> port3;
        output <(port1.type > port2.type) ? port1.type : port2.type> port4;

        // State declarations
        state <type_expression> reg1[type_expression];

        // Firing rules
        rule fire(port1[quantity], port2[quantity], port3[quantity], port4[quantity]) {
                define macro1 = port1 + port2 + param2;
                define macro2 = port1 - port2 - param1;

                reg1[port1] += macro1;
                port3 = macro1;
                port4 = macro2;

                break;

                reg1[port2] = (reg1[port2] > macro1) ? macro2 : 10'h0;

        };

        rule initialize( ) {
                generate (i = 0; i < type_expression; i++) {
                        reg1[$i] = 0;
                };
        };

}
```

Figure 5.7: Cairn Actor Example

A port can be declared to have the same type as another port. For example, $port_3$ in the figure is specified to have the same width as $port_1$. Such a type declaration tells the type resolution algorithm how to propagate types through actors. A type expression can also be an in-line mathematical function written with C operators. For example, the width of $port_4$ is equal to the width of $port_1$ or $port_2$, whichever is larger. This is useful for describing arithmetic actors that combine datums of unequal types. These expressions are evaluated at design time by the type resolution algorithm.

State Declarations

The third section of the actor description declares the actor's internal state. Actor state is not visible to other actors in a composition. It can only be read and written by the computations found within firing rules. State variables are preserved across firings of an actor. The bit width of each state variable is declared using a type expression in the same manner as ports. Optionally, a second type expression in square brackets after the state name declares the depth of the state. This defines a state array or a memory.

Firing Rule Declarations

The final entries in the actor description are firing rules. Each firing rule represents a valid behavior of the actor. The beginning of a firing rule lists the ports where the rule produces and consumes tokens. Optional *quantity* parameters indicate how many tokens are produced or consumed. Omitting this field implies a default value of one.

Unlike the Cal language, there are no guard expressions that condition the firing rule on the values of the tokens. Token values are usually only known at run time, therefore guard expressions imply that the actor performs conditional computation. A basic Sub-RISC processor does not have control logic for conditional computation. Thus, the Cairn actor description language does not include guards as a core language feature. Conditional computation must be specified explicitly using compositions of actors. This will be demonstrated in Section 5.6.1.

Within a firing rule, computation is described using a bit-level feed-forward dataflow language based on the syntax of Verilog. This code computes next state values and output tokens as a function of current state values and input tokens. The Verilog-like syntax precisely describes datatype-level concurrency.

Each line of code describes an assignment to a state variable or output port. All assignments in the rule are guaranteed to execute atomically. No loops or recursion is permitted.

Furthermore, unlike a sequential language such as C, the assignments within a firing rule are evaluated in parallel. State reads occur before state writes. A state variable or output port may not appear as the target of an assignment more than once. This allows the data dependencies within a firing rule to be analyzed to discover all possible data-level concurrency.

The computation within a firing rule is purposefully designed to match what a Sub-RISC processing element can compute using an operation or macro operation. That is, the computation of a mathematical projection and the update of internal state. Later in the Cairn design flow, actors will be mapped to processors and firing rules will be compiled into programs. Note that the ternary operator (?:) that appears in this example is a combinational function and not an instance of conditional computation. When the firing rule executes, the condition and both branches are evaluated in parallel. The result is a simple function of these three values.

Array-type state variables are an exception to the rule that state variables may not be assigned more than once. If the target array indices can be statically determined to be different, then multiple assignment statements are allowed because the underlying state variables are different. However, the array indices may be dependent on run-time data values, making this static analysis impossible. In this case, the firing rule must explicitly sequentialize the assignments. This is done using the *break* keyword. A break statement introduces a sequential boundary between collections of parallel assignment statements. This implies new data dependencies between computations performed before and after the break. If the right-hand side of an assignment statement contains state reads, then the values read will depend on the corresponding state writes performed before the break.

A break statement does not necessarily imply that computations will be performed in separate clock cycles on the target architecture. For example, a PE may have a memory with two independent write ports and hardware that guarantees that one port has priority. This PE may be able to perform both state writes simultaneously while preserving the behavior described by the firing rule.

The *define* keyword is used to help write complex expressions in a modular fashion. This is not a variable that holds its value across sequential boundaries. It is a macro that is expanded in-line at design time.

Finally, the *generate* statement is available to simplify the description of parameterized actors. These statements are evaluated at design time to fill in the contents of a firing rule. This is not a mechanism for describing looping computation, either with static or dynamic loop bounds. The syntax is similar to a C *for* loop. In this example, the depth of the state array is dependent on the actor's context in a model. The *initialize* firing rule uses the generate statement to produce a number of assignment statements equal to the depth of the array.

5.5.2 The Cairn Kernel

Components in a high-level application model communicate using the semantics defined by a particular model of computation. Components in the underlying kernel model communicate using a different, more concrete semantics. In the case of Ptolemy II's kernel, the components are Java objects that communicate using Java function calls. These components implement the abstract semantics defined by the various models of computation that Ptolemy supports.

Cairn uses a different semantics at the kernel level that are specifically chosen to help target implementation on Sub-RISC multiprocessors. Instead of communicating with Java function calls, Cairn's components communicate by explicitly transferring the flow of control amongst themselves. These control flow transfers are a match for the signal-with-data messages that Sub-RISC multiprocessors use for communication.

The components in a Cairn Kernel model are Cairn actors described using the actor description language presented in the previous section. These actors contain firing rules that describe the computation of mathematical projections and the update of internal state. Firing rules are a match for the operations and macro operations that Sub-RISC processing elements execute.

The Transfer-Passing Abstraction

This section describes the semantics of communication between components in a Cairn Kernel model. The only communication primitive in the Cairn Kernel is the *transfer* message. Transfers are so named because they represent the passing of the flow of control from one component to another. When a component receives a transfer message, a single firing rule is invoked. This firing rule performs a computation that produces zero or more new transfer messages. These messages are communicated to other components when the firing rule terminates. The component performs no additional computation after executing the firing rule and producing transfers. Computation continues at the destination of the transfer messages where new firing rules are invoked.

A transfer consists of two parts. A *control* part consists of a single integer value that indicates what firing rule should be invoked at the destination of the transfer. Every component's firing rules are enumerated in alphabetical order at design time to support this.

The second part of a transfer is *data*. This consists of tokens that are made available on the receiving component's input ports during the execution of the given firing rule. A transfer need not provide a token for every input port. A transfer may provide more than one token for some ports.

Figure 5.8 shows how a Cairn actor is encapsulated as a transfer-passing component in the

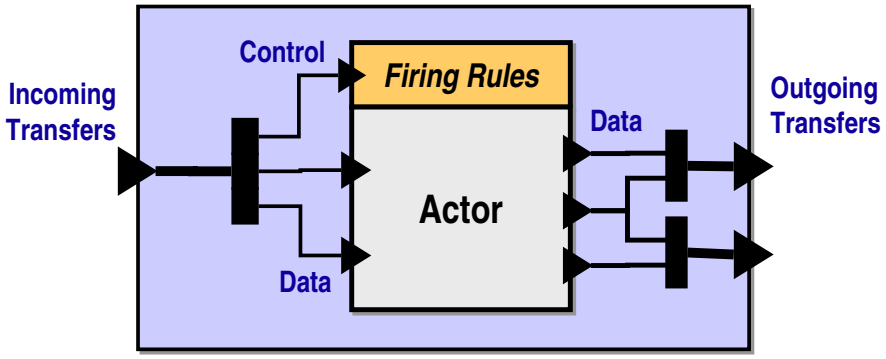


Figure 5.8: Cairn Actor Encapsulated as a Transfer-Passing Component

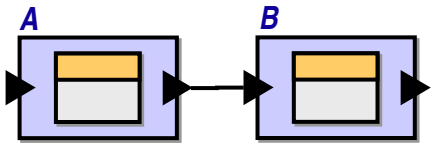


Figure 5.9: Point-to-Point Transfer Passing Connection

Cairn Kernel. Each enclosed actor exports as its external interface a set of input and output ports and a set of firing rules. The Cairn Kernel encapsulates this interface using a set of ports for sending and receiving transfers. There is exactly one port for receiving transfers. The control portion of each transfer is used to select the firing rule to execute. The data portion of each transfer is made available on the enclosed actor's input ports.

After the enclosed actor completes the desired firing rule computation, any tokens produced on output ports are bundled together to form new transfers. These transfers are sent to other Cairn Kernel components via zero or more output ports found on the encapsulating component.

Figures 5.9, 5.10, 5.11, 5.12 and 5.13 show the valid ways of making connections between Cairn Kernel components. Figure 5.9 shows the basic point-to-point connection. Component *A*

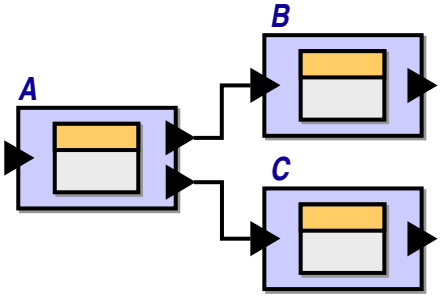


Figure 5.10: One Component Produces Multiple Transfers

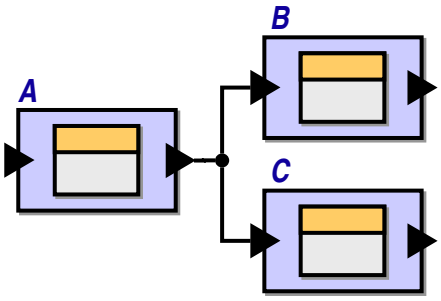


Figure 5.11: Fan-Out of a Transfer Message

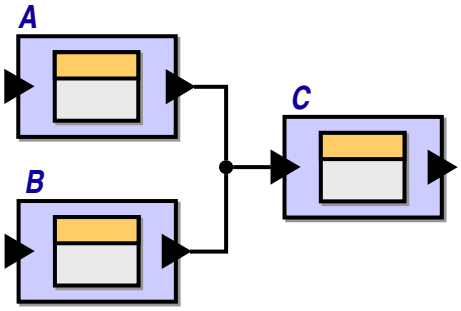


Figure 5.12: Fan-In of a Transfer Message

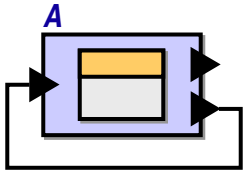


Figure 5.13: A Component Sending a Transfer to Itself

produces transfers that are consumed by component *B*.

In Figure 5.10, component A can produce transfers for components B or C. This can be done separately or together. If A produces two simultaneous outgoing transfers, this represents a fork of the flow of control. Computation continues at both B and C in parallel.

Figure 5.11 demonstrates the fan-out of transfer messages. Transfers from component A are deliberately duplicated and sent to both B and C. This is another way to fork the flow of control.

Components can receive transfers from multiple sources by making multiple connections to the transfer input port. This is shown in Figure 5.12. The semantics of the transfer-passing abstraction require that a component execute all transfers that it receives. No order is specified.

Finally, Figure 5.13 shows that it is possible to create cycles in a Cairn Kernel model. In this case component A sends transfers to itself. Deadlock does not occur in the transfer-passing abstraction because components never block while processing transfer messages.

An important application of self-transfers is to model repetitive processes. Here, component A can execute a firing rule and have the flow of control continue again at A with the same or a different firing rule.

A connection to a component's output port does not indicate that the component *always* produces transfers on that port. The component may have multiple firing rules, some of which produce outgoing transfers and some of which do not. This can be determined by looking at the declaration of each firing rule and checking if the rule produces tokens on the given port. Furthermore, even if a rule produces an outgoing transfer, the control value in the transfer may select the *noop* firing rule on the destination component. Thus computation will not necessarily continue at the destination component.

Transfers containing the *noop* firing rule are used to implement conditional computation in the Cairn Kernel. One component evaluates the condition and uses the result to compute the next firing rule that should be performed. If *noop* is chosen as the next firing rule, then the thread of control does not continue and computation stops.

The transfer-passing abstraction is a way of explicitly modeling crossovers between data and control. Data values that are computed in one component become control words for connected components.

The Cairn Kernel model is purposefully similar to the TIPI multiprocessor architecture abstraction. After all, the computations described in this model are intended to be implemented on such as machine. There is one important difference: how the two models handle the concept of time. In the architecture abstraction, the execution of a signal-with-data message by a processing

element takes a finite amount of time. Each PE can only work on one message at a time. A queue buffers incoming messages until the PE is ready to execute them. When signal-with-data messages are received simultaneously, an arbitrator ensures fair access to the message queue.

There is no concept of time in the transfer-passing abstraction. There is no notion that the actor can only work on one transfer at a time, or that simultaneously arriving transfers must be sequentialized fairly. When multiple transfers are sent to the same actor, they are processed concurrently. Each firing rule executes atomically so there is no danger of concurrent transfers corrupting the internal state of a component. Queues and arbitrators are not components in this model.

5.5.3 Model Transforms

Cairn uses model transform processes to convert application models designed using high-level models of computation into concrete Cairn Kernel models. A model transform fills in implementations for the abstract communication and control semantics defined by a model of computation. This is done by both adding new components to the application model and by making modifications to existing components.

For example, actors in a Click model communicate with abstract *push* and *pull* communication links. The Click model transform replaces these connections with transfer-passing connections. The actors on each end of the connection are modified to produce and consume transfer messages. Intermediate components may be inserted in between the actors to further implement the semantics of *push* and *pull*. The key concept is that transfers are directly implementable on Sub-RISC multiprocessors, whereas *push* and *pull* are not fundamental Sub-RISC behaviors.

Cairn model transforms are written in Java by the engineers who design models of computation and domain specific languages. Java allows these programmers to use whatever graph algorithms and supporting data structures are necessary to make the appropriate modifications to the application model. In contrast, formal metamodeling approaches like GME and AToM^3 force programmers to use graph grammars to specify model transforms. This can be clumsy for some application domains. For example, advanced dataflow domains such as Heterochronous Dataflow require that a complex scheduling problem be solved in order to elaborate and implement the application's process-level concurrency. Scheduling and other optimization problems are not easily formulated using graph grammar production rules. The flexibility of an imperative language makes things easier.

Most model transforms perform the same set of basic steps:

1. *Recurse on Contained Opaque Actors:* Cairn application models employ hierarchical hetero-

geneity. A model may contain opaque composite actors (of the form shown in Figure 5.6(c)) that use different models of computation. Different model transforms are responsible for processing these subcomponents. Therefore, the first step that model transforms perform is to identify opaque composite actors and recurse, calling the appropriate model transform for each subcomponent.

This call will make the abstract communication and control semantics of the subcomponent concrete. It will export the capabilities of the subcomponent as a set of firing rules. Thus the subcomponent exposes the same interface as any normal non-hierarchical actor in the model. The first model transform can now continue without any concern that different models of computation are used deeper in the model.

2. *Flatten Modular Actors:* Normal modular actors of the form shown in Figure 5.6(b) are flattened. These composites do not use a different model of computation, so a recursive model transform call is not required. Removing this layer of encapsulation has no effect on the behavior of the application.

3. *Domain-Specific Type Checking:* Next, the model transform verifies that all of the connections in the application model follow the rules given by the application domain. For example, in the Click language, ports have *push*, *pull* or *agnostic* types. Ports can be connected to ports of the same type or to agnostic ports, but a push port cannot be connected to a pull port. This is a domain-specific type error.

   More sophisticated type checking can also be performed in this step. In Click it is necessary to resolve all agnostic ports to either push or pull types. If a single actor has more than one agnostic port, then they all must resolve to the same type. Therefore the model transform must include code for inferring types and propagating types across actors.

4. *Domain-Independent Type Checking:* In a Cairn Kernel model, domain-specific concepts like push and pull ports are not represented. The only feature that ports have is a bit-width type. All connected ports must have the same bit width. Domain-independent type checking verifies that this is true for actors in the application model. This is a necessary condition for the bit widths to match in the transformed Cairn Kernel model.

5. *Domain-Specific Semantics Checking:* Specific application domains can have rules for making connections between actors other than requiring that types match. There may be restrictions on the number of connections that can be made to certain ports. Cycles may or may

not be allowed in the application model. Fan-out or fan-in connections may only be valid in certain contexts. This step checks that all of these rules are met. If a violation is found, then there is a semantic error in the application model that the programmer must fix.

An example from the Click language is the restriction on fan-in based on whether a connection is push or pull. Multiple push outputs are allowed to fan-in to the same push input port. However, multiple pull outputs are not allowed to fan-in to a pull input port. This is an *ambiguous pull* in Click terminology. The flow of control starts at the pull input. When there are multiple connected pull outputs, the Click element cannot determine which one to choose. It cannot be decided where the flow of control is supposed to go next.

6. *Process Identification:* The model transform uses the model of computation's process-level concurrency rules to locate the processes in the application model. For example, in process networks and CSP-like domains, every actor is a process. In Click, processes are found at the beginning of push chains, the end of pull chains, and at actors that have pull inputs and push outputs (e.g. *Unqueue*). This transform step adds new components to the model to cause these processes to be repetitively invoked.

   A structure like that shown in Figure 5.13 is employed. This component has one firing rule that produces two transfer messages. The first transfer causes one of the original application actors to fire. This is where the domain-specific process is supposed to begin. The second transfer causes the new component to fire the same rule again. The result is a steady stream of transfers that cause the domain-specific process to occur again and again.

7. *Communication Link Elaboration:* This step implements the abstract communication semantics found in the original application model. Connections like Click's push and pull links are replaced with explicit transfer-passing connections. This is done by making extensions to the firing rules found in the application actors. In addition to computing output data tokens, the firing rules will now also compute control words that determine where the flow of control is supposed to go next.

   For example, a push actor in the Click language will output a packet and also a control word that invokes the proper firing rule on the next actor in the push chain. The two outputs are bundled together as a transfer and sent to the next component.

   In some application domains, it is necessary to insert additional components along communication links in order to implement the model of computation's communication semantics. In

Kahn Process Networks, communication links are FIFO queues. A model transform for this domain would insert queue components into the transformed model wherever there is a connection in the original KPN model. The actors on each end of the queue would be extended to communicate with the queue using transfers. One type of transfer would push data tokens into the queue and another type would be used to retrieve tokens from the queue.

8. *Model Encapsulation:* The final step is to export the behavior of a composition of actors as if it were a single opaque actor. This allows the composite model to be used as a component in a hierarchically heterogeneous application model. To accomplish this, the model transform generates a set of firing rules that describe the behavior of the composition as a whole.

   Different models of computation have different notions of what it means to execute a model. In dataflow models, firing the model may mean executing one iteration of the schedule while producing and consuming tokens on the model's input and output ports. A process network, on the other hand, may execute for an arbitrary amount of time until all contained processes halt. A discrete time model may compute one time step.

These steps give an outline for how most model transforms work. Examples of actual model transforms that have been built for Cairn are given in Section 5.6.

5.5.4 Functional Simulation of Cairn Application Models

Functional simulation of application models is critical for effective application design space exploration. Domain experts desire the ability to construct application models piecewise, adding one feature at a time and testing continually. After the entire model is built, it is necessary to test for functional correctness using thorough benchmarks and real-world inputs. Speed and accuracy are important for making this process smooth.

Cairn provides a simulator code generator for models that use the Cairn Kernel transfer-passing abstraction. Applications designed using arbitrary models of computation can be simulated by first applying the appropriate model transform to get a Cairn Kernel model, and then applying the simulator code generator.

The application simulator code generator is exactly the same as the multiprocessor architecture simulator code generator. That generator, described in Section 4.4.3, simulates concurrent processing elements that communicate by exchanging signal-with-data messages. It is not a coincidence that the signal-with-data semantics of Cairn's architecture abstraction are similar to the transfer-

passing semantics of the Cairn Kernel. These models were designed to match so that applications can be correctly and easily mapped onto multiprocessor architectures.

To build a simulator, the transformed application model is interpreted as a multiprocessor architecture. Each application actor is a "processing element" that has one "operation" for each firing rule. Each transfer-passing communication link is interpreted as a point-to-point network-on-chip connection that carries signal-with-data messages. The architecture simulator code generator can now be applied directly, unaware of the original meaning of the given model.

The result is a C++ program that can be compiled and run on the development workstation. This program simulates all of the transfers that the application actors produce and consume.

Since the simulator has a single message queue and dispatch process, the results are representative of one particular sequentialization of the process-level concurrency found in the application. Later, when the application is mapped to a multiprocessor architecture with hardware support for process-level concurrency, some computations may be done in a different order. This is acceptable. The original application model is architecture-independent, so the simulator is providing a functionally correct simulation. If a particular mapping results in unwanted behaviors, there is an error in the original application. The application's process-level concurrency has been incorrectly specified. There may be a concurrency bug in the application, or the chosen model of computation may not be as strict about the allowable behaviors as the programmers believe it to be.

Like the multiprocessor architecture simulator, the application simulator is started by preloading the message queue with transfers. The code generator searches for firing rules named *initialize* throughout the application model. These rules are not allowed to consume any input tokens, but may produce output tokens including outgoing transfers. Before the main simulation loop is started, transfers corresponding to these *initialize* firing rules are placed into the queue. These computations will be done first. They should produce further transfer messages that keep the simulation running. If the simulator transfer queue empties, the system has become dormant and the simulator exits.

5.6 Cairn Application Domains

This section gives examples of high-level application abstractions that are built on top of the Cairn Kernel abstraction. Programmers use these abstractions (domains in Ptolemy II's vocabulary) to model specific application facets. Model transforms convert the high-level models into concrete actors or compositions of actors that use the Cairn Kernel abstraction.

5.6.1 Conditional Execution and Loops

The complexity of the computations that can be written in a Cairn actor description language firing rule is purposefully limited in order to match the capabilities of Sub-RISC processors. A firing rule can calculate a mathematical projection, update state variables, consume input tokens and produce output tokens. More complicated mathematics, such as conditional execution and dynamically-bounded loops, must be described using compositions of actors. Whenever actors need to be composed, models of computation provide abstractions to simplify the job. Therefore Cairn provides high-level abstractions and model transforms for conditional execution and loops. These things are often included as primitives in traditional programming abstractions, but they are modeled explicitly in Cairn.

Conditional Execution

The *if-then-else* control structure is a basic manifestation of conditional execution. First, a Boolean condition is evaluated. If it is true, the computation in the *then* block is performed. If the condition is false, the computation in the *else* block is done instead. This is different from the ternary ?: operator in the Cairn actor description language because it is not a combinational function. Only one of the two branches is executed. This distinction is important when the *then* and *else* blocks perform state updates or produce tokens on different output ports.

Consider Click's *LookupIPRoute* element. This actor pushes packets out one of several output ports based on a decision made by a routing algorithm. In the Cairn actor description language, it is not possible to write a firing rule that conditionally writes a token to an output port. Rules must declare all output ports that they write tokens to, and a token must be produced on all declared output ports.

The correct way to model this behavior in Cairn is to use a composition of firing rules that pass the flow of control amongst themselves using transfer messages. When a packet is pushed into the *LookupIPRoute* element, computation begins with a rule that evaluates the routing decision. This rule then arranges for computation to continue at one of N branch firing rules, where N is the number of output ports on the *LookupIPRoute* element. Each branch firing rule writes the packet to one of the output ports. This way, all of the rules can declare exactly the set of output ports where tokens are produced.

It can be a burden for programmers to think about *if-then-else* structures in terms of transfer messages. Therefore Cairn provides a model of computation to make this task easier. Figure 5.14

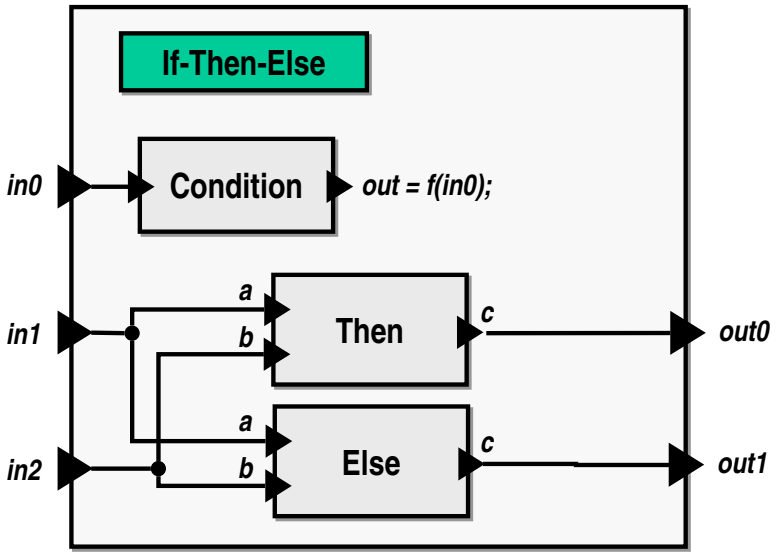


Figure 5.14: If-Then-Else Composite Actor Model

shows how this abstraction is used. Programmers make a composition of three actors (*Condition*, *Then* and *Else*) with an attribute indicating the If-Then-Else model of computation. The semantics of this model are as follows: Computation begins at the *Condition* actor. This actor is expected to have one firing rule that produces a Boolean output value. If this value is true, the *Then* actor will be fired. Otherwise, the *Else* actor will be fired. The semantics of this abstraction are much different from traditional models of computation such as SDF and CSP. Nevertheless, the abstraction defines how components in a composition interact, so the term still applies.

This abstraction hides the transfer messages that are exchanged between the three actors. Programmers leave the output port of the *Condition* actor unconnected. The *Condition*, *Then* and *Else* actors can receive their inputs from any ports on the boundary of the composite model. Likewise, the *Then* and *Else* actors can produce outputs on any of the composite model's output ports.

When the If-Then-Else model transform is applied to this model, the proper transfer-passing connections are automatically filled in. Figure 5.15 shows the transformed model. Incoming transfers are received on a single input port and are sent directly to the *Condition* actor. The model transform extends the *Condition* actor so that in addition to computing the condition, it also produces transfers for the *Then* and *Else* actors.

Figure 5.16 shows all of the modifications to the *Condition* actor. The condition value itself is now an internal variable instead of an external output. This value is used to compute control words for the *Then* and *Else* actors, which are output on the new ports *t1* and *t2* respectively. If the

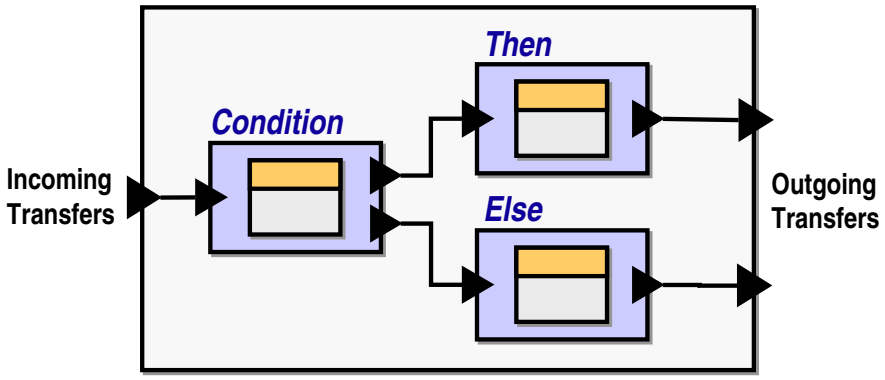


Figure 5.15: Result of the If-Then-Else Model Transform

condition is true, the *Then* actor receives a transfer telling it to fire while the *Else* actor receives a *noop* transfer. If the condition is false, the opposite occurs. Thus computation continues at only one of the two branches.

The transformed *Condition* actor is given additional ports (*a*, *b*, *ta* and *tb*) to pass inputs from the original composite model to each of the branch actors. The exact port configuration and additional firing rule statements are determined from the connectivity found in the original model. This is necessary because the transfer messages for each of the branches must contain the data to work on as well as the control word that tells them what firing rule to execute. These data values are originally present in the transfer messages received by the composite model as a whole. The *Condition* actor gets the messages first and passes the data values along with the thread of control to one of the branch actors.

Other manifestations of conditional execution, such as *if-then* or *switch* control structures, use variations of this abstraction and model transform.

Loops

Dynamic loop control structures allow programmers to express repetitive computations where the bounds of the repetition are not known until run time. In this structure a Boolean condition determines how many times to iterate a loop body. Computation begins with the condition. If it is found to be true, the loop body is executed. The thread of control then continues at the condition, where the decision is recalculated. When the condition is found to be false, the thread of control exits the loop control structure.

Just as in the case of conditional execution, the proper way to express this type of computation is to use a composition of firing rules that communicate with transfer messages. Cairn provides a

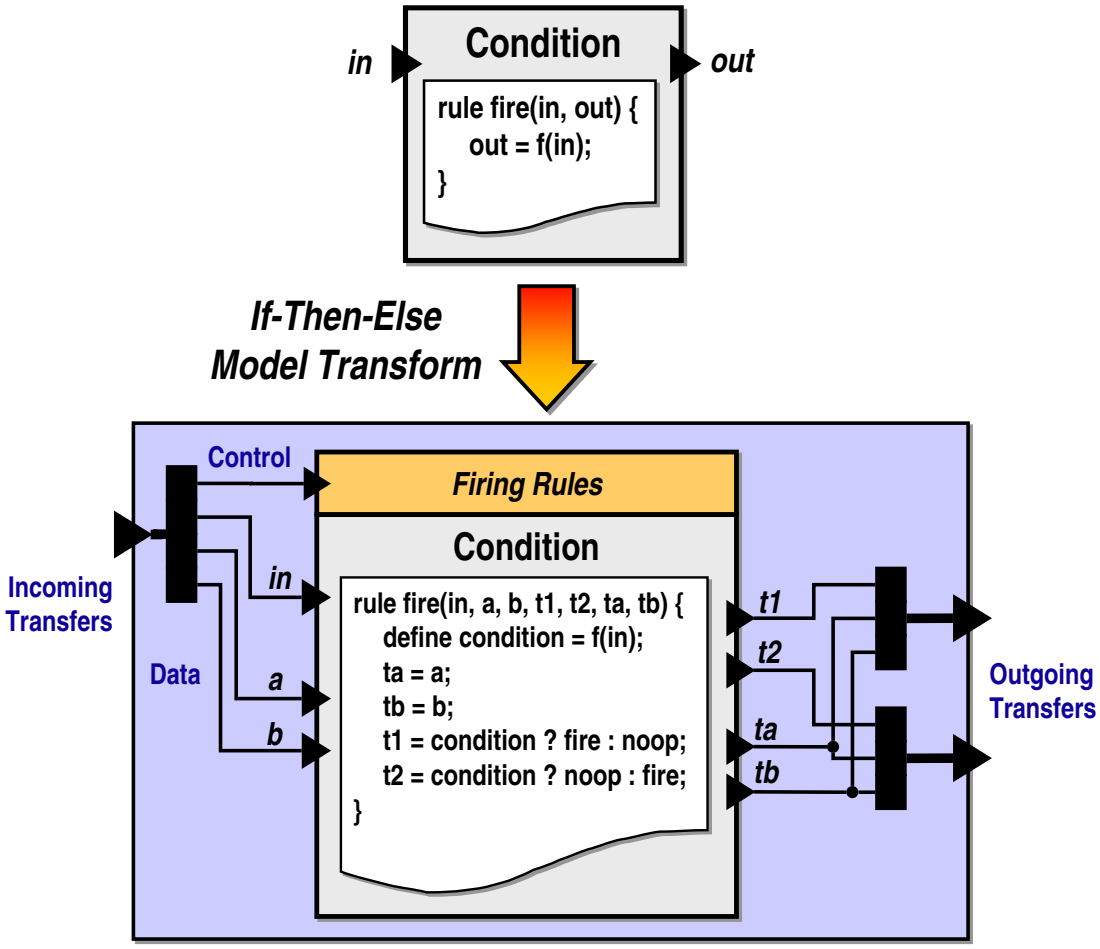


Figure 5.16: If-Then-Else Model Transform Extends the Condition Actor

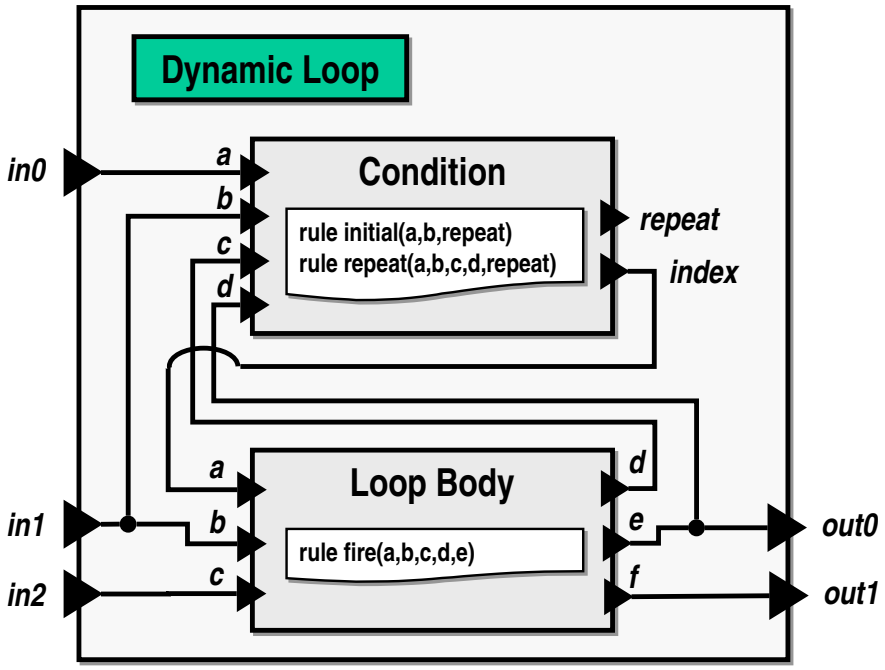


Figure 5.17: Dynamic Loop Composite Actor Model

Dynamic Loop model of computation that programmers can use to express loop control structures without thinking about transfer messages. Figure 5.17 gives an example. The model must contain two actors. The *Condition* actor determines how many times to iterate the *Loop Body* actor.

The *repeat* output of the *Condition* actor is used to decide whether to fire the *Loop Body* actor. Just as in the If-Then-Else model of computation, this port can be left unconnected. The *Condition* actor can have two firing rules that calculate the *repeat* value. One rule can only have dependencies on the input ports of the composite model. This rule determines the initial condition for executing the loop body before any iterations of the loop body have been performed. The second rule is allowed to have dependencies on both the composition's input ports as well as the outputs of the *Loop Body* actor. This enables the repetition condition to be dependent on values calculated inside the loop body.

The *Loop Body* actor is expected to have one firing rule that can be dependent on both composite model inputs and *Condition* actor outputs. In this example, the *index* output of the *Condition* actor is used within the loop body. The *Loop Body* actor also drives the composite model's output ports. When the *Condition* actor decides to stop repeating the loop body, the values on these output ports are those from the last firing of the *Loop Body* actor. If the loop body has never been fired, then there will be no tokens on these ports.

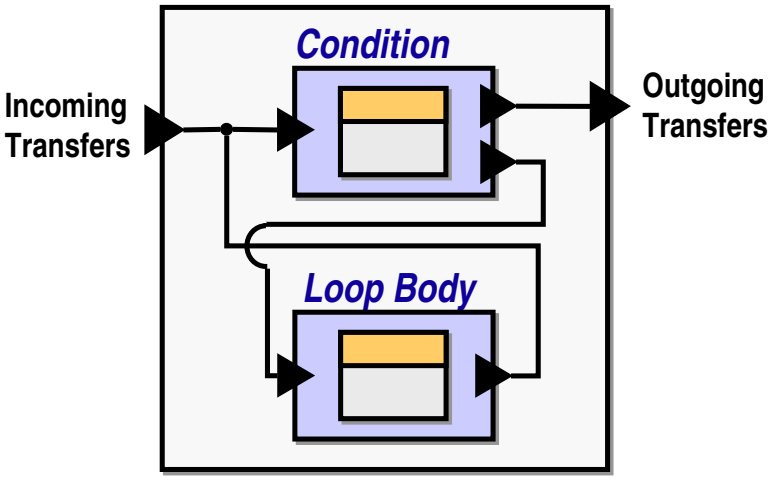


Figure 5.18: Result of the Dynamic Loop Model Transform

Cairn's Dynamic Loop model transform fills in the appropriate transfer-passing connections and makes extensions to the actors' firing rules. The result is shown in Figure 5.18. Incoming transfer messages are directed to the *Condition* actor. This actor decides to pass the flow of control either to the *Loop Body* actor or creates an outgoing transfer that exits the composition. When the *Loop Body* actor fires, control is always passed back to the *Condition* actor. Thus the *Condition* actor can receive control from two different places.

If a Dynamic Loop composite actor is used in a higher-level model with multiple processes, transfers from the *Loop Body* actor may be interspersed with new incoming transfers from outside the model. The dynamic loop is not atomic. Thus, if the *Condition* or *Loop Body* actors contain internal state there will be synchronization issues. The Dynamic Loop model of computation is not concerned with this problem. It is the responsibility of the higher-level model of computation to handle process-level concurrency issues. The Dynamic Loop abstraction cannot make assumptions about what is required at higher levels.

Like the If-Then-Else model transform, the Dynamic Loop transform adds ports to actors and modifies firing rules such that data values are propagated along with the flow of control. The exact changes required are discovered by analyzing the connectivity in the composite model. The original *in0*, *in1* and *in2* input values must be preserved across an arbitrary number of iterations of the *Condition* and *Loop Body* actors. Therefore the *Condition* actor's firing rules are modified to pass these values along to the *Loop Body* actor. In turn, the *Loop Body* actor passes the same values back to the *Condition* actor.

Figure 5.19 shows the modifications to the *Loop Body* actor. The *t_out* port is the control word

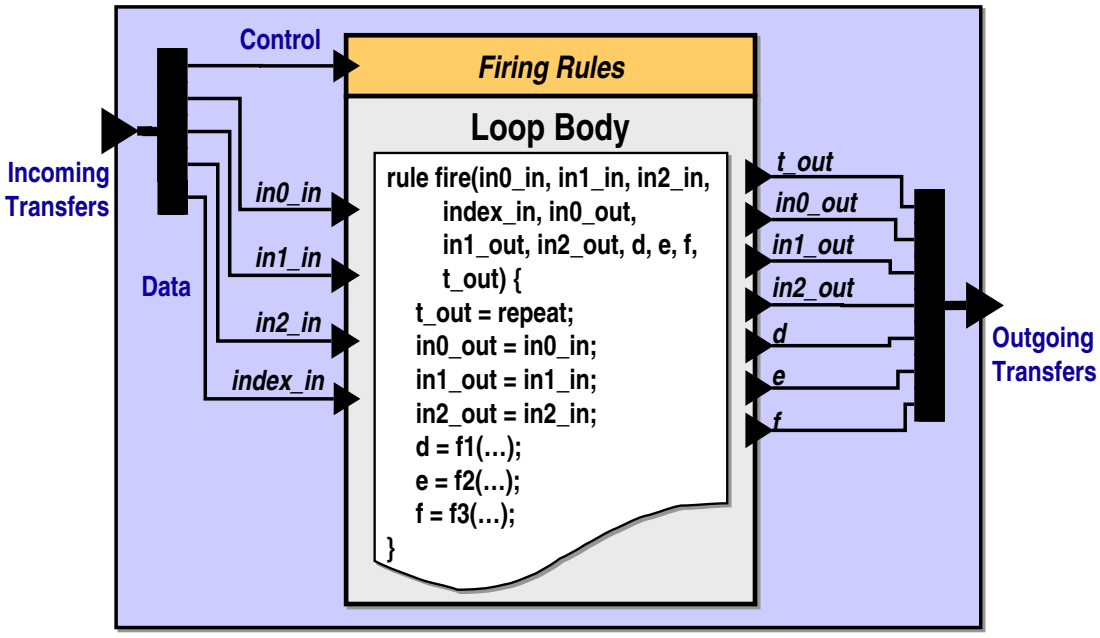


Figure 5.19: Dynamic Loop Model Transform Extends the Loop Body Actor

for the transfer back to the *Condition* actor. This is always the *repeat* firing rule. The *in0_out*, *in1_out* and *in2_out* ports pass the original *in0*, *in1*, and *in2* values through unchanged. No modifications are made to the parts of the firing rule that compute the *d*, *e*, and *f* outputs. All of the input values are allowed to be used to perform these computations.

The extensions to the *Condition* actor are shown in Figure 5.20. The flow of control is directed as in the If-Then-Else model transform by sending *noop* and *fire* transfers to the *t1* and *t2* ports. The *in0*, *in1* and *in2* values are passed straight through to the *Loop Body* actor. The *repeat* firing rule also writes to the *out0* and *out1* using the results of the last loop body computation.

Together, the Dynamic Loop and If-Then-Else models of computation allow Cairn users to explicitly model more complex computations than what can be represented in the Cairn actor description language alone. This explicit modeling satisfies the second core requirement: focus on simple architectures (Section 2.1.2). If dynamic loops and conditional execution were language primitives, then all target processing elements would have to contain hardware to support loops and conditionals. This assumes a higher level of complexity than is necessary for many useful actors. When loops and conditionals are not necessary, the Sub-RISC paradigm allows architects to create smaller, faster and simpler PEs. Explicit modeling ensures that architectural complexity is only incurred when the application demands it.

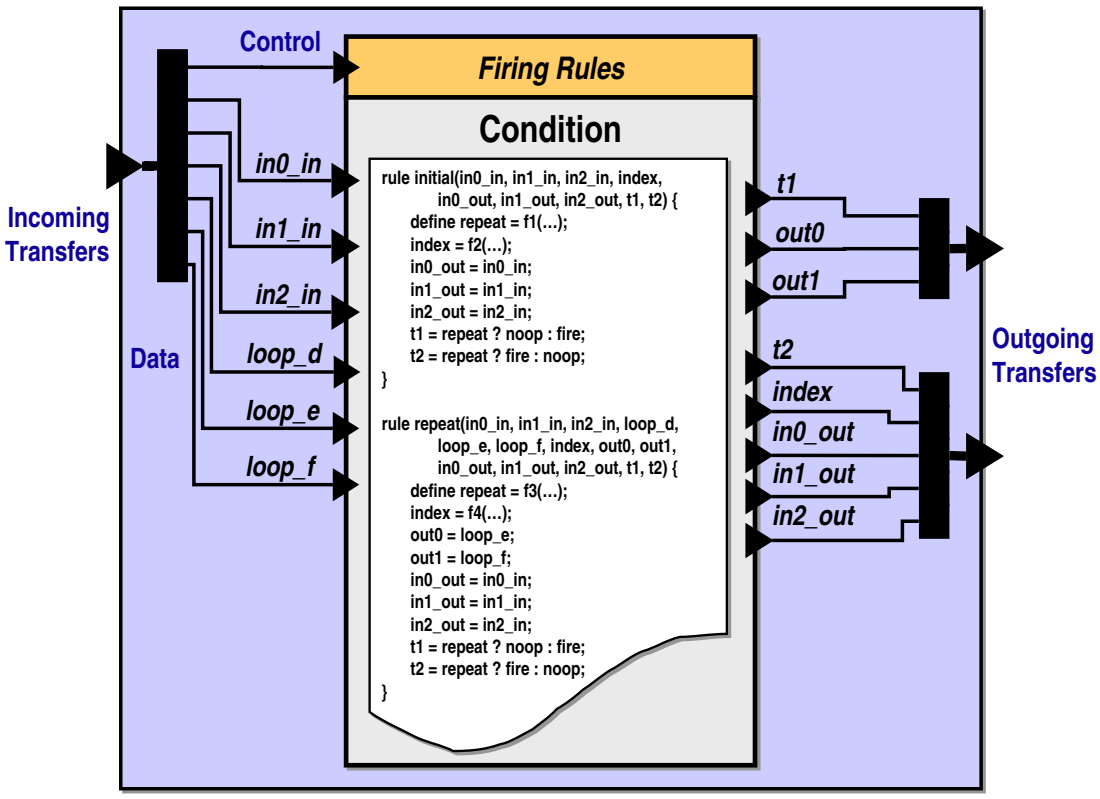


Figure 5.20: Dynamic Loop Model Transform Extends the Loop Condition Actor

5.6.2 The Click Domain

The Click suite by Kohler et al. [72] includes a domain-specific language for describing header processing applications. Domain experts describe an application by assembling *elements* from a pre-written element library. Elements have ports that are connected by directed edges in the application model. The abstraction states that packet headers are passed between elements along the edges. Each element can inspect and update header fields and perform temporal and spatial header processing computations.

The elements encapsulate data-level and datatype-level concurrency. Programmers can work with packet headers as high-level data types and need not concern themselves with the details of the computations that are done to the packets.

Process-level concurrency is captured by the *push* and *pull* communication semantics between elements. Sources of packets, such as network ingress ports, have ports that *push* packets into the element netlist. Network egress ports have *pull* ports that pull packets out of the element netlist. Push and pull ports help define where flows of control start in the application as well as how control is passed from one element to another.

These communication semantics mimic the actual behavior of networking peripherals. When a network card receives a packet, it raises an interrupt that causes a new flow of control to be started. Another interrupt is raised when the card has buffer space available to transmit an outgoing packet.

These communication semantics also match the temporal processing done in the application domain. For example, a *Queue* element buffers packets and has a *push* input and a *pull* output.

These semantics provide an intuitive abstraction for building network processing applications. However, the tools in the Click software suite that support this abstraction do not enable mapping the application onto a custom multiprocessor architecture. The tools only support a top-down implementation methodology that targets single-processor Linux workstations. Since this platform does not consider application-specific concurrency, much of the concurrency found in the high-level Click models is not exploited in the final implementation.

Cairn's Click domain provides the same semantics and element library as the original Click domain-specific language. This domain integrates with the rest of the Cairn methodology to enable mapping applications onto Sub-RISC multiprocessors.

The Click Model Transform

A model transform converts Click models into Cairn Kernel models. The abstract push and pull communication links between elements are replaced with concrete transfer-passing connections. The first step in this process is domain-specific type checking. This verifies that push and pull ports are connected properly and all agnostic ports are resolved to either push or pull types.

The Click type resolution algorithm repeatedly makes type inferences until all agnostic ports are resolved or until a type conflict is found. If an agnostic port is connected directly to a push or pull port, the agnostic port becomes either push or pull respectively (Figure 5.21(a)). The Click semantics state that if an element has agnostic ports, they must all resolve to the same push or pull type. This allows the algorithm to make inferences across actors (Figures 5.21(b) and 5.21(c)). Figure 5.21(d) shows the final result. A Click-specific type error occurs when a push port is directly connected to a pull port, or when agnostic ports on the same element are found to have different types. This is a syntax error that the programmer must fix.

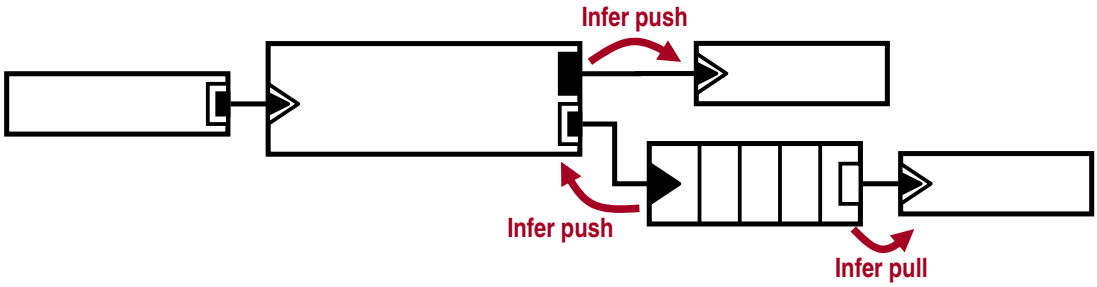
Domain-Specific Semantics Checking

The second step in the model transform is domain-specific semantics checking. Click requires that pull input ports (Figure 5.22(a)) and push output ports (Figure 5.22(b)) have connections. A pull input port must be connected to exactly one pull output port. Figure 5.22(c) shows an illegal *ambiguous pull*. The element cannot decide which of the two queues to pull from. Similarly, push outputs must be connected to exactly one push input port. Figure 5.22(d) shows an illegal *ambiguous push*.

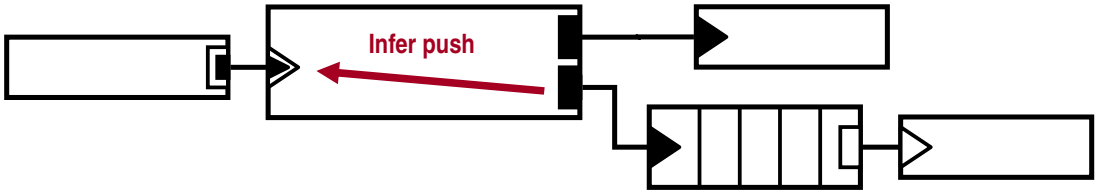
Process Identification

Third, the model transform identifies elements where threads of control begin. A push element with no input ports is the beginning of a *push chain*. A pull element with no output ports is the beginning of a *pull chain*. Elements with pull-to-push personalities (e.g. *Unqueue*) are also the beginnings of threads of control. These examples are shown in Figure 5.23.

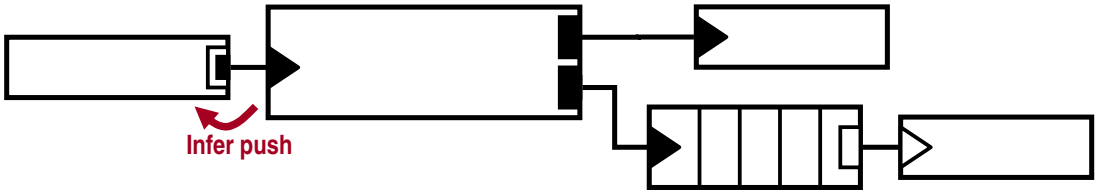
Click elements that start threads of control are transformed into Cairn Kernel structures like the one shown in Figure 5.24. A new component is added that sends *fire* transfers both to itself and to the Click element. This causes the Click element to execute repeatedly.



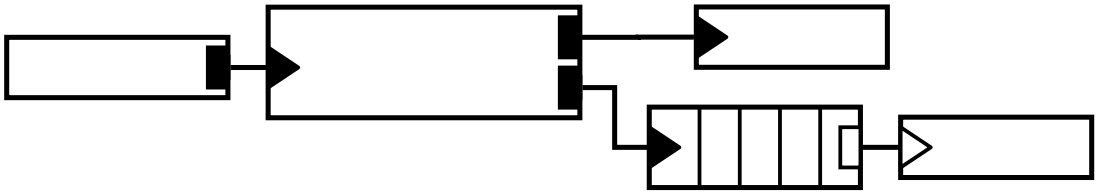
(a) Resolving Agnostic Ports Directly Connected to Push or Pull Ports



(b) Inferring Types Across Agnostic Actors

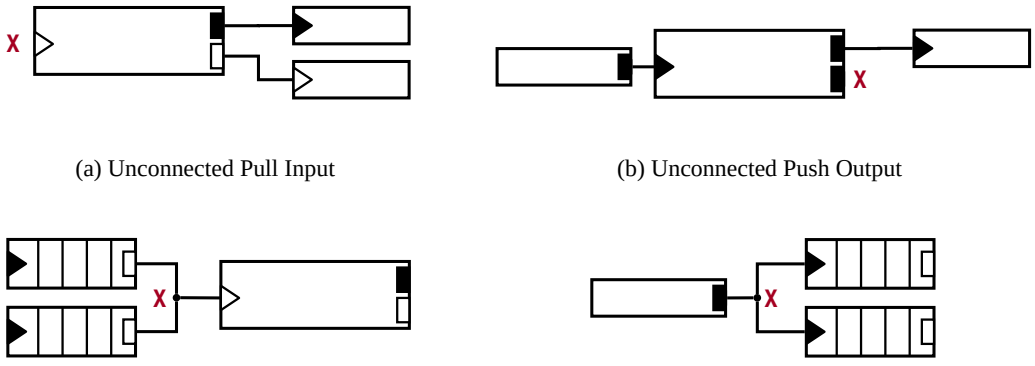


(c) Making Further Type Inferences



(d) All Agnostic Ports are Resolved

Figure 5.21: Click Domain-Specific Type Resolution



(a) Unconnected Pull Input

(b) Unconnected Push Output

(c) Ambiguous Pull

(d) Ambiguous Push

Figure 5.22: Click Connection Errors

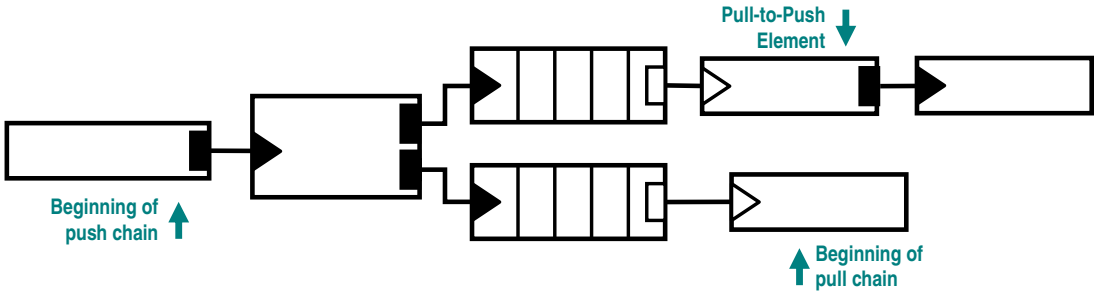


Figure 5.23: Click Process Identification

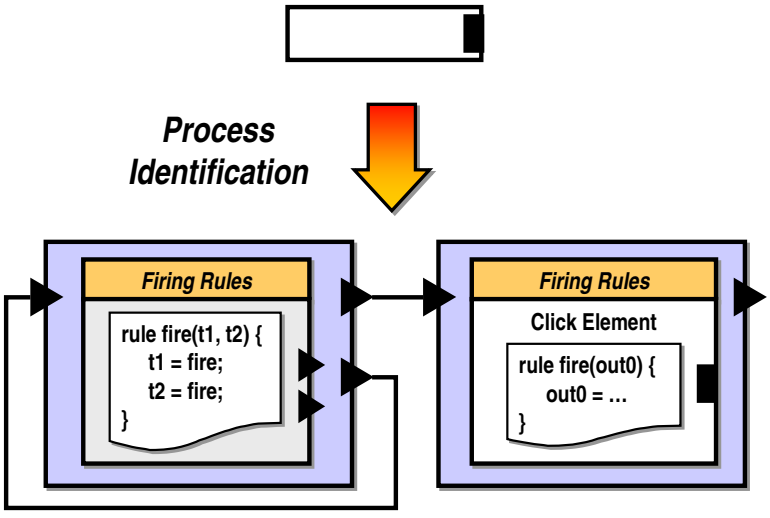


Figure 5.24: Cairn Kernel Structures for Repeatedly Firing a Click Element

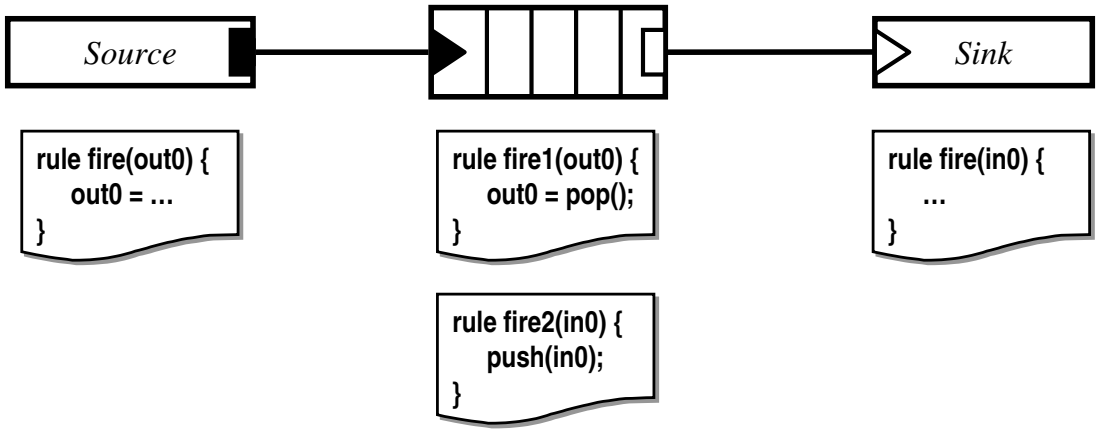


Figure 5.25: Auxiliary Graph for Click Communication Link Elaboration

Communication Link Elaboration

The final step of the Click model transform is to replace abstract push and pull communication links with concrete transfer-passing links. The firing rules within the Click elements are extended so that in addition to performing header processing computations, they also compute a transfer for the next element that should receive the flow of control.

The model transform applies the semantics of the Click model of computation to determine how control flows throughout the model. An auxiliary graph is computed to organize this information. Figure 5.25 shows the starting graph for an example Click model with three elements: a *Source*, a *Queue*, and a *Sink*. The graph contains the given Click model together with additional nodes that represent the firing rules for each element. The model transform will add directed edges to this graph, represented by arrows, that show how the flow of control leads into each firing rule and where it goes after each rule executes.

First, the transform looks for ports where actors can receive the flow of control. An actor receives control when a packet is pushed on a push input port, or when a pull request arrives on a pull output port. There is one of each of these ports in the example, both belonging to the *Queue* element.

Next, the transform searches for a firing rule that is compatible with the push input or pull output. For push inputs, a firing rule that consumes one token from the given port and no tokens from other push input ports is necessary. A rule that requires tokens from multiple push inputs is not compatible with the Click semantics, as this represents a merging of several independent flows of control. The *Queue* has one firing rule, *fire2*, that is compatible with the push input. This is

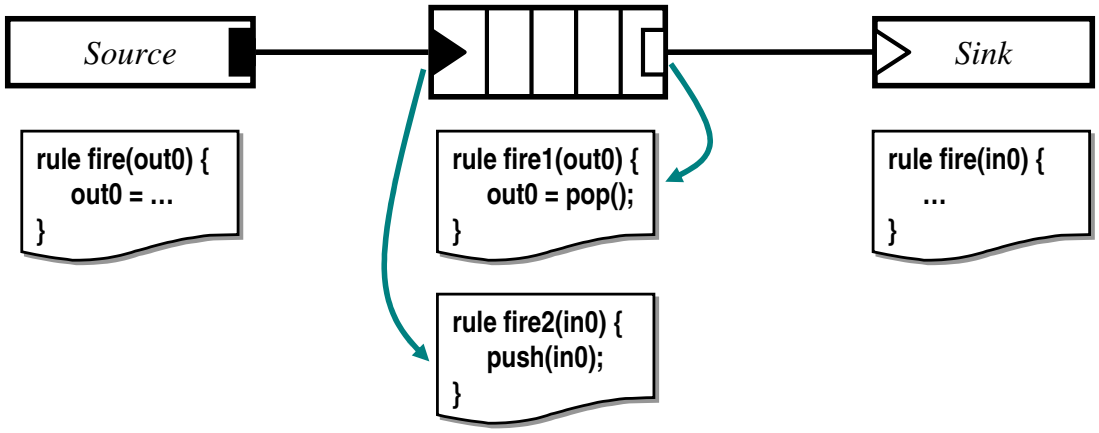


Figure 5.26: Processing Ports where the Flow of Control is Received

the correct rule to execute when control is received on the push input, so an arrow is drawn in the auxiliary graph between the push input and the *fire2* rule. This is shown in Figure 5.26.

For pull outputs, a rule that produces a token on the given output must be found. The *Queue* element's *fire1* rule matches, so an arrow is drawn from the pull output to *fire1*.

If at any point an appropriate firing rule cannot be found, then there is an error in the application model that the programmer must fix. The given actor is not valid in its current context under the rules of the Click model of computation. The same is true if the transform finds more than one firing rule that satisfies the criteria it is looking for. This is an ambiguity that the transform cannot deal with.

Next, the model transform adds the components from the process identification step and determines which firing rules correspond to the beginning of the threads. For the *Source* element, there is only one choice. An arrow is drawn from the *Source Process* node to the *Source* element's *fire* rule (Figure 5.27).

There is only one choice for the *Sink* element as well. This case demonstrates what happens when firing rules have prerequisites: *fire* requires a token on the *in0* port in order to execute. The model transform must figure out where to obtain this token. Figure 5.28 shows how this is done.

All prerequisites can be fetched in parallel. New *merge* and *fanout* nodes are added to the auxiliary graph. The *fanout* node forks control to all of the ports that supply prerequisite data to the *Sink* element. In this case, the *Sink* element's firing rule depends only on *in0*, which receives data from the *Queue* element's output port. Therefore one arrow is drawn between the *fanout* node and the *Queue* element's output port.

When the *Sink* element sends a pull request to the *Queue*, the flow of control is expected to

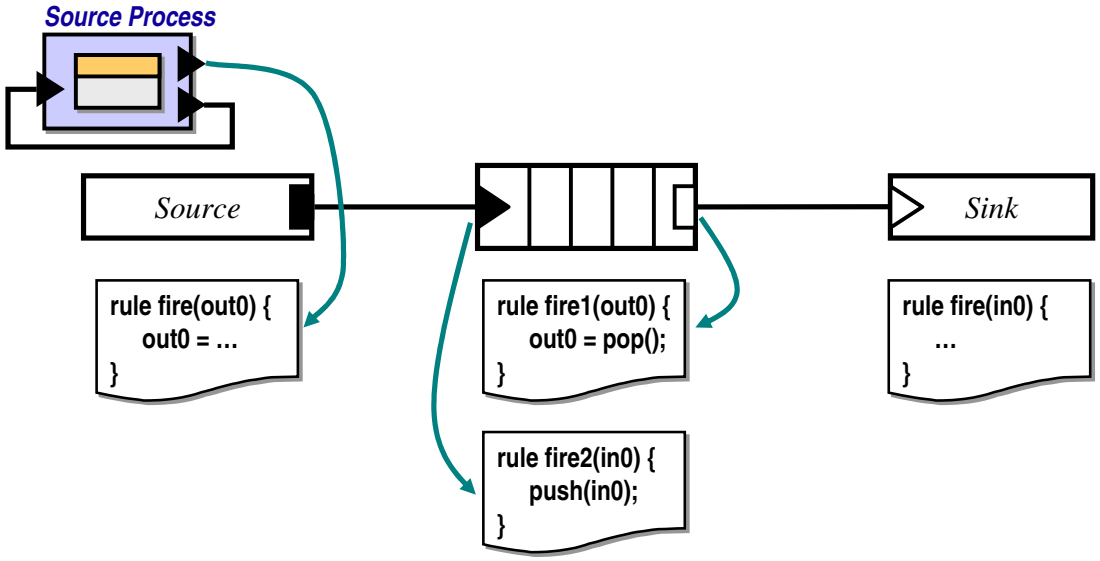


Figure 5.27: Auxiliary Graph Nodes and Edges for Click Processes

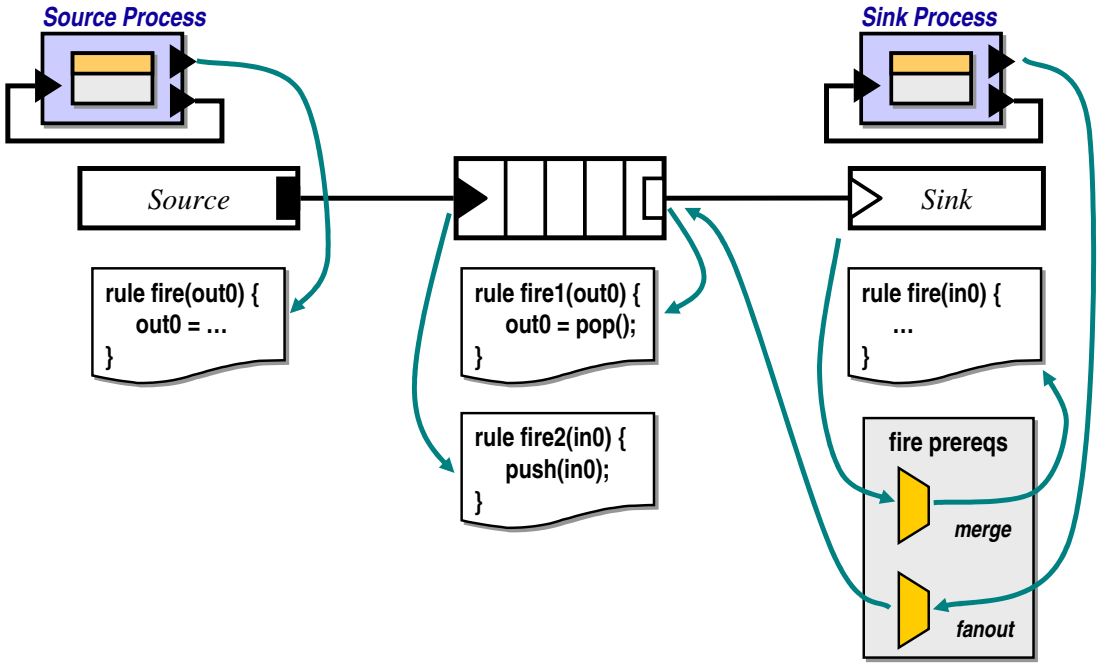


Figure 5.28: Auxiliary Graph Structures for Firing Rules With Prerequisites

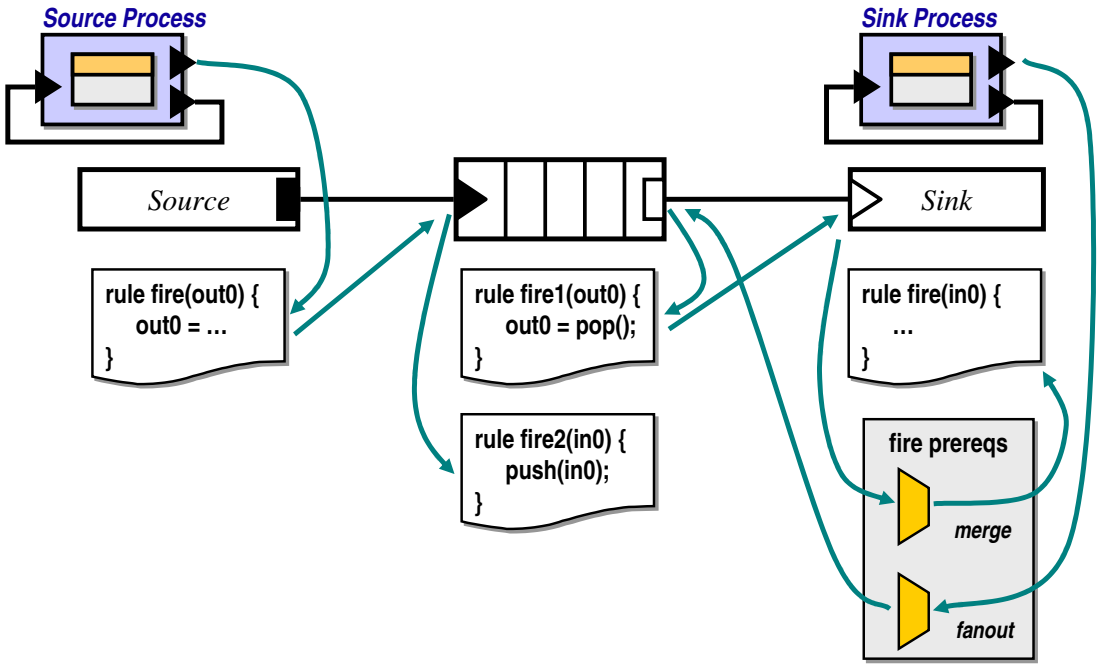


Figure 5.29: Determining Where the Flow of Control Goes After Each Firing Rule

return to the *Sink* element's *in0* port at some time in the future. If the flow of control was forked to fetch multiple prerequisites simultaneously, then there will be multiple flows of control returning to the *Sink* element. The *merge* node collects these flows of control. Once all prerequisites are available, then the *fire* rule can be executed. In this example there is only one prerequisite to fetch, so there is only one arrow exiting the *fanout* node and entering the *merge* node. This is simply the base case of a general algorithm. An example of the general case will be given later in this section.

The last step in communication link elaboration is to visit all firing rules in the auxiliary graph and determine where the flow of control should go next after each rule executes. This is done by looking at the output port declarations in the firing rule and comparing them with the connections found in the original Click model.

The *Source* element's *fire* rule produces one output token on the *out0* port. This is a push output, so computation should continue at the connected push input. An arrow is drawn between *fire* and the *Queue* element's push input port (Figure 5.29).

The *Queue* element's *fire1* rule produces a token on the *out0* port. An arrow is drawn between *fire1* and the *Sink* element's input port. None of the other firing rules produce any output tokens, so no further arrows need to be drawn.

The completed auxiliary graph is now used to make extensions to each of the firing rules. In

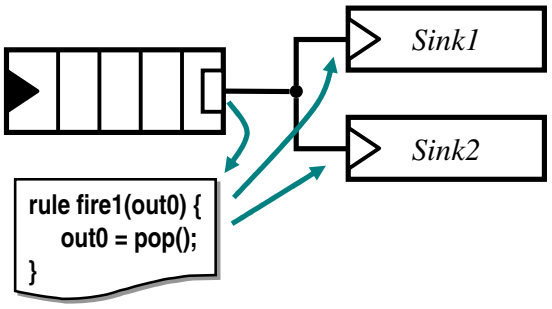


Figure 5.30: Multiple Elements Pulling From One Pull Output Port

addition to performing computation on packet headers, the rules must now also create outgoing transfers. The proper transfer to create can be found by following arrows starting from the firing rule's auxiliary graph node until another firing rule node is reached.

For example, the *Source* element's *fire* rule has an outgoing arrow that leads to the *Queue* element's input port and then to the *Queue* element's *fire2* rule. This indicates that the *Source* element's *fire* rule should be extended to compute a transfer for the *Queue* element telling it to execute the *fire2* rule. The model transform makes this extension and adds the appropriate transfer-passing connections to the Cairn Kernel model. Likewise, the *Queue* element's *fire1* rule is extended to create a transfer for the *Sink* element's *fire* rule.

The *Queue* element's *fire2* rule has no outgoing arrows, so the thread of control dies after this rule is executed. No outgoing transfers are created. The same is true for the *Sink* element's *fire* rule.

The *Sink Process* component is meant to repeatedly execute the pull chain that starts at the *Sink* element. By following the arrows, it can be determined that the first firing rule to execute in this pull chain is actually the *Queue* element's *fire1* rule, and not one of the *Sink* element's rules. The *Queue* must execute first because it provides a prerequisite to the *Sink*. In the Cairn Kernel model, a transfer-passing connection will be made between the *Sink Process* component and the *Queue* component.

If a chain of arrows does not lead to a firing rule, then there is an error in the application model. The model transform was unable to figure out where the flow of control should go next after executing a particular firing rule.

Special Cases of the Click Model Transform

There are two special situations not covered by the above example. The first situation occurs when multiple elements pull from the same pull output port. Figure 5.30 gives an example.

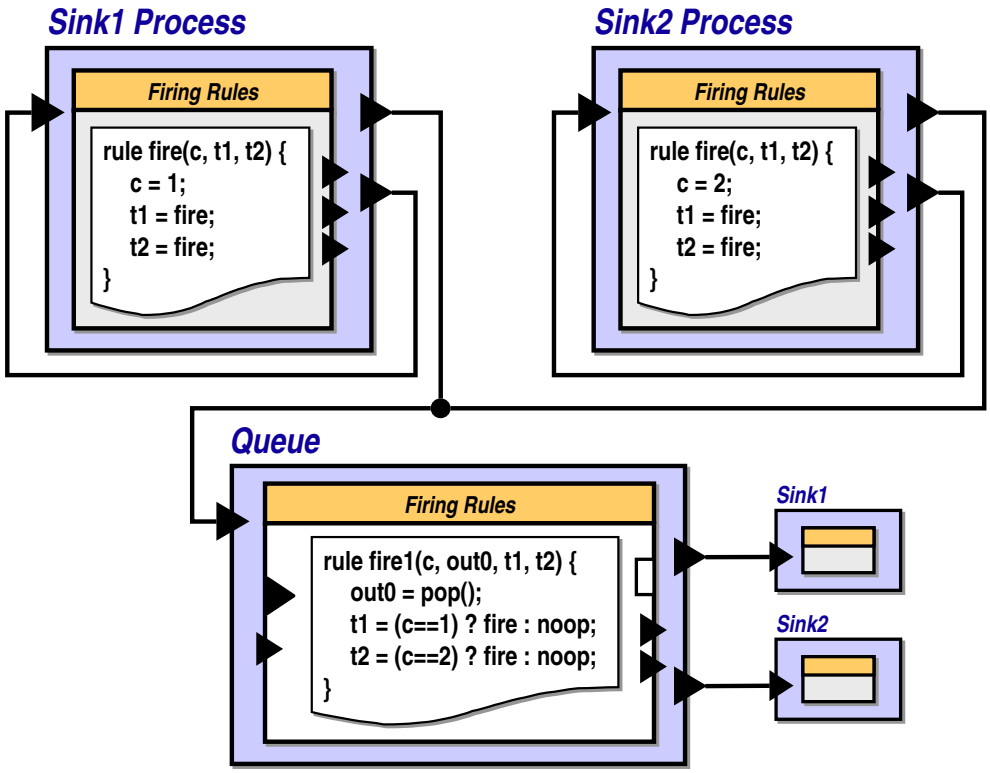


Figure 5.31: Transformed Model with Multiple Elements Pulling From One Port

Sink1 and *Sink2* both pull from the *Queue*. When the flow of control enters the *Queue* at the *out0* port, the model transform knows that the *fire1* rule is the correct rule to execute. However, after this rule executes, there is an ambiguity as to where to send the flow of control next. It could go to either *Sink1* or *Sink2*. The Click semantics require that the flow of control returns to the element that initiated the pull.

In order to achieve this, the *Queue* must be told who initiated the transfer. This is provided in the data portion of the transfers that the *Queue* receives from *Sink1* and *Sink2*. *Sink1* and *Sink2* are extended to provide this information as an integer data field. The *Queue* is extended to conditionally create transfers back to *Sink1* and *Sink2* using the value of the field. The transformed Cairn Kernel model is shown in Figure 5.31. Note that since *Sink1* and *Sink2* are the beginnings of pull chains, the threads of control start at the components labeled *Sink1 Process* and *Sink2 Process*. These components send transfers to the *Queue*, which in turn sends transfers to *Sink1* and *Sink2*.

This technique is analogous to explicit continuation passing. In general-purpose programming languages, continuations require passing call stacks of arbitrary depth. This is because a thread of control can enter a block of code an arbitrary number of times. The stack stores the history of the

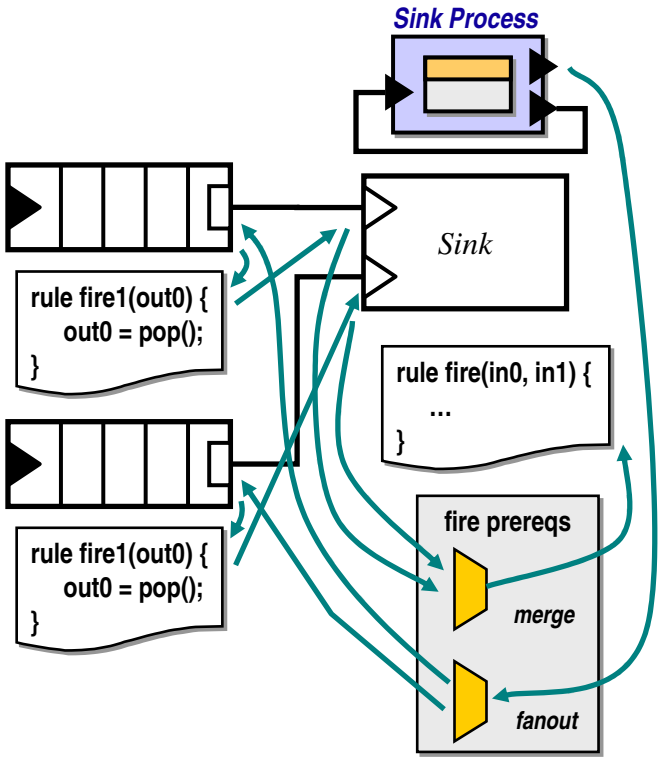


Figure 5.32: Click Element with a Firing Rule that Requires Multiple Pulls

thread, which can be unbounded. As long as there are no cycles in the auxiliary graph, the history of a Click thread is bounded. Therefore a finite amount of data needs to be passed along with transfer messages.

The second special case occurs when a Click element has a firing rule that requires tokens on multiple pull input ports. The model transform uses a *fanout* node to fork the thread of control and fetch all of the inputs simultaneously. A *merge* node rejoins the flows and causes the firing rule to execute. The case where there is only one required pull input was shown in the previous example. An example with two required pull inputs is shown in Figure 5.32.

In this case the model transform inserts two arrows that leave the *fanout* node and two arrows that point into the *merge* node. The functionality of the *fanout* node is implemented in the Cairn Kernel using fan-out of transfer messages. One outgoing transfer port is connected to two incoming transfer ports as shown in Figure 5.11. Transfer messages produced by the *Sink Process* component are duplicated and sent to both *Queue* components.

The *merge* node requires more work. Cairn Kernel components do not wait for multiple transfer messages. They execute transfers as soon as they are received. To synchronize two or more

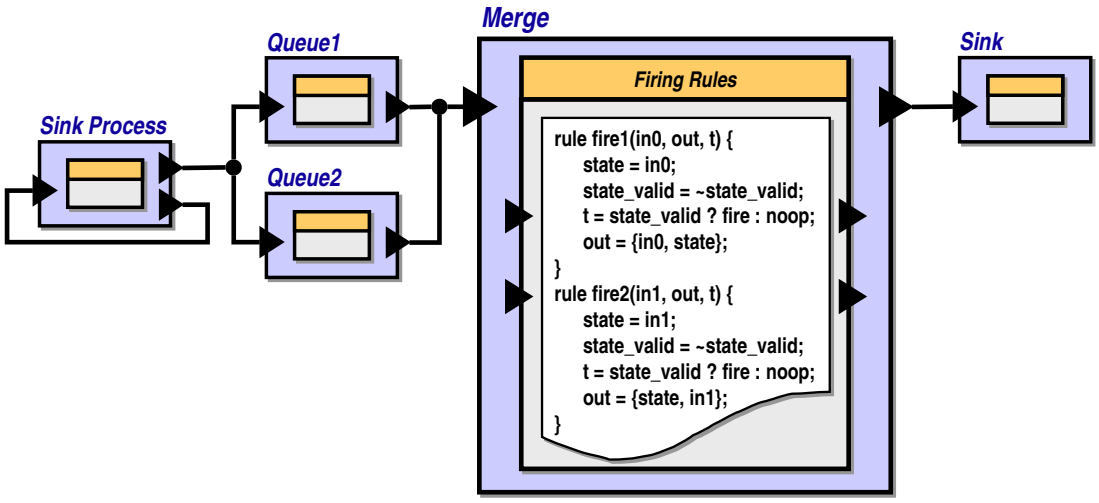


Figure 5.33: Cairn Kernel Model with Merge Components

transfers, an additional component must be inserted into the Cairn Kernel model between the *Queue* components and the *Sink*. Transfers from the *Queue* components are first sent to this *Merge* component. The *Merge* component accepts the transfers independently, in any order. Each *Queue* uses a different firing rule on the *Merge* component. Internal state is used to store the transfers until all of them have arrived. When this happens, the stored data values are grouped together to form a new outgoing transfer destined for the *Sink* component.

Since the Cairn Kernel model explicitly implements the synchronization between the flows of control, the cost of synchronization is made clear. Systems designers pay this cost only when the application requires it. The transformed Cairn Kernel model for this example is shown in Figure 5.33.

The Click model transform illustrates the utility of using an imperative language to describe model transforms. The auxiliary graph is essential to determining the control flow in the application. Current metamodeling approaches lack the flexibility to describe a transform of this complexity in a simple way. A final point is that many of the techniques used in the Click transform have general utility in other application domain transforms, such as inserting process components that repeatedly fire an application actor and the *merge* nodes,

In Chapter 7, a custom Sub-RISC processor for Click header processing will be presented. This processor directly supports the process-level, data-level, and datatype-level concurrency in Click application models to achieve high packet throughput.

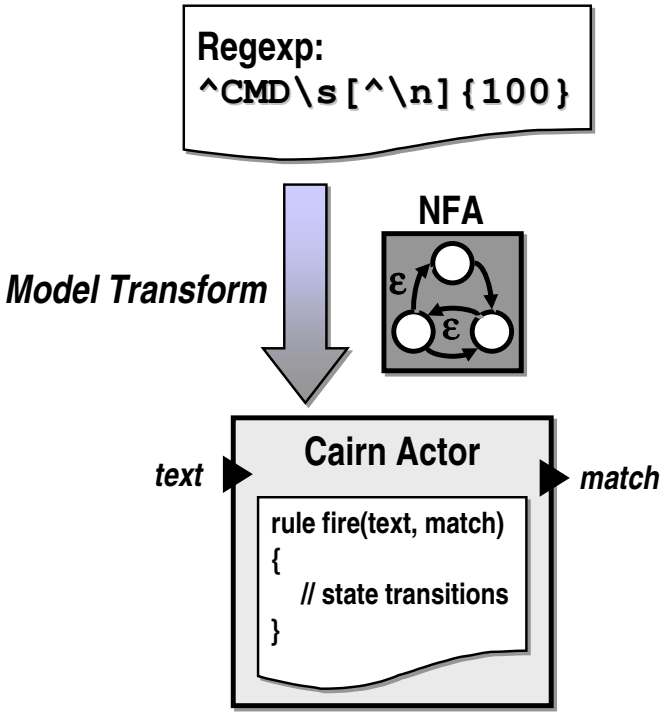


Figure 5.34: Regular Expression to Cairn Actor Model Transform

5.6.3 The Regular Expression Domain

In the Click domain, applications are described as graphs with nodes and edges. Not all applications fit naturally with this style of graphical programming. One benefit of having separate model transforms for each application domain is that the input language can vary from domain to domain. In this section a domain that uses a text-based input language is presented.

One facet of Cairn's network processing design driver is network intrusion detection. This facet performs pattern matching against the contents of packets to search for potential attacks. In the Snort network intrusion detection system, regular expressions are used to model the search patterns. Domain experts focus solely on the patterns they want to search for. The actual technique used to perform the pattern matching is not of interest.

Cairn's regular expression domain provides exactly this abstraction. Experts describe a pattern matching application by simply writing regular expressions. A model transform converts this abstract specification into a concrete implementation as a Cairn actor.

Figure 5.34 shows the inputs and outputs of the model transform. The input is a string in the Perl-Compatible Regular Expression (PCRE) language [94]. This string is parsed and converted into a non-deterministic finite state automata (NFA). The final Cairn actor is an implementation of

this NFA. The actor contains the NFA state and has one firing rule that calculates the NFA's state transition and output functions.

This actor is meant to be used in a streaming context in a larger application model. Text characters are provided on the *text* input port. Every time the actor is fired, one text character is consumed, one NFA transition is calculated, and one match result is produced on the output port. This single-bit result indicates if the regular expression matches at the current position in the text stream.

The technique for converting the textual regular expression into an NFA implementation and then into a set of state transition functions is based on the work of Sidhu and Prasanna [108]. This original work focused on implementing regular expression pattern matching on FPGAs. NFAs are a desirable implementation choice because they require only $O(n)$ memory, compared to deterministic finite automata (DFAs) which require $O(2^n)$ memory in the worst case. On a traditional sequential processor, the trade-off is that calculating an NFA state transition requires $O(n)$ time, whereas a DFA transition requires only $O(1)$ time. Slow processing speed usually rules out NFAs as an implementation choice.

On an FPGA, Sidhu and Prasanna are able to exploit the data-level concurrency found in an NFA to calculate all non-deterministic state transitions in parallel. The main idea is that the NFA can be represented as a one-hot encoded state machine where multiple bits are allowed to be hot at the same time. This represents a superposition of states. State transition logic evaluates all possible next state transitions in one clock cycle. Thus, the implementation has both small size and high speed.

The first step is to create a parse tree for the regular expression. The leaves in this tree represent characters and the nodes represent metacharacters. Each leaf and node has a corresponding circuit block that contains combinational and sequential logic. An edge in the tree is a 1-bit bidirectional link between circuit blocks.

Figure 5.35 shows a simple example for the regular expression a+b$*$. State bits appear in the leaves of this tree. Metacharacters contain combinational logic that calculates the next state values for each state bit.

Sidhu and Prasanna implement this circuit directly on FPGAs. Cairn, on the other hand, interprets the circuit as a bit-level functional model of an NFA pattern matching application. This model is used to construct a Cairn actor. The NFA state bits become the actor's internal state variables. The NFA state transition logic is written out as a set of parallel state update functions in the actor's firing rule. The match output function writes a Boolean token to the actor's output port.

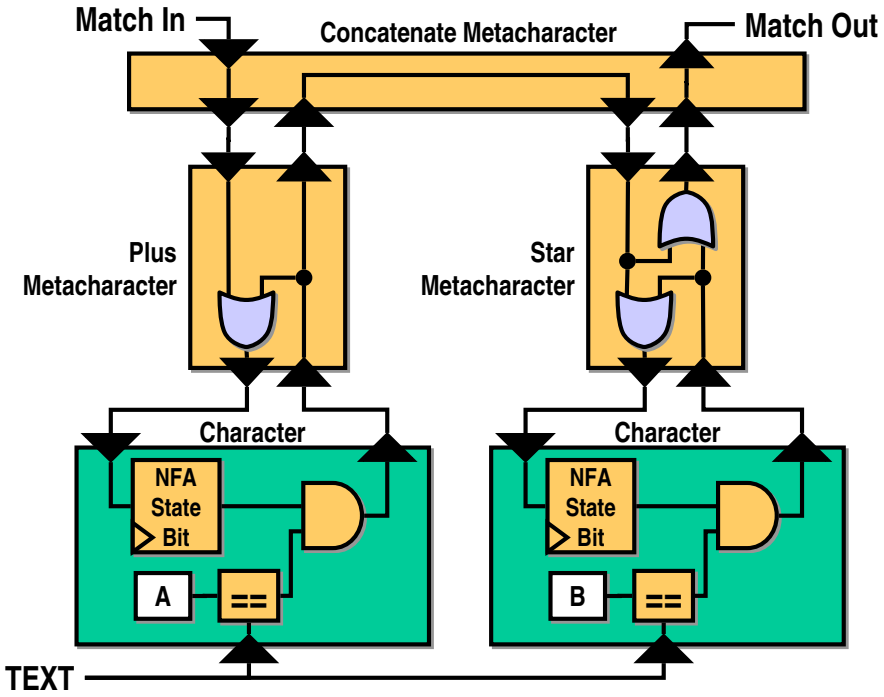


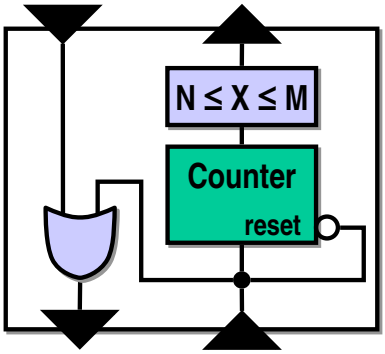
Figure 5.35: Bit-Level NFA Model for a+b$*$

Cairn's model transform also extends the original Sidhu and Prasanna work by defining additional metacharacters, as shown in Figure 5.36. Curly braces add additional state to the NFA in the form of counters that record how many times a subexpression matches. If the counter input is true, the subexpression matches in the current cycle and the counter is incremented. An OR gate also enables the subexpression to attempt another match in the next cycle. The match is propagated up the tree only if the count falls within the given range. If the counter input is false, the number of consecutive matches seen is reset to zero.

Additional metacharacters are special cases of the metacharacters in the figure. The $\{N\}$ metacharacter (match a subexpression exactly N times) is $\{N, M\}$ with N equal to M. Likewise, $\{N,\}$ (match N or more times) uses $\{N, M\}$ with M equal to infinity. The ? metacharacter (match 0 or 1 times) uses $\{0, M\}$ with M equal to 1.

This bit-level functional model is an ideal formulation for the Cairn methodology because it makes all of the concurrency in the NFA explicit. The logic gates and bit-vector wires express datatype-level concurrency. The next state logic can be interpreted as a dataflow graph that describes data-level concurrency. The streaming behavior of the circuit is an example of process-level concurrency. In each iteration, the NFA consumes one character from the incoming text stream and

$\{N, M\}$
match subexpression between N and M times, inclusive



$\{0, M\}$
match subexpression 0 through M times, inclusive

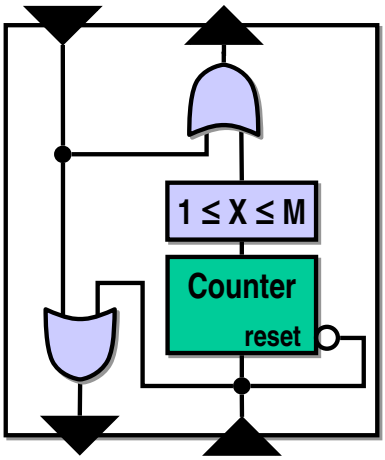


Figure 5.36: Additional Counting Metacharacters

computes the circuit's outputs and next state values. In Chapter 7, it will be shown how this explicit knowledge of the application's concurrency can be used to design a custom Sub-RISC processing element for regular expression matching.

5.7 Summary

This chapter covered the abstractions and processes found at the top of the Cairn design abstraction chart: high-level models of computation, domain specific languages, model transforms, and an underlying application abstraction that captures heterogeneous concurrency. First, network processing was introduced as a design driver. This is an application domain where implementing concurrency correctly is necessary to achieve performance goals. There are multiple granularities of concurrency that must be considered. Network packets have special bit-level data types and use mathematical operations that are different from the standard integer types and ALU operations found in general-purpose processors. Multiple header fields can be processed in parallel to exploit data-level concurrency. Independent packet streams are an instance of process-level concurrency.

Network processing is also a multifaceted, heterogeneous application domain. Different algorithms with different concurrency requirements are applied to packet header processing and packet body processing. Real applications need to do both of these things well, and the cross-cutting concerns between the application facets must be carefully considered.

Furthermore, since network protocols and services change frequently, and bandwidth requirements continue to increase, programmability is a necessity. ASIC designs cannot adapt to these rapid changes in the application domain, making them a risky and expensive solution. Existing design flows that target programmable network processors cannot handle the problems caused by the concurrency implementation gap. This domain is therefore ideal to showcase the Cairn multiple abstraction approach.

The second part of this chapter introduced an application abstraction for capturing heterogeneous concurrency. Two opposing goals had to be balanced: First was to provide intuitive, formal, and useful abstractions that simultaneously assist and restrict programmers. Second was to support a broad range of heterogeneous applications with a diversity of concurrency requirements. The Ptolemy II system promises a good solution to this problem. Programmers use different models of computation for different application facets. These are the restrictive abstractions that make it easy to model specific styles of concurrency. All of the models of computation are implemented on top of a common kernel abstraction that captures diverse concurrency requirements in a common form.

Unfortunately, Ptolemy's focus on simulation on workstations makes it inappropriate for targeting implementation on heterogeneous multiprocessors. Cairn fixes this problem by strengthening Ptolemy's abstractions. Key features such as actor-oriented design, domain polymorphism, and hierarchical heterogeneity are preserved. Cairn makes a precise specification of data-level and datatype-level concurrency by replacing Ptolemy's Java actor description language with a new language that features explicit bit-level types. Abstract interactions between high-level application components are made concrete with *transfers*, a communication primitive better suited for Sub-RISC multiprocessors than Java function calls. These modifications allow an application's concurrency to be expressed in a way that is ideal for implementation on a Sub-RISC system.

This final implementation step is the topic of the next chapter. Designers begin this process with a proper implementation-independent model of the application's requirements. These are the computations that, in an ideal world, the architecture should directly support for high performance. The multiprocessor abstraction presented in chapter 4 exports the architecture's actual capabilities for process-level, data-level and datatype-level concurrency. In the mapping process, designers compare the application's requirements against the architecture's capabilities to characterize the concurrency implementation gap. From there, a concrete implementation of the heterogeneous concurrent application that runs on the multiprocessor platform can be obtained.

Chapter 6

Connecting Applications and Architectures

The third and final component of the Cairn methodology is mapping and implementation. In this design stage, an application model is turned into executable software for a target multiprocessor architecture to create a complete system implementation. The abstractions and processes used in this stage are found in the middle of the design abstraction chart between the application model and the architecture model (Figure 6.1). This design stage is the central part of the Y-chart design methodology (Figure 6.2).

Mapping begins with a complete model of the application and a complete model of the target architecture. Chapter 4 described the architecture side of the Cairn methodology. In this branch of the Y-chart, architects assembled multiprocessor machines out of Sub-RISC processing elements. Cairn's architecture abstraction describes the capabilities of these multiprocessor machines. Individual PEs run small jump-free programs that compute mathematical projections and update internal state. Interactions between PEs are governed by the signal-with-data abstraction. Computation begins at a PE when it receives a message. The results of the computation form new messages that are sent to other PEs in the multiprocessor. Computation stops at the first PE and continues at the destinations of the new messages.

This basic style of inter-processor communication and control allows architects to build hardware support for application-specific process-level concurrency at the multiprocessor level. This complements the ability to customize individual PEs to support the application's data-level and datatype-level concurrency. Simple PEs and simple interconnect ensure a low-cost, lightweight

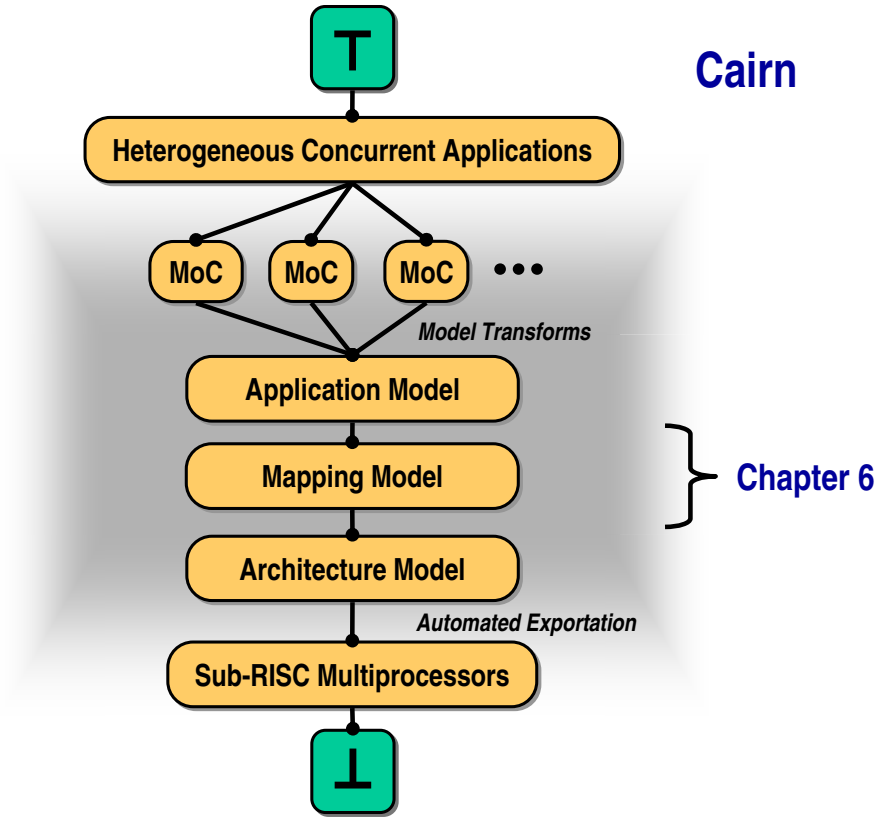


Figure 6.1: Mapping Portion of the Cairn Design Abstraction Chart

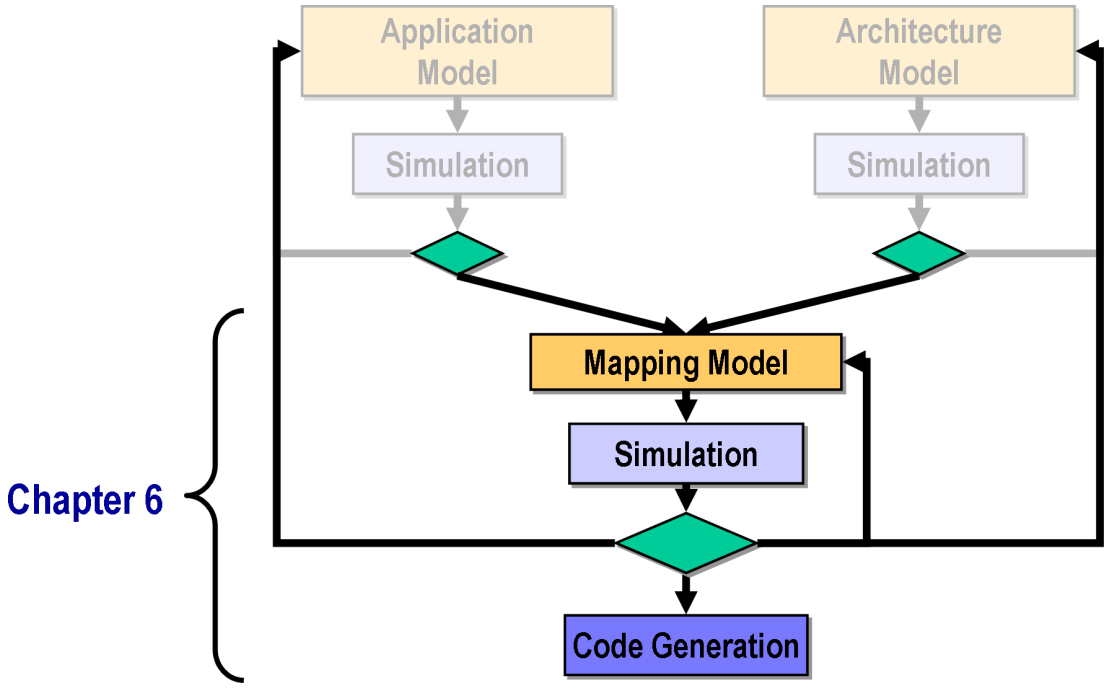


Figure 6.2: Mapping and Implementation Portion of the Y-Chart Design Flow

architecture. Designers only pay for complexity when they purposefully introduce complexity.

Chapter 5 described the application side of the Cairn methodology. This branch of the Y-chart began with high-level architecture-independent application models based on models of computation and domain-specific languages. These formalisms provide domain experts with intuitive abstractions for complex concurrency issues. This makes it possible to quickly construct precise functional models without managing the interactions between components by hand. Programmers focus on issues important to the application domain and defer implementation decisions.

Model transforms automatically converted the abstract high-level models into concrete Cairn Kernel models. The transformed models describe all of the computations necessary to run the application. The Cairn Kernel abstraction was specifically designed to describe these computations in a way that would simplify implementation on a Sub-RISC multiprocessor. Inside individual application actors, explicit datatype-level and data-level computations are described. The complexity of these computations is limited to mathematical projections and state updates. This is designed to match what a Sub-RISC processor can compute. Interactions between actors is governed by the transfer-passing abstraction. The way that actors pass the flow of control amongst themselves is designed to match the signal-with-data messages that Sub-RISC PEs use to interact in a multiprocessor.

In the mapping stage of the Cairn methodology, designers match up the application's computations with the architecture's computational resources. The goal is to use the concrete Cairn Kernel model directly for implementation. If the architecture is programmed to perform exactly the computations specified by this model, then the result will be a functionally correct implementation of the application.

During the mapping process, designers distribute the application's computations across the multiprocessor architecture. This requires designers to make several types of design decisions. First, designers must choose a distribution of the application's computations that exploits the architecture's capabilities for process-level, data-level, and datatype-level concurrency. This is necessary to obtain good performance.

Second, designers must choose how to deal with the concurrency implementation gap. A simple distribution of the application's computations onto the architecture's resources is not possible when mismatches exist between the two. Additional computations like those described in Section 2.1.3 must be used to work around the mismatches. There are many options for accomplishing this, requiring more design decisions to be made.

Cairn seeks to aid designers in performing mapping quickly and accurately. Section 6.1 describes a disciplined mapping methodology wherein designers create explicit *mapping models* that capture the design decisions made during the mapping process. Section 6.2 covers the syntax and semantics of Cairn mapping models. The mapping abstraction used to create these models is the third of the three major abstractions that make up the Cairn methodology, and is a core contribution of this dissertation.

Finally, Section 6.3 describes the processes that make up the bottom of the Y-chart. These processes transform a mapping model into a complete implementation in the form of executable code for the target architecture. Programmers do not have to rewrite their application models in a low-level programming language in order to program the architecture. This would sap productivity and introduce errors. Instead, the similarities between the application and architecture abstractions are leveraged to allow application components to be directly compiled into executable machine code for processing elements. Implementations are thus correct by construction. Designers can run this code on a physical realization of the multiprocessor architecture, or hardware-software co-simulation can be performed as described in Section 6.4.

A fast and accurate mapping methodology ensures that the iterative Y-chart design-space exploration flow is effective. Designers can quickly evaluate the positive and negative qualities of a design, make changes, and repeat until a system that has the desired performance is found.

6.1 A Disciplined Approach to Mapping

A mapping is an assignment of the computations required by a particular application onto the computational resources of a particular architecture. For any given application and any given architecture, there are many possible mappings that can be made. Each will exhibit different characteristics. A correct mapping will preserve the functionality of the application. A correct and efficient mapping will meet the functional requirements as well as the performance requirements that designers seek.

Mapping is non-trivial because of the concurrency implementation gap. Architectures are seldom if ever an exact match for the application's computations. Differences can occur on every level of granularity of concurrency.

An example at the process level is that the number of PEs in the architecture may not match the number of application processes. Otherwise, simple one-to-one assignments of application computations onto architectural resources could be made. If multiple processes must share a PE, then designers must consider factors such as the speed of the PEs and the amount of memory available for storing code. A mapping must not overload the PEs and must still satisfy performance requirements.

Examples of data-level and datatype-level mismatches occur frequently in heterogeneous architectures. Different PE datapaths have different capabilities for data-level concurrency. If a PE cannot exploit all of the data-level concurrency in a process, then some operations must be sequentialized and performance goes down.

At the datatype level, different PEs support different arithmetic and logical operations. For example, some PEs may support floating-point arithmetic while others may not. Some PEs may be designed to accelerate a particular application kernel and may not possess a broad enough set of operations to perform other computations. These mismatches place further constraints on the mapping problem.

Cross-cutting concerns are common as well. A pair of PEs that seem like ideal candidates for parallel application processes from a data-level and datatype-level perspective may not be feasible candidates from a process-level perspective. The on-chip network may not be able to provide the required level of communications between the PEs.

These and other concerns complicate the mapping problem. In practice, mapping is a process wherein designers search for good compromises and workarounds to resolve the concurrency implementation gap mismatches.

In previous programming methodologies, mapping was not an explicit step in the design pro-

cess. Mapping design decisions were something programmers made implicitly in their heads as part of the task of writing separate programs for all of the architecture's PEs. Programmers formulated a mental model of the application's concurrency. Experience and intuition gave clues on how to partition the application in such a way as to leverage the architecture's capabilities and exploit the application's opportunities for parallelism.

Mapping decisions are not tangibly notated anywhere in these methodologies. Instead they are built into the code for the processing elements. This results in an intermixing of the application's requirements with architecture-specific implementation decisions. Some of the code that programmers write implements actual application functionality, while other parts implement workarounds for concurrency implementation gap mismatches. At best, designers will create comments or a whiteboard drawing that explains how and why the application is divided up the way that it is.

This is not acceptable in an iterative design space exploration methodology. When things go wrong, designers have a difficult time determining the root causes and making corrections. Is poor performance the result of an application-level issue, or because of poor mapping design decisions? It is impossible to separate these things. If designers want to experiment with different mapping strategies, a great deal of code needs to be rewritten, including application-level code. Programmers must delve into code that contains a mixture of application computations and architecture-specific implementation computations. These parts must be carefully separated, subjected to the desired modifications, and then carefully reassembled. This is slow and introduces errors. There is no guarantee that a modification to the mapping strategy has not introduced functional errors into the application. Getting even one correct implementation is hard enough. Designers seldom have time to search for an efficient mapping.

6.1.1 Explicit Mapping Models

Cairn solves this problem by making the mapping process a formal step in the design methodology. Designers enter the mapping stage of the design flow with separate models of the application and the architecture. To create a mapping a third model is made: an explicit *mapping model*. The mapping model describes how components from the application model (the application's required computations) are assigned to programmable elements in the architecture. Explicit mapping models capture implementation decisions in a tangible, modifiable form that is separate from the application model and the architecture model. This elevates the process of mapping from an ad hoc design step to a disciplined methodology.

A mapping model captures all of the design decisions that are made during the mapping process. First, it captures the assignment of the application's computations onto the target architecture's computational resources. Second, it captures the extra computations that need to be implemented on the architecture to work around mismatches due to the concurrency implementation gap.

In the ideal case, only the first of these things would be necessary. The computations found in the original application model alone are sufficient to correctly implement the application's required functionality. Due to the concurrency implementation gap, the architecture may not support these computations directly. The previous section gave examples of the types of mismatches that occur. Designers work around these mismatches by assigning additional computations to the PEs in the architecture.

If an application component requires floating-point arithmetic but the architecture provides only integer arithmetic, a software floating-point math library can be used to resolve the difference. Mismatches in on-chip network connectivity can be resolved by using network-on-chip protocols. These protocols are implemented as software that runs on the PEs to mediate access to the on-chip network hardware. Protocols can provide connectivity between PEs that are not directly connected by buses, for example by routing communications through intermediate PEs. Protocols can also provide a different communications semantics on top of the architecture's native semantics.

Cairn's mapping models describe these extra computations in addition to the computations that originated in the application model. These computations are the result of architecture-specific implementation decisions. Designers have many different options for solving concurrency implementation gap mismatches. The mapping model is the correct place to capture all of these design decisions.

6.1.2 Benefits of Explicit Mapping Models

Explicit mapping models make the mapping process fast and accurate. While working with a mapping model, designers focus on implementation concerns. This means making assignments between the application and the architecture and deciding how to solve concurrency implementation gap issues. The application model is not changed during the mapping process, so there is always a consistent model of the application's requirements. Designers need not be concerned about inadvertently changing the functionality of the application. Different mappings will have different performance results, but will implement the same functionality.

Explicit mapping models help designers understand the concurrency implementation gap be-

tween a given application and architecture. A mapping model shows what portion of the system software implements actual application functionality versus what portion exists only to solve concurrency implementation gap mismatches. This information can be used to calculate the cost of the gap in terms of code size and execution time. This characterization of the gap is valuable data for design space exploration. Designers can quickly determine what aspects of the system should be modified for maximum gains in performance.

Explicit mapping models also make it possible to use the three separate feedback paths in the Y-chart design space exploration methodology. Ideas for improvements to the system can be executed with only a minimum of changes. To experiment with different mapping strategies, designers create a new mapping model or make changes to an existing mapping model for the same application and architecture pair. The application and architecture models themselves are not modified.

Designers can experiment with mapping several different applications onto the same architecture, or mapping the same application onto several different architectures. In each case, designers create a new mapping model for each application/architecture pair.

To make changes to the application functionality, designers modify only the original application model. This model describes the implementation-independent aspects of the application. It is not necessary to deal with code that combines application computations and architecture-specific implementation computations.

It is true that modifications to an application model or architecture model may invalidate assignments in the corresponding mapping model. A mapping model is dependent on a given application model and a given architecture model. For example, a PE may be removed entirely from the architecture. The mapping model will have to be changed so that all of the computations assigned to that PE are moved to other PEs. The important fact is that these dependencies are represented explicitly. When such a change is made, designers can look at the mapping model to see exactly what implementation decisions must be updated.

6.1.3 Definition of a Mapping Abstraction

Cairn provides abstractions to help designers create application models and architecture models. The mapping stage of the design methodology is similarly constructed. Designers use a *mapping abstraction* to create mapping models.

The mapping abstraction appears in the middle of the design abstraction chart, between the application abstraction and the architecture abstraction. Based on this position, it is correctly expected

that the mapping abstraction will be more abstract than the architecture abstraction but more concrete than the application abstraction. The mapping abstraction gives designers a restricted view of the architecture that exposes only the features that are necessary for solving the mapping problem. The unimportant details of the architecture are hidden. In this way, the mapping abstraction simultaneously assists and restricts the designer by allowing them to focus solely on mapping concerns.

The mapping abstraction is more concrete than the application abstraction because it associates the application's computations with concrete architectural resources. By itself, an application model only indicates what computations are required to occur. A mapping model adds to this the details of what programmable hardware is supposed to perform the computations. This is closer to a realizable implementation than the application model alone.

The mapping abstraction is also the formalism that gives mapping models a functional semantics. This makes mapping models more than just informal sketches. The constructs in the mapping model have a consistent meaning that can be automatically transformed into software for the target architecture. A complete mapping model describes all of the software for every PE in the multiprocessor system.

6.1.4 Manual Versus Automated Mapping

Creating mapping models is a manual process in the Cairn methodology. The contribution of this dissertation focuses on formulating the mapping problem correctly and providing designers with the right tools and methodologies for creating correct solutions to the mapping problem. Automated techniques for searching the solution space to find optimal mappings are beyond the scope of the "tall, skinny" approach.

Related work in multiprocessor scheduling can be applied to this problem. Scheduling application tasks is comparable to assigning application computations to processing elements. An approach based on mixed integer linear programming (MILP) by Bender [10] finds a mapping that optimizes the total execution time of the application. MILP is a useful technique in general because it allows for complex objective functions. Plishker et al. [98] include code size constraints in their objective function to ensure that a mapping does not overfill any of the PE instruction memories.

Chakuri [19] uses approximation algorithms to solve mapping problems formulated as constraint systems. These techniques are useful when the complexity of the application and the architecture leads to constraint systems that are too big to be solved exactly by MILP in a reasonable amount of time.

A separate category of techniques focuses on dynamic solutions to the mapping problem. A dynamic mapping permits application computations to migrate to different architectural resources at run time. For example, Ghosh et al. [41] dynamically assign application computations to processing elements in such a way as to satisfy hard real time-constraints in the presence of hardware faults.

Cairn allows only static mappings. All dynamic behaviors must be captured by the application model. This is an occasion to use an additional application facet and model of computation to describe the dynamic behavior of the application. The dynamic scheduling algorithm thus becomes part of the application description. When static mapping is applied to this new application model, the correct run-time behaviors of the application get implemented on the architecture.

Any technique for static mapping can be applied to Cairn's mapping methodology. The only requirement is that the mapping abstraction must be augmented to expose the characteristics of the architecture that the mapping algorithm needs to use. For example, the approach by Bender requires knowledge of the time costs for using the various on-chip network communication links, and the time cost of running each application process on each kind of processing element. This data can be obtained through analysis of the architecture model.

6.2 Cairn Mapping Models and the Mapping Abstraction

This section describes the syntax and semantics of Cairn mapping models, and how the mapping abstraction simplifies the task of creating and modifying mapping models.

A mapping model describes the software that is to run on each programmable element in the architecture. This includes computations from the application model as well as additional computations that designers decide are necessary to solve concurrency implementation gap mismatches. The first step in creating a mapping model is to assign these computations to architectural PEs. Designers do not have to write code to make these assignments. The mapping model describes these assignments in a declarative fashion.

This is the coarse-grained part of the mapping process. Designers are describing at a high level how computations are distributed across the architecture. They are not describing how the computations assigned to each PE are actually implemented. That task is the second step in creating a mapping model.

Fine-grained mapping assignments declare how each assigned computation uses PE resources. This includes how the computation's internal state is assigned to the PE's state elements, and how the computation's inputs and outputs are assigned to the PE's on-chip network I/O ports. Like the

coarse-grained assignments, a mapping model captures fine-grained assignments in a declarative fashion. Designers are not writing code by hand during this process.

Fine-grained mapping assignments do not extend to the level of selecting the PE operations that will be used to implement a given computation. Designers stop after performing state and I/O resource allocation. The process of converting computations into executable machine code is done automatically by a compiler-like code generation tool. This and other processes for converting mapping models into executable code are described in Section 6.3.

Programmers who are familiar with existing compilers are used to having fine-grained mapping decisions made for them. For example, C compilers automatically map state variables to the processor's memory. Mapping to a Sub-RISC processor is a more complicated problem because Sub-RISC processors do not follow the architectural design patterns that typical compilers expect. For example, a Sub-RISC datapath may have multiple register files and an unusual memory hierarchy. The I/O resources are also open to customization to support the application's process-level concurrency. Additionally, Sub-RISC machines change frequently during design space exploration, so a compiler cannot be built with a priori knowledge of the datapath architecture.

A priori knowledge of the application's requirements cannot be counted on either. Heterogeneous applications may require different fine-grained mapping strategies for different facets of the application.

For these reasons, Cairn requires designers to make fine-grained mapping decisions explicitly. Tools for automatically performing fine-grained mapping are considered to be in the same category as those that perform exploration of the mapping design space, as described in Section 6.1.4. They are possible, but beyond the scope of this dissertation.

Cairn's mapping abstraction helps designers create mapping models. It works by providing restricted *views* of both the application model and the architecture model. These views expose the characteristics of the application and architecture that are important for making mapping assignments, and hide the distracting details that are not important during the mapping process.

6.2.1 The Mapping View of the Architecture

Cairn's mapping abstraction presents designers with a two-level hierarchical view of the architecture. The first level of the hierarchy shows the PEs in the architecture and the communication links between the PEs. These blocks are resources onto which computations can be mapped.

The second level of the hierarchy shows designers the details inside the PEs that are relevant

to the mapping process. This includes the PE's on-chip network ports and the PE's state elements. The structure of the PE's datapath is hidden from the designer in this view.

Figure 6.3 shows an example. Figure 6.3(a) is an example target architecture. This model was constructed using the methodology described in Chapter 4. There are three Sub-RISC PEs that communicate using point-to-point communication links. PE A can send signal-with-data messages to PE B and PE C. PEs B and C can each send messages back to A, but can not communicate directly with each other. This figure does not show the internal architecture of the PEs, but this information is assumed to be available before mapping begins.

Figure 6.3(b) shows the restricted view of the architecture that is provided by the mapping abstraction. The top level of this view mirrors the top level of the multiprocessor architecture model. The nodes are PEs and the edges represent signal-with-data communication links.

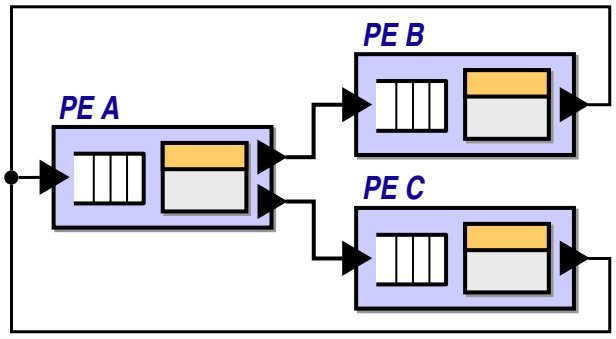
The second level of this view shows designers an abstraction of the contents of each PE. PE C is shown expanded in the figure. Designers are shown the datapath's input ports, output ports, and state elements. The way that the architecture bundles outputs together to create outgoing signal-with-data messages is shown. Similarly, the way that inputs are driven from incoming signal-with-data messages is shown. These architectural features are taken from the original architecture model.

This view of the PE is roughly similar to that shown in Figure 4.17 in terms of how it shows the PEs network-on-chip connections. However, this is not an architectural model of the PE. No control structures appear in this view. None of the datapath elements other than state elements are shown, and none of the datapath interconnect is shown.

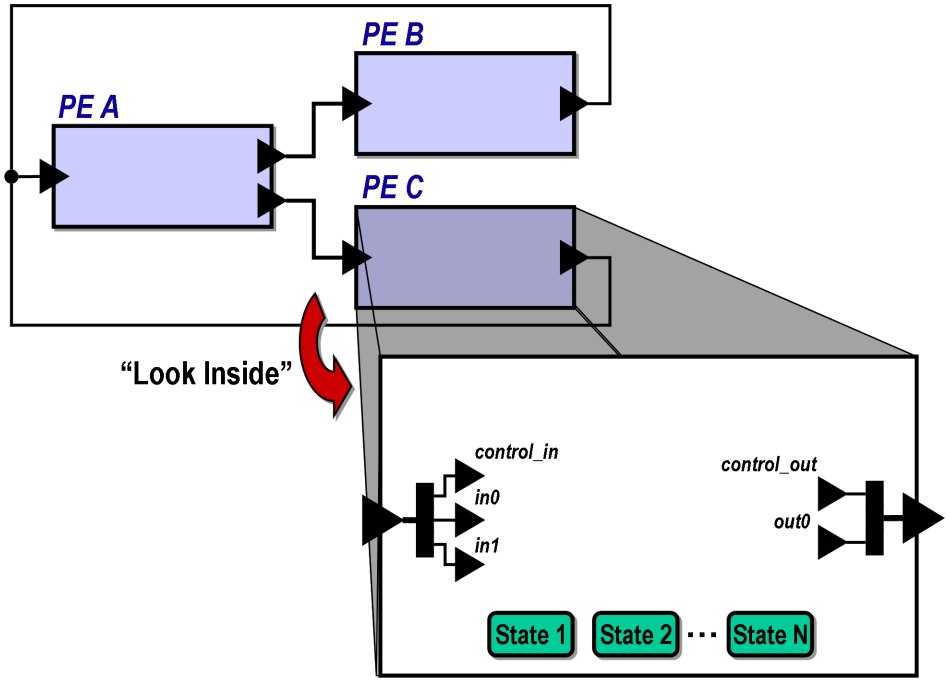
The two levels of this hierarchical view of the architecture are meant to be used during the coarse-grained and fine-grained parts of the mapping process. The top level of the hierarchy shows the parts of the architecture that are important for making coarse-grained mapping decisions. PEs are computational resources onto which application components can be assigned. The edges between PEs show the available on-chip network links that application components can use for communications. The second level of the hierarchy shows the details of each PE that are important for making fine-grained mapping decisions. This includes the architectural state and I/O resources onto which application state, inputs and outputs can be assigned.

6.2.2 The Mapping View of the Application

The mapping abstraction provides a view of the application based on the transformed Cairn Kernel model. The Cairn Kernel model describes the application as a set of components that com-



(a) Multiprocessor Architecture Model



(b) Mapping View of the Architecture

Figure 6.3: The Abstraction of the Architecture Used During Mapping

municate using transfer messages. Each component contains one or more firing rules that compute mathematical projections and update internal actor state. The outputs of a firing rule are interpreted as zero or more new transfer messages that are sent to other actors.

The mapping abstraction's view of the application model shows designers the parts of the application model that are relevant for the mapping process. Like the architecture view, the application view has two parts. The first part contains the information necessary for making coarse-grained mapping decisions. This consists solely of the components found in the Cairn Kernel model and the connections between them. The contents of the components are not important at this stage. Designers are only interested in what computations can be done in parallel and what kind of communications are required between these parallel computations.

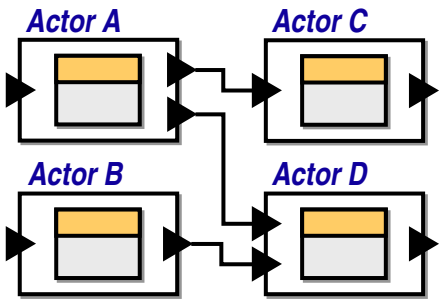
The second part of the mapping abstraction's view of the application model gives designers the information necessary for making fine-grained mapping decisions. Designers are shown a portion of the contents of the application components. This includes the details of how incoming transfer messages are divided to produce inputs for firing rules, and how the outputs of firing rules are bundled together to produce outgoing transfer messages. The final part of each application component that is exposed to the designer is the set of internal state variables. Designers are not shown the contents of the firing rules.

Figure 6.4 shows an example. Figure 6.4(a) is the transformed Cairn Kernel model of the application. This model was constructed using the methodology described in Chapter 5. Programmers used multiple models of computation to make a hierarchically heterogeneous model of the application. Model transforms converted the high-level semantics provided by these models of computation into the concrete Cairn Kernel model.

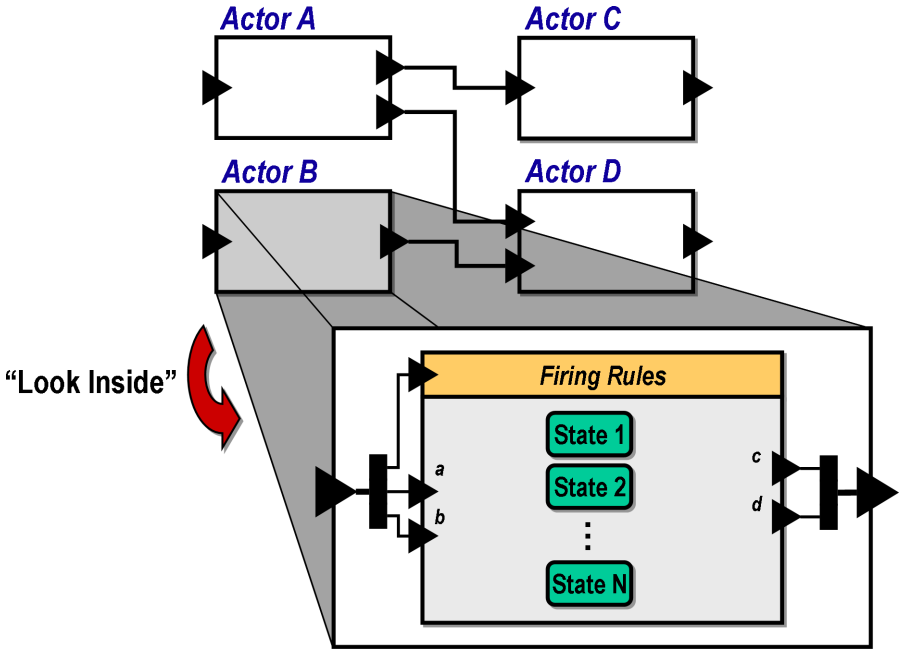
Figure 6.4(b) shows the mapping abstraction's restricted view of the transformed application model. At the top level, this view shows the application's components and the transfer-passing communications links between them. Inside of each component, designers see the details that are necessary for making fine-grained mapping assignments. This view of the contents of the components is similar to that shown in Figure 5.8.

6.2.3 Using the Application and Architecture Views to Create Mapping Models

The mapping abstraction's view of the architecture serves as a template for creating mapping models. Designers create a mapping model by filling in this template. Components from the application view are mapped to PEs by copying them into the template. They are inserted into the second



(a) Cairn Kernel Application Model



(b) Mapping View of the Application

Figure 6.4: The Abstraction of the Application Used During Mapping

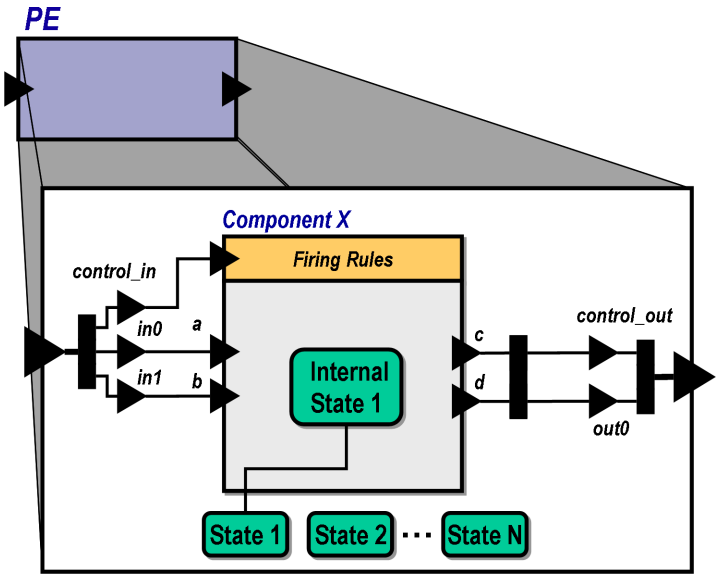


Figure 6.5: Filling In an Architectural Template to Create a Mapping Model

level of the template (i.e. the parts that show the contents of the individual PEs). Designers then add edges between the application components and the PE's ports and state elements. The edges describe how the assigned computations use the PE's resources.

Figure 6.5 shows this graphically. By adding the block labeled *X* to the model of the contents of the PE, the designer has mapped a computation to the PE. Component *X's* firing rules will eventually become software programs that the PE executes in response to incoming signal-with-data messages.

Edges in the mapping model represent constraints between software and hardware. The edges between *X's* input ports and the PE's input ports show how *X* gets its inputs from incoming signal-with-data messages. Similarly, the edges connected to *X's* output ports describe how the results of *X* are bundled together to create new outgoing signal-with-data messages. Component *X* has internal state represented by the block labeled *Internal State 1*. The edge between this block and the PE's *State 1* block means that the computation's state has been assigned to that particular architectural state element.

Later, when firing rules are converted into executable programs for the PE, these constraints will tell the compilation tool what architectural ports to read inputs from and write outputs to, and what architectural state elements to access to update the application's state variables.

This is a high-level description of how mapping models are constructed. A complete description of how designers create mapping assignments and the syntax and semantics of the mapping models appears in the following subsections.

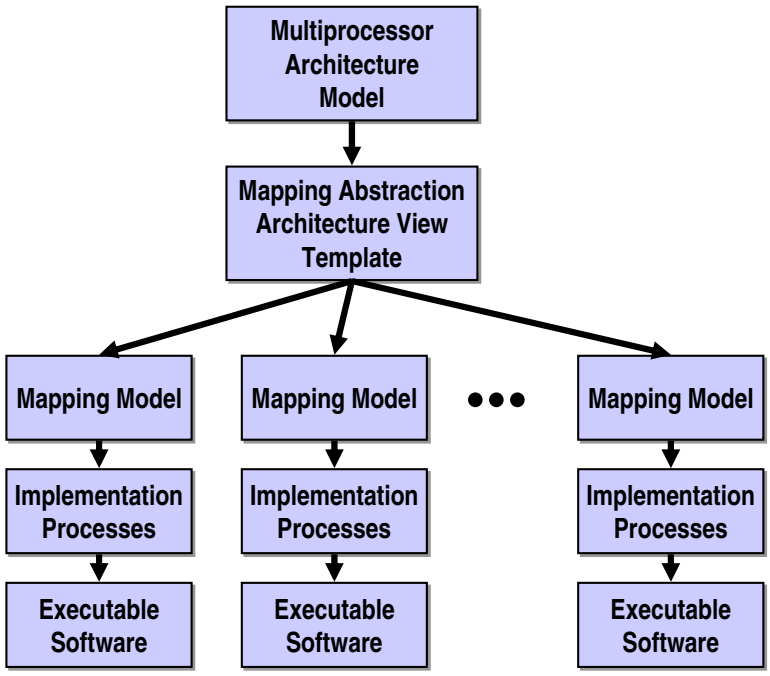


Figure 6.6: Using the Architecture View Template to Create Multiple Mapping Models

To experiment with different mapping strategies, designers create multiple mapping models starting from the same template. This design flow is shown in Figure 6.6. The same processes are applied to each alternative mapping model to obtain executable software implementations. This back-end part of the mapping methodology will be described in Section 6.3.

Making Coarse-Grained Mapping Assignments

Cairn provides a graphical user interface for making coarse-grained mapping assignments. This interface allows designers to interact with the application view and the mapping model in a side-by-side fashion. The mapping model starts out as a copy of the architecture view. This is the template that will be filled in.

A coarse-grained mapping assignment is made by dragging and dropping components from the application view onto processing elements in the mapping model. When this is done, the application component is copied into the mapping model inside of the target PE.

This is demonstrated in Figures 6.7 and 6.8. Figure 6.7 shows the side-by-side presentation of the application view and the mapping model. Actor *B* from the application view is mapped to *PE 0* by dropping it onto the PE.

Figure 6.8 shows the results of this action. Actor *B* now appears inside the contents of *PE*

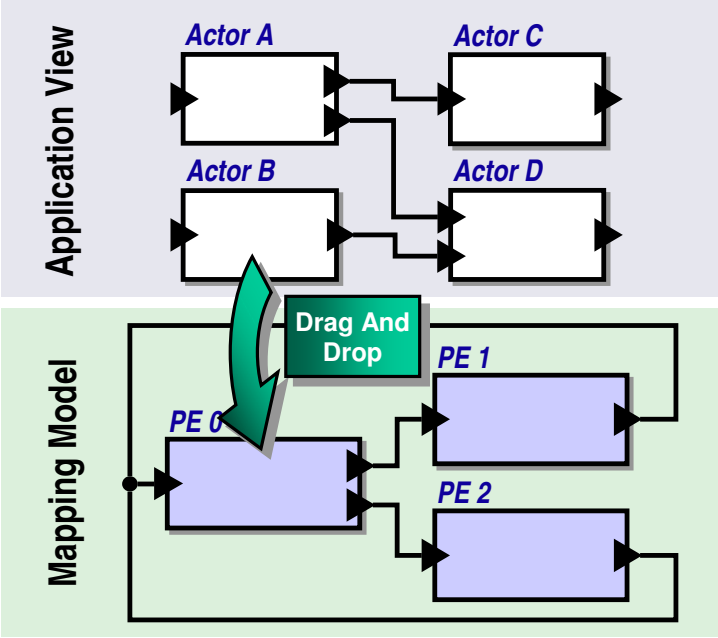


Figure 6.7: Using Drag-and-Drop to Create a Coarse-Grained Mapping Assignment

0. This view of actor *B* shows the characteristics necessary for making fine-grained mapping assignments. The actors inputs, outputs, and internal state variables are shown. Outputs *c* and *d* are connected to a narrow black box adjacent to the actor. This indicates that actor *B* intends to produce outgoing transfer messages by combining the values produced on these ports.

Actor *B* also appears highlighted in the application view. This is a visual cue to the designer that actor *B* has been mapped to a PE in the architecture. The coarse-grained portion of the mapping process is complete when all of the application components have been assigned to PEs.

Designers can remove a mapping assignment by clicking on a component in the application view and pressing the delete key. The corresponding component is automatically removed from the mapping model, and the highlight disappears. The component can then be assigned to a different PE if the designer chooses.

One-to-one and many-to-one assignments are supported between the application and the architecture. A one-to-one assignment means that a single application component has been mapped to a PE. The PE is responsible for implementing only the computations defined by that component. A many-to-one assignment means that two or more application components have been mapped to a single PE. In this case, the PE must implement the union of the computations defined by these components.

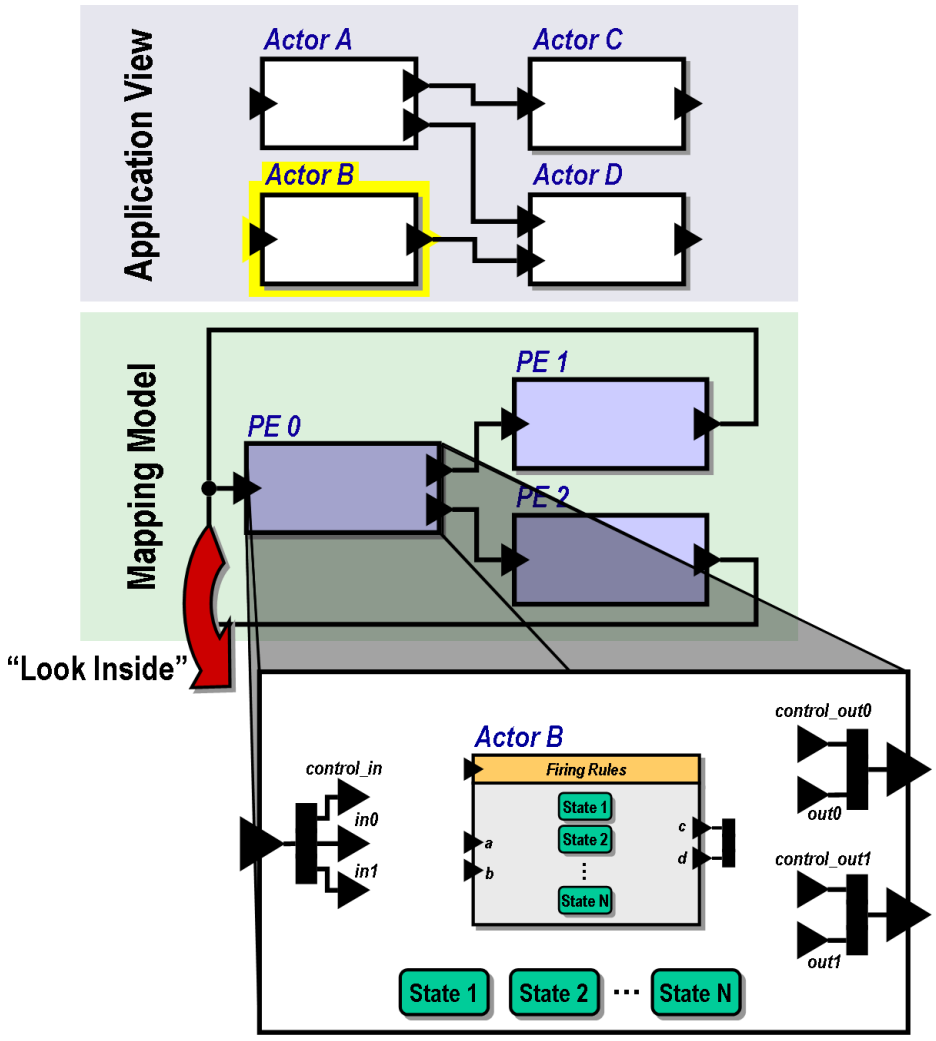


Figure 6.8: Drag-and-Drop Action Adds a Component to the Mapping Model

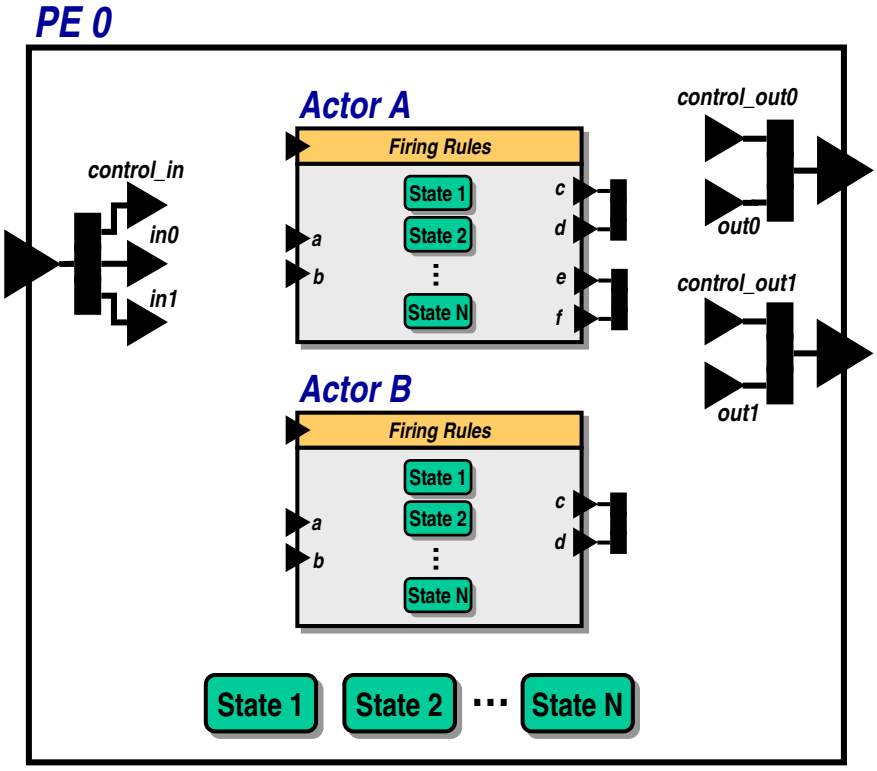


Figure 6.9: Mapping Two Application Components to the Same PE

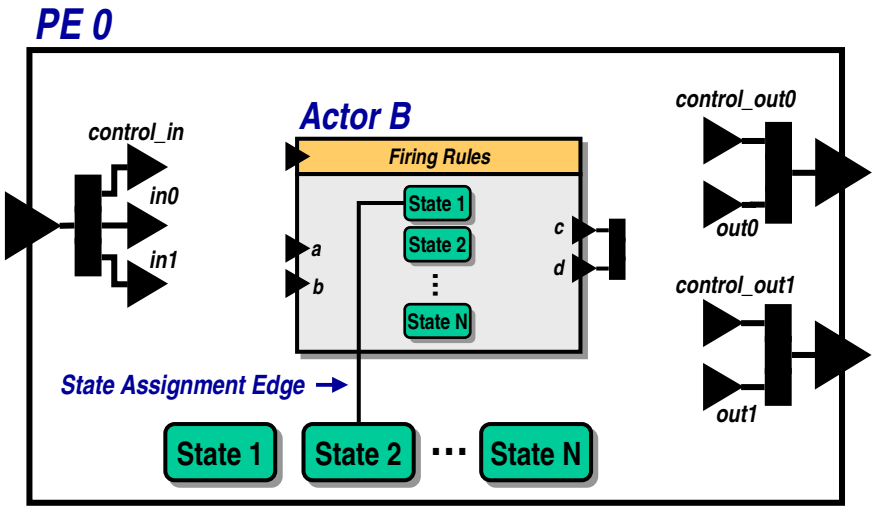


Figure 6.10: State Assignment Edges in the Mapping Model

Mapping several application components to the same processing element may be a compromise that affects the performance of the system. Consider Figure 6.9, which shows the contents of *PE 0* after both actor *A* and actor *B* have been mapped to that PE. If *A* and *B* are parallel processes that can receive simultaneous transfer messages, then this mapping does not fully exploit the application's process-level concurrency. *PE 0* responds to incoming signal-with-data messages sequentially. It must interleave computations for actor *A* with those for actor *B*. This is not as fast as a mapping that assigns actors *A* and *B* to separate PEs, which would allow the separate computations to occur in parallel. This alternative mapping may have other consequences, so designers must consider all of the factors when making mapping decisions.

One-to-many mapping assignments (where a single application component is assigned to several processing elements) are only allowed for application components that contain no state. If a component does contain state, then the one-to-many mapping assignment will cause the state variables to be duplicated. The application model does not expect this and will not contain the computations necessary to keep the distributed state consistent. The result is that the functional behavior of the implementation will be incorrect with respect to the application model.

Making Fine-Grained Mapping Assignments

Designers make fine-grained mapping assignments by drawing edges in the mapping model. There are two kinds of edges that need to be drawn: state assignment edges and I/O assignment edges.

A state assignment edge connects a state variable from a mapped application component to an architectural state resource inside of a PE. Figure 6.10 shows an example of this kind of edge. This edge indicates that the given application state variable is to be mapped into the given architectural state resource. All application state variables must be connected to exactly one state assignment edge. Each architectural state resource can be connected to an arbitrary number of application state variables, provided that the architectural resource is large enough to hold all of the variables.

State assignment edges act as constraints on the code generation tool that produces machine code for PEs. When implementing a firing rule's computations using the PE's operations, the tool must choose operations that read and write from the proper architectural state resource.

I/O assignment edges are used to map an application component's inputs and outputs to a PE's on-chip network ports. When a PE receives a signal-with-data message, it executes a program that computes a firing rule from one of the mapped application components. This program requires that certain input data values be available and guarantees that certain output values will be produced.

The inputs must come from the incoming signal-with-data message. According to Cairn's multiprocessor architecture abstraction, these values will appear on the PE's input ports for the duration of the execution of the program. I/O assignment edges describe which PE input ports provide which program input values.

Similarly, the outputs that the program computes must be bundled together to produce new outgoing signal-with-data messages. Additional I/O assignment edges describe which PE output ports receive which program output values.

Like state assignment edges, I/O assignment edges are constraints for the code generation tool. The tool must choose PE operations that read input values from the proper PE input ports, and write output values to the proper PE output ports.

Figure 6.11 shows example I/O assignment edges for actor *A*. On the input side, the architecture view of *PE 0* shows that incoming signal-with-data messages are broken into three components: *control_in*, *in0* and *in1*. The I/O assignment edges map actor *A's* inputs to these datapath ports.

On the output side, *PE 0* has four output ports: *control_out0*, *out0*, *control_out1* and *out1*. The architecture view shows how these ports are bundled together to form outgoing signal-with-data messages. There are two groups. The *control_out0* and *out0* ports form one group and the *control_out1* and *out1* ports form the other. Each large triangle on the periphery of *PE 0* represents a point-to-point network-on-chip connection that the PE can use to send signal-with-data messages to other PEs in the multiprocessor.

Additional I/O assignment edges map actor *A's* outputs to the four datapath output ports. The

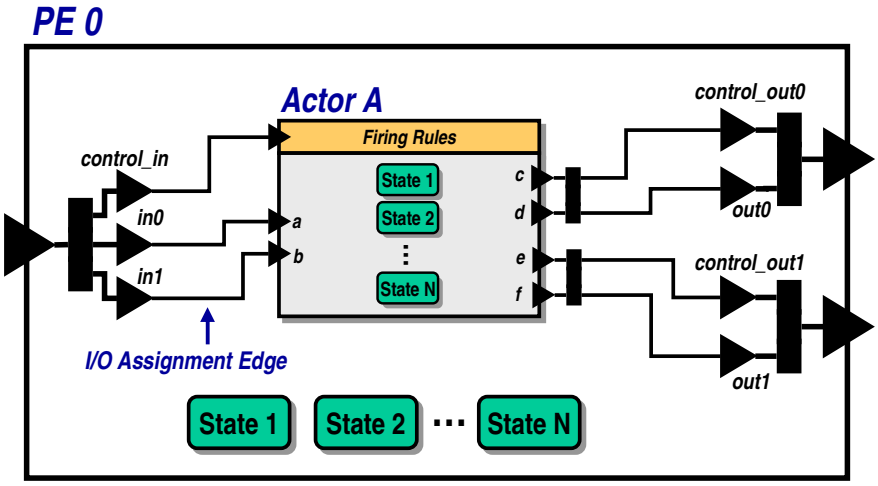


Figure 6.11: I/O Assignment Edges in the Mapping Model

black rectangles adjacent to actor *A's* output ports serve as visual cues that inform the designer how actor *A's* outputs are supposed to be grouped together to form transfer messages. Ports *c* and *d* are meant to form a single message. These must be assigned to datapath output ports that are grouped together to form a single signal-with-data message.

I/O assignment edges describe how the application's transfer messages are implemented using the architecture's signal-with-data communications resources. Figure 6.12 shows a larger example where two communicating application actors are mapped to separate PEs. Actor *A* is mapped to *PE 0*, and actor *B* is mapped to *PE 1*. The edge between actor *A* and actor *B* in the application model indicates that actor *A* sends transfer messages to actor *B*. The mapping model shows how transfer messages from actor *A* to actor *B* are implemented using the interconnect between the PEs.

PE 0 has three output ports (*control_out0*, *out0* and *out1*) that are bundled together to create outgoing signal-with-data messages. *PE 1* has corresponding input ports (*control_in*, *in0* and *in1*). Since *PE 0* and *PE 1* are connected together at the multiprocessor level, these groups of ports will match in terms of number and bit width. Mismatches would have been identified as type errors while the architecture model was being built.

These ports are used to convey the control and data outputs of actor *A* to actor *B*. Actor *A's* output port *c* provides the control portion of the transfer message for actor *B*. The attached I/O assignment edge declares that this value should go into the *control_out0* portion of the signal-with-data message that *PE 0* sends to *PE 1*. Likewise, actor *A* port *d* provides data for actor *B* that goes into the *out0* portion of the signal-with-data message. These values are unbundled at *PE 1* and connected to the appropriate ports on actor *B*. A complete mapping model will have I/O assignment

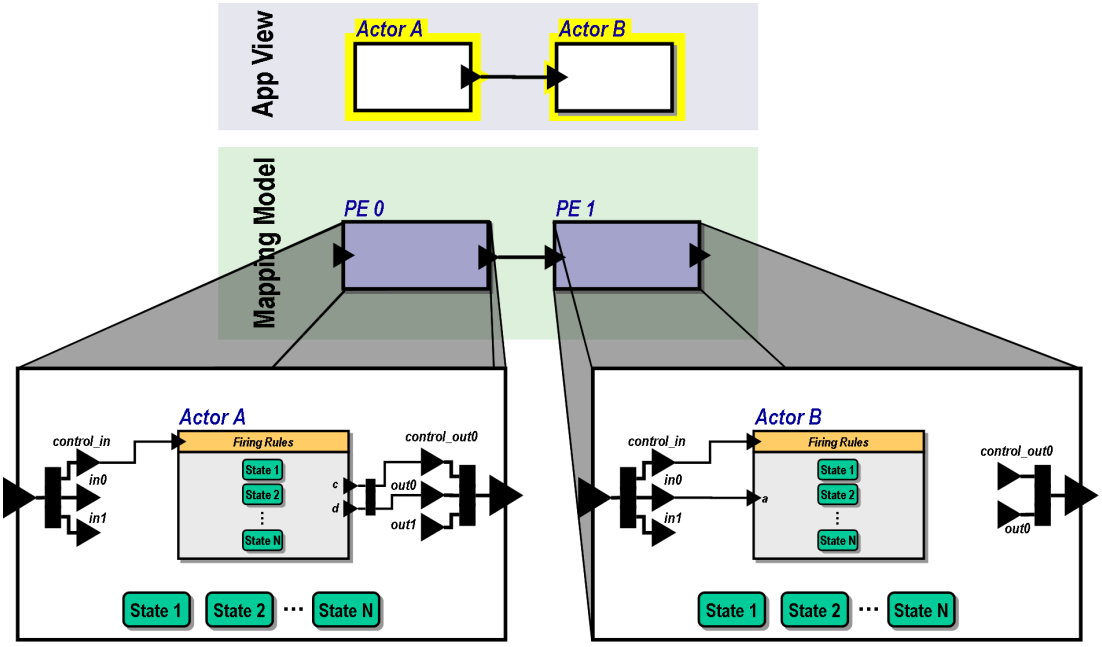


Figure 6.12: Implementing Application Transfer Messages Using the On-Chip Network

edges such as these for all of the transfer-passing connections found in the application model.

In this example, there is an unused pair of ports on the PEs (ports *out1* and *in1*). This simply indicates that the application components are not utilizing the full width of the communication link between the PEs. The architecture is capable of sending more complex signal-with-data messages between the PEs.

When multiple application actors are mapped to the same PE, there can be I/O assignment edges for multiple actors that connect to the same architecture ports. This is used to indicate that multiple transfer messages are implemented using the same on-chip network hardware. Figure 6.13 shows an example. Actor *A*, mapped to *PE 0*, sends transfer messages to actors *B* and *C*, which are both mapped to *PE 1*.

The transfer messages for actors *B* and *C* have different formats. Actor *B* receives a message with one data value whereas actor *C* receives a message with two data values. The on-chip network connection between the PEs is large enough to send two data values and a control value, so there is no mapping difficulty. The I/O assignment edges inside *PE 0* map the transfer between actors *A* and *B* to the *control_0* and *out0* ports. The transfer between actors *A* and *C* shares the *control_0* and *out0* ports, and additionally uses the *out1* port. Inside *PE 1*, a corresponding set of edges distribute the values to the inputs of actors *B* and *C*.

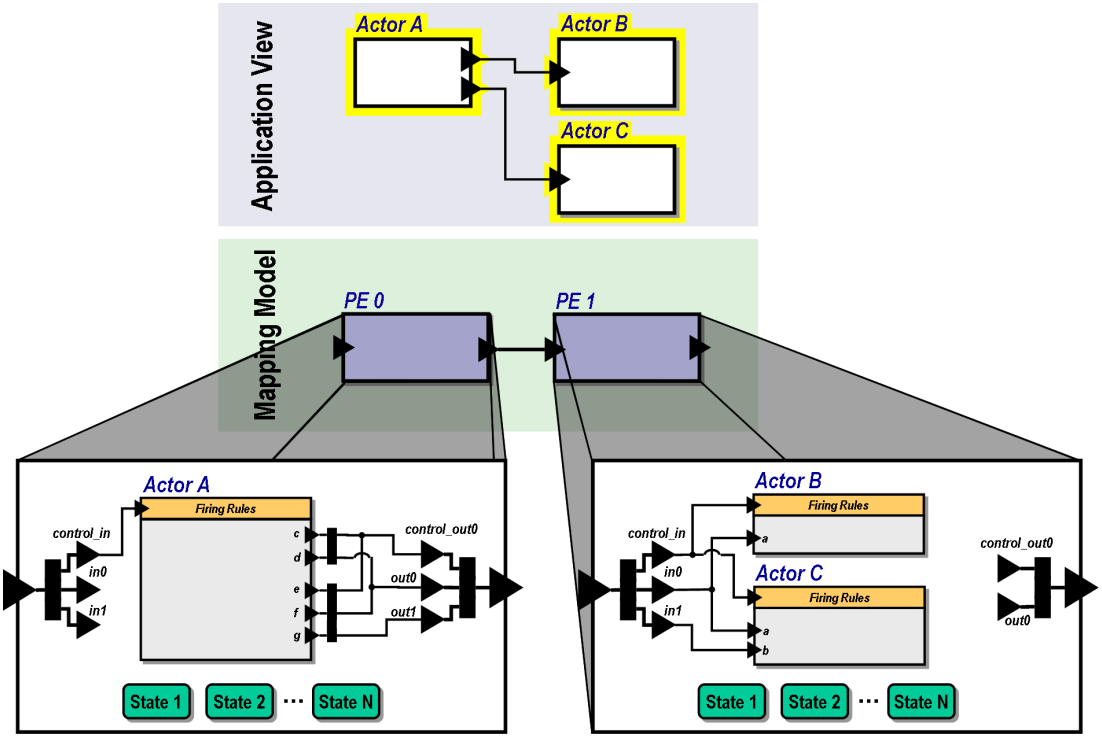


Figure 6.13: Mapping Multiple Transfer Messages to One Signal-with-Data Channel

PE 1 must be able to distinguish whether incoming signal-with-data messages are intended for actor *B* or actor *C*. Therefore the control portion of the messages must uniquely identify the firing rule to execute. Within the firing rules for actor *A*, symbolic values can be used to compute control words. For example, one of actor *A*'s firing rules may state:

```
rule fire() {
        c = ActorB.fire;
        ⋮
}
```

This rule produces an outgoing transfer that tells actor *B* to execute its *fire* rule. After a mapping model is made, Cairn will automatically fill in this symbolic value with an integer value chosen to uniquely identify the program corresponding to actor *B's fire* rule on *PE 1*. This ensures that transfer messages that share the same on-chip network hardware are not ambiguous.

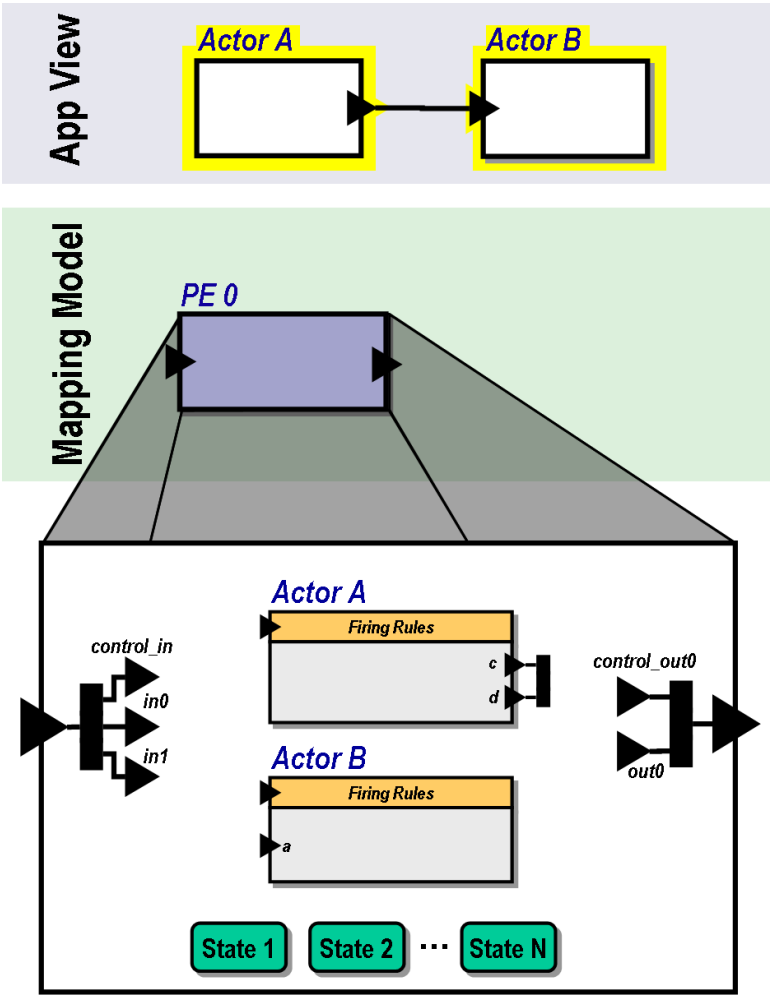


Figure 6.14: Mapping Two Communicating Application Actors to One PE

Intra-PE Communications

The above examples describe cases where communicating application components are mapped to separate PEs, and the transfer messages between them implemented as signal-with-data messages between the PEs. This section describes cases where communicating application components are mapped to the same PE. The communications between these components occur within a single PE.

Figure 6.14 shows an example. Application actors *A* and *B* are mapped to the same PE. The communications link between *A* and *B* must be implemented on that PE as well.

There are two ways that this can be done. The first possibility is to arrange for computation to continue directly at actor *B* after actor *A* executes without producing any signal-with-data messages. This involves concatenating the firing rules for actors *A* and *B* to make one program for the PE. The

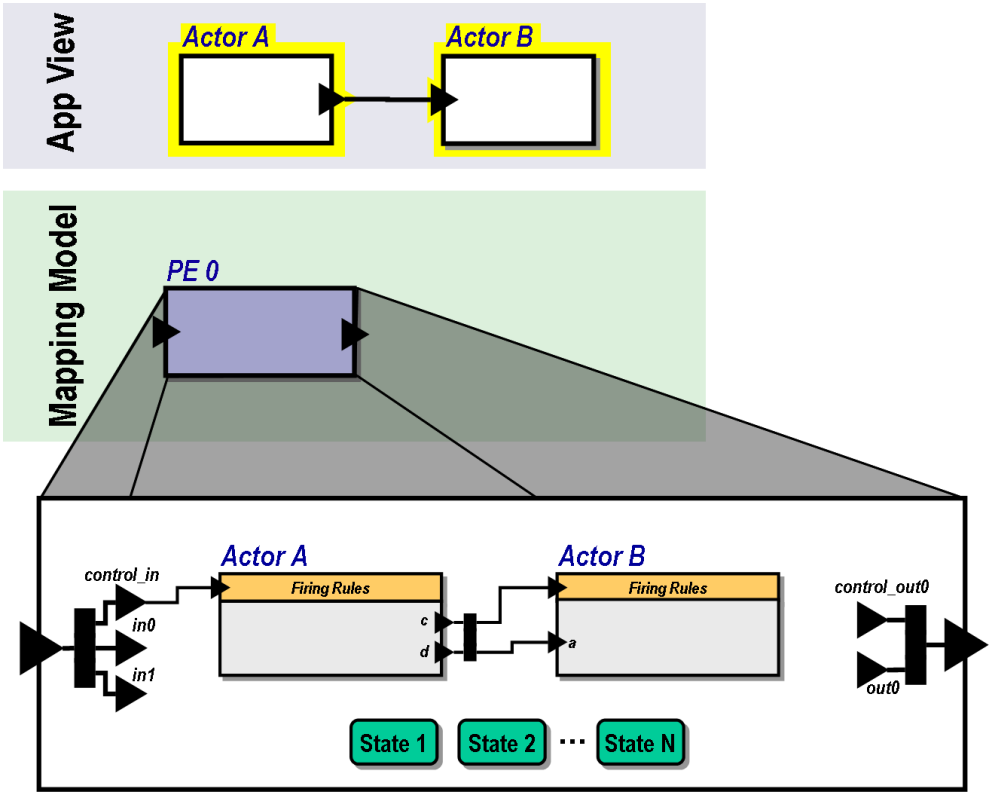


Figure 6.15: Making a Direct Connection Between Two Application Components

data values communicated between the two actors are simply intermediate values in this combined program. When the PE receives a signal-with-data message telling it to execute the program corresponding to actor *A's* firing rule, it executes a program that performs the computations for both actor *A* and actor *B*. This is equivalent to having the flow of control jump directly from actor *A* to actor *B*.

This case can only be used when it is statically determinable that the flow of control *always* continues at actor *B* after actor *A* fires. This is true when actor *A* produces a transfer message for actor *B* that has a constant control value. Recall that Sub-RISC PEs do not execute programs with jumps or conditional execution. If actor *A's* transfer message has a data-dependent control value, then it is not certain that computation should continue at actor *B* after actor *A* fires. Therefore it is not always safe to simply concatenate the programs together.

Figure 6.15 shows how this can be described in the mapping model. Instead of drawing I/O assignment edges between the application components' ports and the architecture's I/Os, designers draw edges directly between application component ports. In this figure actor *A's* c and d outputs

are connected directly to actor *B's* control input and a data input. Actor *B* gets its control value directly from actor *A*, and not from the PE's control input port. This means that actor *B's* firing rules cannot be directly executed in response to incoming signal-with-data messages. They are only executed in conjunction with one of actor *A's* firing rules.

The second possibility for mapping intra-PE communications handles the case where the flow of control does not always continue at actor *B* after actor *A* fires. This is true when actor *A* produces a transfer message for actor *B* that has a data-dependent control value. For example, this situation occurs when actors *A* and *B* are part of an application-level loop or conditional execution block.

In this case, designers arrange for the target PE to send a signal-with-data message to itself after the program for actor *A* executes. This self-message causes the PE to continue execution with the program for actor *B*. Since actor *A* can create a message with a *noop* control value, the PE will not always continue with actor *B's* computation. The feedback loop from the PE's data outputs to its own control inputs allows the flow of execution to be data dependent.

In order for this second mapping possibility to work, designers must map the actors to a PE that is capable of sending a signal-with-data message to itself. Figure 6.16 shows the same application as Figure 6.15, only this time the actors are mapped to a PE with a loopback communications link. I/O assignment edges in the mapping model connect actor *A's* outputs to the PE's outgoing signal-with-data ports, and actor *B's* inputs to the PE's incoming signal-with-data port. Messages from actor *A* to actor *B* loop around outside of the PE. Incoming signal-with-data messages can trigger the execution of either actor *A's* or actor *B's* firing rules.

Using Network-On-Chip Protocols

The above examples cover the cases where the transfer-passing edges between application components are implemented directly on point-to-point architectural communication resources. Realistic multiprocessor platforms will have more complex on-chip networks that include features such as shared buses and switching nodes. It will not always be possible or desirable to map directly-connected application components to directly-connected PEs. Instead, the messages between application components may need to be routed through several PEs before they reach their destination.

Network-on-chip protocols allow designers to build the abstraction of direct point-to-point communication links on top of a complex on-chip network. These protocols are extra computations that are mapped to PEs to mediate how application transfer messages are implemented using architectural signal-with-data messages.

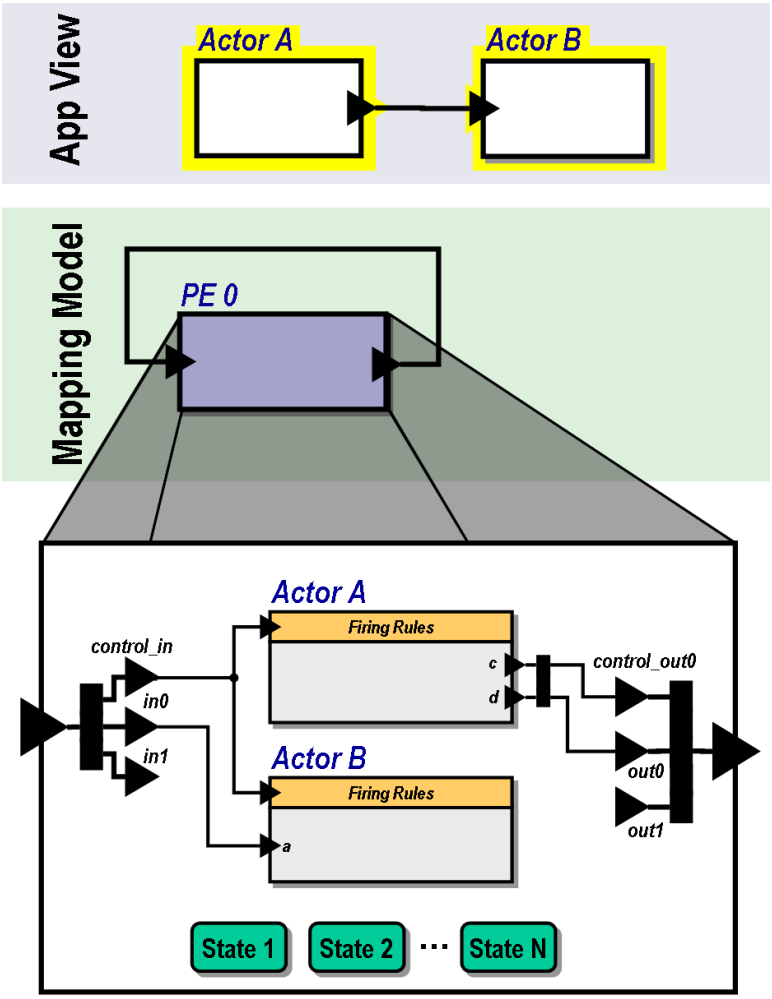


Figure 6.16: Mapping to a PE that Sends Signal-with-Data Messages to Itself

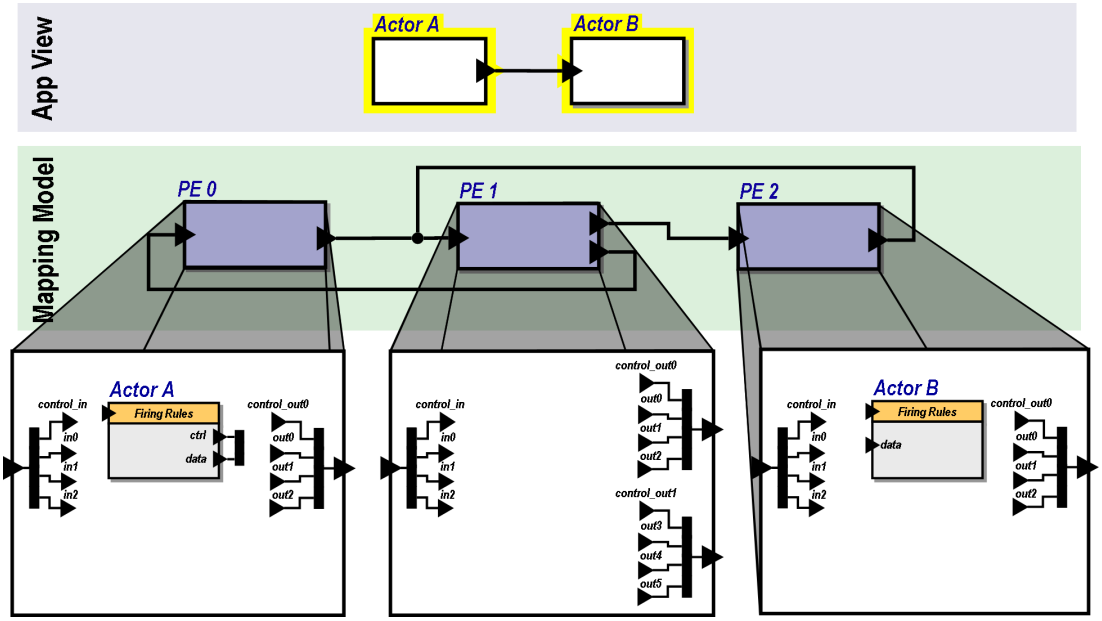


Figure 6.17: Application Actors Mapped to PEs that are Not Directly Connected

On-chip networks and protocols are an extensive topic that will not be discussed in detail here. Jantsch, et al. [66] provide an overview of this field. This section gives a short example on how designers can include network-on-chip protocols in Cairn mapping models.

Consider mapping a pair of connected application actors to PEs that are not directly connected by a signal-with-data communication link. Figure 6.17 shows an architecture with three PEs. PEs *0* and *1* can exchange signal-with-data messages, as can PEs *1* and *2*. However, *PE 0* cannot send a signal-with-data message to *PE 2*.

Actor *A* on *PE 0* needs to be able to send a message to actor *B* on *PE 2*. One way to do this is to assign a computation to *PE 1* that provides the service of forwarding messages between the left-hand and right-hand sides of the architecture. *PE 1* behaves as an on-chip network router.

Figure 6.18 shows the additional components that are mapped to the PEs to realize this scheme. On *PE 0*, a network-on-chip protocol component is placed between actor *A's* output ports and the PE's output ports. The meaning of this connection is the same as when two application components are directly connected (as shown in Figure 6.15). The flow of control continues at the protocol component after any of actor *A's* firing rules are executed.

Actor *A* computes control and data for actor *B* on ports *ctrl* and *data* respectively. Next, the protocol component uses these values to compute a signal-with-data message for *PE 1*. They are placed in the data portion of this signal-with-data message. Additionally, the protocol component

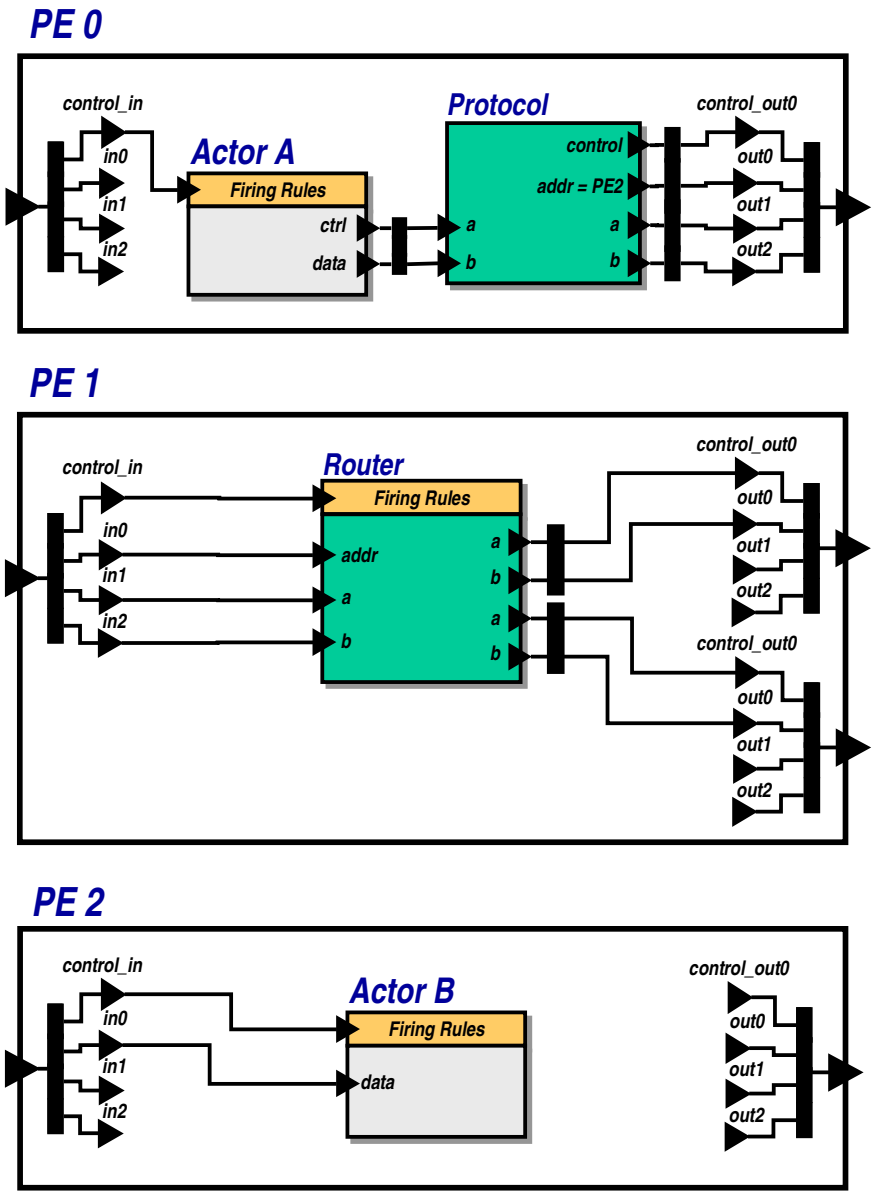


Figure 6.18: Mapping Models that Use a Network-on-Chip Routing Protocol

adds an *addr* data value to the message. The original message intended for actor *B* is encapsulated in a "network-on-chip packet" that is sent to *PE 1*.

When *PE 1* receives this signal-with-data message, it executes a program for the router component. This program takes the *ctrl* and *data* fields out of the signal-with-data message and bundles them into a new signal-with-data message. The value of the *addr* field is used to decide which of *PE 1's* outgoing signal-with-data ports should be used to send the new message. If the *addr* field is set for *PE 2*, then the original *ctrl* and *data* values computed by actor *A* will get passed on to *PE 2* where the computation for actor *B* will be performed.

The protocol component on *PE 0* and the router component on *PE 1* together implement a virtual point-to-point signal-with-data connection between PEs *0* and *2*. This has the desired effect of allowing actor *A* to communicate with actor *B*. No changes are required to either application actor. The network-on-chip protocols are transparent from the point of view of the actors.

Designers add network-on-chip protocol components to the mapping model by instantiating them from a library of pre-designed components. These components should be parameterized and modular so that they can be easily reused across a variety of applications and architectures.

In this example, the computations performed by the protocol components are rather simple. This is intended to show the basic theory of how network-on-chip protocols are used in Cairn mapping models. In a real deployment scenario, Cairn's mapping methodology allows designers to use more complicated protocols and more sophisticated network-on-chip techniques.

For example, a larger multiprocessor architecture than the one shown here may provide multiple paths between *PE 0* and *PE 2*. The intermediate PEs may make dynamic routing decisions based on the congestion levels of the links for better performance. Another example is that designers may wish to provide different quality-of-service guarantees for the communication between different application components. Network-on-chip protocols can do this as well. There is a rich set of design options here that can all be implemented using Cairn's disciplined mapping methodology.

6.3 Design Point Implementation

After creating a mapping model, designers have a complete model of a system design point. This includes a complete model of the architecture and a model that describes the software that is to run on each programmable element in the system. All that remains to produce a realizable implementation is to convert the mapping model into executable machine code for the PEs. Cairn provides an automatic code generation process to do this.

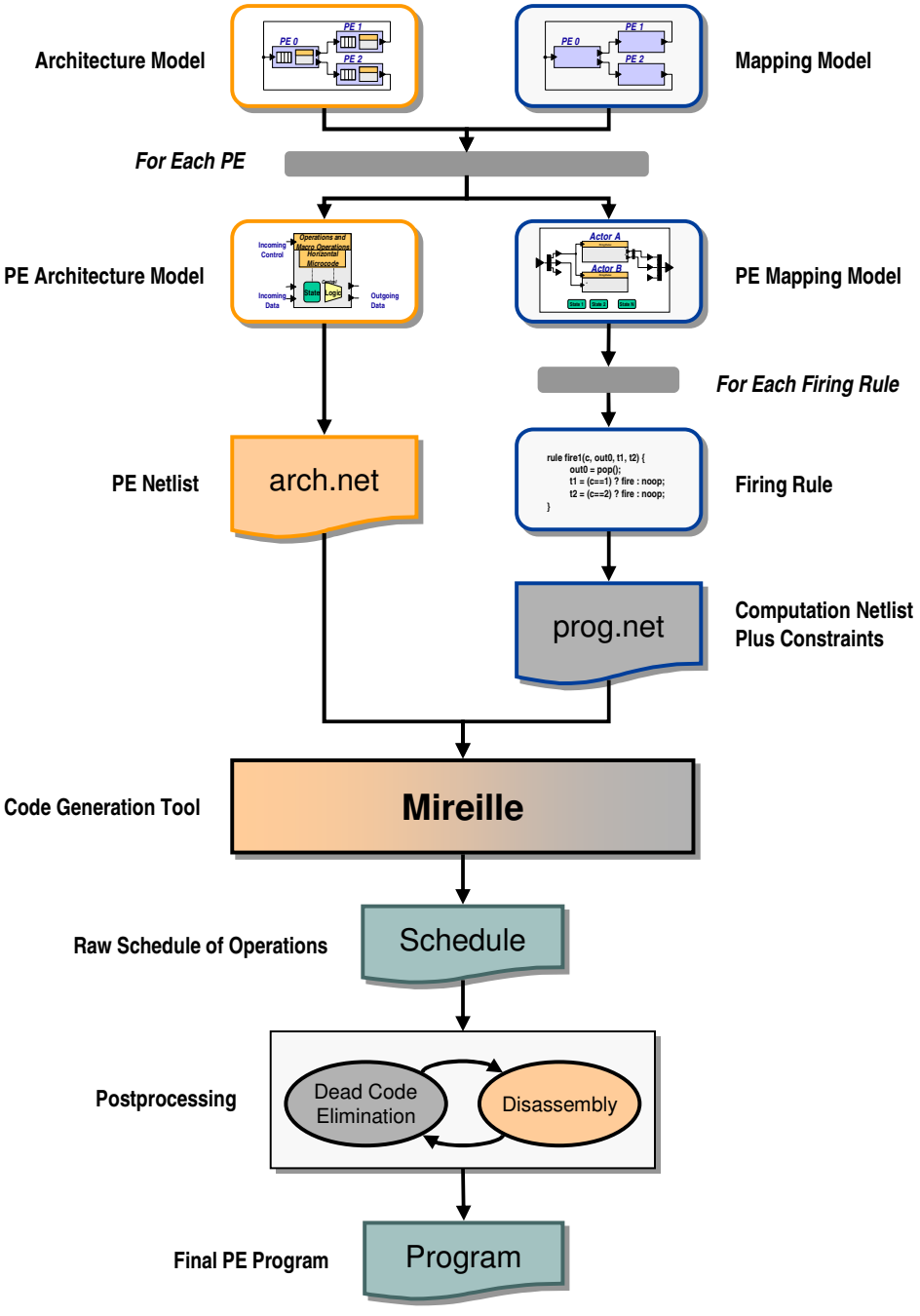


Figure 6.19: Code Generation Flow Chart

The core of this flow is an independently-developed tool called Mireille, which is a retargetable code generator for programmable processors. Mireille is used to convert the firing rules found inside of Cairn actors into schedules of operations for Sub-RISC PEs. The Cairn project makes extensions to Mireille to allow it to generate code for the kind of Sub-RISC processors that appear in Cairn multiprocessor architectures. That is, processors that execute programs in response to incoming signal-with-data messages, read the program inputs from the PE's on-chip network ports, and write the results as outgoing signal-with-data messages. Mireille is an ongoing research project by Matt Moskewicz at the University of California at Berkeley. The portions of this project that are used by Cairn are complete.

Figure 6.19 shows the flow chart for the design point implementation process. Each processing element in the multiprocessor is considered individually. Every firing rule found in the mapping model for that PE is converted, using Mireille, into a machine code program for that PE. These are the programs that the PE executes in response to incoming signal-with-data messages.

In an iterative design methodology based on the Y-chart, designers do not have time to write a compiler for each PE in the architecture. Heterogeneous architectures have a variety of different PEs that change frequently during design space exploration. This requires a potentially large number of compilers and a great deal of effort spent keeping the compilers up-to-date.

Further compounding this problem is the fact that Sub-RISC processors can have unusual datapath architectures that do not follow traditional architecture design patterns. There may be multiple register files. There may be incomplete connectivity between the register files and the datapath function units. Traditional compiler algorithms and optimization techniques cannot always be applied to Sub-RISC architectures.

Code generation for Sub-RISC architectures also has the possibility of failure. There may not exist a schedule of operations that implements the desired computation. Amongst the possible reasons for this, the PE might not have the correct function units required by the computation. Another possibility is that there is not enough register file space for intermediate values.

This is a problem that designers who are used to traditional processors do not expect to encounter. Since the Sub-RISC approach allows architects to build small and lightweight processors with limited capabilities, there is always the possibility that the architects will create a PE with capabilities that are too limited. This is an example of a concurrency implementation gap mismatch. The problem can be resolved by using the Y-chart feedback paths to make changes to the architecture, application, or mapping.

Fortunately, there are two points that help simplify the code generation process. First, the ex-

act capabilities of each Sub-RISC PE are well-known. An automatic operation extraction process identifies the atomic state-to-state behaviors that the PE can perform and summarizes this information within an operation set and a conflict table. This gives the code generation process a useful abstraction of the architecture that reveals the features that are important for building software and hides the details that are not important.

The second beneficial point is the simple nature of the computations that are to be implemented on the PEs. Cairn firing rules are feed-forward dataflow networks that compute mathematical projections and update internal state. There are no branches, jumps, or conditional execution. The code generation tool only has to work on problems the size of a basic block, as defined by traditional compilers. There are no control flow issues to deal with.

In order to support a variety of Sub-RISC architectures, and to be flexible to architectural changes during design space exploration, Cairn uses a retargetable code generation approach. The Mireille tool takes as inputs both a netlist representing a PE architecture and a netlist representing the firing rule to implement. Code generation is formulated as a search problem that attempts to find a schedule of the PE's operations that implements the computation specified by the firing rule. The following sections describe the details of this process.

6.3.1 Preparing Inputs for the Code Generation Tool

The first step is to prepare the architecture and firing rule netlists for the code generation tool. One architectural netlist is created for each PE in the multiprocessor architecture. Then, the process iterates over every component mapped to that PE in the mapping model. Each mapped component has one or more firing rules. These firing rules are individually converted into computation netlists. The Mireille tool will be invoked separately for each computation netlist to create the set of programs for the PE. Once all of the firing rules for a PE have been transformed into software, the process moves on to the next PE in the architecture.

The architecture netlist is derived from the PE architecture model. It describes the PE's on-chip network interface ports as well as the operation set and conflict table. Features such as the size of the register files and other state elements are also included. These are the aspects of the architecture that are important for code generation.

The computation netlist is taken directly from the firing rule written in the Cairn actor language. This involves only a change of syntax. The netlist describes the outputs and next state values that are to be computed as functions of the inputs and current state values. The code generation tool will

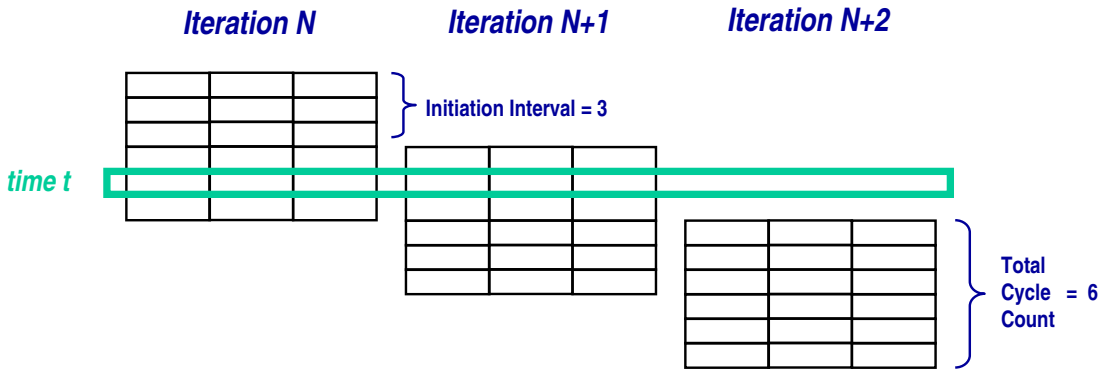


Figure 6.20: Initiation Interval and Total Cycle Count Constraints

solve for a schedule of operations that computes all of the required values.

The following additional constraints are included in the computation netlist:

- *Total Cycle Count:* This constraint bounds the length of the schedule of operations that the code generation tool will try to find. All of the firing rule's next state variables and output values must be calculated using this many or fewer processor cycles.

  The value for this constraint affects the complexity of the scheduling problem, and hence the run time of the code generation tool. If too large of a value is chosen, then the space of possible schedules will also be large and the search may take a long time. If the value is much too small, then the tool will quickly identify that a solution cannot be found by exhausting the possibilities. If the value is only slightly too small, for example one cycle less than the optimum, then the tool will also be slow. The problem will be only barely unsolvable, so many schedules that are nearly correct will be explored before the tool gives up.

  The ideal strategy for setting this constraint is to choose a value 3 or 4 cycles larger than the expected optimum, and then to decrease the value one cycle at a time until the exact optimum is found.

- *Initiation Interval:*

  When a PE is expected to receive the same signal-with-data message repeatedly, performance gains can be realized by interleaving the iterations of the corresponding program. The initiation interval constraint tells the code generation tool how many cycles will pass before the PE will start executing the next iteration of the program. If this value is less than the total cycle count, then adjacent iterations of the program will overlap in time.

An example is shown in Figure 6.20. Here, the PE is running a program that takes 6 cycles to complete. With an initiation interval of 3, the PE starts working on the next iteration of the program after the previous iteration is half complete. Over the long term, this leads to a throughput of one program iteration every three cycles. At each time t, the PE is executing operations for two separate iterations of the program. This exploits the data-level concurrency of the PE to perform more computations in each cycle, thus achieving higher datapath utilization and better performance.

The initiation interval constraint places restrictions on the schedule that the code generation tool is attempting to find. Normally, in each cycle of the program, the code generation tool tries to schedule operations that are conflict-free according to the PE's conflict table. The initiation interval constraint adds to this the possibility of conflicts across different cycles. In the figure, the fourth cycle of iteration n is executed at the same time as the first cycle of iteration $n + 1$. The operations across both of these cycles must be conflict-free according to the conflict table. With a sufficiently small initiation interval, more than two iterations of the program may overlap.

In order to find a schedule that meets these constraints, the architecture must have sufficient datapath resources to perform computations from different iterations at the same time. This is another case where Sub-RISC code generation can fail. An initiation interval constraint that is too small may over-constrain the problem to the degree that there is no solution. If this occurs, designers should try increasing either the initiation interval constraint or the total cycle count constraint. Both of these actions will loosen the constraints on the problem and may allow a schedule to be found.

- *I/O and State Constraints:* The I/O assignment edges found in the mapping model are used to tell the code generation tool which architectural ports are used for the firing rule's inputs and outputs. This constrains the tool to choose operations that read and write from the correct ports. By the end of the program, the PE must write the firing rule's output values to the output ports indicated by the I/O assignment edges. The hardware will bundle these values together to create new signal-with-data messages that are sent out on the network-on-chip.

  Similarly, the state assignment edges tell the tool which architectural state elements are used for which program state variables. The tool must choose operations that read current state values from the proper state elements and write next state values back to those same state elements. All of the next state values must be written by the end of the program.

6.3.2 Running the Code Generation Tool

After constructing the netlists, the Mireille tool is invoked to solve for a schedule of operations that performs the desired computations subject to the given constraints. Mireille uses a combination of symbolic simulation and pseudo-Boolean satisfiability to find such a schedule. This technique is derived from an approach based on integer linear programming [122].

The first step is to perform symbolic simulation on the architecture over a number of cycles equal to the total cycle count constraint. The input variables for the program are initially placed on the input ports indicated by the I/O assignment constraints. Likewise, the program's current state variables are placed in the state elements indicated by the state assignment constraints.

In each cycle, the symbolic simulator attempts to apply each of the PE's operations. This enumerates the entire set of symbolic values that can be computed for the next cycle using the symbolic values that are present in the architecture in the current cycle.

If an operation reads from a port or state element that does not contain a known symbolic value, then that branch of the simulation is pruned. It is not valid to apply that operation in the current cycle because the prerequisite data is not available.

An operation is considered successful if it produces a new symbolic value that exists somewhere in the computation netlist. If the operation does not produce a value that is needed by the program, then this branch of the simulation is pruned.

Since the PE's operation set describes all possible state-to-state behaviors, this simulation is guaranteed to explore the entire state space of the architecture. Pruning unnecessary operations keeps this state space small.

For each cycle and each operation, a Boolean clause is created for a SAT formulation of the scheduling problem. If this variable is true, it means that the given operation is issued to the datapath in the given cycle. If the variable is false, it means that the operation does not appear in the schedule at the given cycle.

The PE's conflict table is used to add additional Boolean clauses that constrain which operations can be issued in the same cycle. Additional conflict clauses are added when the initiation interval constraint is less that the total cycle count constraint. These clauses represent conflicts across different cycles that appear when iterations of the program are interleaved.

Pipeline registers in the architecture are another source of Boolean clauses. When an operation writes a symbolic value to a pipeline register, that value is only available in the next cycle. This clause states that if this operation is issued in cycle t, then an operation that reads from the pipeline

register must be issued in cycle $t + 1$. Ordinary state elements such as register files and memories do not get these clauses because they can store symbolic values indefinitely. An operation can write a symbolic value to a register file and leave it there to be read by another operation an arbitrary number of cycles in the future.

The sizes of state elements such as register files and memories are encoded using pseudo-Boolean cardinality constraints. When an operation writes a new symbolic value into a state element, the number of values stored in that element is incremented. A schedule that satisfies these constraints will not overfill state elements with intermediate values.

A final set of SAT clauses ensures that the program's output values are written to the PE ports specified by the I/O assignment constraints, and that the program's next state values are written to the state elements specified by the state assignment constraints.

At this point, Mireille performs a sanity check to make sure that all of the outputs and next state values found in the computation netlist were obtained during symbolic simulation. If this fails, then there will be no solution to the scheduling problem and it would be a waste of time to run the pseudo-Boolean SAT solver. Otherwise, the solver goes ahead.

If a solution is found, the resulting schedule of operations can be obtained by reading off the Boolean variables that correspond to the operations that are issued in each cycle from the SAT certificate. This schedule is written as a macro operation for the PE. It is a table of operations with one row for each cycle and one column for each operation that is issued in each cycle (as shown in Figure 4.15).

Mireille also produces a data structure that indicates what symbolic values appear in which state elements over the course of the schedule. This is a table with four columns: cycle number, state element identifier, state element address, and symbolic value identifier. This information is necessary for the post-processing step described below.

6.3.3 Processing the Outputs of the Code Generation Tool

A consequence of using pseudo-Boolean satisfiability to solve the scheduling problem is that the tool may find a schedule that contains unnecessary operations. For example, the schedule may compute a required output value or intermediate value more than once. This is because the SAT formulation only constrains the total length of the schedule. It does not encode the concept that the processor should avoid unnecessary computations. Since the SAT solver returns only one of perhaps a large number of solutions, the solution found is often imperfect in this regard.

In practice this happens frequently when the total cycle count constraint is larger than necessary, or when the PE datapath has more data-level concurrency than the program needs. The extra issue slots in the schedule are filled up with redundant operations.

To clean up the schedule, a post-processing step is performed. A combination of dead code elimination and disassembly is used to remove from the schedule operations that do not directly contribute to program outputs or next state values.

Figure 6.21 shows the pseudocode for this process. First, the table showing the symbolic values that appear in each state element over time is analyzed. For each output and next state value, the algorithm finds the first cycle in the schedule where this value is calculated. In case the schedule computes the value more than once, the earliest version will be used.

Next, the algorithm goes backwards through the schedule starting with the last cycle and ending at the first cycle. In each cycle, it is determined which outputs and next state values appear for the first time. These values are added to a list of *valid writes*. The operations that write these values are known not to be dead code.

The algorithm now disassembles all of the operations issued to the PE in this cycle. Disassembly identifies the state elements and output ports that are written by the operation. Operations that do not perform valid writes are identified as dead code and removed.

Disassembly also identifies the state elements and input ports that are read by the valid operations. These values must be computed in a previous cycle of the program. Therefore they are added to the list of valid writes.

The algorithm now continues backwards through the schedule. At each step the valid write list contains the values that are necessary for operations that appear later in the program, as well as outputs and next state values that are being calculated for the first time in the current cycle. Once an operation is found that writes a value found in the valid write list, that value is removed from the list. This ensures that intermediate computations are performed only one time.

The result of dead code elimination is an optimized schedule of operations that implements the computations required by the firing rule. This is executable machine code that can be run by the PE. Designers can take this code and use it to program a physical realization of the multiprocessor architecture, which will then implement the behavior specified by the original high-level application model. Another option is to perform co-simulation of the architecture and the software, which is described in the following section.

```
list symbolsAlreadyFound;
map<int, list> cycleToSymbolList;

// Find the first cycle that a given output or next state variable is calculated
foreach entry in MireilleStateElementTable {
        {cycle, stateID, stateAddr, symbol} = MireilleStateElementTable[entry];
        if !symbolsAlreadyFound.contains(symbol) {
                symbolsAlreadyFound.add(symbol);
                cycleToSymbolList[cycle].add(symbol);
        }
}

// Prune dead code
list validWrites;
from last cycle to first cycle of MireilleSchedule {
        // Outputs and Next State Variables calculated for the first time are live
        validWrites += cycleToSymbolList[cycle];

        operationsThisCycle = MireilleSchedule[cycle];

        list liveOperationsThisCycle;
        foreach operation in operationsThisCycle {
                {operationWrites, operationReads} = disassemble(operation);
                if operationWrites ⊂ validWrites {
                        liveOperationsThisCycle += operation;
                }
        }

        // Prepend good operations to final program
        finalProgram = liveOperationsThisCycle + finalProgram;

        // Remove completed writes from validWrites
        // Add prerequisite reads to validWrites
        foreach operation in liveOperationsThisCycle {
                {operationWrites, operationReads} = disassemble(operation);
                validWrites −= operationWrites;
                validWrites += operationReads;
        }
}
```

Figure 6.21: Pseudocode for Mireille Schedule Post-Processing

6.4 Co-Simulation of Hardware and Software

Co-simulation of hardware and software can be performed by generating a behavioral simulator of the multiprocessor as described in Section 4.4.3. The program schedules found by the code generation process are added to the set of macro operations for each PE. When the simulator dispatches a signal-with-data message to a PE, the PE executes a macro operation that implements a firing rule from a mapped application component. Thus, the simulator models the architecture running the software that was produced by the mapping process. Designers can analyze the results of this simulation to determine the performance characteristics of the given design point.

This type of simulator is classified as a compiled-code simulator because the software that the PEs execute is built into the simulator. The main *switch* statement in the body of each PE's simulator class object (Figure 4.28) contains one case for each macro operation. The individual operations that make up this macro operation are implemented as one block of C++ code. This is faster than other simulation techniques because the individual operations do not need to be decoded one at a time. Operation decoding is a major bottleneck in processor simulators [121].

Behavioral simulation is not one hundred percent accurate because it does not model the communication delays or signal-with-data message queuing and arbitration behaviors of the multiprocessor architecture. If designers require higher accuracy, an RTL model of the architecture can be generated and then simulated with a discrete-event simulator. This RTL model is loaded with the code generated by the mapping process to simulate the execution of the mapped application.

6.5 Summary

This chapter covered the abstractions and processes found in the middle of the Cairn design abstraction chart. Designers use a mapping abstraction to create explicit mapping models that describe how the application's requirements are matched with the architecture's capabilities.

In previous design methodologies, mapping was not an explicit step in the design process. Designers made mapping decisions implicitly as part of the process of writing code for the individual PEs in the multiprocessor. This ad hoc approach is not acceptable for systems in which concurrency is a primary concern. Implementation decisions are hopelessly intermixed with the application specification. If designers want to explore different mapping strategies, much of the code will have to be rewritten. This is slow and has the danger of introducing errors into the application.

To understand the issues caused by the concurrency implementation gap and to develop good

solutions to these issues, designers must make a more formal comparison of the architecture's capabilities and the application's requirements. Cairn's disciplined mapping methodology enables this. Explicit mapping models capture implementation decisions in a tangible, modifiable form that is independent of both the application model and the architecture model. Only a minimum of changes are required to experiment with different mapping strategies. Designers do not have to write low-level code or make modifications to the application model. This ensures that design space exploration is fast and accurate.

Explicit mapping models help designers characterize the concurrency implementation gap. The models show what portion of the system software implements actual application functionality versus what portion exists only to solve mismatches between the application and the architecture. This reveals the cost of the concurrency implementation gap in terms of code size and execution time. These results drive the design space exploration process. Designers can independently modify the application, architecture, or mapping to achieve better performance.

Cairn's mapping abstraction assists designers in creating mapping models. It provides restricted views of both the application and the architecture that expose the features necessary for solving the mapping problem, while hiding the unnecessary details. The mapping abstraction also gives mapping models a functional semantics. A mapping model fully describes the software that is to run on each PE in the architecture.

Cairn provides processes to automatically convert a mapping model into an executable machine code implementation. This is the final step in each iteration through Cairn's Y-chart design methodology. If the performance results of the current design point meet expectations, then the design process is complete. If not, then designers can use the Y-chart feedback paths to explore different design points. The architecture, application, and mapping design spaces can all be individually explored.

In the next chapter, a full design example will be presented to demonstrate the complete Cairn methodology. A high-level specification of a network processing application will be created. A heterogeneous Sub-RISC multiprocessor with application-specific process-level, data-level and datatype-level concurrency will be designed. The high-level application models will be used to program the architecture through the mapping process. Then, it will be shown how the architectural design space can be easily explored to find a better match for the application's requirements. The final performance results prove that the Cairn methodology can produce fast and lightweight solutions with high designer productivity.

Chapter 7

Design Example: Gigabit Security Gateway

To demonstrate the effectiveness of the Cairn methodology, a complete design example will now be shown. This example will cover modeling an application, designing an architecture, and creating a mapping. Analysis of the performance results of the initial design point will reveal shortcomings that can be fixed through design space exploration. Revisions to the architecture will be made to improve performance and the final results will be presented.

To show that the Cairn approach is capable of handling heterogeneous applications with multiple styles of concurrency, a multifaceted network processing application will be chosen. This is an application domain where there are opportunities for concurrency at every level of granularity: process-level, data-level and datatype-level. Implementing this concurrency correctly in hardware is critical for achieving high network throughput.

Programmable systems are an attractive implementation platform because they are flexible to rapidly changing application requirements. However, existing network processor architectures have achieved notoriety for their programming difficulties. Architects and programmers alike have relied on ad hoc approaches for handling heterogeneous concurrency, and this has led to serious concurrency implementation gap problems. Programmers are unsure how to leverage the architecture's capabilities for concurrency, and architects are unsure if their designs provide features that actually benefit the application domain.

This design example will show how the Cairn approach helps designers solve the concurrency implementation gap by providing the right abstractions for concurrency. High-level models of com-

putation allow architects to quickly and easily build functional models of the application. The Sub-RISC paradigm allows architects to construct lightweight, high-performance multiprocessors with support for application-specific process-level, data-level and datatype-level concurrency. Cairn's disciplined mapping methodology allows designers to program the architecture using the high-level application models directly, without writing low-level code by hand. This enables fast and effective design space exploration. By using the right abstractions for concurrency, designers can realize the promise of programmable systems: superior results and high productivity.

This chapter is organized as follows: Section 7.1 describes the example application and shows how it can be modeled with multiple models of computation. Section 7.2 covers the design of a Sub-RISC multiprocessor with support for the application's concurrency. The mapping process and initial performance results are given in Section 7.3. These results generate ideas for design space exploration, which is shown in Section 7.4.

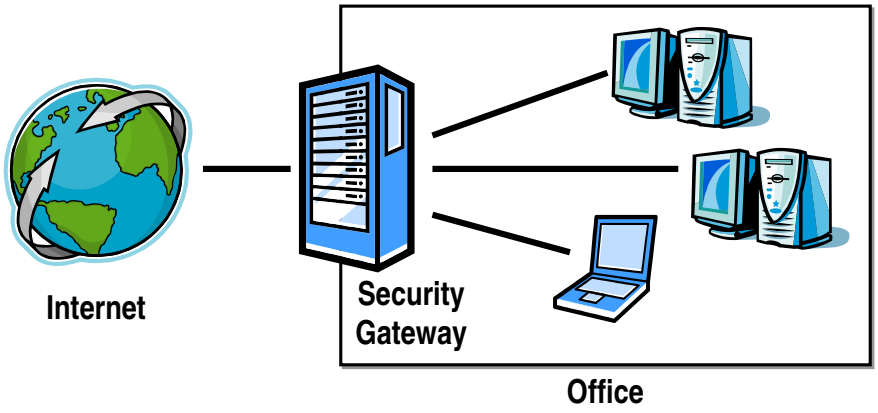
7.1 Designing the Gigabit Security Gateway Application

The application for this design example is a gigabit security gateway. This is a network router that sits at the edge of the Internet to forward packets between end users and the rest of the Internet. The "gateway" part of the application refers to the IPv4 forwarding facet. This part of the application is responsible for switching packets between various networks.

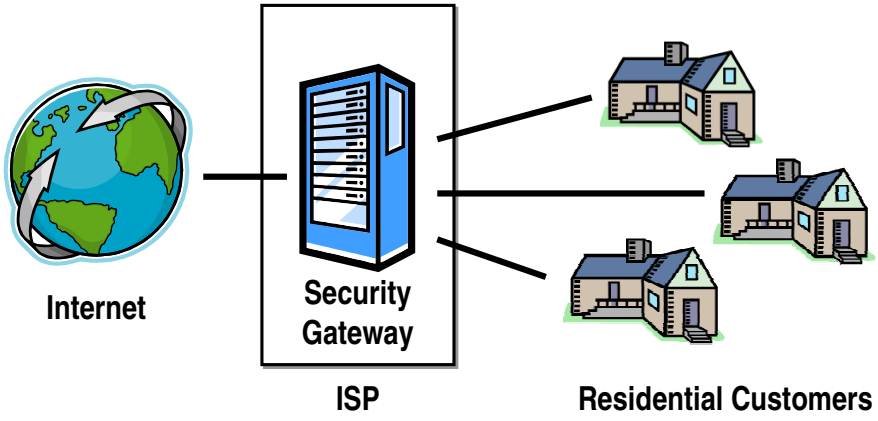
The "security" part of the application refers to network intrusion detection. Network intrusion detection systems inspect streams of packets, searching for traffic patterns that match the behavior of known malware, trojans and viruses [110]. A common strategy employed by crackers is to infect end-user computers with hidden malware that can be used to execute distributed attacks on Internet services. The malware is delivered either by tricking users into installing it or by exploiting operating system or network service security holes. Then, the infected computer is used to send spam or join in distributed denial-of-service attacks, often without the knowledge of the owner.

Network intrusion detection has not traditionally been a feature of network gateways. Since the gateway is a neutral intermediary between end-user computers and the rest of the Internet, and since end users are often unaware of the problems with their machines. this is the ideal location to deploy the security system. The gateway searches through streams of packets to determine if any end hosts are misbehaving. If evidence of infection is found, the appropriate administrative actions can be taken before the problem gets out of hand.

Figure 7.1 shows two possible deployment scenarios for the security gateway. In Figure 7.1(a),



(a) Office Deployment Scenario



(b) ISP Deployment Scenario

Figure 7.1: Deployment Scenarios for the Security Gateway

a large office uses the gateway to connect its internal subnets to the whole Internet. The company is interested in two things. First is to identify possible incoming attacks and protect the internal computers against these attacks. Second is to identify possible outgoing attacks, to ensure that the company's computers are not infected with malware and that the company's Internet connection is not being hijacked for the purpose of creating Internet mayhem. For example, a visitor may bring in an infected laptop from off-site. If this machine starts sending suspicious packets, the IT staff can learn of the problem quickly and take care of it before the malware spreads.

In the scenario shown in Figure 7.1(b), the security gateway is deployed by an Internet service provider between its residential customers and the rest of the network. The same benefits apply. The ISP can sell protection from known attacks as a service to its customers. It can also learn when a customer's computer begins misbehaving. The key to stopping spam and other distributed attacks is to identify the problem at its source, and to stop that machine from sending packets on the network until it can be fixed.

The "gigabit" part of the application refers to the packet throughput rate that must be attained in order for the security gateway to be practical. The computational requirements of network intrusion detection are high. General-purpose processors operating at gigahertz frequencies can search at rates up to a few hundred kilobytes per second [38]. This is not fast enough for the given deployment scenarios, where the Internet uplinks run at gigabit rates and faster. ASIC designs could outperform general-purpose processors, but they are not flexible to rapidly changing application requirements. The database of attack patterns needs to be updated constantly in order for the security system to be effective. This calls for a programmable solution.

The gigabit security gateway application is therefore an ideal candidate to demonstrate the Cairn approach. The application is multifaceted. It performs both packet header and packet body processing. Each of these facets has different concurrency requirements. Since independent packet streams can be processed in parallel, there is a large amount of process-level concurrency. The unusual bit widths of packet header fields and the arithmetic and logical operations performed on these fields provide opportunities for data-level and datatype-level concurrency. The network intrusion detection facet of the application performs pattern matching on the bodies of the packets, which provides different opportunities for data-level and datatype-level concurrency.

In order to achieve gigabit packet throughput rates, it is necessary to have architectural support for all styles and granularities of concurrency. Lastly, programmability is a crucial requirement. A Sub-RISC multiprocessor should make an ideal platform for this application.

The first part of this design example describes the process of modeling the application using

Cairn's application abstraction. Designers use different models of computation to model different application facets. Then, automatic model transforms are applied to convert the high-level application model into a concrete Cairn Kernel model. The resulting Cairn Kernel model will be used during the mapping process.

The network security gateway application is divided into three facets. The first facet performs the header processing and IPv4 forwarding parts of the application. The second facet describes the regular expression matching that is done by the network intrusion detection system. The final facet models how the gateway separates incoming packets into headers and bodies. The headers are sent to the header processing facet, while the bodies are stored in a memory until they are ready to be transmitted on an outgoing network interface.

7.1.1 The Header Processing Application Facet

The primary task of the header processing facet is to perform IPv4 forwarding. Forwarding consists of switching packets between various networks. In this design example, the security gateway will have 8 network interfaces, all of which are gigabit Ethernet. Packets arriving on any of the 8 interfaces should be retransmitted on one of the 7 other interfaces. The destination IP address field in the packet header is used to make the forwarding decision. Packets should not be forwarded out the same interface on which they arrived. This is an invalid routing loop that fails to get the packet closer to its final destination.

To model this facet of the application, the Click model of computation will be used. Click is designed specifically for header processing applications.

Figure 7.2 shows an outline of the Click model for this facet. Central to this model is a Click element called *LookupIPRoute*. This is the element that makes the forwarding decision. A longest prefix match algorithm is applied to the destination IP address field of packets that are pushed into this actor on its push input port. The result of this algorithm determines the push output port on which the actor will send the packet. The contents of the input chain and output chain boxes will be shown later.

The *LookupIPRoute* element contains a routing table that holds $(address,\ mask,\ port)$ tuples. The $address$ and $mask$ fields specify a range of IP addresses. A destination address dst matches this range if $dst \wedge mask = address$. The $\wedge$ operation is a bit-wise Boolean AND. If there is a match, the packet will be sent on the port indicated by the $port$ field. A destination address may match multiple routing table entries. The longest prefix match algorithm will choose the match

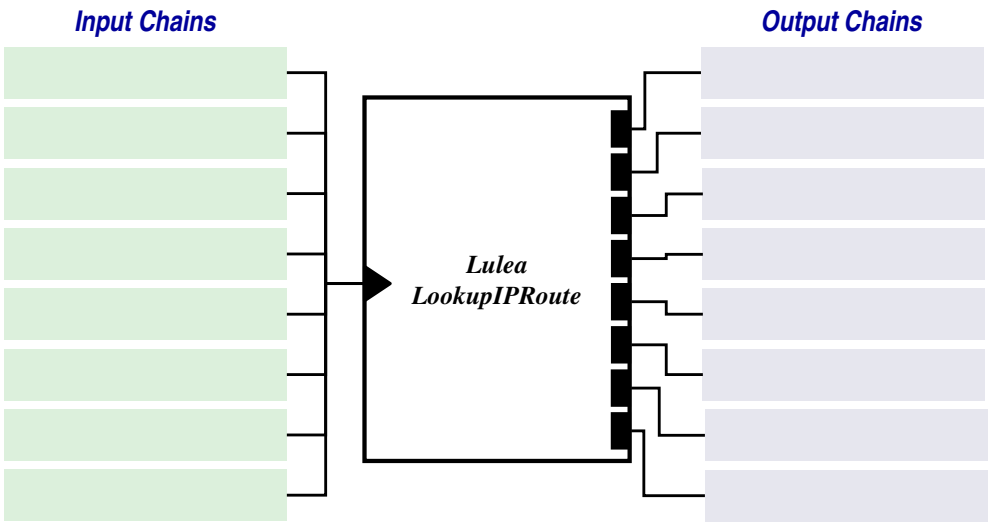


Figure 7.2: Click Model for IPv4 Forwarding

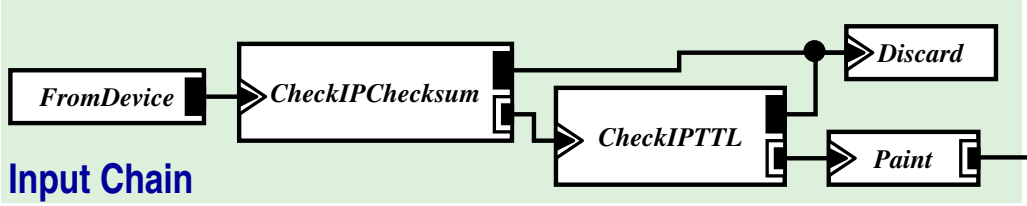


Figure 7.3: Detailed Click Model of the Forwarding Input Chain

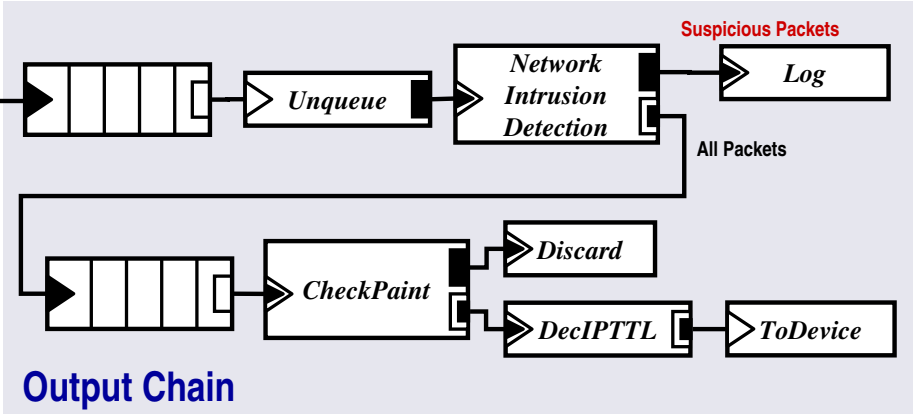


Figure 7.4: Detailed Click Model of the Forwarding Output Chain

with the longest $mask$, i.e. where the most number of bits from the $address$ field match the given destination address.

In practice, routing tables can contain tens of thousands of entries. These tables need to be updated frequently in response to dynamic changes in the network, for example if a particular route goes down. A mechanism for efficiently storing and accessing this data is necessary. The *LookupIPRoute* element used in this design example employs the Lulea lookup approach [30]. This technique compresses the routing table so that it uses a small amount of memory and requires only a small number of table accesses to perform each longest prefix match. Different versions of the *LookupIPRoute* element can be swapped into the application model that use different algorithms to make the forwarding decision. Choosing the right algorithm is part of the application design space exploration process.

Figure 7.3 shows the details of the input chain of the Click model. These elements pre-process incoming packets before they are sent to the *LookupIPRoute* element. There is one copy of this input chain for each of the eight network interfaces on the security gateway.

The first element in the input chain, *FromDevice*, models a network interface as a source of packet headers. Two different checks are applied to incoming packets. First, the header checksum is verified by the *CheckIPChecksum* element. Invalid packets are discarded.

Second, the time-to-live header field is checked by the *CheckIPTTL* element. This field is decremented every time the packet passes through a router. If it is zero, then the packet has been forwarded an unusual number of times. This is evidence that there may be a routing cycle in the Internet. Such packets are also discarded.

If the packet passes these checks, then it is ready to go on to the *LookupIPRoute* element. The final element in the input chain, *Paint*, annotates the packet with the number of the network interface it was received on. This will be checked later to make sure packets do not depart on the same interface they arrived on.

Figure 7.4 shows the details of the output chain of the Click model. Packets are first placed in a queue before being processed by the *Network Intrusion Detection* element. This allows forwarding and intrusion detection to run as decoupled processes that can process packets at different rates. The *Network Intrusion Detection* element normally copies all packets to its bottom output port and into another queue. Packets that are determined to be suspicious are additionally copied to a *Log* element. Administrators can use the information collected by that element to decide how to deal with potential malware.

The second half of the output chain performs final header processing before transmitting a

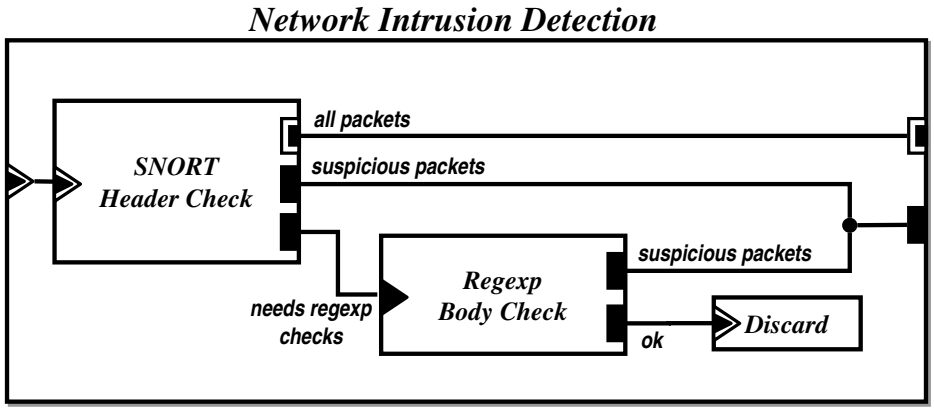


Figure 7.5: Contents of the Network Intrusion Detection Element

comment	=	FTP CMD overflow attempt
protocol	=	tcp
destination_port	=	21
pcre	=	`^CMD\s[^\n]{100}`

Figure 7.6: Example Snort Pattern

packet. The paint annotation is checked first. Then, the time-to-live field is decremented. The *ToDevice* element models an outgoing network interface as a sink of packets. Whenever the interface is ready to transmit, a thread of control starts here and attempts to pull a packet from the upstream elements in the output chain.

Figure 7.5 shows the contents of the *Network Intrusion Detection* element. This is an example of transparent hierarchy in a Cairn application. The contents of this actor are still covered by the Click model of computation.

This subsystem compares packets against patterns from the Snort database [110]. Each Snort pattern checks for a certain kind of suspicious behavior. Figure 7.6 shows an example pattern. There are both header and body components to the pattern. The packet must match all of these to be considered suspicious.

This pattern looks for a known buffer overrun attack against FTP servers. The *protocol* and *destination_port* fields of the pattern check the packet header to verify that this packet is part of a connection to an FTP server. The *pcre* field contains a Perl-Compatible Regular Expression. This checks the body of the packet for an FTP command string that is excessively long.

The *Network Intrusion Detection* element has two parts. The first part checks packet header fields against the patterns in the Snort database. This alone may be enough to determine that a

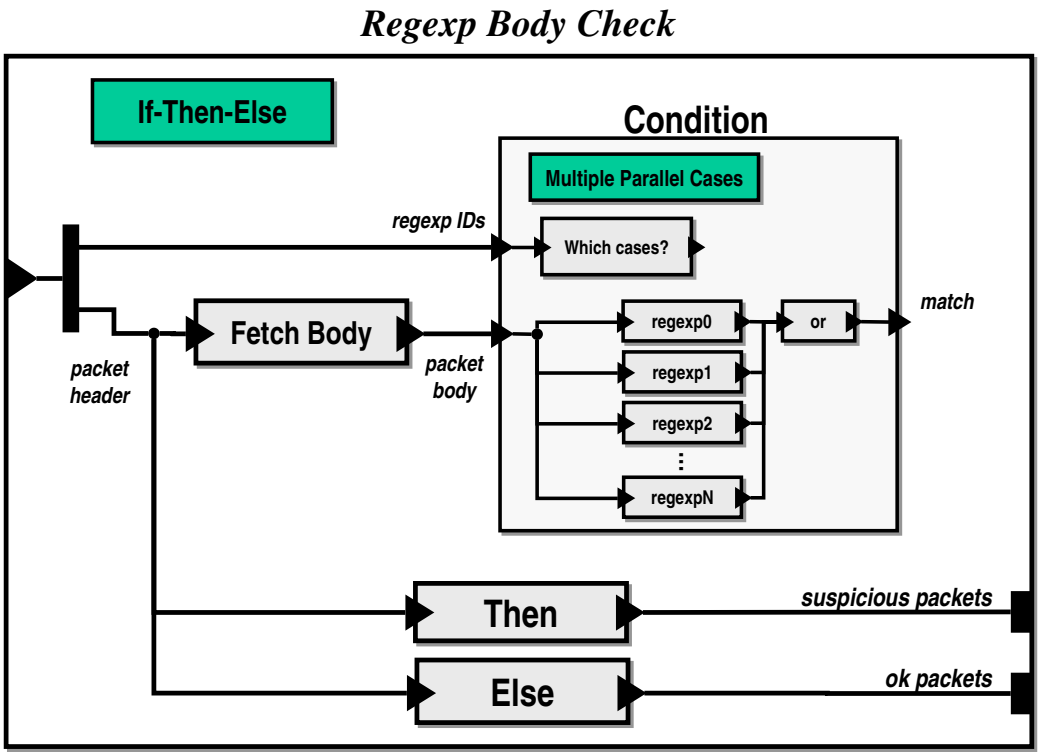


Figure 7.7: Hierarchically Heterogeneous Regexp Body Check Actor

packet is suspicious. If the header matches one or more patterns that have one or more *pcre* fields, then the packet is sent to the *Regexp Body Check* element. This element searches for the requested regular expressions in the body of the packet. If a match is found, the packet is declared suspicious. If not, then the complete Snort pattern has not been matched and this copy of the packet can be discarded.

7.1.2 The Regular Expression Matching Application Facet

The regular expression matching done by the *Regexp Body Check* element has different computational requirements compared to the header processing facet of the application. The focus is on pattern matching within individual packets, and not the spatial and temporal processing of numerous streams of packets. Therefore the Click model of computation is not a good choice for this application facet. Instead, Cairn's conditional execution and regular expression domains will be used to describe the contents of the *Regexp Body Check* element. This introduces hierarchical heterogeneity into the application model.

Figure 7.7 shows how this actor is defined. The basic structure uses Cairn's If-Then-Else do-

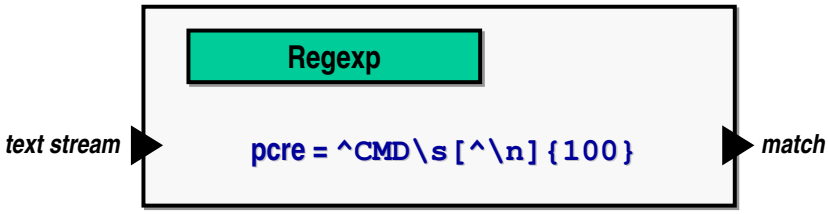


Figure 7.8: Contents of a Regular Expression Actor

main. The incoming data consists of the identifiers of the regular expressions that are to be matched and the suspicious packet header. The *Fetch Body* actor retrieves the packet body that corresponds to the packet header. This actor encapsulates an interaction between the header processing and header/body separation facets of the application. This will be detailed in Section 7.1.3. The *Condition* actor checks if the packet body matches the regular expressions. If so, the packet header is sent on the *suspicious packets* port back to the header processing application facet. If not, the *Else* actor sends the packet header on the *ok packets* port.

The *Condition* actor is itself hierarchical and uses a new model of computation called *Multiple Parallel Cases*. This domain behaves similarly to a C *switch* statement, but it allows multiple cases to be evaluated in parallel. Each case searches the packet body for one of the regular expressions in the Snort database. If the *Snort Header Check* actor determines that multiple regular expressions need to be tried, these can all be done in parallel. The results are combined with a logical OR operation. This is a conservative match. If any one of the requested regular expressions match, then the packet is declared suspicious.

The *regexp* actors are also hierarchical. Figure 7.8 shows the contents of one of these actors. Cairn's Regular Expression domain (Section 5.6.3) allows these actors to be defined simply by giving the regular expression as a text string. The input is a stream of characters and the output is a true or false value indicating if the regular expression was matched.

7.1.3 Separating Packet Headers and Packet Bodies

The final task in modeling the security gateway is to describe how packet headers and bodies are processed separately by the gateway. A common technique in router design is to separate incoming packets into headers and bodies. The header processing facet of the gateway only performs computations on the headers. The bodies are stored in a memory until header processing is finished. Then, when the packet header is ready for transmission, the body can be fetched from the memory and the entire packet can be sent out. The benefit of this approach is to minimize communica-

tion between application processes. If the header processing actors are given the entire packet, this could lead to a large amount of unnecessary on-chip network traffic when the application is later implemented on a multiprocessor.

Figure 7.9 shows the contents of one of the *FromDevice* actors. There are two subcomponents. A *Network Interface* actor interfaces with a gigabit Ethernet port and produces packet headers and packet bodies. The header and body together are sent as a transfer message to a *Packet Body Repository* actor. The repository implements a map that stores $\{header, body\}$ pairs. The control word in the transfer message to the repository selects the *insert* firing rule to add a new pair to the map.

At the same time, the *Network Interface* produces a transfer message containing only the packet header. This message is intended for the *FromDevice* element's push output port. It will be sent to the next element in the input chain of the Click model.

Figure 7.10 shows the contents of one of the *ToDevice* actors. When the actor pulls a packet header from the Click output chain, the repository's *remove* firing rule is activated to fetch and remove a $\{header, body\}$ pair from the map. The flow of control continues at a *Network Interface* element which transmits the entire packet.

The last point of interaction between the header processing and body processing facets of the application is the *Fetch Body* actor in Figure 7.7. The contents of this actor are shown in Figure 7.11. This structure is similar to the *ToDevice* actor, except that the repository's *lookup* firing rule is used instead of *remove*. This fetches the packet body corresponding to the given header without removing the pair from the map. The data will still be available when the packet finally leaves the router through a *ToDevice* element.

The *Packet Body Repository* actor appears multiple times in these models. These are all references to the same object. Hence, this application model is no longer strictly hierarchical. There is one reference within each input chain (*FromDevice*), and two in each output chain (*FetchBody* and *ToDevice*) for a total of 24 references to the repository actor.

Storing and fetching packet bodies can be a performance bottleneck. Therefore, the repository should not be implemented as a single memory but instead as a set of parallel memories. Wide memories can be used to store and retrieve entire packets in one access. This is straightforward and will not be discussed further in this design example.

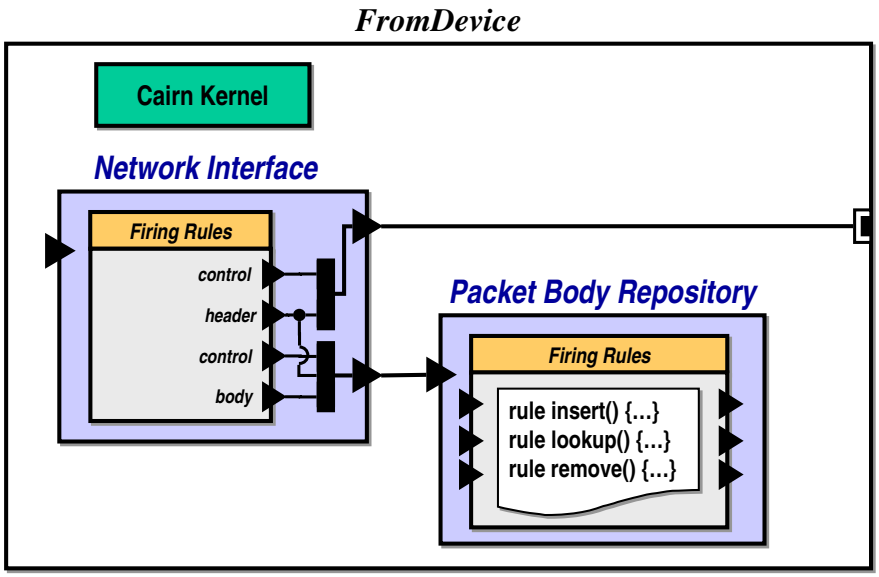


Figure 7.9: Details of the FromDevice Click Element

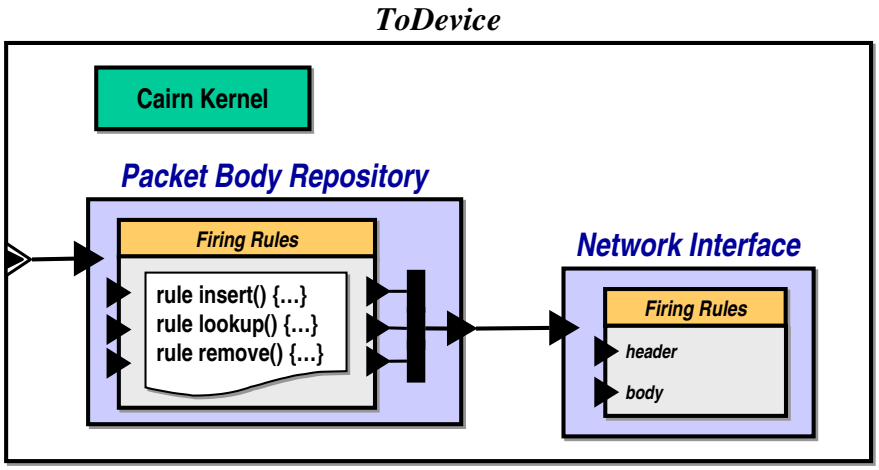


Figure 7.10: Details of the ToDevice Click Element

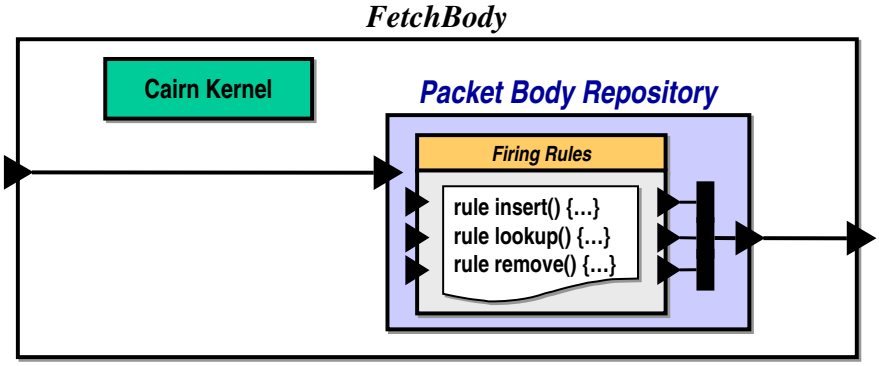


Figure 7.11: Details of the Fetch Body Actor

7.1.4 Applying Model Transforms to Create a Cairn Kernel Model

After the high-level application model is complete, automatic model transform processes are applied. This converts the abstract application model into a concrete Cairn Kernel model. The result is a flattened graph of components that interact by passing transfer messages.

Model transforms start at the leaves of the model hierarchy and work backwards to the root. Some leaves, such as the *FromDevice*, *ToDevice*, and *FetchBody* actors, are already defined as Cairn Kernel models. Therefore no modifications need to be performed.

The *regexp* actors in the security facet of the application use Cairn's Regular Expression model of computation. They are defined using a text string that indicates the regular expression that is to be matched. The model transform for this model of computation converts this actor into a Cairn Kernel component that implements a non-deterministic finite automata. This process was described in Section 5.6.3.

The *regexp* actors are contained in a model that uses the Multiple Parallel Cases model of computation. The model transform for this domain elaborates the control flow amongst the *regexp* actors. They operate in parallel to match a single packet body against multiple regular expressions.

The next higher level of the application model uses the If-Then-Else model of computation. The model transform for this domain elaborates the flow of control between the *Then* and *Else* components as described in Section 5.6.1. The combined result of the *regexp* actors is used to produce a transfer message for either the *Then* component or the *Else* component but not both. These branches copy the given packet header to one of the two output ports.

Finally, the root of the application model uses the Click model of computation. This model transform was described in Section 5.6.2. All of the abstract push and pull communication links are transformed into transfer-passing connections.

When the model transform process finishes with the root of the application model, all of the abstract communication links have been replaced with transfer-passing connections. Additional components have been inserted where necessary to implement communication resources and application processes. The resulting model is now ready to be mapped to a multiprocessor architecture.

7.2 Designing an Application-Specific Heterogeneous Sub-RISC Multiprocessor

While application designers are focusing on high-level application concepts, architects are thinking about concrete implementation details. This section describes how architects can build an application-specific Sub-RISC multiprocessor in parallel with the application design process. Three different PEs will be shown. The first two are intended to perform the computations found in the header processing application facet. The third is intended to perform the regular expression matching found in the security facet. Since these facets have different opportunities for data-level and datatype-level concurrency, it is sensible to build different datapaths to run them. Each PE will be a lightweight machine, free of clutter, that is inexpensive and excels at the given task. This follows the second core requirement: consider simple architectures.

7.2.1 Sub-RISC PEs for Header Processing

How does an architect approach the problem of designing a new PE for a particular application domain? There are three categories of knowledge that guide the design process:

- *Application Knowledge:* First, architects can look at the actors that appear in the application model and analyze the computations they perform. Even before the application model is complete, it is clear that common Click elements like *Paint*, *CheckPaint*, *CheckIPChecksum*, *DecIPTTL*, *CheckIPTTL*, and *LookupIPRoute* will be used. A PE designed specifically for Click header processing should support all of these computations with good performance.

- *Architectural Knowledge:* Second, architects can look at the target implementation platform and see what features are available to construct the new PE. In this design example, the Xilinx Virtex 2 Pro FPGA will be used as an implementation platform for the Sub-RISC multiprocessor. Since this is a configurable platform, the new PEs should be customizable elements.

  One important customization is extensibility. It should be possible to extend a Click-specific PE to add support for additional Click elements. As a complement to this, the PE should also be reducible. It should be possible to trim the Click-specific PE down to support just one or two elements for greater performance and smaller area requirements.

- *Prior Experience:* Third, architects can study other custom PEs for network processing to see what is possible and what the potential pitfalls are.

  Longest Prefix Match: A common observation is that the *LookupIPRoute* element is a bottleneck. Several architectures have been proposed to speed up this operation. Henriksson and Verbauwhede propose a pipelined architecture based on a 2-bit trie [51]. The key to performance is to saturate accesses to the routing table stored in external SRAM. This has the potential to do one route lookup every clock cycle.

  Taylor et al. [117] use 8 PEs to saturate access to an external routing table. The PEs perform a more complicated routing algorithm that saves significant memory over the trie approach.

  The Lulea algorithm [30] is compelling for a Sub-RISC PE because it offers several ways to exploit datatype-level concurrency. One operation in this algorithm is a reduction add, which counts the number of set bits in a 16-bit word. Also, the Lulea algorithm can be modified to take advantage of the 18-bit wide BRAM blocks on the Xilinx FPGA, thus permitting a compressed routing table to be stored on-chip.

  Manipulating Header Fields: A second observation is that it is good to be able to quickly modify irregularly-sized bit fields in packet headers. Sourdis et al. [113] build PEs with header field extraction and modification engines to do this efficiently.

  Custom Control Flow: The Linkoping architecture [50] recognizes that standard sequential control flow is not the best match for packet processing. This PE offers an application-specific data-driven control scheme. In one cycle, the PE can perform a field matching operation and compute the next control word for itself.

ClickPE Architecture

To guide the integration of all of these ideas, designers rely on architectural design patterns [31] and try to follow a minimalist approach. The resulting Sub-RISC architecture for Click header processing, called the ClickPE, is shown in Figure 7.12. This schematic is the top level of a hierarchical design that matches the hierarchical arrangement of fields in an Ethernet packet header.

At the top level the datapath is 344 bits wide. This allows the PE to process a complete packet header in parallel. The 344 bit number is the sum of a 20 byte IPv4 header, a 14 byte Ethernet header, and a 9 byte Click header for annotations such as *Paint* values and timestamps. There is a 344-bit register file at this level, and network-on-chip I/O ports for sending and receiving headers

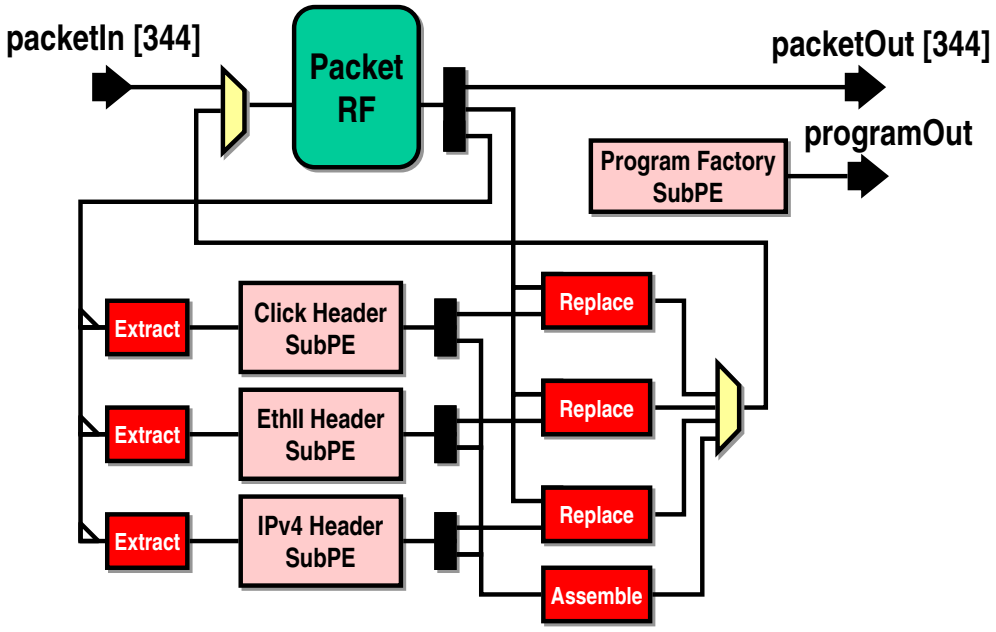


Figure 7.12: Sub-RISC PE for Click Header Processing

as the data portion of signal-with-data messages.

The operations at this level of the ClickPE are to break up the header into IPv4, Ethernet, and Click sub-headers and pass these components to sub-PEs for further processing. Additional operations provide for reassembling modified packet fields and storing the result in the packet header register file.

The Program Factory sub-PE is used to calculate control words for the signal-with-data messages that are transmitted to other PEs. Control can be generated from an immediate value in the instruction word, by a lookup within a small memory, or by a hardware if-then-else operation. Each of the other sub-PEs can perform comparison operations to drive the if-then-else operation. These connections are not shown in the diagram for clarity.

The sub-PEs do the bulk of the header processing. The IPv4 sub-PE is diagrammed in Figure 7.13. The remaining sub-PEs are similar in structure. There is a primary register file for storing the IPv4 sections of packet headers. The sub-PE has operations to break out the fields into smaller register files and reassemble them. This hierarchy of register files allows computation to occur on different parts of the header in parallel.

The IPv4 sub-PE has function units for doing checksum calculations, testing the values of fields, decrementing fields, and incrementally adjusting the checksum. All of these operations are inspired by the elements in the Click application model. This combination of function units and

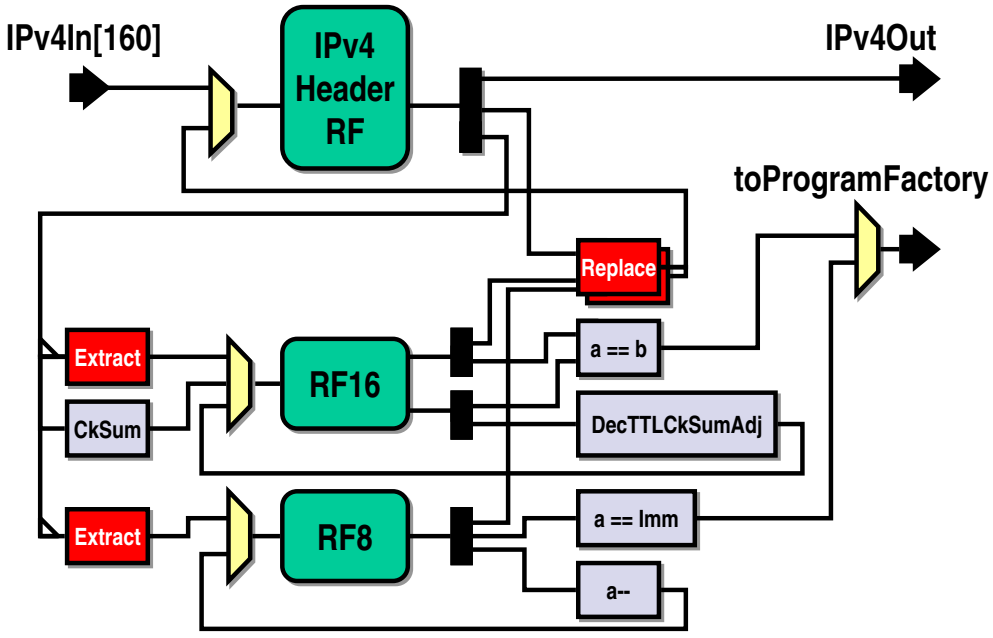


Figure 7.13: IPv4 Sub-PE

register files of various widths supports the datatype-level and data-level concurrency in the application.

The ClickPE is also interesting for what it omits. There is no program counter or instruction fetch and decode logic. Instead, the ClickPE will execute in response to signal-with-data messages from itself or from other PEs in the multiprocessor architecture.

By omitting the standard RISC control logic, the ClickPE architecture is not only simplified, it also realizes a style of process-level concurrency that is a better match for the requirements of the application. The PE will run programs in response to signal-with-data messages that correspond to Click push and pull communications events.

LuleaPE Architecture

The LuleaPE is a trimmed-down version of the ClickPE designed to perform the Lulea longest prefix match algorithm. It is shown in Figure 7.14. This architecture trades off reusability (since it supports fewer Click elements) for higher performance and lesser resource utilization. This PE is meant to perform one stage of the three-stage Lulea algorithm. In Section 7.2.3, a pipeline of three LuleaPEs will be built to do the entire lookup algorithm.

The Lulea approach involves performing several table lookups using parts of the IPv4 destina-

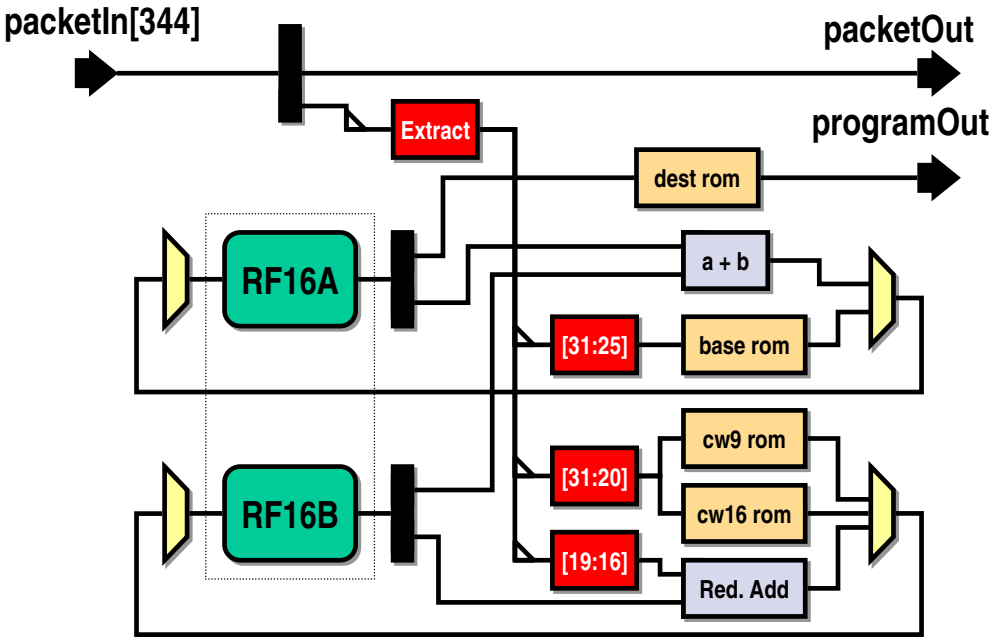


Figure 7.14: Sub-RISC PE for Lulea Longest Prefix Match

tion address as indices. In [30], the sizes of these tables are constrained by the byte-level granularity of a traditional processor's memory space. The FPGA offers embedded memories that are 18 bits wide, and the Lulea algorithm can be modified to take advantage of this. Several of the lookup tables can be made smaller as a result, allowing a larger routing table to be stored with fewer BRAM blocks. These lookup tables are implemented as function units directly in the LuleaPE datapath. They can be accessed concurrently, thereby improving data-level concurrency.

Another modification from the original Lulea algorithm is in the reduction add operation. This was originally done using a lookup table. The LuleaPE provides a hardware operation for this, eliminating extra memory references and better matching datatype-level concurrency.

Several of the ClickPE's wide register files are absent in the LuleaPE. This is because the LuleaPE only requires read access to the packet header. No temporary storage for modified headers is required. The LuleaPE reads the header directly from the *packetIn* port, where it arrives as the data component of an incoming signal-with-data message. This saves extra cycles that the ClickPE needs to move data to sub-PEs.

7.2.2 A Sub-RISC PE for Regular Expression Matching

To construct a processing element for regular expression matching, architects follow the same approach. Analyzing the computations found in the application *regexp* actors reveals opportunities for data-level and datatype-level concurrency. Related work in pattern matching suggests what architectural features will be important.

- *Application Knowledge:* Diagrams like Figure 5.35 show what kind of computations are necessary for performing regular expression matching with non-deterministic finite automata. These models can be interpreted as bit-level dataflow diagrams that show how next state variables and outputs are computed as functions of current state variables and inputs. Since all of the data dependencies are explicitly represented, all of the potential data-level concurrency is visible. Each of the next state and output functions can potentially be evaluated in parallel. The model also reveals the application's datatype-level concurrency. Operations like single-bit Boolean ORs, ANDs, and multi-bit character equality comparisons are frequently used.

- *Architectural Knowledge:* The application's single-bit computations should be a good match for the fine granularity of the target FPGA. A lightweight datapath with small area requirements is expected. The application does not require a wide ALU or a wide register file for storing intermediate results.

- *Prior Experience:* Many systems for pattern matching have been proposed. Some of these approaches only perform exact string matching and not full regular expression matching, and are therefore unable to detect all of the attack scenarios found in the Snort database. Nevertheless, it is still useful to study the successes and pitfalls of this prior work.

  Haagdorens et al. [46] focus on the process-level concurrency in pattern matching applications. In this work multiple network flows are processed in parallel against the same pattern database. The target architecture is a dual-processor Xeon with Hyperthreading. Despite hardware support for 4 concurrent threads, only a 16% improvement in throughput is found. Synchronization and communication between processes is a significant bottleneck. Focusing on process-level concurrency alone is not sufficient for a scalable parallel implementation.

  To exploit data-level and datatype-level concurrency, FPGAs are often used. Baker and Prasanna build a circuit to accelerate the Knuth-Morris-Pratt algorithm for exact string matching in [4]. This design can process one text character per cycle for a throughput of 2.4 Gb/sec.

The search pattern is programmed into the FPGA's embedded memories. However, network intrusion detection requires hundreds of patterns of the size presented (16 to 32 characters).

The Granidt approach uses a content-addressable memory (CAM) to compare an entire text string against many patterns simultaneously [43]. This scales up to 32 patterns of 20-byte length. After this, the frequency of the design drops and performance suffers.

Several works try to scale further by comparing fewer text and pattern characters at the same time. Baker [5] and Sourdis [112] use partitioning algorithms on the pattern database to avoid instantiating redundant comparators. This leads to a reduction in area in exchange for larger fan-outs and longer routing delays for the comparator output signals.

Cho and Mangione-Smith [22] instantiate comparators for only the prefix of a pattern, and store the suffix in an on-chip ROM. If a prefix match is found, the suffix is retrieved and compared one character at a time to complete the match. The sequential comparator used for suffix matching further reduces the design area. These three approaches achieve gigabit throughput rates against hundreds of simultaneous patterns. Bloom filters offer the possibility of scaling to thousands of patterns, if one accepts a small probability of false positives [32].

Full regular expression matching is often necessary to match Snort patterns. Deterministic finite automata can be implemented in either hardware or software, but require exponential space in the worst case. Non-deterministic finite automata (NFAs) do not have this problem and can be implemented on FPGAs. The original work by Sidhu and Prasanna generates a hard-wired circuit that implements a fixed NFA [108]. These circuits fully exploit data-level concurrency by evaluating all non-deterministic state transitions in parallel in one cycle. The required space is linear in the size of the regular expression and the machine runs in constant time. Franklin et al. [38] achieve between 30 and 120 MHz depending on the size and complexity of the regular expression. This equates to 30-120 MB/sec of throughput.

A common downside to the NFA-based approaches is scalability. Complex regular expressions lead to circuits with long combinational delays which degrade performance. Second, it is not possible to explore different area/performance trade-offs. There is only one design point which is a complete parallelization of the algorithm. Third, these designs depend on the configurability of the FPGA to be able to change the search patterns. One must synthesize, place and route the design each time. The generality of the FPGA comes at the cost of large silicon area, high power requirements, and low operating frequencies.

```
match_output  :==  expr
next_state    :==  expr
expr          :==  constant
                 |  state
                 |  compare(state, text, pattern)
                 |  expr or expr
                 |  count_range(expr, counter, N, M)
```

Figure 7.15: NFA Next State Equation BNF

To avoid these shortcomings, a new Sub-RISC PE for regular expression matching will be designed. This NFA PE executes non-deterministic finite automata generated from regular expressions with only a 1-bit wide datapath and 3 function units. Software programmability avoids the dependence on FPGA reconfiguration to change the search patterns. Therefore, the NFA PE architecture can be realized as an ASIC for higher throughput and lower cost. Unlike the synthesis approaches, more complex regular expressions correspond to longer PE programs, not larger circuits. The fixed datapath gives predictable performance in all cases.

NFA PE Architecture

If one writes out the next state equations and output equations for a Sidhu and Prasanna NFA (e.g. Figure 5.35) by hand, a simple pattern emerges. All of the signal values in the NFA circuit can be calculated by recursively applying a small set of functions. This can be easily described by the BNF syntax shown in Figure 7.15. NFA outputs and next state variables are both *exprs*. An *expr* can be a constant (0 or 1) or a current state bit. It can also be the result of the conditional comparison function found in the leaves of the Sidhu and Prasanna tree.

There are two ways to build an *expr* recursively. One is to take the Boolean OR of two *exprs*. This is an operation found in many metacharacter nodes. The other is to apply the count-and-range functions found in the curly-brace metacharacters to an *expr* (Figure 5.36).

These are the only operations necessary to match a regular expression. The NFA PE architecture, shown in Figure 7.16, is a simple 1-bit wide datapath that can perform exactly this set of operations.

There are two register files. The main one-bit wide RF stores NFA state bits and the results of intermediate computations. The counter RF stores state for curly-brace metacharacters.

It may seem that this figure is a simplification, but it is in fact a complete Sub-RISC architecture specification. The TIPI design tools allow architects to leave many aspects of the control logic

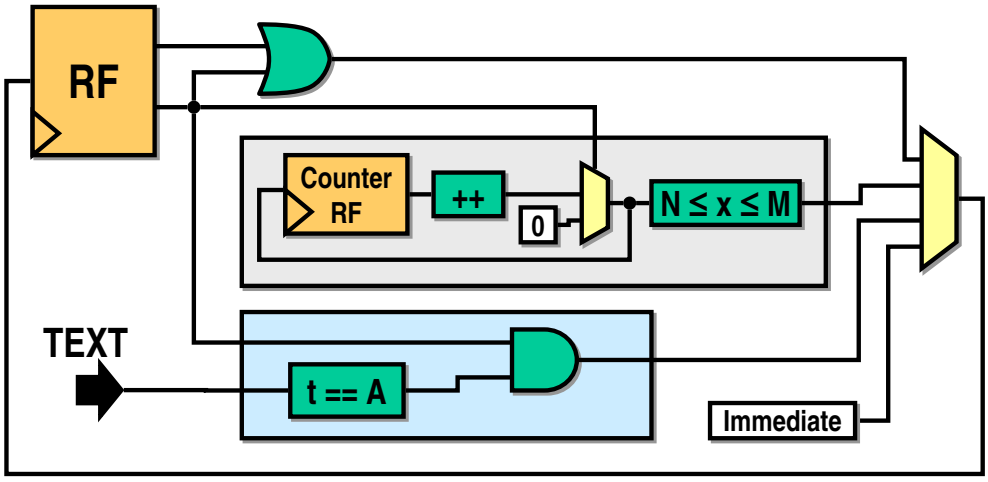


Figure 7.16: Sub-RISC PE for Snort Regular Expression Matching

unspecified. In this diagram there are no read address, write address, or write enable signals on the register files. The 4-to-1 multiplexer has no select signal. The comparators are missing inputs for the *N*, *M*, and *A* values. In the Sub-RISC approach, these signals are implicitly connected to a horizontal microcode control unit. The operation extraction algorithm discovers the settings that have valid meaning and can be used by statically-scheduled software.

There are five valid operations for this datapath. The first is simply the *noop* operation where the machine idles for a cycle. There is an operation that performs a Boolean OR, one that does the count-and-range function, one for the conditional compare function, and one that writes an immediate constant into the register file.

This datapath is also parameterized as follows:

- *Register File Depth:* This controls how many NFA state bits the PE can store. With a deeper register file, the PE can evaluate more complex regular expressions. In this design example a default value of 256 is used.

- *Counter RF Depth:* Each curly-brace metacharacter in a regular expression requires its own counter. One pattern in the current version of the Snort database requires 17 counters, and the rest use 14 or fewer. NFA PEs with 16 and 32 counters will be mentioned in Section 7.4.

- *Counter RF Width:* This determines the maximum values of *N* and *M* that can be used in a curly-brace metacharacter. The NFA PE datapath uses type-polymorphic TIPI components to automatically match this parameter. The largest value found in the Snort database is 1075, so the counters are set to be 11 bits wide.

- *Character Width:* Normal ASCII characters are 8 bits wide, but in this design example 9 bits are used to allow for ample out-of-band characters (e.g. the $ which matches the end of a text stream).
- *Program Memory Depth:* This is the maximum length of the program the PE can run to execute an NFA. A value of 512 is chosen for implementation on a Xilinx FPGA, as this is the shallowest BRAM configuration. For ASIC designs, a 256-entry program memory is used.

This datapath architecture is a way to time-multiplex the logic found in an NFA circuit. It can execute any regular expression up to the complexity bounded by the architectural parameters. The more complex the expression, the more datapath clock cycles it takes to evaluate each NFA iteration.

7.2.3 Complete Multiprocessor Architecture

Figure 7.17 shows the proposed multiprocessor architecture for the security gateway. It is composed of ClickPEs, LuleaPEs, and NFA PEs. This architecture closely matches the structure of the application shown in Figure 7.2. Separate PEs serve the input chains, output chains, and *LookupIPRoute* components of the application.

On the left side of the multiprocessor schematic, one ClickPE services each Ethernet interface. These PEs will implement the Click elements found in the input chain of Figure 7.3, and the portion of the output chain after the security elements in Figure 7.4. Incoming packets will first be processed at these ClickPEs, and then sent to a pipeline of three LuleaPEs that implement the longest prefix match algorithm.

The LuleaPEs implement the three stages of the Lulea algorithm. They are identical in datapath structure, except that the depth of the lookup tables increases in the second and third PEs.

After the routing decision is made, packets go to one of eight output chains in the application model. The first part of each output chain performs network intrusion detection. The multiprocessor provides groups of ClickPEs and NFA PEs to perform these tasks. There is one group for each of the eight network interfaces. The ClickPE in each group performs the header matching component of the network intrusion detection application, while the cluster of NFA PEs serve as coprocessors that perform the regular expression matching. The quantity of NFA PEs in each cluster is based on the performance of a single NFA PE and the complexity of the rules in the SNORT database.

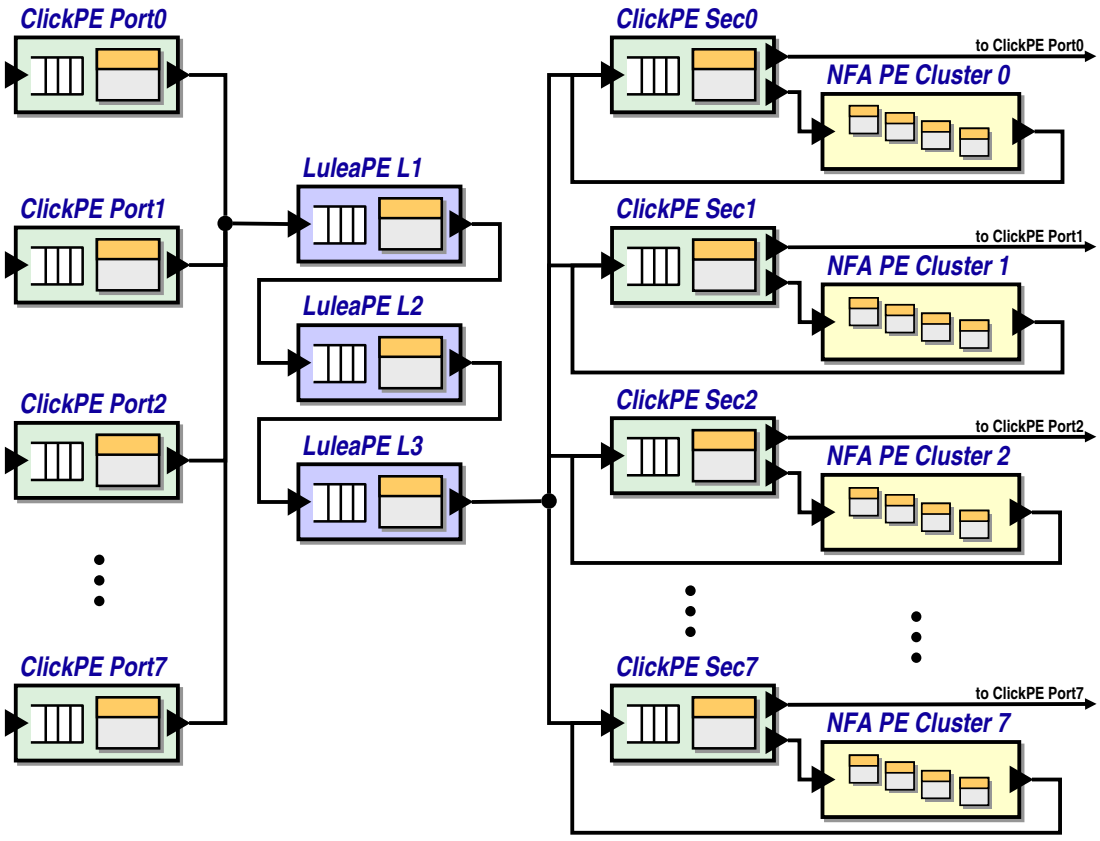


Figure 7.17: Multiprocessor Architecture for the Security Gateway

Architects use design space exploration to choose a cluster size large enough to meet the desired gigabit packet throughput rate.

After the security processing is complete, packets are returned to the ClickPEs on the left side of the schematic. These PEs perform the last computations in the Click output chains and then transmit the packets on the network interfaces.

7.3 Mapping and Implementation Results

The next step in the design process is to map the security gateway application onto the multiprocessor architecture. This creates a complete model of one particular design point. Analysis of the software and hardware performance of this design point provides useful information for design space exploration.

The layout of processing elements and the interconnect network of the multiprocessor in Figure 7.17 closely match the Click application model. This permits a straightforward mapping as described in the previous section. No special network-on-chip protocols are necessary for this example.

The contents of the mapping model for the PE labeled *ClickPE Port0* are shown in Figure 7.18. This PE implements all of the input chain elements for the first network interface, as well as the *CheckPaint*, *Discard*, *DecIPTTL*, and *ToDevice* elements of the corresponding output chain. PEs labeled *ClickPE Port1* through *ClickPE Port7* are mapped in the same way.

The three stages of the Lulea lookup are assigned to the pipeline of LuleaPEs. The *regexp* actors in the security facet of the application are mapped to the clusters of NFA PEs. The remaining actors in the security facet are mapped to the ClickPEs labeled *ClickPE Sec0* through *ClickPE Sec7*.

To obtain the software performance results of this design point, designers use Cairn's code generation process to convert the mapping model into executable software for the processing elements. This indicates how many processor cycles it takes to execute the various application computations.

No code is written by hand during the mapping and implementation processes. The transformed application models are used directly to program the multiprocessor. Thus, the functionality expressed in the original application models is preserved in the implementation. This fast and accurate implementation methodology is a key benefit of the Cairn approach.

To convert software cycle counts into actual packet throughput rates, it is necessary to characterize a hardware implementation of the target multiprocessor. Designers use Cairn's RTL code generator to build a synthesizable Verilog model. This model is run through the Xilinx FPGA tools

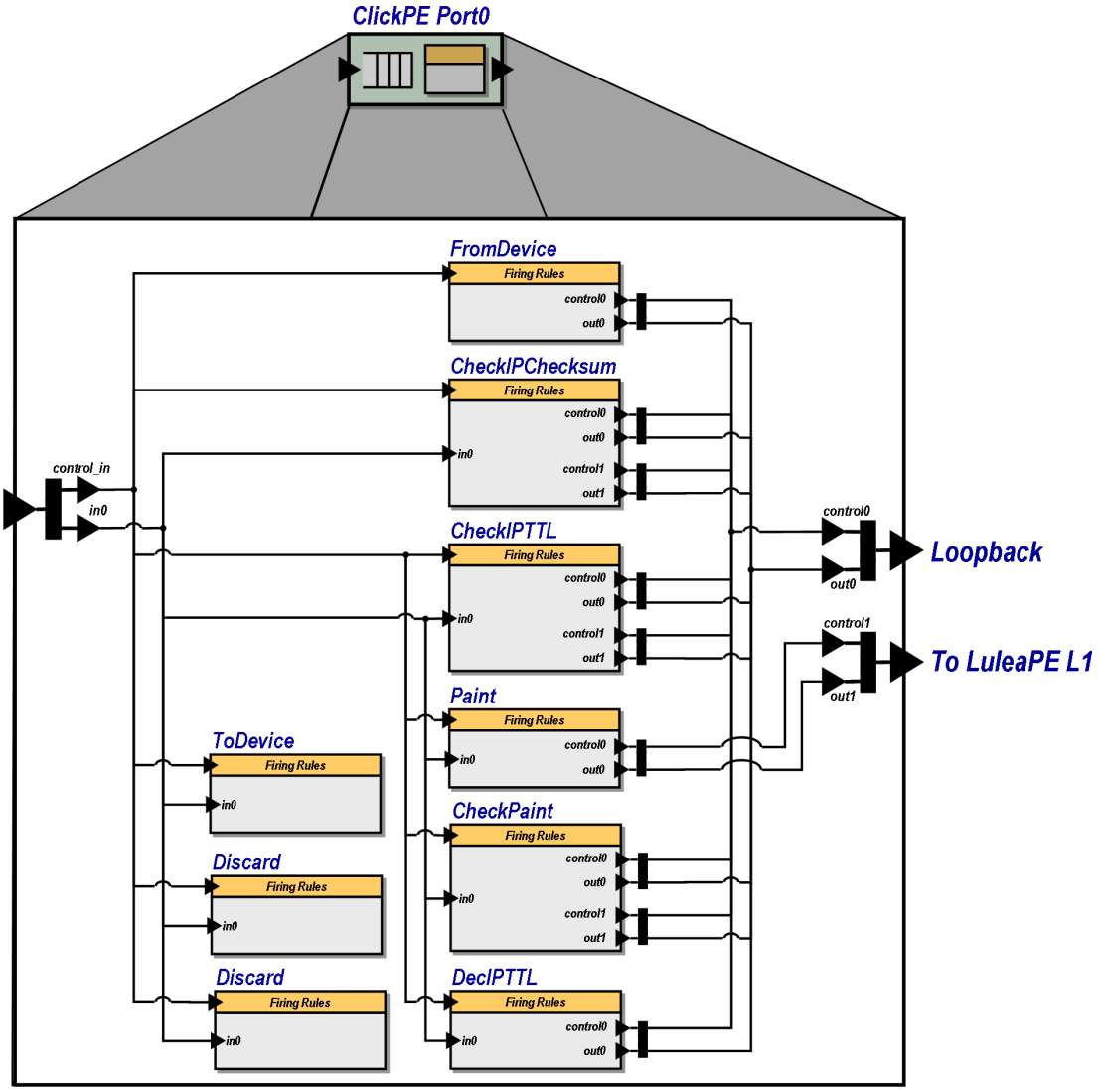


Figure 7.18: Contents of the Mapping Model for ClickPE Port0

Click Element	*ClickPE Cycle Count*
Paint	5
CheckPaint	4
CheckIPChecksum	12
DecIPTTL	10
CheckIPTTL	5
16-bit Longest Prefix Match	10

Table 7.1: ClickPE Cycle Counts for Mapped Click Elements

Architecture	*Slices*	*BRAMs*	*Clock Rate*
ClickPE	2841	12	110 MHz

Table 7.2: ClickPE FPGA Implementation Results

to obtain speed and area utilization data. When this is combined with the software performance numbers, the overall system performance figures are obtained. These results are summarized for the ClickPEs, LuleaPE, and NFA PEs in the following sections.

7.3.1 ClickPE Performance

Table 7.1 gives optimal ClickPE cycle counts for the Click elements found in the security gateway application. Table 7.2 gives post-place-and-route implementation results for a single ClickPE on a Xilinx Virtex 2 Pro 2VP50 FPGA.

The ClickPEs labeled *ClickPE Port0* through *ClickPE Port7* handle incoming and outgoing packets for the gateway's eight network interfaces. Each incoming packet is processed by the *Paint*, *CheckIPChecksum* and *CheckIPTTL* elements, requiring 22 cycles of computation. An outgoing packet requires 14 cycles for the CheckPaint and DecIPTTL operations, for a total of 36 cycles per packet at these PEs. At 110 MHz the ClickPE can process over 3 million packets/sec. Assuming a minimum-sized 64-byte Ethernet packet with a 64-bit preamble and a 96-bit inter-frame gap, this corresponds to a line rate of 2 Gb/sec. This is ideal for a full-duplex gigabit Ethernet interface.

For this experiment, additional hardware was included in the IPv4 sub-PE for performing a 16-bit longest prefix match. This is not shown in Figure 7.13 for clarity. This operation is used in the PEs labeled *ClickPE Sec0* through *ClickPE Sec7* to perform the header field matching component of the network intrusion detection application. The lookup tables in the datapath are large enough to support 10,000 prefix table entries. The longest prefix match operation always takes 10 cycles regardless of the number of entries in the table.

Architecture	*Slices*	*BRAMs*	*Clock Rate*
Lulea L1, L2, L3 Pipeline	2487	219	125 MHz

Table 7.3: LuleaPE FPGA Implementation Results

Lulea Stage	*LuleaPE Cycles*	*Packets/sec*	*Rate*
L1	8	15.6 M	10.5 Gb/sec
L2	10	12.5 M	8.4 Gb/sec
L3	10	12.5 M	8.4 Gb/sec

Table 7.4: Performance Results for the LuleaPE

7.3.2 LuleaPE Performance

Tables 7.3 and 7.4 show the hardware and software performance results for the pipeline of three LuleaPEs. The lookup tables contained within these three PEs are large enough to hold a 100,000-entry routing table. They are implemented in a total of 435 KB of on-chip BRAM memory.

The *LuleaPE L2* and *LuleaPE L3* PEs have larger tables than the *LuleaPE L1* PE. The extra two cycles these PEs use to process a packet is due to extra pipelining in these large RAMs.

The L1 PE matches the first 16 bits of the IPv4 destination address. If a match in the routing table is found at that point, the result is forwarded through the L2 and L3 PEs without requiring any further computations. This achieves the maximum throughput of one packet every 8 cycles, or 10.5 Gb/sec.

The L2 PE matches the next 8 bits of the destination address, and the L3 PE matches the final 8 bits. For packets that match prefixes of lengths between 17 and 32 bits, one or both of these PEs must perform computations, so the lower throughput rate of one packet every 10 cycles (8.4 Gb/sec) is achieved. A random mix of incoming packets will result in a processing rate of between 8.4 and 10.5 Gb/sec of traffic. Again, this performance is ideal for eight gigabit Ethernet interfaces, even for worst-case packets.

As expected, tuning the LuleaPE to specialize at one computation results in a PE that is both smaller and faster than the ClickPEs. Three LuleaPEs together occupy about the same number of slices as a single ClickPE.

7.3.3 NFA PE Performance

To evaluate the software performance of the NFA PE, it is programmed with regular expressions extracted from the Snort pattern database. Each regular expression is used to configure a Cairn

Regexp Component	*Cycle Cost*
Single character	1
|, +, ∗ metacharacters	1
{ } metacharacter	2
? metacharacter	2
Concatenate metacharacter	0

Table 7.5: NFA PE Cycle Counts

Regular Expression domain actor using the model transform described in Section 5.6.3. This actor is then mapped to an NFA PE, and the automatic code generation process is used to produce a software implementation.

The resulting program performs one iteration of the non-deterministic finite automata that matches the desired regular expression. One character from the incoming text stream is consumed and the PE calculates the automata's match output and next state functions. This program is repeated for each character in the text stream. The result is a stream of match bits that indicate if the regular expression has matched at each point in the text stream.

The more complex the regular expression, the more NFA PE cycles it will take to evaluate each iteration of the automata. This cycle count can be approximated by adding up the cost of each regular expression component, as given in Table 7.5. There are 385 unique regular expressions in the Snort pattern database. This approximation was calculated for all of them, and the results are plotted in the histogram shown in Figure 7.19.

The *X* axis shows the length of each regular expression program in NFA PE cycles. The *Y* axis shows the number of Snort regular expressions with each length. While most regular expressions can be matched in under 40 cycles per text character, there are several that require 180 cycles or more. Especially troublesome is the fact that 20 regular expression require more than 250 cycles per text character. The worst regular expression requires 731 cycles. This is not only bad for performance, but it requires the NFA PEs to have larger instruction memories than expected. The NFA PE has a parameterized instruction memory depth with a default value of 256 entries. If the regular expression program is longer than this, it will not fit in the memory and the NFA PE will not be able to match the desired regular expression at all.

Analysis of the worst-offending regular expressions gives clues to the cause of this problem. These regular expressions make frequent use of the [ ] grouping operator to perform set comparisons. In the current datapath this must be expanded using the | metacharacter, e.g. $[\mathrm{a–c}] = (\mathrm{a}|\mathrm{b}|\mathrm{c})$. This

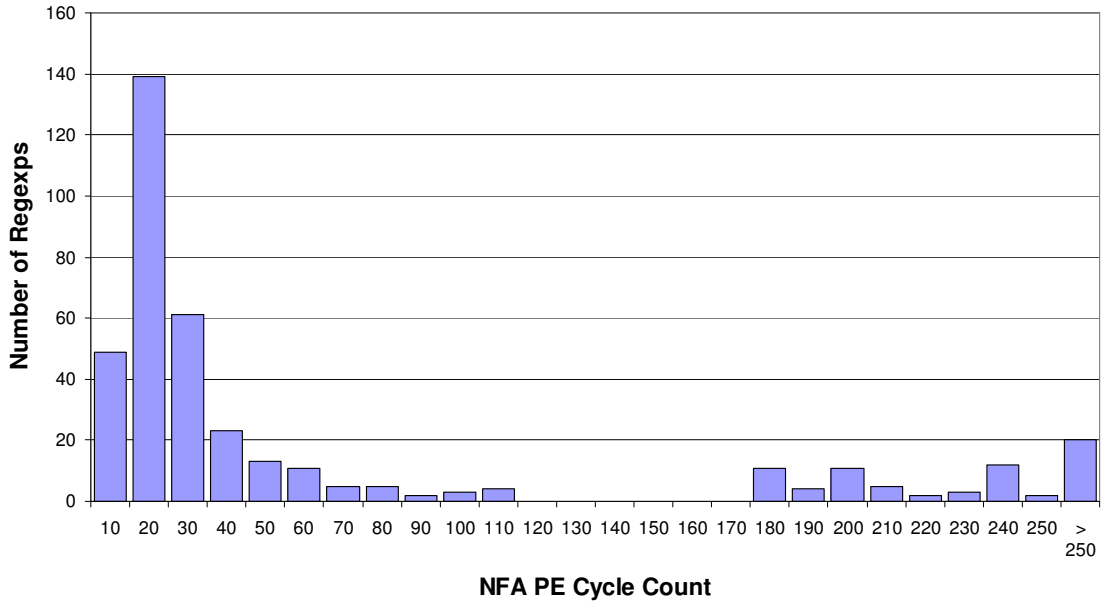


Figure 7.19: Histogram of Snort Regular Expression Program Cycle Counts

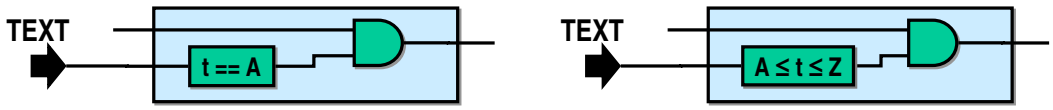
costs a large number of cycles, especially when the dash character is used inside of the square brackets to specify ranges of characters. For example, [0–9A–Fa–f] matches any hexadecimal digit, but requires 22 single-character equality comparisons and 21 | metacharacters to OR the results together.

Other constructs that appear frequently in the worst-case regular expressions are the \d, \s, and \w special characters. The \d character matches any digit, \s matches any whitespace character, and \w matches any word character. These also get expanded out to a long series of single-character comparisons that must be combined with the | metacharacter.

The NFA PE will perform much better if it can support set comparisons directly in hardware. In the next section, architectural design space exploration will be used to make modifications to the NFA PE to better match the application's requirements.

7.4 Design Space Exploration

The software performance results given in the previous section suggest that the NFA PE's single-character equality comparison function unit is not powerful enough to support regular expressions that make frequent use of [ ] set comparison leaves. One idea to solve this problem is to replace the equality comparator ($t = \mathrm{A}$) with a range comparator ($\mathrm{A} \leq t \leq \mathrm{B}$). This would allow



(a) Before Comparator Replacement (b) After Comparator Replacement

Figure 7.20: Experimenting With a Different Text Comparator

the NFA PE to execute an expression like [0–9A–Fa–f] with only three comparison operations and two OR operations: a significant savings over the previous 22 comparisons and 21 ORs.

The TIPI architecture design tools make it trivial to experiment with this kind of change to the architecture. Architects simply replace the equality comparator in the schematic diagram with a range comparator. This new component can be found in the library of datapath elements. Figure 7.20 shows before-and-after schematics of the relevant part of the NFA PE datapath.

It is not necessary to make connections to the A or B inputs of the new comparator. These signals are implicitly connected to the PE's horizontal microcode unit. The bounds of the range comparison, like the value of the equality comparison earlier, become immediate fields in the instruction word.

In some other architecture description languages, a change to the datapath like this creates an inconsistency between the ISA specification and the PE architecture specification. Architects may have to manually update the ISA to reflect the change, for example by changing the instruction encodings. This is not necessary in TIPI. The operation extraction process automatically updates the PE's operation set. It determines that a new range comparison operation is available, and that the previous equality comparison operation is no longer available. The rest of the operations are shown to be unchanged.

Changes to an ISA often require changes to a compiler, but again this is not necessary in the Cairn approach. Cairn's retargetable code generation process takes the PE's operation set as an input, so it is flexible to this sort of architectural change. Designers simply run the process again to build new machine code programs for the modified PE. It is not necessary to change either the application model or the mapping model. Cairn helps designers see the results of their architectural changes quickly and with a minimum of effort.

Based on the success of this first experiment, architects can try additional changes to further improve the NFA PE's performance. The next idea is to add support for the \d, \s and \w characters. To do this, architects add three hard-wired comparators in parallel with the range comparator. AND

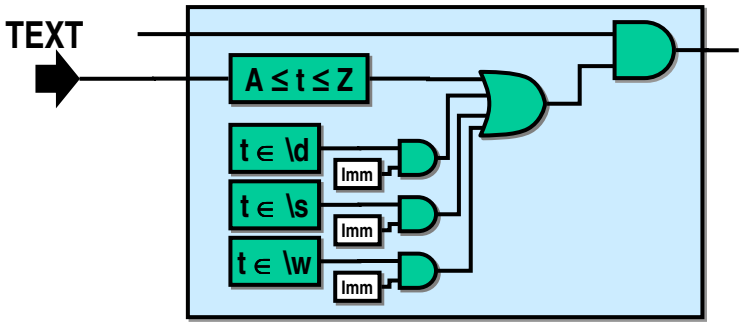


Figure 7.21: Adding Support for the \d, \s, and \w Special Characters

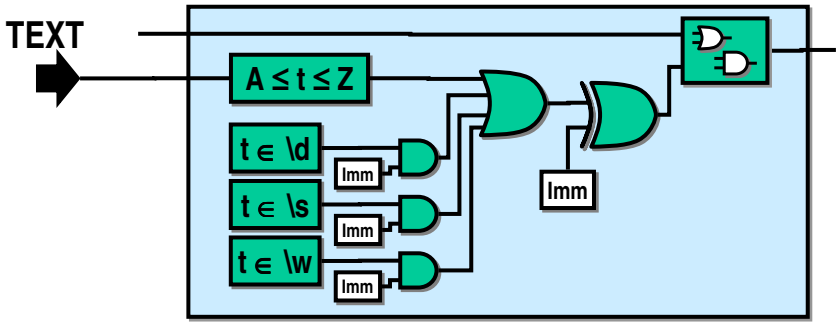


Figure 7.22: Adding Support for Inverse Set Leaves With Multiple Ranges

gates allow the results of these comparisons to be conditionally included with the result of the range comparison. Figure 7.21 shows this modification.

This allows the \d, \s and \w characters to be matched in only one cycle. Additionally, if any of these characters appears in a [ ] set along with a regular character or range, they are free. For example, [0–9A–Fa–f] can be written as [\dA–Fa–f]. The \d and A–F comparisons can be done in parallel by the \d hard-wired comparator and the range comparator respectively. The range comparator performs the a–f comparison on the next cycle, and then only a single OR operation is necessary to combine the results. A regular expression that used to take 22 comparisons and 21 ORs can now be done in a total of 3 cycles.

The next idea is to add hardware support for the [^] syntax. This is an inverse set comparison that matches when the given character is not a member of the set. To do this, an XOR gate is added to the datapath to conditionally invert the result of the character comparators. One input of this gate is left unconnected, and is thus driven by an immediate field in the microcode.

To speed up [ ] leaves with multiple items (e.g. [0–9a–f]), the AND gate and OR gate in the original datapath are replaced with logic elements that can perform either operation. Figure 7.22 shows this modification. The expression [0–9a–f] becomes $State \wedge ((0 \leq t \leq 9) \vee (\mathrm{a} \leq t \leq \mathrm{f}))$.

Regexp Component	*Cycle Cost*
Single character or \d, \s, \w	1
[ ] with N characters or ranges ($N > 1$)	$N + 1$
[^] with N characters or ranges	N
|, +, $*$ metacharacters	1
{ } metacharacter	2
$\{0, M\}$ metacharacter	3
? metacharacter	3
Concatenate metacharacter	0

Table 7.6: Modified NFA PE Cycle Counts

Similarly, [^0–9a–f] is $State \wedge \neg(0 \leq t \leq 9) \wedge \neg(\mathrm{a} \leq t \leq \mathrm{f})$. This change allows [ ] leaves with N characters or ranges to be matched in only $N + 1$ cycles. Inverse [^] leaves with N characters or ranges require only N cycles.

The final design space exploration activity is to make three small changes to the datapath. First, the datapath can benefit from pipelining. TIPI allows users to create irregular pipelines by adding pipeline registers wherever they are necessary to break critical paths. It is not necessary for architects to restrict themselves to the traditional 5-stage RISC architectural design pattern. Synchronous read ports are added to the register files, and a pipeline stage is added before the inputs of the 4-to-1 mux. The combinational paths through the count-and-range function unit and the text comparison unit are also broken by pipeline registers.

Second, the immediate field that writes a constant into the register file can be removed entirely by exploiting special cases of the count-and-range function unit. A zero is obtained by executing the count-and-range operation with $N = \infty$ and $M = 0$. These values are reversed to obtain a one. The write enable for the counter RF is disabled in these cases to avoid disrupting counter state.

Lastly, four pairs of single-bit input and output ports are included for communicating with neighboring NFA PEs in a tiled cluster. This allows a group of NFA PEs to send signal-with-data messages to each other, and also back to the ClickPE that does the header matching component of the network intrusion detection application. Match results from several NFA PEs running in parallel can be combined using this mechanism and used by the ClickPE to decide how to process a packet.

Figure 7.23 shows the final NFA PE architecture. Table 7.6 shows the updated cycle costs of the various regular expression components. All 385 unique Snort regular expressions were evaluated again and the resulting histogram is shown in Figure 7.24. The new cycle count distribution is shown as solid bars. For comparison, the old distribution is shown as a line.

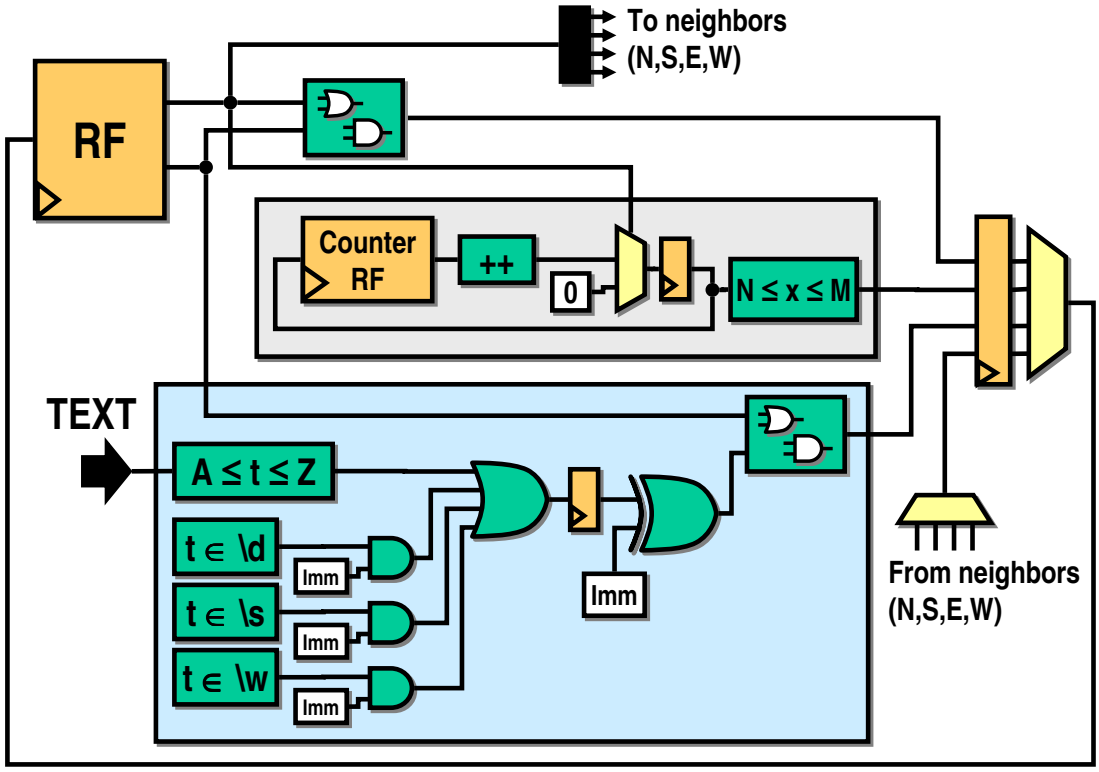


Figure 7.23: Final Modified NFA PE Schematic

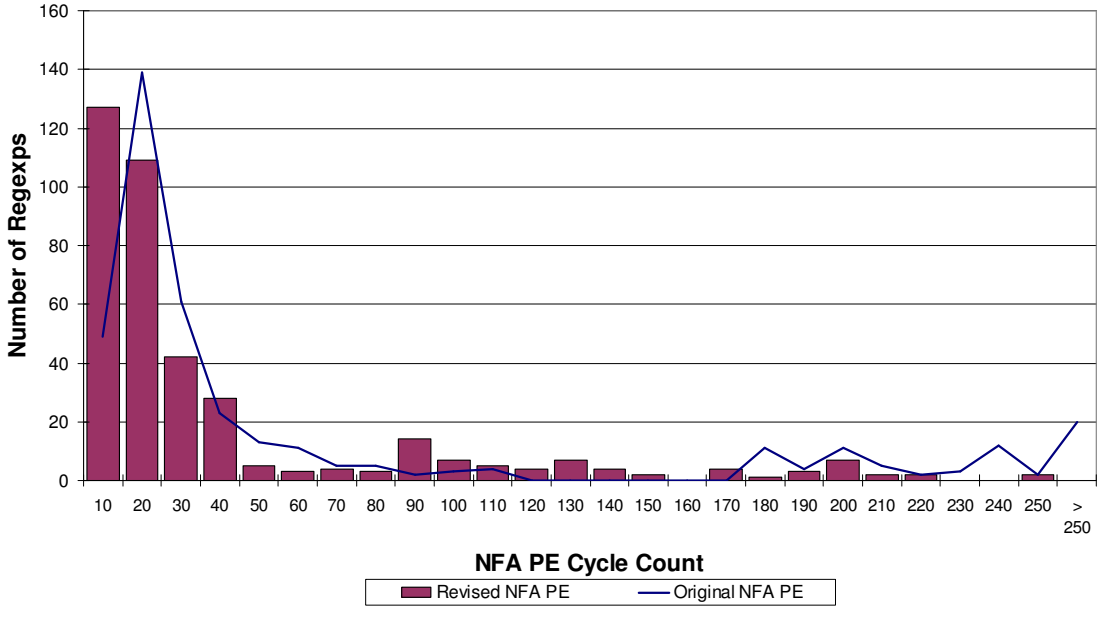


Figure 7.24: Modified NFA PE Cycle Count Histogram

Architecture	*Slices*	*BRAMs*	*Clock Rate*
16 Counters	142	3	185 MHz
32 Counters	152	3	185 MHz

Table 7.7: Modified NFA PE FPGA Implementation Results

With the modifications, no Snort regular expression requires more than 250 cycles per character. This solves the most pressing problem that some expressions were too big for the PEs instruction memory. The most complex regular expression requires 246 cycles, while the majority take under 40 cycles. Even small regular expressions benefit from the architectural changes. The peak in the histogram has moved down from the 10-20 cycle category to the under 10 cycles category.

Over all 385 regular expressions, there are 7,599 single character, \d, \s, and \w comparisons. The [ ] metacharacters require an additional 2,354 comparisons. The range comparator proves its worth: If all ranges were expanded to single character comparisons, 14,010 comparisons would be required instead. The total cost of all 385 regular expressions is 14,058 PE cycles. The NFA PE spends 70% of its time doing comparisons and 30% of its time evaluating metacharacters.

To support all 385 regular expressions, each cluster of NFA PEs should have at least 55 PEs. This is the minimum number of PEs necessary to have enough total instruction memory to hold all of the programs if each PE has a 256-entry memory. Each cluster could potentially evaluate all 385 regular expressions in parallel. Every 256 cycles, the cluster updates all 385 NFAs and consumes one text character. In practice, header field matching will determine that a packet only needs to be searched for a subset of the regular expressions. Therefore the PEs will normally be used to process multiple independent packets in parallel.

Modified NFA PE FPGA Implementation Results

To measure the hardware performance of the NFA PE, two different Xilinx XC2VP50-5 designs are considered. One has a 16-entry counter register file, and the other has a 32-entry register file. The 16-entry PE is capable of evaluating expressions with up to 16 curly-brace metacharacters. The most of these metacharacters that appear in any one Snort regular expression is 17. Therefore at least one PE in each NFA PE cluster should be a 32-counter version.

Area requirements and clock rates for the two designs are shown in Table 7.7. Embedded BRAMs are used for the instruction memories. These are configured for the minimum allowed depth of 512 words even though only 256 words are addressed.

Architecture	*Area* (mm^2)	*Frequency* (MHz)		
		Typ	*Typ/IO*	*Best/IO*
16 Counters	0.143	980	1042	1299
32 Counters	0.155	980	1053	1299

Table 7.8: Modified NFA PE ASIC Implementation Results

An NFA PE cluster with 55 members consumes 7,810 slices and 165 BRAMs. At 185 MHz, this machine matches one character against 385 regular expressions every 256 cycles for a worst-case throughput of 722 KB/sec. This worst-case performance is comparable to the average-case performance of a general-purpose processor running at a much higher clock rate.

Modified NFA PE ASIC Implementation Results

The NFA PE does not depend on the reconfigurability of the FPGA in order to change the search patterns. Only the software stored in the instruction memory needs to be changed. Higher performance can be achieved by implementing the NFA PE as an ASIC design instead of an FPGA design. This avoids the high area and clock frequency overhead of the reconfigurable logic.

Both NFA PE variants were synthesized with Synopsys Design Compiler for one of Infineon's 90 nm technologies. For the large instruction memory, a macro was generated. The other memories were synthesized.

For typical operating conditions, a clock frequency of 980 MHz is obtained, as shown in Table 7.8. The critical path in this experiment was dependent on external I/O, i.e. getting the stream of text characters into the datapath. Note that the text character input does not change every clock cycle. It only changes once per program iteration, which is typically 10 to 40 cycles. Therefore this critical path is pessimistic. I/O pipelining could also be used to break up this critical path. The most critical internal path allows for a 1.042 GHz clock rate (Typ/IO). For best operating conditions, the design can run at 1.3 GHz (Best/IO).

A 55 PE cluster will consume 7.9mm^2 and achieves throughput of 31.25 Mb/s at 1 GHz and 40.63 Mb/sec at 1.3 GHz. This is less than the maximum throughput of 1 Gb/sec that can be transmitted on each of the security gateway's network interfaces. However, only a fraction of the packets passing through the gateway will require regular expression matching. Header field matching is enough to determine that most packets are not suspicious.

Also, not every packet must be searched for all 385 regular expressions. To achieve a full gigabit line rate, a single NFA PE at 1.3 GHz can afford to spend 10 cycles per character (1300

Mcycles/sec / 125 Mchars/sec $\approx$ 10 cycles/char). If the packet stream can be broken up into 55 distinct flows, all of the NFA PEs in the cluster can operate in parallel on separate packets. This increases the cycle budget to 550 cycles/character. This is enough to process at least 13 average-sized regular expressions, which is ample for the patterns in the Snort database.

7.5 Performance Comparison

SMP-Click achieves 493K packets/sec on a dual-processor Xeon Linux workstation for a 2-port routing application that does not perform intrusion detection [21]. For 64-byte Ethernet packets with a 64-bit preamble and 96-bit inter-frame gap, this is approximately 320 Mb/sec of throughput. Synchronization and cache coherency between PEs are major bottlenecks in this implementation. When the number of processors was scaled up beyond two, no further performance gains were realized. The Sub-RISC multiprocessor in this design example better matches the process-level concurrency of the application and is more scalable. Eight gigabit Ethernet ports are supported at full line rate, and network intrusion detection is performed as well.

NP-Click implements a full 16-port Fast Ethernet router at rates between 880-1360 Mb/sec on the Intel IXP1200 network processor [106]. This application also does not perform intrusion detection. The ClickPEs and LuleaPEs perform better than the IXP microengines, although the NP-Click numbers include the cost of interfacing with actual Ethernet MACs. The NP-Click code spends about 50% of its time moving packet data on and off the IXP. A full Cairn design should surpass this limitation by using more PEs for packet I/O. The raw lookup performance of the LuleaPE pipeline matches the lookup rate of a hand-coded design for the IXP2800 [83]. The Cairn approach is competitive with this advanced network processor architecture but offers higher designer productivity. No hand coding is required at all.

StepNP can achieve a 10 Gb/sec forwarding rate, but this requires 48 ARM processors, each with 16 hardware threads and a 500 MHz clock [93]. These hardware requirements are significant, and performance is dependent on network-on-chip latency.

The Soft Multiprocessors approach achieves 2.4 Gb/sec on an FPGA with 14 MicroBlaze cores at 100 MHz [67]. This design requires 11,000 slices. Additional cores would be required to add support for network intrusion detection.

The Cliff router is able to handle 2 Gb/sec of traffic in about 4,000 slices [73]. This is a structural synthesis approach that converts a Click header processing application into a custom FPGA circuit. The area utilization numbers include the logic to interface to two gigabit Ethernet

MACs, but not the MACs themselves. The comparable Cairn design would have 2 ClickPEs and a pipeline of 3 LuleaPEs, requiring 8,200 slices plus additional logic for MAC interfaces. For the extra area, designers gain programmability and lookup performance headroom to scale to several more Ethernet ports.

The performance of the NFA PE can be compared against the work of Franklin, Carver and Hutchings [38] where a complete NFA circuit is implemented on an FPGA. This circuit is a completely unrolled design whereas the NFA PE is a completely time-multiplexed design. The authors use 8-bit equality comparisons, so [ ] expressions with ranges are expensive to implement (as was the case with the first version of the NFA PE). For all 385 regular expressions, a total of 20,742 single-character comparisons are required. Also, the curly-brace { } metacharacter is not considered.

For an NFA of this size approximately 1.25 slices/character are required for a total of 25,000 slices. This is slightly larger than the XC2VP50 used in this design example. Snort regular expressions do not feature deeply nested metacharacters, so the combinational delay is kept under control. Extrapolating from the published results, a frequency of 50 MHz should be attainable. This machine consumes one character every clock cycle for a throughput of 50 MB/sec. This high throughput is possible because the completely unrolled design fully exploits the data-level concurrency of the application.

The NFA PE trades off data-level concurrency for small area requirements. To achieve a comparable throughput, a single NFA PE running at 1.3 GHz can afford to spend 26 cycles per character (1300 Mcycles/sec / 50 Mchars/sec = 26 cycles/character). This is enough to match one short regular expression. However, the NFA PE requires only 150 FPGA slices whereas the complete NFA circuit requires 25,000 slices.

The fine granularity of the Sub-RISC approach allows designers to target more points on the performance/area trade-off curve. To match all 385 expressions at 50 MB/sec, ten clusters of 55 NFA PEs each can be employed. If designers do not want to allocate this much area to network intrusion detection, they can use a smaller number of PEs.

Also, the hardware performance of the NFA PE is not dependent on the complexity of the regular expression. Large expressions with deeply-nested metacharacters will lead to long critical paths with large fan-outs in a complete NFA circuit. The datapath will always run at the same frequency. Only the software programs become longer.

7.6 Summary

This chapter presented a complete design example to demonstrate the effectiveness of the Cairn methodology. All of the steps in the design process were shown. Designers begin by using models of computation and actor-oriented design principles to create a model of a multifaceted application. This model captures multiple styles and granularities of concurrency. By starting with a precise model of the application's opportunities for concurrency, designers will be able to exploit these opportunities later in the implementation process.

At the same time, architects construct a heterogeneous Sub-RISC multiprocessor architecture. The custom ClickPE, LuleaPE, and NFA PE datapaths are a first attempt at matching the process-level, data-level, and datatype-level concurrency requirements of the application. These processors are lightweight and fast, containing only the function units required by the application. After mapping the application to this architecture, designers can quickly see how well they did.

Cairn's disciplined mapping methodology allows designers to implement the application on the architecture quickly and correctly. Programmers do not have to rewrite the high-level application models in a low-level programming language for the individual PEs in the architecture. Instead, simple mapping assignments are used to describe how the application's computations are distributed across the PEs. An automatic code generator converts mapping models into executable software.

The initial performance results of the NFA PE led to ideas on how to improve the architecture to obtain better performance. Separate application, architecture, and mapping models make it possible to perform design space exploration with a minimum of effort. Architects can quickly modify the structural datapath schematic to realize the desired changes. It is not necessary to update an ISA specification, modify a compiler, change the application, or modify the mapping model to obtain new performance figures. The retargetable code generation process automatically implements the same regular expressions on the new architecture. Architects can perform multiple iterations of design space exploration quickly and easily.

This example shows that thorough and effective design space exploration leads to superior results. Programmable Sub-RISC PEs are lightweight, yet still provide high performance because they are a good match for the process-level, data-level, and datatype-level concurrency required by the application. The fast and accurate deployment methodology removes the need to write low-level code by hand for individual PEs. Instead, high-level application models are used directly for programming the architecture. By providing proper methodologies for concurrency, Cairn allows designers to realize the benefits of programmable architectures.

Chapter 8

Conclusion

Programmable multiprocessors have a great deal of potential as a successor to RTL-based design methodologies for a broad range of applications. Datapaths are a larger, coarser basic block than RTL state machines. This allows architects to design more complex systems with higher productivity. Programmability adds a software aspect to the design flow. Performing software iterations is much cheaper than performing hardware iterations. Additionally, software can be updated in the field to match changing application requirements, increasing the longevity of the architecture. These are desirable alternatives to the costs and risks of traditional ASIC design.

In the network processing domain alone, dozens of heterogeneous multiprocessor architectures have been created [105]. One thing they all have in common is that they are notoriously difficult to program. Heterogeneous multiprocessor architectures are concurrent systems. The fact that there are multiple datapaths running in parallel is only the beginning. In addition to process-level concurrency, the architectures also exhibit data-level concurrency and datatype-level concurrency.

In order to meet performance goals, programmers are looking to exploit all of the architecture's capabilities for concurrency. Their job is to match up the opportunities for concurrency that exist in the application with hardware resources that can implement this concurrency. If they are successful, then the application runs on the architecture with the expected performance. If not, then the architecture appears to be "difficult to program" [81].

The *concurrency implementation gap* makes this a challenging problem. Current methodologies exhibit a *systemic* ad hoc treatment of concurrency for systems in which concurrency is a first-class concern. Application designers are not using good abstractions for capturing the various styles and granularities of concurrency in their applications. Architects are not using good abstractions to export the concurrency capabilities of their architectures to programmers. The result is that

the concurrency capabilities of architectures are seldom suitable for the concurrency opportunities of applications. No one knows if the application is leveraging the architecture to the fullest, or if the architecture provides features that actually benefit the application.

Producing one functional implementation of a given application on a target architecture is the definition of success. There is no time to explore the design space. However, design space exploration is the very thing that makes programmable multiprocessors interesting. By working with coarser basic blocks, architects are supposed to have the power to freely experiment and find an architecture that is a good match for the intended application domain. Software programmability is supposed to allow programmers to build their applications incrementally and to make changes quickly and cheaply. The concurrency implementation gap ruins the promise of programmable multiprocessors by making it difficult to perform this necessary iterative design.

The goal of Cairn is to provide a new design methodology for heterogeneous multiprocessors that treats concurrency as a first-class concern. Instead of ad hoc approaches, designers use formal abstractions for concurrency. This includes the concurrency found in multifaceted applications as well as the concurrency found in heterogeneous multiprocessor architectures. Cairn considers implementation on a novel class of target architectures that can exploit this concurrency in hardware. The proper consideration of concurrency allows designers to understand and cross the concurrency implementation gap quickly and efficiently. This makes it possible to perform effective design space exploration and arrive at superior systems.

8.1 Satisfying the Three Core Requirements

The central hypothesis of this dissertation is that there are three core requirements that a design methodology must meet in order to solve the concurrency implementation gap.

8.1.1 Use High-Level Application Abstractions to Capture Concurrency

The first core requirement is to use a formal high-level abstraction to model concurrent applications. This abstraction captures the application's inherent opportunities for parallelism in an implementation-independent way. It includes all of the styles and granularities of concurrency found in multi-faceted applications. By starting with a precise model of the application's inherent requirements, designers will better be able to exploit these opportunities later in the design process.

Application abstractions increase productivity by giving domain experts an intuitive mecha-

nism for modeling complex concurrency issues. Concurrency can be described implicitly instead of explicitly. Then, programmers can focus on concepts that are important to the application domain. These abstractions work by simultaneously assisting and restricting the programmer. By following a set of pre-defined rules chosen to be natural for the application domain, designers are freed from the burden of describing implementation details.

Inspired by systems like Ptolemy II, Cairn uses models of computation and the concept of actor-oriented design as the basis for application abstractions. Designers use compositions of models of computation to model multifaceted applications. Models of computation describe the concurrency between actors. Cairn provides an actor language based on Cal to model the data-level and datatype-level concurrency within actors.

Model transforms automatically convert the abstract semantics of these high-level application models into concrete Cairn Kernel models. In the Cairn Kernel, communications and control between actors are implemented with transfer messages. Transfers are designed to be a good match for the primitive communication semantics of Sub-RISC multiprocessors. This representation of the application is ready to be mapped to the target architecture.

Pros and Cons of Cairn's Application Abstraction

Cairn's application abstraction is designed for domain experts who understand what they want to do but are unsure how to best accomplish their goals in hardware. From the programmer's point of view, Cairn makes it possible to write programs that contain complex concurrency correctly. Models of computation allow programmers to avoid the incomprehensible behaviors found in thread-based programs [77]. Since the application abstraction hides hardware implementation details from the programmer, these application models work well for experimenting with implementing the same application on several different architectures. The application model by itself is not tied to any one particular architecture.

Cairn avoids the shortcomings of C-based behavioral synthesis approaches (e.g. Cynthesizer, CatapultC, and others described in Section 3.3.2) because it does not depend on the C language. C works best as an architectural abstraction for sequential general-purpose processors. It does not succeed at describing the capabilities of application-specific embedded programmable elements. Also, it does not succeed at describing the multiple granularities of concurrency found in embedded applications. Despite its popularity, C is inappropriate for concurrent systems.

Approaches based on single domain-specific languages (e.g. NP-Click, Cliff, Accelchip, and

others described in Section 3.3.2) fall short when heterogeneous applications are considered. Languages like Click and MATLAB excel at modeling certain application facets, but are inappropriate or non-intuitive for others. Cairn avoids this pitfall by allowing programmers to create hierarchical application models that use compositions of models of computation.

A primary downside to the Cairn application abstraction is that it requires application designers to learn several new syntaxes and models of computation for programming. Each application facet should be described with a different model of computation. This can be a barrier to adoption. However, when a language does a good job at helping the designer visualize the problem they are trying to solve, the challenge of learning a new language is easily overlooked.

Another trade-off associated with providing an implementation-independent abstraction to the programmer is that complexities are moved into the model transform processes (e.g. those described in Section 5.6). These transforms must make some design decisions on how to implement the semantics of the high-level model of computation using Cairn Kernel transfer messages. From this point of view, the model transform begins to look like a synthesis process. Fortunately, average users do not have to write model transforms. This only needs to be done by domain-specific language designers. If a transform can implement the high-level semantics in different ways, it is up to the language designer to decide how and when to make these options visible to the domain expert. The theory of how to combine arbitrary models of computation to make heterarchical application models also remains an open research question.

In summary, Cairn's application abstraction is ideal for programmers who want to get something running quickly with minimal effort. These users are not interested in writing assembly-level code to extract every bit of speed out of their target architecture. Rather, this is a tool for users who want to work with high-level models and use these models for implementation and not just experimentation.

8.1.2 Focus on Simple Sub-RISC Architectures

The second core requirement is to consider simple architectures. Common and successful general-purpose RISC processors are simultaneously too simple and too complex to be good basic blocks for heterogeneous multiprocessors. The "too simple" argument stems from the fact that general-purpose processors lack support for application-specific concurrency. General-purpose processors are "too complex" because in trying to be applicable to any application, they inevitably include features that are not useful for certain applications. These arguments apply to all three

granularities of concurrency: process-level, data-level and datatype-level.

Customizable processors such as Tensilica's Xtensa are a step in the right direction. They allow architects to add application-specific function units to a core datapath to better support datatype-level concurrency. However, there is only a limited capability to modify the core datapath, for example to add register files or to change the control logic. Therefore it is not possible to fully support application-specific data-level and process-level concurrency.

Sub-RISC processors are a compelling basic block for programmable multiprocessors because they allow designers to break traditional architectural design patterns. Function units, register files, and pipelines can all be customized to match the application domain's requirements. Since these datapaths can be built from the ground up, designers only pay for complexity when it is necessary. This ensures a lightweight, low-cost architecture.

Cairn's architectural abstraction for multiprocessor Sub-RISC machines provides a mechanism to capture the capabilities of these novel architectures. The data-level and datatype-level concurrency of individual datapaths is captured by an automatic operation set extraction process. Modifications made during design space exploration are automatically identified and exported to programmers.

The process-level concurrency of a multiprocessor is captured by a signal-with-data abstraction. The data outputs of one PE become data and control inputs for other PEs through the exchange of signal-with-data messages. This allows architects to build hardware support for application-specific process-level concurrency by making novel compositions of PEs.

Pros and Cons of Cairn's Target Architectures and Architectural Abstraction

The intended users for the architectural side of the Cairn methodology are architects who are familiar with the microarchitectural trade-offs involved in making datapaths and who want to fine-tune their architectures. Cairn provides abstractions and tools ideally suited for these individuals. The Sub-RISC approach allows architects to build clever solutions with few transistors. The single-processor and multiprocessor architecture abstractions allow architects to focus on the exciting parts of the design process: Trying new ideas and quickly seeing the results. The tools hide the tedious parts of the design process: the per-instance overhead of figuring out instruction encodings, maintaining consistency between various architecture models, writing code generators, and writing low-level RTL models.

A potential downside of the Sub-RISC approach is that for a broad and complex application

domain, the dream of specialized PEs may succumb to the need for generalized, reusable architectures. Despite attempts to build small and clever datapaths, all of the PEs may end up looking like RISC processors in the end. If this is so, then architects would be better off assembling common, proven, and supported pre-designed processors.

Also, if a broad set of general-purpose features are the inevitable conclusion, this calls into question the value of Cairn's signal-with-data abstraction. This abstraction is designed for PEs that compute simple mathematical projections. If most PEs are capable of conditional execution and recursive computations, the signal-with-data abstraction may be unnecessarily detailed.

Fortunately, it is still possible to satisfy the second core requirement without accepting Sub-RISC architectures. If RISC cores are neither too simple nor too complex for the system, then they can be used instead. The signal-with-data abstraction can be replaced an abstraction that better matches the capabilities of the chosen cores. It is important to model the architecture separately from the application and the mapping, and to maintain an effective abstraction for exporting the multiprocessor architecture's concurrency to programmers.

8.1.3 Use a Disciplined Mapping Methodology

The third core requirement is to use a disciplined mapping methodology to map applications onto architectures. The major difficulty in crossing the concurrency implementation gap is comparing an application's requirements for concurrency with an architecture's concurrency capabilities, and then producing an implementation based on that comparison. In current methodologies, designers mentally organize both the application's and the architecture's concurrency and then proceed to write separate programs for the individual processing elements in the architecture. All of the design decisions on how the application's concurrency is partitioned across the architecture are made implicitly. Application concepts, architecture concepts, and implementation decisions are blended together in the form of low-level code.

If the result fails to meet expectations, it is never clear what part of the system is to blame. The original application may be faulty, or the architecture could be insufficiently powerful. The partitioning decisions made during the implementation process may have introduced inefficiencies. Whatever the problem is, the solution will involve rewriting large amounts of code. This slows down the implementation process and introduces errors, making effective design space exploration impossible.

Cairn meets this core requirement by making the mapping process a distinct step in the design

flow. Designers create explicit mapping models that describe how the application's computations are assigned to architectural resources. These models capture all of the design decisions that are made to match up the application with the architecture in a tangible, modifiable form. This allows designers to understand, characterize and solve the concurrency implementation gap.

Cairn's automatic code generation process converts mapping models into executable code for the processing elements in a multiprocessor. Designers do not have to rewrite their high-level application models in another language in order to produce an implementation. The code generation process is fully retargetable. If architects modify PE datapaths, there are no compilers to update. The same application computations can simply be rescheduled onto the modified PEs, and designers can observe the results of their modifications quickly.

Pros and Cons of Cairn's Mapping Methodology

Cairn's mapping methodology is the key enabler of the iterative Y-chart design flow. Mapping models are separate from the application model and the architecture model. This enforces a strong separation of concerns that makes it possible to use all of the design space exploration paths in the Y-chart. Designers can experiment with changes to the application, architecture, or mapping with only a minimum of effort. It is easy to map different applications to the same architecture, or the same application to different architectures. It is easy to explore different mapping strategies for the same application and architecture pair. Programmers can use their high-level application models directly for implementation without writing assembly code by hand.

One downside to Cairn's mapping methodology is that it is the responsibility of the designer to manually create mapping models. This can be complicated. Designers must partition their application across potentially hundreds of processing elements, and arrange for the signal-with-data messages sent between these PEs to be communicated over the on-chip network efficiently. Due to the interactions between different streams of messages over the same network links, it can be difficult to characterize the behavior of the entire application running on the architecture. It can also be difficult to understand how changing the mapping will affect the total system performance.

The goal of Cairn's "tall, skinny" approach is to formulate the mapping problem correctly rather than to find a universal automatic solution to the mapping problem. The formulation itself is successful. The mapping abstraction exposes exactly the details of the architecture that are necessary for application implementation. Designers see exactly what design decisions must be made in order to finish the implementation. This formulation preserves the separation of concerns described

previously. This is a significant improvement over ad hoc design methodologies. Future work will address the issue of helping designers visualize the performance trade-offs of different mapping decisions. Eventually, it should be possible to make many kinds of mapping decisions automatically and perhaps even optimally.

8.2 Final Remarks

The Cairn project contributes a multiple abstraction approach that meets the three core requirements. With multiple abstractions, designers can use the right tools for the different stages of the design process. There is no single abstraction that covers all facets of embedded system design. Application designers and architects have different concerns. An abstraction that has too broad of a scope will not benefit productivity. Strong abstractions simultaneously assist and restrict designers. Compositions of these abstractions give designers the flexibility they need to model heterogeneous systems.

The results demonstrated in this dissertation show that high-level abstractions are compatible with high performance. By properly modeling the concurrency in a system, designers can arrive at a lightweight, low-cost implementation. A formal application model reveals the application's exact requirements. A formal architecture model exports the capabilities of the target platform. A disciplined mapping methodology allows designers to formally compare these things, which leads to ideas for improvements to further increase performance and reduce costs.

Design discontinuities occur when engineers reach the limits of existing methodologies and begin to search for alternatives that can handle increasing complexity. In modern systems, concurrency is the cause of this complexity. If processors are to succeed as the "NAND gates of the future" designers must be able to solve the concurrency implementation gap. Cairn accomplishes this and enables greater productivity and superior results.

Bibliography

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *International Conference on Computer Architecture*, pages 248–259, 2000.

[2] Perry Alexander and Phillip Baraona. Extending VHDL to the systems level. In *Proceedings of the VHDL International Users' Forum*, pages 96–104, October 1997.

[3] Peter Ashenden, Philip Wilsey, and Dale Martin. SUAVE: Extending VHDL to improve data modeling support. In *IEEE Design and Test of Computers*, pages 34–44, April 1998.

[4] Zachary Baker and Viktor Prasanna. Time and area efficient pattern matching on FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 223–232, February 2004.

[5] Zachary K. Baker and Viktor K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 135–144, April 2004.

[6] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[7] F. Balarin, P. D. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishing, 1997.

[8] Felice Balarin, Jerry Burch, Luciano Lavagno, Yosinori Watanabe, Roberto Passerone, and Alberto Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *IEEE International High-Level Design Validation and Test Workshop*, pages 129–133, November 2001.

[9] Felice Balarin, Luciano Lavagno, Claudio Passerone, and Yosinori Watanabe. Processes, interfaces and platforms: Embedded software modeling in Metropolis. In *International Workshop on Embedded Software (EMSOFT)*, pages 407–421. Springer-Verlag LNCS 2491, October 2002.

[10] Armin Bender. MILP based task mapping for heterogeneous multiprocessor systems. In *Proceedings of the European Design Automation Conference*, pages 283–288, 1996.

[11] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[12] Jeffrey Bier, Edwin Goei, Wai Ho, Philip Lapsley, Maureen O'Reilly, Gilbert Sih, and Edward Lee. Gabriel: A design environment for DSP. *IEEE Micro*, 10(5):28–45, October 1990.

[13] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July-August 1999.

[14] Shekhar Borkar. Low power design challenges for the decade. In *Asia and South Pacific Design Automation Conference*, pages 293–296, January 2001.

[15] Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November 2005.

[16] Jerry Burch, Robert Passerone, and Alberto Sangiovanni-Vincentelli. Using multiple levels of abstractions in embedded software design. In *International Workshop on Embedded Software (EMSOFT)*, pages 324–343. Springer-Verlag LNCS 2211, October 2001.

[17] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 178–188, 1987.

[18] Adam Cataldo, Elaine Cheong, Huining Thomas Feng, Edward A. Lee, and Andrew Christopher Mihal. A formalism for higher-order composition languages that satisfies the Church-Rosser property. Technical Report UCB/EECS-2006-48, EECS Department, University of California, Berkeley, May 9 2006.

[19] Chandra Chakuri. *Approximation Algorithms for Scheduling Problems*. PhD thesis, Stanford University, 1998.

[20] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[21] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor PC router. In *Proceedings of the USENIX 2001 Annual Technical Conference*, pages 333–346, June 2001.

[22] Young H. Cho and William H. Mangione-Smith. Deep packet filter with dedicated logic and read only memories. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 125–134, April 2004.

[23] CoFluent Design. http://www.cofluentdesign.com.

[24] Elanix Corp. Elanix SystemView user's guide, 2001.

[25] Intel Corp. Intel Internet Exchange Architecture Software Development Kit. http://www.intel.com/design/network/products/npfamily/sdk.htm.

[26] Intel Corp. *Intel IXP1200 Network Processor Product Datasheet*, December 2001.

[27] Intel Corp. *Intel IXP2800 Network Processor Product Brief*, 2002.

[28] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: Application modeling for signal processing systems. In *Design Automation Conference*, pages 402–405, June 2000.

[29] Juan de Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In *ETAPS Fundamental Approaches to Software Engineering (FASE)*, 2002.

[30] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *ACM SIGCOMM*, pages 3–14, 1997.

[31] André DeHon, Joshua Adams, Michael DeLorimier, Nachiket Kapre, Yuki Matsuda, Helia Naeimi, Michael Vanier, and Michael Wrighton. Design patterns for reconfigurable computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2004.

[32] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel Bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[33] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972.

[34] Doron Drusinsky and David Harel. Using Statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(7):798–807, July 1989.

[35] Stephen Edwards. The challenges of hardware synthesis from C-like languages. In *Design, Automation and Test in Europe*, pages 66–67, 2005.

[36] J. Eker and J. Janneck. Embedded system components using the CAL actor language, October 2002. http://www.gigascale.org/caltrop.

[37] A. Fauth, J. van Praet, and M. Freericks. Describing instruction set processors using nML. In *European Design and Test Conference*, pages 503–507, 1995.

[38] R. Franklin, D. Carver, and B. L. Hutchings. Assisting network intrusion detection with reconfigurable hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–120, April 2002.

[39] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer, 2000.

[40] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[41] Sunondo Ghosh, Rami Melhem, and Daniel Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):272–284, March 1997.

[42] GNU Multiple Precision Arithmetic Library. http://www.swox.com/gmp.

[43] Maya Gokhale, Dave Dubois, Andy Dubois, Mike Boorman, Steve Poole, and Vic Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In *International Con-*

ference on Field Programmable Logic and Applications (FPL), pages 404–413, London, UK, 2002. Springer-Verlag.

[44] Matthias Gries and Kurt Keutzer. *Building ASIPs: The Mescal Methodology*. Springer, 2005.

[45] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, 2002.

[46] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *International Workshop on Information Security Applications*, page 188, August 2004.

[47] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Design Automation Conference*, pages 299–302, 1997.

[48] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Design, Automation and Test in Europe*, pages 485–490, March 1999.

[49] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[50] Tomas Henriksson, Ulf Nordqvist, and Dake Liu. Embedded protocol processor for fast and efficient packet reception. In *IEEE International Conference on Computer Design (ICCD)*, pages 414–419, 2002.

[51] Tomas Henriksson and Ingrid Verbauwhede. Fast IP address lookup engine for SoC integration. In *IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, April 2002.

[52] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[53] A. Hoffman, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[54] Impulse Accelerated Technologies. http://www.impulsec.com.

[55] AccelChip Inc. MATLAB as a system modeling language for hardware, 2005. http://www.accelchip.com.

[56] Celoxica Inc. Handel-C language reference manual, 2003. http://www.celoxica.com.

[57] Celoxica Inc. Software-compiled system design, 2004. http://www.celoxica.com.

[58] Coware Inc. SPW: Design automation technology for digital signal processing applications, 2005. http://www.coware.com.

[59] Synfora Inc. PICO Express datasheet, 2004. http://www.synfora.com.

[60] Synplicity Inc. True DSP synthesis for fast, efficient, high-performance FPGA implementations, January 2005. White paper. http://www.synplicity.com/products/synplifydsp.

[61] Tensilica Inc. Tensilica XPRES Compiler. http://www.tensilica.com/products/xpres.htm.

[62] Tensilica Inc. Tensilica Xtensa LX processor tops EEMBC networking 2.0 benchmarks, May 2005. http://www.tensilica.com/html/pr_2005_05_16.html.

[63] Tensilica Inc. Xtensa architecture and performance, 2005. White paper. http://www.tensilica.com.

[64] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, March 2005. http://www.xilinx.com.

[65] Axel Jantsch and Ingo Sander. Models of computation in the design process. In Bashir M. Al-Hashimi, editor, *SoC: Next Generation Electronics*. IEE, 2005.

[66] Axel Jantsch and Hannu Tenhunen. *Networks on Chip*. Springer, 2004.

[67] Yujia Jin, Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. An automated exploration framework for FPGA-based soft multiprocessor systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, September 2005.

[68] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*, pages 471–475, 1974.

[69] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[70] Bart Kienhuis, Ed Deprettere, Pieter van der Wolf, and Kees Vissers. A methodology to design programmable embedded systems. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling and Simulation (SAMOS)*, pages 18–37. Springer-Verlag LNCS 2268, 2002.

[71] Eddie Kohler, Robert Morris, and Benjie Chen. Programming language optimizations for modular router configurations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 251–263, October 2002.

[72] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[73] Chidamber Kulkarni, Gordon Brebner, and Graham Schelle. Mapping a domain specific language to a platform FPGA. In *Design Automation Conference*, pages 924–927, June 2004.

[74] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *IEEE International Workshop on Intelligent Signal Processing*, May 2001.

[75] Edward Lee and Stephen Neuendorffer. Concurrent models of computation for embedded software. *IEE Proceedings on Computers and Digital Techniques*, 152(2):239–250, March 2005.

[76] Edward A. Lee. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, University of California, Berkeley, July 2003.

[77] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[78] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1):75–108, 1998.

[79] Stan Liao, Steve Tjiang, and Rajesh Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Design Automation Conference*, pages 70–75, June 1997.

[80] Zhonghai Lu, Ingo Sander, and Axel Jantsch. A case study of hardware and software synthesis in ForSyDe. In *International Symposium on System Synthesis (ISSS)*, pages 86–91, October 2002.

[81] Craig Matsumoto. Net processors face programming trade-offs. *EE Times*, 30 August 2002. http://www.eetimes.com/story/OEG20020830S0061.

[82] S. McCloud. Catapult C synthesis-based design flow: Speeding implementation and increasing flexibility, October 2003. Mentor Graphics Corp. whitepaper.

[83] D. Meng, R. Gunturi, and M. Castelino. IXP2800 Intel network processor IP forwarding benchmark full disclosure report for OC192-POS. In *Intel Corp. Tech Report for the Network Processing Forum (NPF)*, October 2003.

[84] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The Olympus synthesis system. *IEEE Design and Test of Computers*, 7(5):37–53, October 1990.

[85] Giovanni De Micheli and David Ku. HERCULES - a system for high-level synthesis. In *Design Automation Conference*, pages 483–488, June 1988.

[86] Message Passing Interface Forum. http://www.mpi-forum.org.

[87] Trevor Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, April 2001.

[88] Henk Muller. *Simulating Computer Architectures*. PhD thesis, University of Amsterdam, 1993.

[89] Sanjiv Narayan, Frank Vahid, and Daniel D. Gajski. System specification and synthesis with the SpecCharts language. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 266–269, November 1991.

[90] Stephen Neuendorffer. *Actor-Oriented Metaprogramming*. PhD thesis, University of California, Berkeley, December 2004.

[91] JoAnn Paul. Programmers' views of SoCs. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 156–161, 2003.

[92] Pierre Paulin, Clifford Liem, Trevor May, and Shailesh Sutarwala. DSP design tool requirements for embedded systems: A telecommunications industrial perspective. *Journal of VLSI Signal Processing*, 9(1–2):23–47, January 1995.

[93] Pierre Paulin, Chuck Pilkington, and Essaid Bensoudane. StepNP: A system-level exploration platform for network processors. *IEEE Design and Test of Computers*, 19(6):17–26, 2002.

[94] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org.

[95] Dac Pham, Tony Aipperspach, David Boerstler, Mark Bolliger, Rajat Chaudhry, Dennis Cox, Paul Harvey, Peter Hofstee, Charles Johns, Jim Kahle, Atsushi Kameyama, John Keaty, Yoshio Masubuchi, Mydung Pham, Jürgen Pille, Stephen Posluszny, Mack Riley, Daniel Stasiak, Masakazu Suzuoki, Osamu Takahashi, James Warnock, Stephen Weitzer, Dieter Wendel, and Kazuaki Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation Cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, January 2006.

[96] Andy Pimentel, Louis Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed Deprettere. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, November 2001.

[97] William Plishker. *Automated Mapping of Domain Specific Languages to Application Specific Multiprocessors*. PhD thesis, University of California, Berkeley, 2006.

[98] Willian Plishker, Kaushik Ravindran, Niraj Shah, and Kurt Keutzer. Automated task allocation for network processors. In *Network System Design Conference*, pages 235–245, October 2004.

[99] Wolfram Putzke-Röming, Martin Radetzki, and Wolfgang Nebel. Modeling communication with Objective VHDL. In *Proceedings of the VHDL International Users' Forum*, pages 83–89, March 1998.

[100] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Heterogeneous parallel programming in Jade. In *ACM/IEEE Conference on Supercomputing*, pages 245–256, 1992.

[101] John Salmon, Christopher Stein, and Thomas Sterling. Scaling of Beowulf-class distributed systems. In *ACM/IEEE Conference on Supercomputing*, pages 51–64, November 1998.

[102] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, January 2004.

[103] John Sanguinetti and David Pursley. High-level modeling and hardware implementation with general-purpose languages and high-level synthesis. In *IEEE/DATC Electronic Design Processes Workshop*, April 2002.

[104] Christian Sauer, Matthias Gries, and Sören Sonntag. Modular domain-specific implementation and exploration framework for embedded software platforms. In *Design Automation Conference*, pages 254–259, 2005.

[105] Niraj Shah. Understanding network processors. Master's thesis, University of California, Berkeley, 2001.

[106] Niraj Shah. *Programming Models for Application-Specific Instruction Processors*. PhD thesis, University of California, Berkeley, 2004.

[107] Niraj Shah, William Plishker, and Kurt Keutzer. NP-Click: A programming model for the Intel IXP1200. In *Workshop on Network Processors at the International Symposium on High Performance Computer Architecture*, February 2003.

[108] Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–238, April 2001.

[109] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[110] Snort - The De Facto Standard for Intrusion Detection/Prevention. http://snort.org.

[111] Donald Soderman and Yuri Panchul. Implementing C designs in hardware: A full-featured ANSI C to RTL Verilog compiler in action. In *Proceedings of the VHDL International Users' Forum*, pages 22–29, March 1998.

[112] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 258–267, April 2004.

[113] Ioannis Sourdis, Dionisios Pnevmatikatos, and Kyriakos Vlachos. An efficient and low-cost input/output subsystem for network processors. In *Workshop on Application Specific Processors (WASP-1)*, 2002.

[114] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley Systems Programming Series, 1984.

[115] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using Kahn process networks: The Compaan/Laura approach. In *Design, Automation and Test in Europe*, volume 1, pages 340–345, February 2004.

[116] S. Swamy, A. Molin, and B. Covnot. OO-VHDL: Object-oriented extensions to VHDL. *IEEE Computer*, 28(10):18–26, October 1995.

[117] David Taylor, John Lockwood, Todd Sproull, Jonathan Turner, and David Parlour. Scalable IP lookup for Internet routers. *IEEE Journal on Selected Areas in Communications (JSAC)*, 21:522–534, May 2003.

[118] Willian Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196, April 2002.

[119] Scott Weber. *TIPI: Tiny Instruction Processors and Interconnect*. PhD thesis, University of California, Berkeley, 2005.

[120] Scott Weber and Kurt Keutzer. Using minimal minterms to represent programmability. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, pages 63–68, September 2005.

[121] Scott Weber, Matthew Moskewicz, Matthias Gries, Christian Sauer, and Kurt Keutzer. Fast cycle-accurate simulation and instruction set generation for constraint-based descriptions of programmable architectures. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, pages 18–23, September 2004.

[122] Tom Wilson, Gary Grewal, Ben Halley, and Dilip Banerji. An integrated approach to retargetable code generation. In *International Symposium on High-Level Synthesis*, pages 70–75, May 1994.

[123] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, 1998.

[124] Claudiu Zissulescu, Bart Kienhuis, and Ed Deprettere. Communication synthesis in a multiprocessor environment. In *International Conference on Field Programmable Logic and Applications (FPL)*, August 2005.