Automated Mapping of Domain Specific Languages to Application Specific Multiprocessors

William Lester Plishker

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-123
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-123.html

October 5, 2006

Automated Mapping of Domain Specific Languages to Application Specific Multiprocessors

by

William Lester Plishker

B.S. (Georgia Institute of Technology) 2000

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 Rastislav Bodik
Professor Robert Cole

Fall 2006

Automated Mapping of Domain Specific Languages to Application Specific Multiprocessors

William Lester Plishker

Abstract

Automated Mapping of Domain Specific Languages to Application Specific Multiprocessors

by

William Lester Plishker
Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kurt Keutzer, Chair

Application specific multiprocessors are capable of high performance implementations while remaining flexible enough to support a range of applications. Architects of these systems achieve high performance through domain specific optimizations such as multiple processing elements, dedicated logic, and specialized memory and interconnection. However, these features are often introduced at the expense of programming productivity. For application-specific programmable systems to succeed, it is necessary to deliver high performance implementations quickly. Three of the most important and time-consuming steps to arriving at implementations on these platforms are (1) to extract parallelism from applications descriptions, (2) to arrive at a model of the architecture, and (3) to map the parallelized application to the architectural model. We examine this problem for the networking domain and target a commercial family of network processors: Intel's IXP series. We propose and demonstrate a solution that starts with a domain specific language, which allows the extraction of parallelism without designer intervention. We show that high level application information enables optimizations and automated transformations to a task graph. A task graph is a model of an application that exposes its computation, data, and communication. The elements of the task graph can be mapped to processing elements, memory, and interconnect of the target architecture. We formulate

the mapping problem as an integer linear programming (ILP) problem. We demonstrate that this method finds efficient solutions with fast run times on the data plane of network applications of different scales and complexity including IP forwarding, differentiated services, network address translation, and web switching. Using this design flow designers enjoy design times up to 3.5 times shorter than other new productive approaches. The resulting implementations are within 17% of hand mapped approaches. While the demonstration vehicle for this work has been network processing, we believe it will have wider applicability to other application domains that are starting to employ more single chip multiprocessing.

Professor Kurt Keutzer
Dissertation Committee Chair

To my parents,

whose support, wisdom, and love have made this all possible.

Contents

List of Figures

List of Tables

Acknowledgments

I want to thank each member of the Mescal team for the help contributed to this work. I was fortunate enough to work with Scott Weber and Andrew Mihal for my first project with the group. They got me started on the right foot. I next want to thank Yujia Jin, N.R. Satish, Jike Chong, and Matt Moskewicz for the friendship and sound technical advice that they have given me over the years. Bryan Catanzaro and Martin Trautmann have shown great patience by working with my research tools and improved my project through their work. They all have been great sounding boards for ideas and have lent much needed eyes for proofreading this work. Dave Chinnery has been a great source of information a variety of topics and was always happy to talk with me. Most especially I want to thank Niraj Shah, whose research paved the way for this work. His mentorship has shaped much of this work both technically and non-technically and for that I thank him greatly.

My friends and family have been great during this time in my life. Their support and understanding have allowed me to get the most out of this experience. Starting graduate school with Doug Densmore, Donald Chai, Kaushik Ravindran, Adam Zoss, and Roger O'Brient has made classes, homeworks, projects, and working late that much easier. All of the members of the Mayfield Fellows Program have made my time here at Berkeley so interesting. Brenda Haendler has been amazing. I appreciate all of her support, proofreading, and lending of a sympathetic ear. My parents have supported me through this entire experience. I have always felt I could fly as close to the sun as I want, and they would be there to catch me if my wings melted.

My dissertation committee members have provided invaluable feedback. They have donated their time to hear me talk, to read this dissertation, and to discuss my research. Other people at Berkeley have made working here such an enjoyable experience including Ruth Gjerde, Jennifer Stone, and Dan MacLeod. A number of professors have taken the time to discuss my research including Wen-mei Hwu, David Patterson, and Drew Isaacs. For this work, for my graduate career, and for many internships, I owe a great deal of gratitude to my adviser Kurt Keutzer. His guidance to me ranging from research, career, and life has been indispensable. I will miss our

lunchtime conversations and taking his money in poker.

Chapter 1

Introduction

Application specific multiprocessors (ASMPs) represent an enormous potential boon to electronic system designers. As applications become more complex and more computationally intensive, ASMPs are capable of providing high-performance implementations through domain specific architectural optimizations, while retaining flexibility through programmability. However the difficulty of programming these devices has greatly inhibited their adoption. Long design times and rigid implementations of programs have eliminated many of the benefits of these architectures. The key to unlocking the potential of these architectures is a fast, robust tool flow that can take a natural application description, automate difficult design decisions, and quickly produce an efficient implementation. The goal of this dissertation is to present a framework which satisfies each of the objectives for a representative application area using a popular application description language mapped onto commercial ASMPs.

1.1 Emerging Trends

The growing proliferation of applications on embedded devices continues to change how the world works. This explosion of platforms increases what is possible in many application domains. In networking routers and edge devices are securing and accelerating our networks using intrusion detection, denial of service attacks defense, and packet content based routing. In media new video codecs enable full motion playback on handheld devices like cell phones. In the automotive industry drive-by-wire technologies are creating more robust and responsive car subsystems. In medicine segmentation and three dimensional rendering allow doctors to more quickly and accurately diagnose patients.

Electronic system designers have facilitated this explosion by creating the products that implement these applications. Among their many design decisions, they must set the platform on which each of the product's features will be implemented. Electronic system designers have a variety of options for this including application specific integrated circuits, general purpose processors, field programmable gate arrays, and application specific standard parts, allowing them to trade-off efficiency, implementation time, *non-recurring engineering* (NRE) cost, and per part cost. The technology summary and market suitability of each of these options is shown pictorially in Figure 1.1.

Of all the solutions available to an electronic system designer, those based on *application specific integrated circuits* (ASICs) achieve the highest performance. An

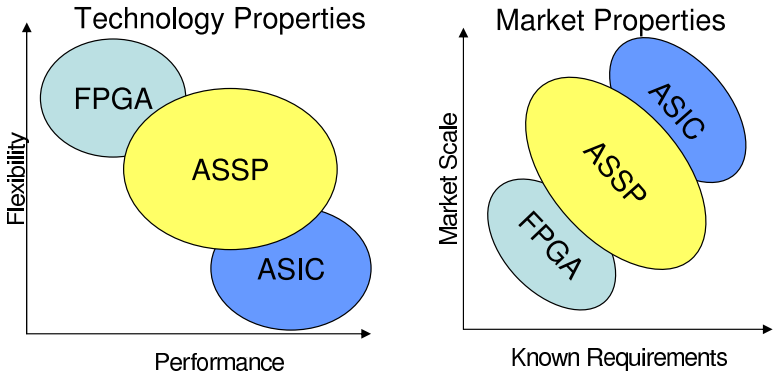

Figure 1.1: Market and technology summary of solutions available to system designers *Source: Allan Armstrong from Ryan Hankin Kent (RHK), 2004*

ASIC is a custom designed silicon chip specifically created for a single application. The design flow starts with a *hardware description language* (HDL). Designers explicitly define how concurrently executing computational blocks connect and relate temporally to each other. To arrive at an implementation, automated tools perform extensive optimizations to improve performance, power, cost, and yield. The result is an implementation that for any given process generation, is the fastest, smallest, lowest power solution. However, this result comes at a long design time with an increasingly high NRE, which is up to $25 million for 90nm ASIC [6]. These design costs are due largely to the increasing cost of development, verification, and masks. While ASICs offer an attractive solution to designers, their NRE costs make them suitable for only the highest volume markets. Further, because ASICs are hardware solutions, they are inflexible to changes in the applications they implement. Any flux in the application's behavior will cause a new long design iteration including re-verification and the purchase of new masks. Consequently, ASICs are increasingly used for only

high-volume applications.

Field programmable gate arrays (FPGAs) are another general purpose solution for electronic system designers. With programmable lookup tables for boolean functions and configurable interconnections, programmers can create arbitrary combinational and sequential logic out of this fabric. Like ASICs, the typical design flow for FPGAs starts with a hardware description language, again used to define computational blocks and their interconnection. This description facilitates utilization of the inherently parallel blocks of the fabric. Automated tools perform the typical steps of hardware design, but specialized for targeting fabric instead of raw silicon. Since FPGAs are a general purpose off-the-shelf technology, designers can implement their target application quickly compared to traditional hardware flows. Their field programmability is also an asset to nascent applications that may need to change after they are deployed. This fast time-to-market and flexibility comes at a cost. FPGA's underlying technology puts them at a severe performance disadvantage to application specific solutions. Compared to ASICs they require 40 times the area, 10 times the power, and are 3 times slower than ASICs [42]. Furthermore, while they are an off-the-shelf solution, their per part costs are generally the highest of the options, making them suitable for lower volume markets, where NRE costs are not easily overcome.

General purpose processors (GPPs) are another solution not specifically tuned to an application domain. They provide the easiest path to a low performance implementation of an application. GPPs for embedded applications like ARM, Intel's

Pentium M, IBM's PowerPC are tuned for general purpose computation and each has robust tool flows and support for multiple operating systems. These devices typically have large and diverse communities of users ensuring that tools used to arrive at implementation will be robust for a variety of applications. Regardless of the application's original description language (provided the language has functional support to execute it), minimal effort is required by the designer to arrive at a functional implementation. Electronic system designers enjoy the benefits of a software solution, flexibility, fast design times, and low NRE cost, but these come at a heavy price of low performance, poor power efficiency, and potentially high per post cost. As a result, designers use this solution only for computationally light or control dominated tasks. With more design effort, designers can significantly improve performance (be it clock frequency or power) and lower per part cost.

Application Specific Standard Parts (ASSPs) are off-the-shelf chips that are tailored to a set of applications. ASSPs are capable of achieving higher performance by incorporating domain specific architectural features such as special purpose hardware, multiple multithreaded processing elements, distributed heterogeneous memories, and special purpose peripherals. ASSPs have different degrees of flexibility, ranging from simple parameterization to being completely programmable capable of running arbitrary programs. For example, NEC's D/A Converter for Audio System [55] is a parameterizable ASSP. Designers can configure the device in a few ways including its system clock and the data format of its serial input. This flexibility makes

this ASSP suitable for multimedia terminals, MPEG audio equipment, video CDs, game machines, and electronic musical instruments. By contrast a network processor, like Intel's IXP2800 [35], has fully programmable processing elements which have an instruction set and a memory architecture tailored for packet processing. These processing elements can be programmed for any desired feature, giving the programmer of the device complete flexibility. Because of their high performance, low NRE costs, and relatively low per part costs, ASSPs have been making gains in the market place. Figure 1.2 displays the ASSP trend. ASSP have yet to approach their full market potential. Impeding adoption and reducing their utility to designers is the fact that completely programmable ASSPs are difficult to program. The reasons for this is discussed in depth in Section 1.2.1, but the implication is that while these devices are capable of implementing high performance, complex applications, they are not currently used for it. While the market for ASSPs is large and growing, its full potential will only be tapped when electronic system designers are able to use them for complex, performance intensive applications, even in small, niche markets. ASSPs will then fuel the continued explosion of new applications across many areas.

1.2 Application Specific Multiprocessors

Application specific standard parts (ASSPs) present electronic system designers with the potential to service a set of applications with high performance implementations, fast design times, and low per part costs. Single-chip multiprocessing is becom-

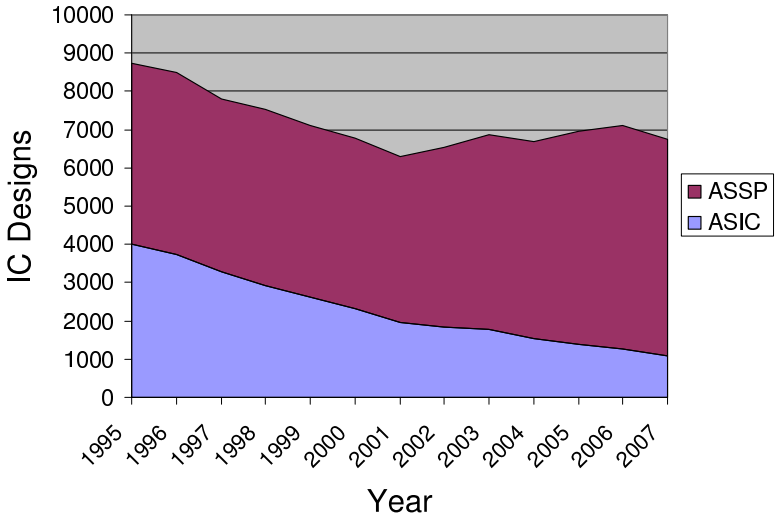

Figure 1.2: ASIC versus ASSP design starts *Source: Handel Jones, IBS 9/23/2002*

ing increasingly popular in mainstream computing to conquer power problems, but it has existed for several years in certain application domains to exploit the highly parallel nature of a set of applications. Single chip multiprocessing has been successfully deployed in embedded computing markets including graphics, digital signal processing, and networking.

Narrowly focused ASSPs such as a D/A converter do not employ fully programmable processing elements and consequently have limited flexibility. An ASSP based on multiple processing elements, such as Intel's IXP2xxx series, has flexibility limited only to the programs which can compile to it. In practice for the IXP2xxx, the set of applications which can be efficiently supported by the architecture includes a variety of network applications [49] [14] [53] along with other applications that happen to map well to the architecture [70]. High performance is achieved on these kinds

of platforms through multiple processing elements, distributed memory, specialized hardware, and a customized interconnect topology. We call a platform which utilizes these features to deliver implementations an *application specific multiprocessor* (ASMP). Once an electronic system designer has selected an ASMP to implement an application, programmers must be able to produce high quality software quickly in order to tap into the potential of the device.

1.2.1 Programming Application Specific Multiprocessors

The ability to buy these high performance solutions off-the-shelf makes ASMPs an attractive option to electronic system designers. Once chosen, the task of harnessing the power of the platform and implementing the features needed falls to a programmer. Programmers would like to utilize these devices by:

- Producing *efficient* implementations that exploit the power of the architecture
- Programming *productively* such that they may arrive at implementations quickly
- Having *portable* application descriptions that can target multiple devices
- Creating a *verifiably* correct implementation

While all of these are desired properties of a programming environment, the time it takes to arrive at an implementation and its resulting efficiency are the two critical features which determine the success or failure of it. The programming environments for a given ASMP will largely determine its success in the marketplace.

Domain Specific Languages

Domain Specific Languages (DSLs) provide programmers with a natural way of describing applications. Developed by domain experts, DSLs match their mental model of an application domain. DSLs enable higher productivity by providing component libraries, communication and computation semantics, visualization tools, and test suites tailored to a given application domain. In digital signal processing, MATLAB [46] uses simple syntax for vectors and an extensive library of digital signal processing kernels to enable natural design capture. Simulink [47] is an actor oriented approach for capturing dataflow style or control applications. The built-in simulator and visualization tools create an environment where application designers can describe and iterate on an application design quickly and accurately. LabVIEW [54] is another modular, dataflow-like language specialized for testing instruments. For networking, a popular DSL is Click [39], which is an actor oriented language customized for packet processing and packet flow visualization.

Programming Environments

While domain specific languages are commonly employed for application exploration, the programming environments for ASMPs tend to force programmers to code at a much lower level. On the whole, programming environments for these devices have opted for exposing much of the architecture to the programmer. The ability to arrive at high performance solutions is given the highest priority and the programmer

manages all of the features of the architecture. Most of these devices initially ship with only assemblers, giving programmers the ability to tighten their kernels at the instruction level.

The perceived ability to arrive at efficient implementations comes at a heavy price. To enable the use of special purpose hardware and exposed memories, vendors of these ASMPs often extend popular sequential models. Programmers end up overwhelmed with details of these idiosyncratic devices. Utilizing memories, balancing computation, and coordinating communication are daunting tasks when handled in low level programming environments that are built on Assembly or C. Even mundane tasks such as communicating with special purpose hardware can be a serious time sink. Saddled with the task of negotiating all of the architectural detail and programming at a low level, programmers require long design cycles to arrive at reasonable implementations.

Implementation gap

The task of describing the functionality of an application and the process of implementing that application is separated by a wide chasm that we call the *implementation gap* as shown graphically in Figure 1.3. Regardless of the application domain and the target ASMP, many obstacles are necessary to overcome to arrive at efficient implementation.

Besides the mundane but real annoyances of application design for these devices,

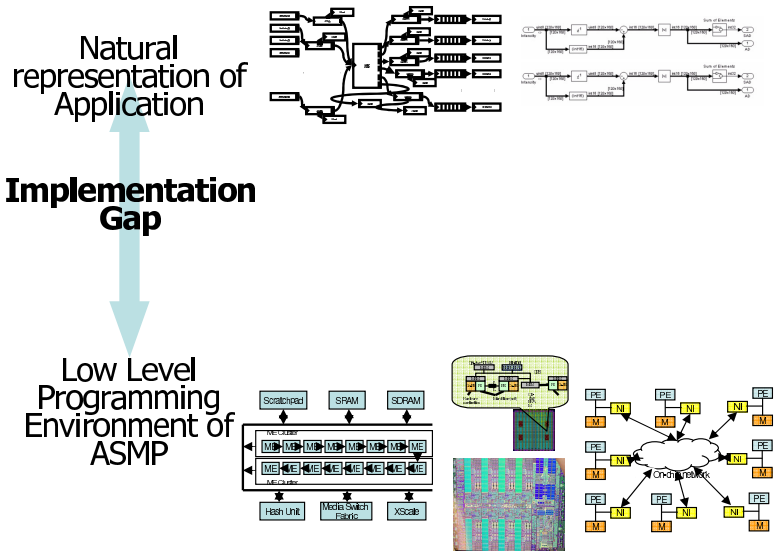

Figure 1.3: Implementation Gap

there exist three fundamental problems to effectively traversing the implementation gap shown pictorially in Figure 1.4.

- Creating an architectural model capturing the performance salient features.
- Constructing an application representation that exposes enough information to be efficiently implemented.
- Mapping this representation of the application to the architecture.

The paradigm of manually writing programs for each of thread of each processing elements creates many difficult design decisions for programmers. Programmers must balance the computation across processing elements to ensure that there is no single computational bottleneck on any processing element. If the architecture has

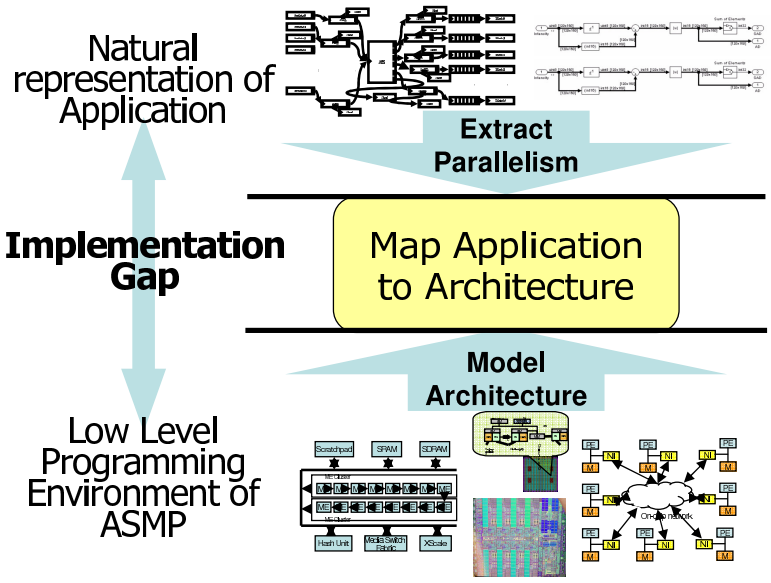

Figure 1.4: Keys to crossing the Implementation Gap

exposed distributed memories, programmers must take care to locate program data in appropriate memories. Small frequently accessed data should generally be located in memory local to a processing element, while larger infrequently accessed data should be located in off-chip memory. Multiprocessing complicates these rules of thumb by forcing the designer to account for how data is connected to programs on different processing elements. For example, a piece of data simultaneously shared between two programs on different processing elements cannot be mapped to the local memory of one processing element if the memory is not accessible to the other processing element. More generally, set of architectural restrictions describing which processing elements may directly communicate to which memories and other processing elements is referred to as an architecture's *topology*. Topology must be respected by the de-

signer to ensure that an application's communication patterns may be mapped to the interconnect resources in the architecture. Because of the architectural details, the problem space for the mapping problem is large and irregular. Since evaluating a design point requires a significant amount of time due to writing, debugging, and testing, a model of the architecture is needed for fast design space exploration. The model must be constructed with care capturing those features that have the greatest impact on performance while hiding those which needlessly complicate the model and increase the evaluation time.

This model is used by programmers or tools to make mapping decisions about how the application is to be distributed onto the architecture. Exacerbating the problem is the fact that each of these design decisions (allocating computational tasks to processing elements, laying out data in memory, and assigning application communication to architectural interconnect) are interrelated. A programmer can be easily overwhelmed by this large, irregular design space. As the number of processing elements and distributed memories increases and the topology gets more complicated with every generation, the problem becomes increasingly difficult.

To make these mapping decisions, the designer must have a description of the application that these difficult design decisions can be applied to. Often the application description is devoid of the information needed to do this. For example, a whiteboard or prose description of an application may not have state information for the computational tasks, so a feasible solution could not be made that included memories. A

model of the application must be constructed that exposes the critical features of the application.

While architects have been spending much effort maximizing the performance of their devices, the real key to ensuring the success of an ASMP is enabling the efficient and productive traversal of the implementation gap shown in Figure 1.3. Providing programmers the ability to harvest most of the performance of the architecture is more important than its raw performance.

1.2.2 Existing Common Design Flow

While the key to unlocking the potential of these devices lies in the ability to cross the implementation gap effectively, the design flow that commonly exists in industry is woefully flawed. While it varies from industry to industry and company to company, the typical design flow for one of these application descriptions is shown in Figure 1.5.

First an application team makes the application level choices about what features to implement, the basic algorithms for the kernels needed, and based on their estimates, the target platform to be used. Describing the application can be as informal as a whiteboard sketch or English prose, but it often takes the form of a functional description. This is commonly done in a popular sequential programming language, but for larger application domains, domain specific languages are growing in popularity.

While capturing the functionality of an application can be done quickly, the job of

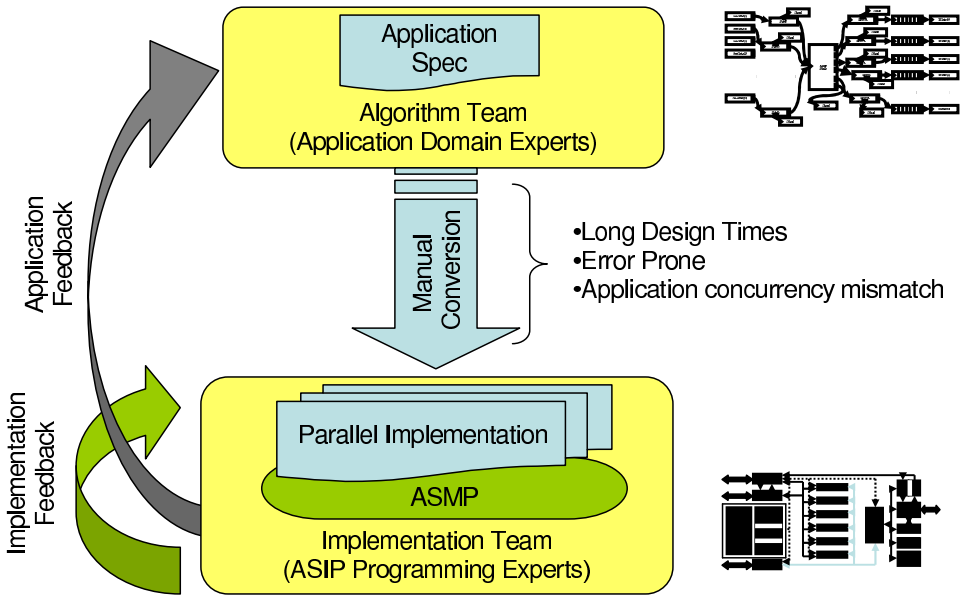

Figure 1.5: Existing design flow to cross implementation gap

implementing it on ASMPs is notoriously difficult. Driven by performance pressures, system designers perceive that to reap the benefits of using an ASMP, they often start implementation entry at the lowest level. Typically this task is done by one or more individuals who are experts of the target platform, called the *implementation team*. The implementation team is by and large required to overcome each of the fundamental obstacles of crossing the implementation gap. Their model of the architecture is usually implicit and based on their own experiences. Mapping decisions are made manually based on intuition. Arriving at an implementable form of the application is done by writing code in the target environment. Programmers must write multiple, often multithreaded, sequential assembler or C programs according to this mapping.

Humans manage to persevere despite the difficulty presented by these environ-

ments and settle on a mapping of the application. Programmers eventually create a functional design, but arriving at a working implementation is not enough, as the *raison d'être* of these devices is to deliver high performance solutions. Simplicities of the model and biases of the programmers invariably leads to a solution with an unforeseen bottleneck. Programmers must utilize information at this fully evaluated design point to refine the solution by rewriting kernels, redistributing the computational load, or relocating data in memories. These changes to the application require additional testing and design time, but fortunately they are feedback loops contained within the implementation team. The dependencies and feedback loops of this design process is given more rigorous consideration using a Design Structure Matrix [64] in Appendix B. Figure B.2 indicates that teams are well grouped for this design flow, but that significant intergroup dependencies and feedback exist.

The advantage of using an ASMP is its ability to provide high performance with short design time. The software path to implementation should afford short design times with fast changes. However, a flawed design flow to cross the implementation gap mitigates each of these. Unfortunately, this current design methodology leads to long design times and to being locked into a particular system mapping.

1.3 Objectives of Research

The work presented here seeks to provide a framework with supporting algorithms and tools to make the job of crossing the implementation gap more productive, while

still producing implementations comparable to hand crafted designs. Specifically, it focuses on the fundamental problems of crossing the implementation gap between an application described in a domain specific languages to ASMPs: architectural model creation, application representation construction, and application to architecture mapping this representation of the application to the architecture.

To study this throughly and practically, this work focuses on one particular application domain. While there are many potential areas to choose from to study this domain, networking is well suited to the task for reasons that are discussed in the following chapter. We select a representative domain specific language and common, high performance commercial network processors, and then we construct a framework. It is with these two endpoints the fundamental problems of crossing the implementation gap have been examined.

1.4 Organization of this Dissertation

This dissertation is organized in the following manner. Chapter 2 gives an overview of the field of networking, including the representative applications that will test this approach. Chapter 3 describes our overall approach to crossing the implementation gap along with some related work. To efficiently target these devices, programmers must first capture what are the salient features of the architecture, while abstracting away those details that are not important. Chapter 4 presents these features for network processors, based on my own experience with common commercial architec-

tures. We present a mathematical model of these features so that it is amenable to automated design space exploration. There are optimizations and transformations that are enabled by starting with a more abstract, domain restricted representation of the application. Chapter 5 discusses techniques that we have arrived at that produce the task graph automatically and provide significant performance improvements. To demonstrate this approach, Chapter 6 presents results of this approach by mapping the data plane of representative network applications onto high performance commercial network processors. Chapter 7 summarizes and concludes this work, while presenting some possible future directions.

Chapter 2

Network Processing

For an in depth examination of the problem of an application specific implementation gap, we focus on the field of networking. The networking domain continues to create more functionality in the network through new applications running within the network. Performance pressure coupled with changing functionality has prompted the rise of network processors. Indicative of the wider application specific multiprocessor trend, network processors achieve high performance through architectures customized to exploit parallelism inherent in the networking domain. While many programming approaches have appeared for these powerful devices, their focus on performance has lead to difficult design cycles. Domain specific languages in networking provide an ideal alternative to describing applications using packet abstractions and the common communication style of networking applications. The implementation gap between networking domain specific languages and existing network processor programming

environments is wide.

2.1 Motivation for Networking Focus

The field of networking has been a plentiful area of research and innovation. Years of study have produced a variety of applications across different parts of the network and different layers of the protocol stack. Besides being interesting from a research standpoint, networking is a huge market. According to the the Telecommunication Industry Association, telecommunication industry spending reached $856.9 billion in 2005 and is projected to keep growing at around 9% annual growth rate [26]. Growth was lead by network equipment and new applications on the network such as voice over IP, web conferencing, and video conferencing. These trends are indicative of a field which has maturity and size, but is still growing and evolving based on new applications.

These new applications drive not only new features in the network, but continue to drive the Internet's growth in bandwidth. In fact, data traffic on the Internet is doubling every 12 months [20], outpacing Moore's Law. The desire for new features along with performance pressures driven by increasing traffic demands precipitated the rise of high performance programmable solutions: network processors. These application specific multiprocessors are tailored to packet processing and provide the potential of high performance software solutions to networking's most demanding problems.

The networking area is large but still evolving. The interest in new applications has fueled the creation of domain specific languages. The sheer size of the domain supports a large community of users with a common set of rules or design philosophies. The size and continuing interest in new applications in networking have made it a fertile area for domain specific languages. The independent creation of domain specific languages and application specific multiprocessors makes networking an ideal niche for exploring the key problems of crossing the implementation gap.

The extensive study of how to utilize parallelism on single chip multiprocessors makes networking of particular interest to mainstream computing. The diversity of architectures, application, and programming methodologies can provide useful guidance to the general purpose community. Existing solutions to general purpose parallel programming will suffice for a small number of processors, thanks to the parallelism of independent jobs. But as the general purpose cores increase and multiprocessor parallelism must be exploited directly by applications, the key problems of the implementation gap exhibited in networking may manifest in the general purpose processing. A careful examination of the challenges and potential solutions in networking could hold broad insights for the mainstream multiprocessor world of the future.

2.2 Network Applications

Network applications are primarily defined by their location in the network. The *edge* of a network is the boundary between one managed network and another. An

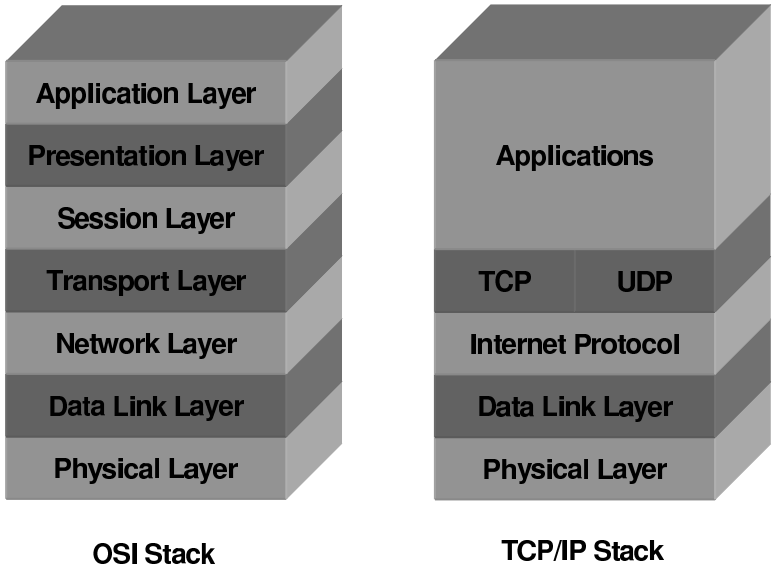

Figure 2.1: OSI and TCP stacks

edge network device may serve as the gateway for a handful of hosts to the Internet or a high bandwidth device that connects two Internet Service Providers. Within the boundaries of a managed network is the *core*. Core networking devices provide the services of the network. In the Internet core routers provide high bandwidth routing with best effort. A corporate network that merges data and voice traffic has core devices which provide quality of service guarantees along with basic routing functionality.

Besides location in the network, applications can also be classified by where they exist in the Open Systems Interconnection (OSI) model [11]. Layers of the OSI stack represent various levels of abstraction of communication from bits to packets to flows to application data shown in Figure 2.1. Much of the innovation on network de-

vices occurs at the Network and Transport layer. The Network layer covers routing packets, segmentation/reassembly, and sending error packets. The Transport layer ensures packets are reliably delivered in-order and manages flow control and congestion. In practice these correspond to the Internet Protocol (IP) for the Network layer and Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) for the Transport layer. The separation of layers allows designers to design network applications with a particular abstraction of packets and packet flow.

Thanks to its long history, many traditional network benchmarks have defined the area to measure the performance of the devices. However a longstanding uniform test-bench has been elusive because the demands on the application change rapidly from generation to generation. For example, *Internet protocol* (IP) packet routing has been a basic application for decades, but each generation of equipment has different port bandwidths, physical protocols, routing table sizes, and number of ports.

Along with the canonical networking benchmarks, new applications have appeared in different parts of the network. These have added new features to networks to service applications that need more than the best effort routing of most networks. Another driving force for new applications in the network is the desire to distributed more of the computational workload to the network. This steady trend of exporting more functionality out of the end hosts onto the network includes intrusion detection, quality of service guarantees, protocol translation, and routing based on packet content. The following section gives examples of common networking benchmarks along with

new ones that are driving the networks of the future.

2.2.1 IP Forwarding

Internet Protocol version 4 (IPv4) packet forwarding [9] is a common kernel of many network applications. The Network layer application forms the foundation of packet processing in the core of networks including the Internet. It provides the functionality needed to provide end-to-end delivery of packets. IP forwarding enables global connectivity with lossy, best effort routing of packets. There are no guarantees on bandwidth, latency, or in-order delivery of the packets. This limited feature requirements allows designers to deliver the highest performing implementations so that core routers of the Internet are able to achieve the highest possible throughput.

For the base case of IP forwarding, a packet is first received by the router. Being a Network layer application, the computation is confined to the IP header of the packet. Many implementations separate packet headers from the packet payload to limit the amount of memory passed between processing elements. The packet header is checked its validity by ensuring various fields are consistent with the IP fields including checksum, header length, and IP version. The egress port of the packet is determined by a performing a longest prefix match route table lookup of the destination address. The time-to-live (TTL) field which indicates the maximum number of hops a packet can traverse is decremented. Finally the packet header is recombined with the payload and transmitted on the appropriate port.

2.2.2 Network Address Translation

Network address translation (NAT) [63] is a widely used edge application that enables communication between networks with conflicting address spaces. A traditional configuration is shown in Figure 2.2. On one side of the router with NAT functionality is a *local area network* (LAN) while on the other is a *wide area network*. As with many local networks, the address space used by the LAN is not coherent with respect to the larger WAN. This can happen for a variety of reasons including a shortage of WAN addresses or a legacy address allocation. A NAT device allows a single WAN address or a set of WAN addresses to be used among hosts on the LAN. This conserves address of the WAN and decouples the LAN infrastructure from the WAN.

To allow these two networks to communicate, a NAT router rewrites the packets that it forwards between these two networks. Hosts on the LAN with data requests or responses to hosts on the WAN have their packets forwarded to the NAT router. These packets' addresses are rewritten to use the address or addresses reserved to the NAT router in the WAN's address space. In order to keep address rewriting consistent and to forward responses from the WAN back to the appropriate hosts in the LAN, NAT routers keep a record of which packet flows are active between the LAN and WAN. Packet flows are often classified by a flow identifier (flow ID) which is a unique identifier of a network session. With TCP/IP packets, this four-tuple is typically: (source IP address, source TCP port, destination IP address, destination

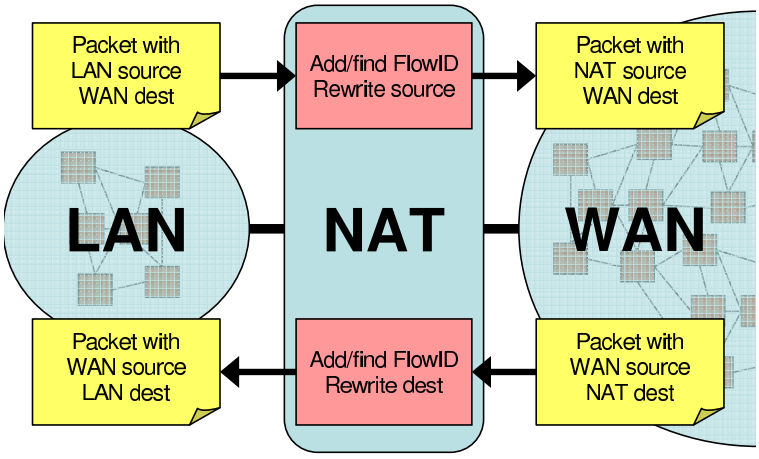

Figure 2.2: Block diagram of a NAT router

TCP port).

A typical scenario is depicted in Figure 2.2 in which a single WAN IP address is used for an entire LAN. A host on the LAN sends a request packet with a destination on the WAN. The packet is forwarded to the NAT router, where it searches its table of flow IDs for an existing connection. If one is found, it rewrites the packet's source address and port to match the entry in its table. If it is not found, the NAT router reserves a new port for the flow and adds it to the flow ID table. The packet header is rewritten and forwarded on to the WAN. A response packet from the WAN undergoes similar processing except that the destination address and port are rewritten instead of the source address. Connections are eventually removed from the table based on a time-out or a packet signaling the end of the session.

2.2.3 Differentiated Services

The basic functionality provided by the Internet's infrastructure is that of connectivity. Routers forward packets with best effort in an attempt to deliver as much data as fast as possible to their destinations. While this is sufficient for many applications which run on the network, others require more than just delivery to destinations. For example, people having a voice conversation over the network are sensitive to latency. Conversely for most web page data throughput is more important than latency. To provide bandwidth, latency, or jitter guarantees to packet flows, network administrators employ Quality of Service (QoS) architectures.

One type of QoS architecture is *Differentiated Services* (DiffServ) [12]. It is a provisioned model in which network administrators design the network infrastructure with broad categories of traffic. Interior nodes of the network apply different *per hop behaviors* (PHBs) to various classes of traffic. The classes of PHBs recommended by IETF include:

- Best Effort - no guarantees of packet loss, latency, or jitter
- Assured Forwarding - 4 classes of traffic, each with varying degrees of packet loss, latency and jitter
- Expedited Forwarding - low packet loss, latency and jitter

Routers are augmented with special schedulers, queue sized specifically for each type of packet class, and with bandwidth rating mechanisms to police the number of pack-

ets annotated in each flow. End applications bin their packets into each of these flows to indicate which data require which guarantees.

2.2.4 Web Switch

Much of the performance growth in the Internet can be explained by an increase in network functionality. By locating more of the computation in the network, end hosts are able to scale more with demand. A web switch is the offloading of one such application. A web switch uses the content of a packet to route it. For web requests, this could include the directory of the URL requested, the type of file, or the cookie of the client. With this Application layer information, a web switch can intelligently distribute traffic across a set of heterogeneous servers. Like NAT, it is transparent to both client and server.

Figure 2.3 shows an example of a configuration with a web switch which classifies HTTP requests into regular page requests, secure page requests, and image requests. This allows a single website to be hosted by multiple servers without a single host bottleneck. Other options for load balancing requests across multiple servers include random selection of a server, selection based on the least loaded server, and selection based on hashing. Each strategy can make use of a web switch by offloading the distribution computation to it.

Since even simple requests can involve exchanging multiple packets between client and server, session state is often stored in the web switch to ensure coherence between

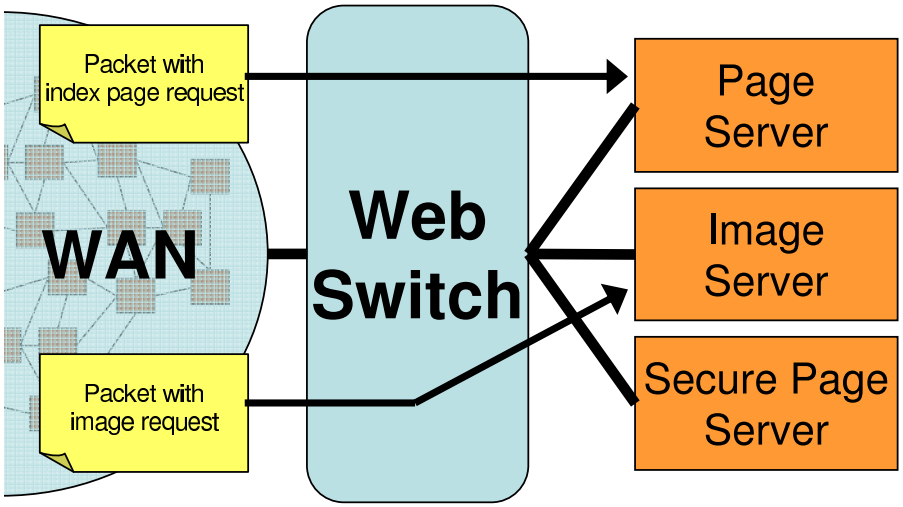

Figure 2.3: Web Switch Functionality

client and server. Such session state is part of the Transport layer which keeps track of states such as connection setup, packet ordering, and connection tear down. To ensure that the web switch is transparent to both client and server, web switches often employ "connection splicing" in which the router handles the initial setup connection with the client before the request flows are sent to the appropriate server. This allows the webswitch to see the first real packet of application data so that it can use that information to select the correct server.

2.3 Network Processors

The variety and complexity of networking applications is a natural fit for a programmable solution. This fact coupled with the increasing performance demands of these parallel applications has feuled the rise of network processors which is repre-

sentative of the larger trend toward application specific multiprocessors. Complex applications with varied feature sets that may change over time necessitated the use of a programmable solution, while application performance requirements demanded application specific optimizations to the platforms. A variety of network processors have come into existence in the past decade. They are distinguished by where they are deployed in the network and the principles the architects believed would bring the highest performance for the intended applications. A recent survey of network processors identified the key features and the major architectural axes of network processor designs [4]. These include number and type of processing elements, issue width, pipeline depth, memory structure, hardware accelerators, and on-chip topology.

To retain high performance, architects have employed a variety of techniques. These include customizing instruction sets to packet processing, using multiple processing elements and multithreading to exploit the inherent parallelism between packets and packet flows, tailoring memory architectures to packet header and payload movement, and specialized hardware for compute intensive packet operations such as hash engines and cryptography engines. The architectural details create a wide diversity of network processors. Differences include the instruction set used, the number of processing elements, how they are connected, and the size and number of distributed memories used. The following sections examines a few representative network processors to highlight some of the commonalities and differences between network processor architectures.

2.3.1 Intel IXP1200

The IXP1200 [32] is one of Intel's network processors based on their Internet Exchange Architecture. It has six identical RISC processors, called *microengines* (MEs), plus a StrongARM processor as shown in Figure 2.4. The StrongARM is used mostly to handle control and management plane operations. The microengines are geared for data plane processing and each has hardware support for four threads that share an instruction store of 1K-2K instruction words. Fast context swapping is enabled by a hardware thread scheduler that takes only a few cycles to swap in ready thread. The IXP series uses cooperative multithreading, meaning threads swap out voluntarily and are not preempted by other threads. The register file itself is split logically into individual areas for each thread, so a thread's state does not have to be pushed to memory. There is also a shared region which threads can used for sharing data between threads.

To cope with small size of the instruction store, each of the threads on a processing element may share instruction memory. Register addressing within an instruction may operate in different modes: *Context relative addressing* is addressing a registers in the region local to a thread and *Absolute addressing* is the registers in the shared pool that are common to all threads. In context relative addressing and instruction adds an offset such that the register in the appropriate pool is addressed by any thread that enters that context.

The memory architecture is divided into several regions: large off-chip SDRAM,

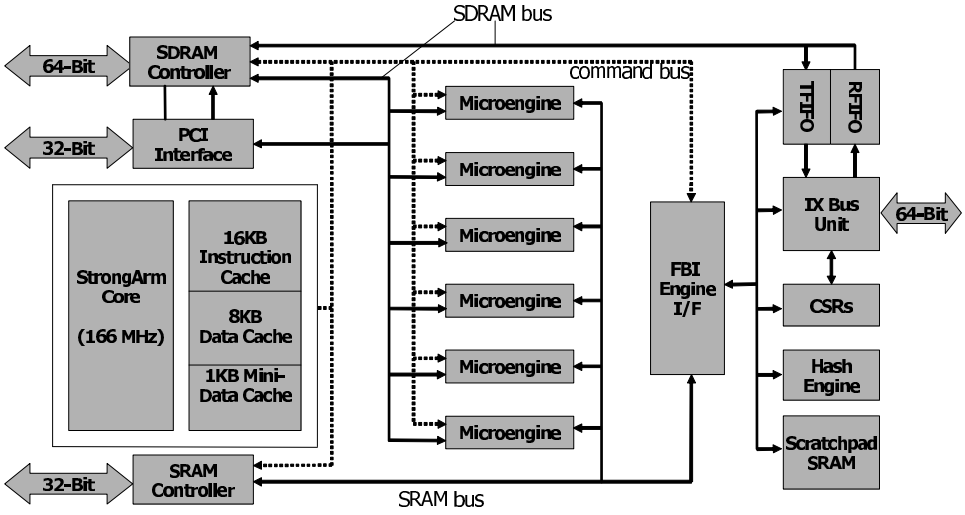

Figure 2.4: Block Diagram of the Intel IXP1200

faster external SRAM, internal scratchpad, and local register files for each microengine. Each region is under the direct control of the user and there is no hardware support for caching data from slower memory into smaller, faster memory (except for the small cache accessible only to the StrongARM). The interconnection of the memories and processors to peripherals is customized for packet movement. A dedicated bus between DRAM and the ingress and egress buffers is intended for movement of the packet payload to and from memory without impeding the progress of existing computation. Interconnection also exists between microengines and the ingress and egress buffers too allowing designers to move packet headers directly to the register files so that they may be immediately operated on.

2.3.2 Intel IXP2xxx

The IXP2xxx is Intel's most powerful family of network processors to date, capable of up to 23.1 GIPs [33] [35]. Each member of the family has an XScale processor that is intended for control and management plane operations. Like the IXP1200, there are multiple RISC processors with instruction sets tuned for packet processing called *microengines* (MEs). With instruction set architectures geared towards data plane processing, each microengine has hardware support for eight threads with a shared instruction store. To permit fast context swapping, each also has a hardware thread scheduler monitoring the readiness of each thread. To keep these cores busy, the memory architecture is divided into several regions: large off-chip DRAM, faster external SRAM, internal scratchpad, next neighbor registers (NNs), and local memories (LMs) and register files for each microengine. Each region is under the direct control of the user, and there is no built-in cache structure for the microengines. Next neighbor registers allow producer-consumer relationships between neighboring microengines to avoid communicating through slower, globally shared memory. Shown in Figure 2.5, the IXP2400 is a midrange performer member of the IXP2xxx family with 8 microengines running up to 600 MHz, each with 8 hardware supported threads.

2.3.3 EZChip NP-2

EZChip's third generation family of network processors is NP-2 [22]. Geared for 10Gbps of packet processing, the NP-2 integrates multiple processing elements,

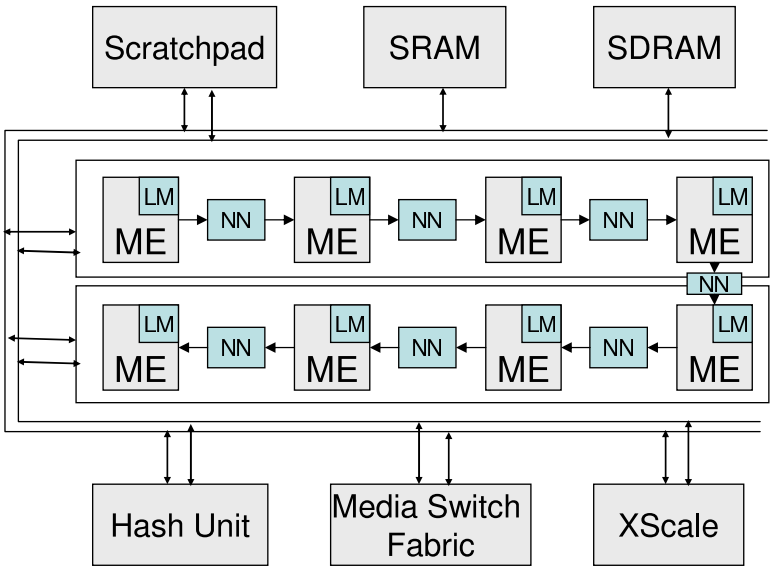

Figure 2.5: Block Diagram of the Intel IXP2400

traffic managers, classification engines, and Gigabit MACs onto a single die. To achieve performance on certain network applications, EZChip employs heterogeneous processing elements called task optimized processors (TOPs). They have customized data paths and instruction set tuned for a certain functionality. There are four types of TOPs each tuned for a particular stage of packet processing: parsing the packet, lookup and classification, forwarding the packet and making decisions on quality of service, and modifying the packet. The TOPs are connected by a "super-scalar" architecture so that many may operate in parallel. TOPs are not tied to a particular port allowing them to service any packets ready for processing. Hardware dynamically manages scheduling packets to a particular TOP and ensures that packets remain in-order for each port.

2.3.4 Cisco SSP

The highest performing network processor to date is found in the Carrier Routing System (CRS-1) from Cisco. CRS-1 is a high performance router capable of processing 92 Terabits per second. This core router is based on a network processor called the *Silicon Packet Processor* (SSP) [68], which was developed jointly with IBM and Tensilica. The SSP is capable of 40Gbps of throughput and uses 188 Xtensa processing elements, which is by far the most processing elements for a network processor to date. Extra processors are also on-chip to improve the yield of fabrication. The programmability is utilized to instantiate different feature sets, so the SSP can be used across a many product lines and in different parts of the network. Significant chip real estate is spent on hardware managers for controlling peripherals and distributing packets to the processing elements.

2.4 Commercial Programming Environments for Network Processors

Network processors represent an enormous performance potential by delivering many processing elements, customized memory hierarchy, special purpose hardware, and a topology geared toward efficient packet movement. Tapping into the potential of these devices requires addressing each of the key challenges outlined in Section 1.2.1. This burden falls to either dynamic management, offline tool chains, or programmers.

Several commercial approaches have risen to facilitate programming these devices. While they have chosen to distribute the burden differently, each approach is a general purpose, performance oriented design entry environment.

The philosophies for the how to program these devices have tended towards two extremes: burdening the programmer and burdening the hardware. The first is to completely expose the programmability of the architecture to the application designer. With no intermediate abstraction layers, programmers are expected to arrive at an efficient mapping and implementation themselves. The second is to hide the details of the underlying architecture through hardware managers. This approach takes control out of the hands of developers and offline tools so custom mappings are not possible. Such a hardware runtime approaches introduces either overhead which attempts to solve the mapping problem suboptimally or restricts the possible mappings to more trivial solutions. In either case, the result of the difficult implementation process is an unportable application description. Underutilized in these commercial approaches are the use of tools and automation and software runtime systems. The following section covers programming approaches that exist commercially today.

2.4.1 Assembler

As many of these architectures are based on programmable RISC cores with instruction sets tailored to packet processing, the easiest programming environment to ship with such a device is an assembler. Programmers write each instruction to be

executed on the platform and select the specialized instructions that can be used to accelerate their algorithms. On architectures with exposed distributed memory, programmers explicitly manage the data in those memories. Shared resources such as hardware acceleration units are also handled directly by the programmer. To properly interact with some peripherals, programmers must write code to explicitly interface with state machines that control them.

Using assembly for design entry gives the programmer complete control over the architecture. For highly exposed architectures this allows the programmer to directly utilize the platform specific features of the hardware. With non-preemptable processing elements, like the microengines on the IXP, critical sections and race conditions can be managed temporally with cycle counts. Programmers control when threads swap on and off the processing element, so designers can build critical sections by not having any thread swaps in the region. Programmers are not forced to use locking variables to ensure the fidelity of a critical section, which saves memory space and instructions, but makes the program unportable to network processors with preemptable cores.

Some network processors limit the programmers control of the architecture through hardware. For the NP-2 programmers write a single sequential program. This creates an image with no parallel programming or multithreading expressed. The allocation of the TOPs to processing elements to incoming frames, passing messages between the TOPs as well as maintaining the ordering of frames is performed in hardware

transparently to the programmer. Assembly still allows the designers to maximize the architecture, but the interface to the architecture is more restrictive and has significant hardware overhead.

If the programs to be run on these processing elements are small and the number of different programs to run are few, this model works well. Difficulties in using assembly can be handled when applications require few instructions and homogeneity in the application limits the parallel complexity for the programmer. As soon as designers need to implement a large or complex application, using assembly as a description language becomes problematic. Along with all of the other difficulties of crossing the networking implementation gap, designers must deal with all of the architectural features with little or no tool support. Furthermore, resulting applications are tied to the particular architecture.

2.4.2 C-variants

While many of the network processor processing elements have special instructions, they are still typically RISC cores that are somewhat amenable to compilation. Looping constructs, branching, dead code elimination, and register allocation are all productivity enhancing features of using C over assembly. Architecture specific features are captured by using intrinsics which are function that represent special instructions not targeted well by compilation. Exposed memory regions are exported with specific data types. For example, the microengine C compiler contains typing

constructs that indicate which memory the variable resides in. With such constructs in the language, these network processor C variants have additional features which must be learned.

Architectural restrictions may also remove constructs from C. For example, the IXP series has no intrinsics for pushing the entire variable set for a function. This means for liveness and register allocation, the program must be statically analyzable. Therefore Microengine C [34] disallows recursion and function pointers. Algorithms already designed in C may have to be rewritten to respect these restrictions imposed by the architecture.

The productivity benefit of using a C over assembly is relatively minor as this still leaves a huge programming task to the designer. Programmers must manually arrive at an assignment of those computational tasks to processing elements along with layout of data to memory and intertask communication to physical communication links. These intertwined design decisions result in a large, irregular design space that has a large impact on the implementation's final performance. As vendors add more cores and more memory, this programming challenge becomes more difficult with each generation.

2.4.3 TejaNP

TejaNP [65] is commercial product which provides a robust interface for programming a diverse set of platforms, one of which is the IXP2xxx series. The general

approach for describing and implementing applications focuses on portability, performance, and ease of use. The basic design flow for targeting the IXP2xxx is described in Figure 2.6. Programmers start by customizing library elements that are to be used in their application. These elements include state machines, data structures, queues, mutual exclusion constructs, and device IO. State machines are used to organize the computation, and programmers describe the operations performed by writing C like code for each edge of the state machine. The application model is general in that describing computation and organizing it with state machines is not domain specific.

These components including the state machines are stitched together to completely describe the functionality of the application. The data and control plane of an IP forwarder represented in this connected form is shown in Figure 2.7. The green boxes represent computation described by the state-machines, orange boxes are shared data structures, and the blue cylinders are communication channels. Present in the environment is an analogous architectural representation made up of processing elements and memories hooked together by buses or point-to-point links. After the application is completely described, programmers manually map each of these application components to their corresponding architectural one. State-machines are mapped to threads of processing elements, data structures to memories, and communication channels to hardware links.

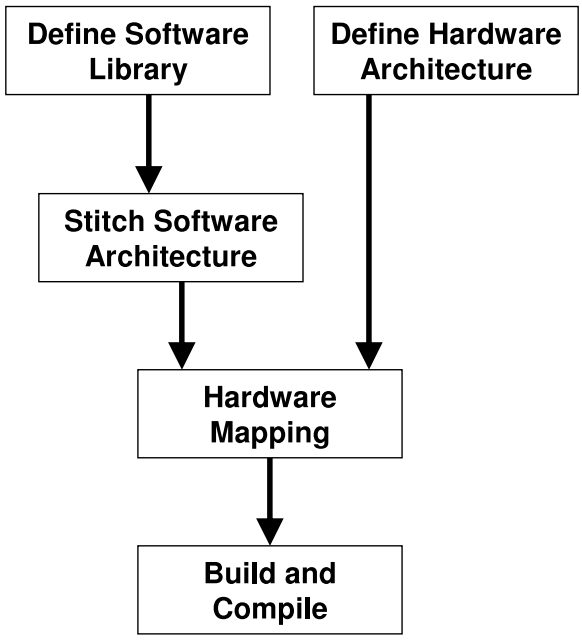

Figure 2.6: Design flow of TejaNP

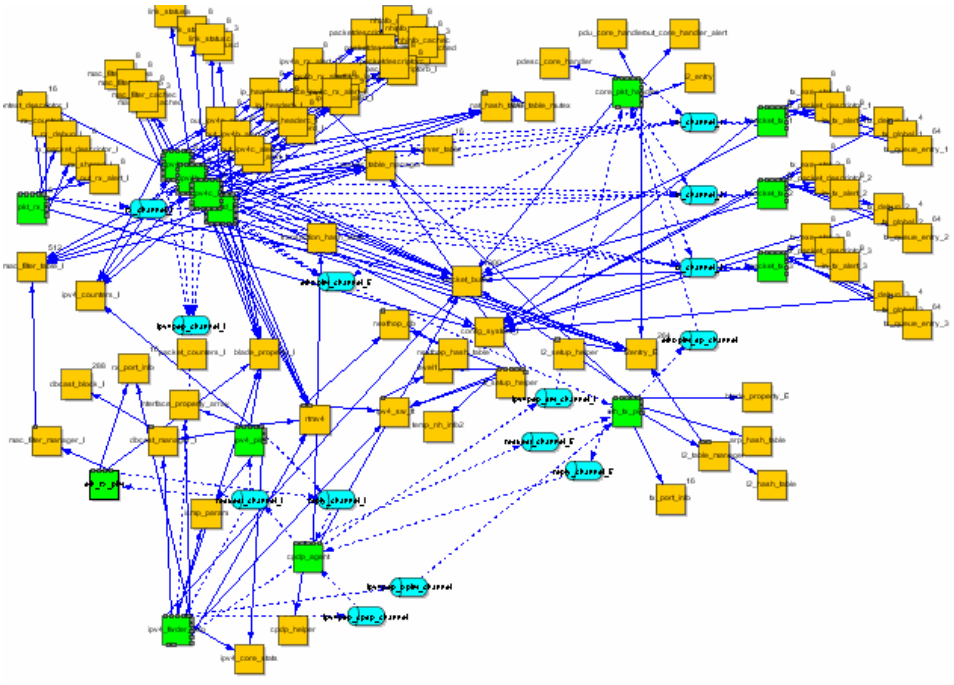

Figure 2.7: Application representation in TejaNP

2.4.4 Microblocks

The difficulty of using C-variants or assembler was recognized by designers early on. Intel released the Microblocks framework originally with the IXP1200 [31] and now exists for the IXP2xxx [36]. It institutes some design methodology along with a library of elements optimized for performance. Built on top of existing tools for the IXP family, Microblocks organizes code in blocks of computation that may be connected together to describe an application. Each element is highly configurable as there are no invariants it must strictly adhere to. Programmers manually assign each of the computational blocks, their corresponding data, the their interconnection to the appropriate architectural resources. Furthermore, programmers must write the corresponding glue code for each of these assignments. For example, if computational block needs to be moved to another Microengine, the programmer must manually relocate the code and rewrite the code that connects it to the rest of the application.

Since design space exploration is difficult in this system, Intel has provided an architectural tool for the fast performance estimation of a mapping on the architecture. It derives its estimates quickly from an analytic model or more slowly and accurately from a transaction simulation, neither of which are cycle accurate. These assist the design space exploration in that new designs need not be implemented for performance evaluation.

2.5 Domain Specific Languages

Commercial approaches for programming network processors are used by a small community of users, tightly tied to the target platform, and require many difficult design decisions to arrive at an implementation. Designers have an immense amount of freedom when programming, but this lack of restriction leads to the creation of unportable, unstructured code.

Domain specific languages in networking provide an attractive alternative to design entry by making useful restrictions to design entry along with using communication and computation semantics natural to the application domain. These are based on a few of the fundamental networking principles discussed in Section 2.2. First is the isolation of the layers of the networking stack. This allows designers to use a particular abstraction of the data. Second, domain specific languages utilize the OSI Network layer principle of packet independence. For those applications which use Transport layer semantics, the independence of packet flows is captured. Finally, the methods for invoking execution in a network application are through requests and responses. Requests correspond to available resources to send or process packets or periodic events to schedule computation. Conversely, responses involve data that appears in the network ready to move through the computational paths of the application.

The following section summaries how a representative group of domain specific languages realize these concepts.

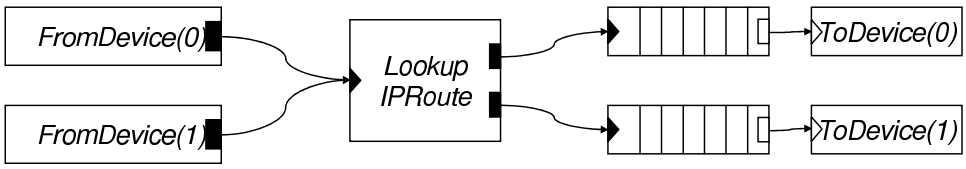

Figure 2.8: A Simple Click 2 Port Forwarder

2.5.1 Click

Click [39] is an environment designed for describing networking applications. A Click description of an application is functionally complete, such that a programmer may test and refine his application using freely available tools on a general purpose platform. Based on a set of principles tailored for the networking community, Click has been used to described a variety of network applications [40] [59]. Applications are built by composing computational elements, which correspond to common networking operations like classification, route table lookup, and header verification. Prior work has shown that the parallelism can be extracted and utilized for general purpose multiprocessor platforms [18]. Figure 2.8 shows a Click diagram of a simple 2 port packet forwarder, in which packets ingress through *FromDevice*, have their next hops determined by *LookupIPRoute*, are queued, and finally then egress through *ToDevice*.

Elements have input and output ports that define communication with other elements. Connections between ports represent packet flow between elements. Ports have two types of communication: push and pull. Push communication is initiated by the source element and effectively models the arrival of packets into the system. Pull

communication is initiated by the sink that can model space available in hardware resources for transmitting packets. Each actor has push or pull ports that may be composed with other elements to form paths of push elements, and paths of pull elements, called *push chains* and *pull chains*, respectively. Note that these push chains and pull chains form independent computational paths through the graph, indicating parallelism inherent in the Click description.

Figure 2.8 shows a Click diagram of a simple 2 port packet forwarder. The boxes are elements and the small triangles and rectangles within elements represent input and output ports, respectively. Black ports are push ports, while white ports are pull ports. The arrows between ports represent packet flow. The *FromDevice* element abstracts an ingress port. As a packet enters the system, it is received by *FromDevice*, which then passes the packet to *LookupIPRoute*. *LookupIPRoute* contains a table that maps destination IP addresses to output ports. After *LookupIPRoute* determines the packet's output port, it is enqueued. When there is free capacity in the MAC port buffer, *ToDevice* dequeues a packet and sends it to the appropriate MAC port.

Click is implemented on Linux and uses C++ classes to define element behavior. Element communication is implemented with virtual function calls to neighboring elements. The sources of push paths and the sinks of pull paths are called schedulable elements. Firing the schedulable element of a path will execute all elements on that path in sequence. Each of the computational paths formed by these sequences can operate independently, while packets on a particular chain stay in the same order as

they arrived at the beginning of the chain.

2.5.2 NesC

NesC [25] is programming language designed for embedded networking applications on a small scale. Used in conjunction with TinyOS [29] it is used to deploy applications on many small, wireless nodes. Like Click, NesC is actor oriented language in which programmers build their application out of *components*. Designers first write these components in a C-like language and then "wire" them together. Component ports are typed as either *events* or *commands*. This typing of ports allows for a split-mode operation in which long latency events may be decoupled from an execution. Commands correspond to the request of an activity (e.g. start a timer) and events represent the mechanism for response (e.g. a timing event has a occurred).

Modules contain events and tasks, which are used to model concurrency. Link in TinyOS, tasks run to completion and are not preemptable by other tasks. When tasks are ready to be run, they are sent to the scheduler and executed when there are idle cycles on the processor. Conversely events may interrupt tasks and events. They are used to model hardware interrupts from peripherals. In this way they are somewhat analogous to the push and pull of Click, where push events often correspond to schedulable elements with rated execution, while pull events represent a peripherals ability to control the execution of the application. To handle shared state, users can declare sections of code atomic. These constructs permit the static

analysis of the application description. Code that is only reachable from tasks is run completely synchronously while that which is reachable from an event is considered asynchronous. With this information offline tool can find potential race conditions involving shared variables.

2.5.3 Baker

Baker is a domain specific language developed by Intel. It is the entry language for the Shangri-La design flow [27] which is summarized later in Section 3.3.3. Like other networking DSLs, it too is actor oriented with the basic block of computation being a *packet processing functions* (PPFs). A PPF can have local data, be the source and sink of packet processing, and describe a packet operation. These are connected together via communication channels which permit static analysis of the application description. These channels are asynchronous, directional queues that connect input and output endpoints of PPFs. Packets flow along these communications to create a self-described data flow graph known as packet flow graphs. Like *fromDevice* and *toDevice*, the PPFs that correspond to packet reception and transmission abstract away the details of the actual hardware devices.

2.6 Networking Implementation Gap

Like the situation discussed in Section 1.2.1, network application designers are faced with the problem of crossing the networking version of the implementation gap as shown in Figure 2.9. A networking domain specific language description of an application enables natural design capture. They do this by providing libraries of basic packet operations and primitives for packet access. As in the general case, these natural descriptions of the applications are quite different from programming environments commercially used with network processors Their emphasis on performance rather than productivity has given rise to general purpose, low level programming environments that expose much of the features of the architectures. Like the flow described in Section 1.2.2, they suffer from long design times and produce rigid implementations.

The networking implementation gap exhibits each of the three keys to be solved for producing efficient implementations. Application designers must extract the concurrency enabled by packet and/or flow independence. Designers must internalize or explicitly create a model of the network processor which captures performance critical features such as multithreading, next neighbor registers, and chained interconnection topology. Finally, they must exploit the parallelism of the architecture by mapping the application to the architecture.

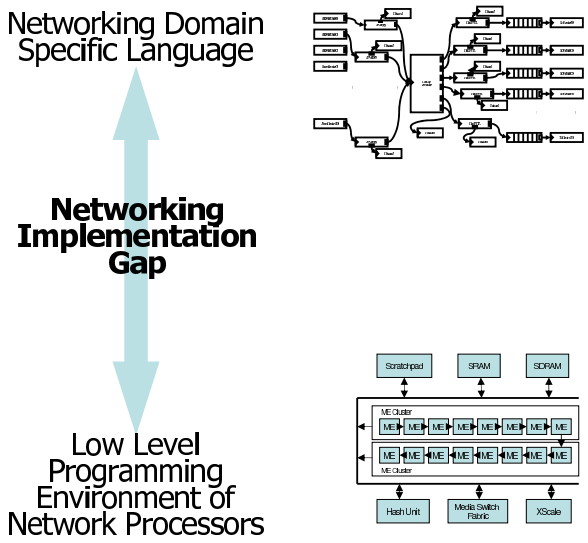

Figure 2.9: Networking implementation gap

Chapter 3

Crossing the Networking Implementation Gap

An ideal solution to crossing the networking implementation gap permits natural application capture, utilizes the architectural resources of the target network processor, and arrives at solutions quickly that can be verified for correctness. Our solution achieves many of these objectives by implementing a novel design framework that utilizes automation to arrive at efficient implementations quickly. In the design framework, programmers start by describing the application to be implemented with a domain specific language. Application level optimizations can be made on this application description. Using a target specific library of computational elements, the application transformed into to an application model which has an analogous architectural model. The mapping problem is formulated as a integer linear programming

problem. Optimal solutions to the mapping problem are found using a powerful general purpose solver. Based on these assignments, code is generated for compilation on individual processing elements. The solution presented in this thesis moves the body of research focused on crossing the implementation gap closer to an ideal domain specific solution.

3.1 Properties of an Ideal Solution

While the starting and end points of the network implementation gap have developed, the ability to bridge between them remains a difficult problem to electronic system designers. An ideal solution to the networking implementation gap would have a few key properties. Application entry would ideally be in something akin to a domain specific language to facilitate productive design capture. The application description would be portable to other network processors. Second, a design flow would take this application description and produce efficient implementations. Efficient implementations on application specific multiprocessors implies that the ideal solution would load balance computation and assign data into distributed memories in such a way that performance is maximized.

These implementations would be produced quickly. Performance oriented networking applications invariably benefit from designer insights and successive refinement of the application based on performance results. Faster time to implementation enables more design feedback cycles and therefore a higher quality solution. The

ideal solution would be able to incorporate arbitrary user guidance without effecting the rest of the design flow. It is often the case in high performance systems that certain design restrictions or legacy code force a part of the application to a certain part of the architecture. An ideal design flow would be able to use such information seamlessly when generating software to be run on multiple processing elements.

3.2 Proposed Approach

The divide between applications described in domain specific languages and their implementation platform is wide. Our proposed framework for traversing this chasm is described in Figure 3.1. A designer describes the application in a DSL, while ideally not being concerned with the final implementation platform. Having an abstract description of the application provides the unique opportunity for high level optimizations. After optimizations the application is transformed into a model that is mappable to the target architecture. To be mappable the application model includes information needed to arriving at an efficient implementation on the target architecture. In addition to the computation described in the application, this information could include packet distribution, data size and scoping, communication bandwidths between communication blocks. Much of this information comes

This abstract representation of the application is optimized to produce a task graph suited for high performance systems. The task graph is made up of computational elements written specifically for the target architecture. Performance and

resource usage parameters of these elements along with a model of the target architecture are used by the mapping stage. This stage includes the allocation of computational tasks to processing elements, the layout of data in memory, the assignment of communication to interconnect, and the scheduling of tasks. Based on the derived mapping, the task graph is divided into the individual programs to be run the processing elements. A compiler specific to each processor finishes the flow to implementation by producing executables from the individual programs. Since prior stages use approximations of tasks properties, a profile of the resulting executable feedbacks to the high level optimization stage and the mapping stage. Consequently, feedback helps guide both the DSL to task graph transformation and the mapping of the task graph. After a sufficient number of iterations, the final result of the entire flow is a program optimized for performance within resource constraints. The following sections covers each of these steps, while certain key steps will be discussed in detail later chapters.

3.2.1 Domain Specific Language Application Representation

The ideal method for describing an application productively is to use a domain specific language which are discussed in Section 2.5. A domain specific language like Click (Section 2.5.1) has gained popularity in the networking community because of its ease of use. A programmer can describe an application in a few lines and immediately have a functional implementation. There is no overt performance benefit to utilizing

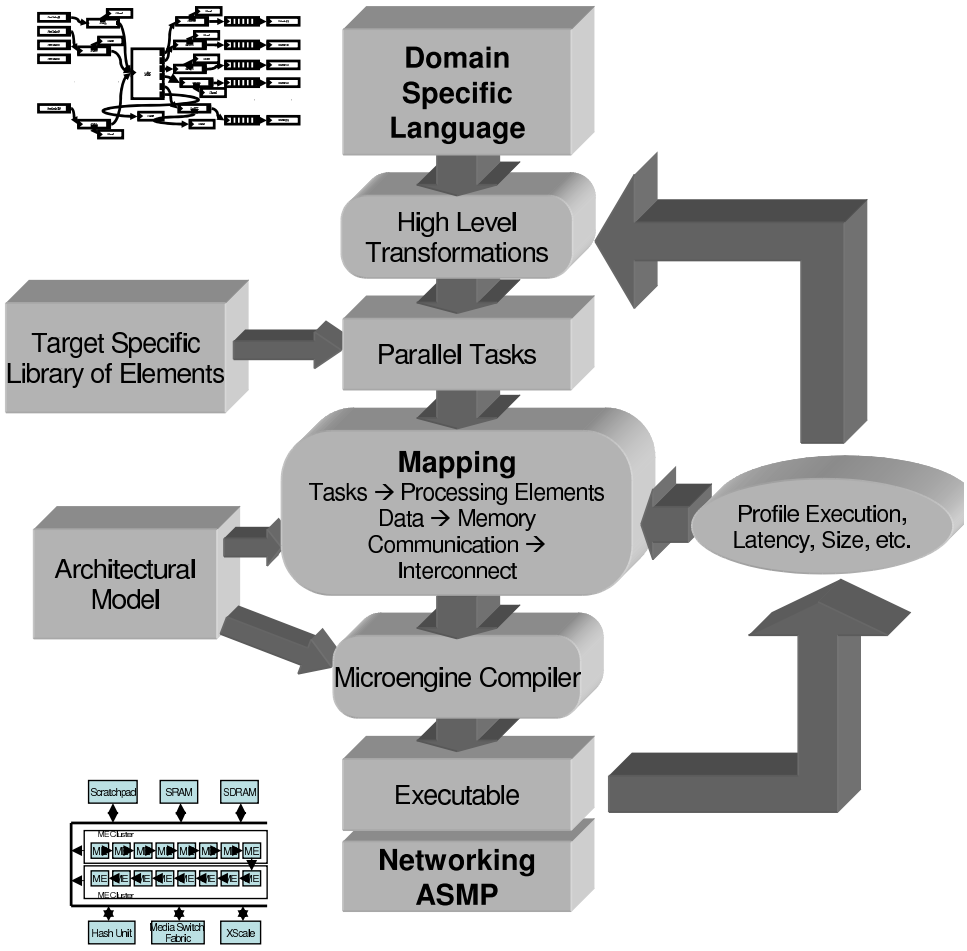

Figure 3.1: Proposed flow to cross implementation gap

this approach as Click is built on top of C++. For a designer to use it, the primary motivation is to arrive at an implementation more quickly.

General purpose languages suffer from being so expressive that it becomes intractable to extract much task level parallelism. Besides productivity benefits, DSLs offer a unique opportunity for the extraction of parallelism inherent in the application model. Previous implementations of Click on parallel machines found that it is amenable to exposing the parallelism in networking [18]. The structural description of the application is amenable to automated extraction of parallelism. Furthermore, the high-level representation of the application with domain specific restrictions makes DSLs a good starting point for optimizations and transformations to an explicitly parallel description to a mappable application model.

3.2.2 Application Level Transformations and Optimizations

With an application described at an abstract level, many application level transformations are possible. A low-level representation of the application like C or assembly obscures this structure. Interprocedural optimizations or the altering of thread boundaries are traditionally difficult to do with sequential code stitched together with shared state and communication channels. However, such optimizations are readily possible with an application description that limits what is expressible. A DSL provides invariants useful to an optimizing tool.

This framework employs these invariants in multiple ways. Replication of certain

computational blocks can be done safely thanks to the communication and concurrency semantics of Click. Since blocks of code must expose shared data, replicating tasks can be attached to the appropriate shared data and replicating any local data. With data elements revealed, inter-element merging of data can also occur along with the elimination of redundant initialization code.

The structural nature of networking domain specific languages coupled with their restricted communication semantics enables the pipelining or retiming the application. By inserting queues, execution paths can be made shorter and overall performance can be increased. These optimizations are covered in Section 5.4. The structure also makes it possible to transform to a representation of the application that may be mapped to the target architecture: a task graph.

3.2.3 Application Model

Our application model is called a task graph. It is a representation of the application which contains information needed to arrive at an efficient implementation. The construction of the task graph is specific to a pairing of an application domain and an architecture. It is a balance of two competing goals:

- **Including information** - By providing more visibility into the application requirements, the mapping engine has more opportunities to arrive at an efficient mapping.

- **Creating a simple model of the application** - Constructing a model of

the application that contains only the salient information to make the mapping problem easier

While the structure of the task graph is specific to an application domain and architecture pairing, mapping network applications to network processors can be described by an *interactivity graph* of the application graph. Interactivity graphs expose the blocks of computation that may be run in parallel, the data that is connected to them, and the communication links. These components of the task graph are the application representation of the key problems of mapping the application to the architecture. Furthermore, the interactivity graphs produced by this flow represent the "steady state" of an application, in which tasks dependencies are decoupled by queues. At steady state most queues are assumed to have some elements so that each computational blocks has the inputs in needs to fire. In other words there are no links to indicate dependent execution between tasks.

Certain properties of the components of the task graph are necessary for arriving at a good mapping from network application to a network processor. For tasks these annotations include execution time, rate of execution, instruction store footprint, and time spent in long latency events. Data is annotated with size, number of accesses, and the kind of access. Connections between tasks and data include information regarding the rate of data transferred over it and the directionality of the link. Section 4.4 describes formally the task graph.

3.2.4 Architectural model

The architectural model is analogous to the application model. It is constructed with the same balance between exposing features and abstracting away complicating details. Leveraging our own experience with the architecture to identify its key features, we model the architecture as a directed graph of architectural components. Nodes in the graphs are processing elements and memories which are connected by interconnect. Each of these components of the architecture model are annotated with features of the architecture. Processing elements provide compute cycles at certain rate, memories house data with a certain access time, and interconnections connect processing elements and memories together at a certain access time. Section 4.5 describes the architectural model in detail.

3.2.5 Mapping

The architectural model and application model are constructed specifically so that one maps onto the other directly. We formulate the mapping problem as a set of constraints using these two models. Using a boolean selection matrix and a set of linear constraints, we utilize a powerful general purpose *integer linear program* (ILP) solver to find an assignment of the components of the task graph to the appropriate architectural components. The solutions are optimal or, to reduce the solve time, within a guaranteed bound of the optimal. Such a framework satisfies the designers desire to be able to arrive at efficient solutions quickly. Designer feedback can be

incorporated into the mapping engine by including custom constraints. The details of this formulation are covered in Section 4.6. These assignments are used by the code generation and compilation phase of the framework.

3.2.6 Code Generation and Compilation

Based on the mapping assignments determined and the code contained in the element specific library, blocks of sequential code are generated that can be used by processing element specific compilers. The target specific elements are written as stylized C code. Besides adhering to the communication standards of the domain specific language, the code describing the application includes tags to indicate where variables are to be annotated with their memory locations. Like static linking, these memory declarations sections are rewritten based on the derived assignment. The way to connect elements is contained in the structural information of the domain specific languages. The connections are similarly generated by rewriting the calls to downstream or upstream elements. Task allocation is realized by the synthesis of drivers for each of the processing elements. Tasks allocated to a processing elements have their associated schedulable element called inside of the appropriate main function. The result of each of these static rewriting is a set of sequential programs which can be tackled by traditional compilation.

3.2.7 Feedback

This design flow requires either estimates or profiled cycle counts. The number of execution cycles consumed and latency cycles can be determined by running the library elements on processing elements with typical or worst case inputs. Memory and communications access times need also to be profiled for accurate memory assignment costs. Sizing of data can also be memory dependent. For example a one byte datum on a 4 byte word machine may consume an entire word, especially if the compiler is unable to consolidate smaller variables into a single word. Instruction store footprint size is highly dependent on the instruction set architecture of the processing element.

Design flows for performance oriented systems benefit from feedback cycles. By refining profile and estimation metrics, optimizations and transformations can be incrementally improved. Besides raw performance refinement, designers often gain insight into the best implementation from observing implementations on various workloads.

3.3 Other solutions to crossing the implementation gap

Other solutions exist in networking to cross the implementation gap to a multiprocessor platform. Here we examine solutions in which design entry is done in a language natural to the application domain. Many of these solutions use C or assembly at some part of the design flow, but they do not begin with them. They target

a variety of multiprocessor platforms. Specifically noted with each of the solutions are how the solutions tackle the three keys to crossing the implementation gap as described in Section 2.6. For some, architectural features are abstracted away, while others put certain features under the control of the programmer. Determining how the application is to be distributed across the device is left to the programmer, automated tools, a software runtime system, hardware mangers, or some combination therein. Extracting the parallelism from the architecture is done through automated tools or exposed by the programmer. For each of these solutions bridging across the implementation gap requires a decision on each of these critical features, which determines. The following section covers a set of such approaches.

3.3.1 SMP Click

A natural extension of Click is to port it to multiprocessor architecture to take advantage of the inherent parallelism in processing packet flows. Push chains and pull chains capture this parallelism. While there are some shared resources between chains, each is computationally independent. A multithreaded version of Click that targets a Linux implementation [18] takes advantage of this fact. To get Click to execute in parallel on a *symmetric multiprocessor* (SMP) platform, the authors modify the original Click scheduler and securing shared memory with critical sections. The execution chains within an application expose parallelism that is readily exploited by the SMP platform. Data and instruction memory are cache coherent so a waiting

task can safely be run on any processor.

Such freedom allows the authors to experiment with the different load balancing techniques. They try a dynamic load balancing scheme which assigns a single thread to each processing element. Each thread keeps a list of the schedulable elements it is in charge of executing. Imbalances between processing elements are evened through adaptive load balancing. The authors found that statically scheduling some of the tasks can provide a specific performance benefit. Imbalances in computation may lead to a better mapping for elements which are communicating directly. By being mapped onto the same processing element, they share an L1 cache for communication, lowering the execution time for each. Scheduling algorithms are considered in detail in [13]

3.3.2 PacLang

PacLang [21] is a research effort to divorce application description form the architecture. This tenet lead the authors to the design flow described in Figure 3.2. First applications are described in a *linear typed language*. By linear typing packets, tools can statically assert that a packet is never accessed simultaneously by two different threads. Threads do not need to use locking mechanisms to modify packet contents. Furthermore, compilers can optimize the program by safely moving packets between physical chips and create local copies of packet data for faster access. These linear typed blocks are then connected to each other through queues.

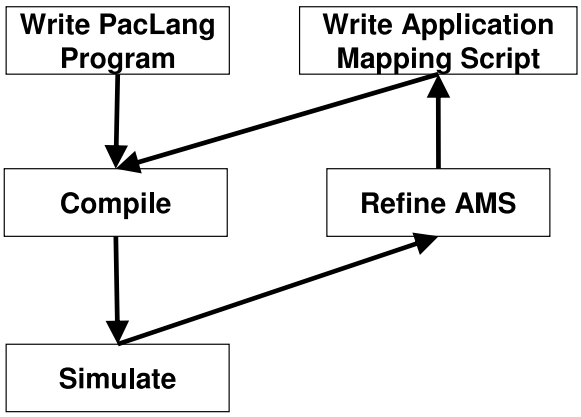

Figure 3.2: Design flow using PacLang

This application description is mapped to the architecture through an *architecture mapping script* (AMS). The script can direct the assignment of tasks but also guide optimization. Such optimizations can include clustering or retiming the application.

3.3.3 Shangri-La

Shangri-La [27] targets Intel's IXP2xxx series of network processors and is the closest relate work to this approach. The central tenets are to start from a DSL (*Baker* described in Section 2.5.3), to target independent building blocks, and to utilize run time reconfigurability. The proposed flow is shown in Figure 3.3. The programmer describes their algorithm in Baker. Programmers must write the application in such away the any block of computation may be replicated an arbitrary number of times. A functional profile of the code is made which is annotated on the intermediate representation of the application. A pipeline compiler groups these

code segments in to larger groups called aggregates. The pipeline compiler assigns the data to memory and aggregates to processing elements based on the functional profile. The aggregates are compiled by the aggregate compiler producing binaries for the appropriate processing elements. These tasks are then managed using a runtime compiler. It uses the previous annotations as guides to the assignment of aggregates to processing elements and then adapts those assignments based on traffic load.

Shangri-La uses an architectural model which exposes processing elements and memories. The performance metrics are based on longest chain annotations. Mapping the application to the architecture is a combination of offline automated and manual annotations along with runtime support for reconfiguration. To exploit the parallelism of the architecture, the programmer must specify this application code that exposes thread level parallelism. While at design entry they may not be burdened with the choice of how many threads to allocate to particular segment of code, they must write the code such that these choices may be made later.

The choice of relying on a hardware system to dynamically manage the mapping has a number of implications. Since the application may be fine tuned at runtime, the mapping burden on the application designer is lighter. However a dynamic runtime system on an architecture with little hardware support (such as the targeted IXP family of network processors) implies significant overhead for reconfiguring the system. For example, if computation section in a pipeline of packet processing is to be moved to another processing element, the computation's instructions must be moved along

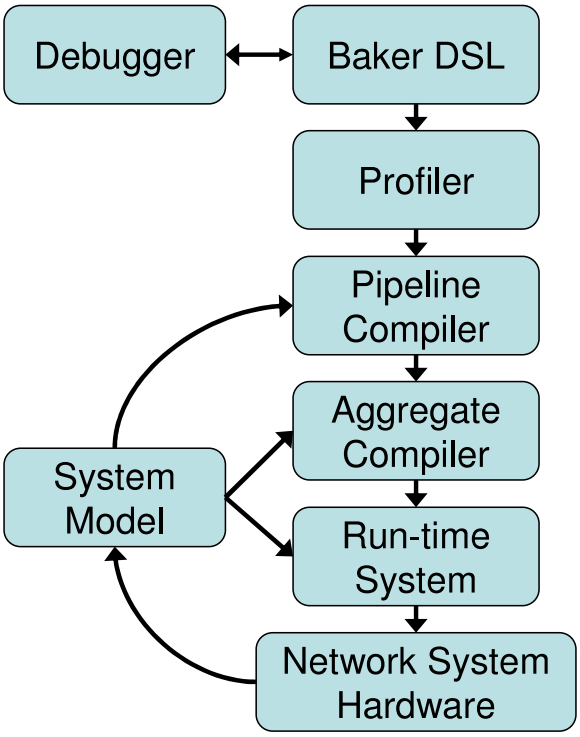

Figure 3.3: Shangri-La flow

with any active state. To ensure correctness the original block must be halted to move the internal state as well as the packets in flight to it. If many packets are in a queue waiting to be processed, many cycles will be spent moving the data from one memory to another. Meanwhile the packet pipeline is stalled waiting for this reconfiguration to complete. The granularity and periodicity of reconfiguration is limited by the architecture, and may not permit fine grain optimization of computational elements. Furthermore, a centralized manager of computation and data mapping is not likely to scale to many processing elements.

3.3.4 NP-Click

Our initial attempt at closing the implementation gap, is NP-Click, a programming model for the Intel IXP1200 [61]. (The "NP" indicates that this model is for network processors.) Rather than promote a new style and syntax of programming, we based our model on Click. Like Click, NP-Click builds applications by composing elements that communicate using push and pull. In NP-Click, we implement the elements in Microengine C to leverage the existing IXP1200 compiler. To improve implementation efficiency, NP-Click complements the abstraction of Click with features that provide visibility into salient architectural details. Specifically, NPClick enables programmers to

- control thread boundaries to effectively manage processor and thread
- map data to different memories
- separate design concerns of arbitration of shared resources and functionality.

Our experience with programming hardware multithreaded architectures shows that arriving at the correct allocation of elements to threads is a key aspect to achieving high performance. Thus, we enable the programmer to easily explore different mappings of elements to threads. We outfit NP-Click with a similar mechanism to SMP Click, in which schedulable elements are the entry point int parallel blocks of computation. To implement the latter, we synthesize a scheduler that fires each path within that thread.

The amount of parallelism present in the target architecture places pressure on shared resources. For example, the IXP1200 has 24 threads that can each simultaneously query a control status register. This can lead to bus saturation, which in turn can cause lengthy delays in requests. Such situations lead to potential hazards that necessitate arbitration schemes for sharing resources. To recognize the importance of sharing common resources, we separate arbitration schemes from computation and present them as interfaces to the elements. The two main resources that require arbitration on the IXP1200 are control status registers and the transmit FIFO.

There are two main usage modes of NP-Click: implementing elements and implementing the application. If the elements required for an application are not in the library, the user must implement them. Although we implement NP-Click elements in IXP-C, they are much simpler to write. In most cases, implementing an element is easy because the inputs and outputs are defined, access to shared resources is via interfaces, and the assignment of data to memory is deferred. This focuses the programmers effort and attention on writing the code that implements an elements function.

After writing all the elements required for the application, the programmer now focuses on assembling the elements to describe the application functionality. He can then provide additional information on mapping to the architecture. This decomposes the architectural model into two: one is highly visible for writing the small sequential blocks in Microengine C, while the other abstracts away features so that the mapping

problem can be effectively considered. There is still a significant burden on the programmer to manually map the application, but the burden is lighter and more focused on the key features of the architectures than low-level commercial approaches. It was from these insights that we developed the framework presented in this thesis.

3.3.5 Differences to this approach

The approach presented in this work fills a void left by the existing solutions in this space. First, it leverages a popular platform independent domain specific language (Click) as its starting point. Click has gained popularity with only general purpose platforms as its implementation vehicle. With a vast set of tools available for networking designers on general purpose platforms, Click's popularity implies that it has intrinsic appeal to application designers. Second, it uses our own experience with a set of popular commercial network processors to develop application and architectural models that are accurate and yet amenable to automated mapping. The NP-Click work gave us insight into the value of having a library of elements that is written specifically for the target processing elements. The mapping problem itself is built on a general purpose constraint based framework which can produce optimal results. It may be tuned to trade-off solution time and how close to optimal it is guaranteed to be. Finally, application concurrency extraction may be done directly and automatically from a domain specific description of the application. The structural information coupled with communication and concurrency semantics inherent in a domain specific

language such as Click exposes a good deal of parallelism. It is with these differences in mind that mapping and application level transformations are considered in detail in the following chapters.

3.4 Summary

Sequential languages (like C, C++, Pascal, Fortran, Java, etc.) have served as a valuable role of being the *de facto* program entry environment for most applications. While many other programmable tools have assisted the process of programming (make files, UML, compiler optimizations), the vast majority of a sequential programmer's time is spent writing and debugging code in the common sequential language. With the rise in single chip multiprocessors and the issues of parallelism, this domination is waning. In addition to sequential set of criteria, programming environments must also be considered for their ability to utilize parallelism productively. In this section we cover parameters of programming approaches that defines how designers will utilize parallelism in their application and on the architecture.

While parallelism extraction, architectural modeling, and mapping the application to the architecture are the most important features to crossing the implementation gap, choosing a programming environment for implementing a particular application on a network processor will involve a variety of other considerations which are summarized in Table 3.1. The programming environments emphasize either productivity, efficiency, portability, or correctness as discussed in Section 1.2.1. Traditionally, net-

Table 3.1: Design environment characterization

Approach	Emphasis	Concurrency Model	Partitioning	Scheduling	Retargetability	Guarantees
Assembler	Performance	Explicit	Manual	Manual	None	None
C Variants	Performance	Explicit	Manual	Manual	None	None
SMP Click (Dynamic)	Productivity Portability	Push/Pull	Implicit	Dynamic	Linux SMPs	None
SMP Click (Static)	Productivity Performance	Push/Pull	Implicit	Manual	Linux SMPs	None
TejaNP	Productivity	State Machine	Manual	Manual	NPs, FPGAs	None
PacLang	Productivity	Abstracted	Manual	Manual	IXP2400 only	None
Shangri-La	Productivity Perfomance	Dataflow-like	Automated	Semi-Automated	IXP2xxx	None
Our Approach	Productivity Performance	Push/Pull	Automated	Automated	IXP1200 IXP2xxx	Mapping Optimality

work applications have valued performance above all other features, so many of the early network environments have emphasized that. With more complex applications and the scale of the problem increasing, design environments have started to focus on productivity.

Exposing concurrency as a construct to designers will determine much of the environments performance and productivity trade-off. Environments that require the designer to manage the parallelism of the architecture directly when expressing the application are capable of high performance since there is no restriction on the final implementation. But managing the concurrency is then a large part of the design process, increasing the time to implementation. At the other extreme environments which have no concurrency constructs rely on tools or runtime systems to exploit any parallelism in the target. The programming task is much easier as it reverts to the traditional, well known, sequential programming problem, but increasing performance is much harder. Programmers have little control over exposing or exploiting

more parallelism. A happy medium is to use an abstraction for concurrency which structures the kind of parallelism that the programmer can express. A concurrency abstraction can range from an API like OpenMPI [24] or OpenMP [2] to model of computation [44] like synchronous dataflow [45] or Kahn Process Networks [38]. Driven by the approach's emphasis, earlier, performance oriented environments required explicit management of the parallelism without structure, while recent approaches have employed various kinds of abstractions to achieve performance without placing an undue burden on the designer.

Another prominent feature of a design approach is the ability to provide guarantees regarding the final implementation. This could include temporal assertions to assure the designer that deadlines will be met or guarantees regarding deadlock or livelock. Providing guarantees typically comes at the cost of expressiveness or design time while not influencing performance. Consequently, little effort has been made to incorporate such features into network processor programming environments.

Essential to the design flow is what it can target. The ability to target multiple platforms gives designers flexibility to change the platform. Some environments target a purely theoretical architecture and not have a path to a real implementation. Most typical with network processors is to targets a particular commercial product or product family such as Microengine C. A step from this is targeting a set homogeneous architectures such as those that support OpenMPI. The most ideal would be the ability to target any heterogeneous multiprocessor architecture.

Table 3.2: Summary of how existing solutions solve the keys to crossing the implementation gap

Approach	**Architectural Model**	**Mapping**	**Concurrency Extraction**
Assembler	Completely visibility	Manual	Manual
C Variants	Completely visibility besides some instructions	Manual	Manual
SMP Click (Dynamic)	Completely abstracted	Dynamic	Automated
SMP Click (Static)	Multiple processing elements cache performance	Manual	Automated
TejaNP	Processors, memory, interconnect	Manual	Manual
PacLang	Processors	Scripted	Manual
Shangri-La	Processors, memory	Automated	Semi-Automated
Our Approach	Processors, memory, interconnect	Automated	Automated

The most important and pertinent to this work is the treatment of the three keys to crossing the implementation gap. Table 3.2 summaries how each of the related approaches handle the key problems to crossing the implementation gap: the model of the architecture used, how mapping is performed, and how concurrency needed by the architecture is extracted from the original application description. Besides the work presented here, only the Shangri-La framework utilizes tools to assist the designer with difficult decisions. Our work presents a flexible, powerful design flow which uses powerful solvers to arrive at optimal solutions to accurate models that we have constructed from our own experience.

Table 3.3 summaries the comparison with related work. Those approaches that start with a domain specific language and target an application specific multiprocessors are the most similar to this work. They tackle the same implementation gap. Some mapping approaches in the supercomputing realm have provably optimal or

Table 3.3: Related Work Summary

	DSL Starting Point	Application specific target	Automated mapping	Optimality Guarantee
Assembly		Most NPs		
C Variants		Some NPs		
Microblocks		IXP		
SMP Click	Click		Dynamic	
TejaNP		Various		
PacLang	PacLang	IXP2400		
NP-Click	Click	IXP1200		
Shangri-La	Baker	IXP2xxx	Semi-Automated	
Our Approach	Click	IXP	Automated	✓

close to optimal solutions, but such solutions have yet to port to the application specific multiprocessor world. ASMPs have unique architectural features that must be accounted for when modeling the architecture that complicate the mapping algorithms designed for supercomputing clusters. The work presented here incorporates these features *and* can provide optimal solutions to the mapping problem with respect to these models.

Chapter 4

Automated Mapping

Mapping an application to an architecture is a critical step in crossing the implementation gap. The problem space is large and irregular prompting a variety of solutions to the problem. We solve this problem by using a model of the architecture and application that exposes the information needed to arrive at an efficient mapping while abstracting away details of only minor utility. We formulate the mapping problem between these two models as an integer linear programming *(ILP) problem. ILP is an ideal vehicle for a solution as it is flexible and can leverage powerful general purpose solvers. Platform specific and user constraints can be readily incorporated into a core framework. The mapping problem proves to be an NP-complete problem.*

4.1 Motivation

Based on our own experience with application specific multiprocessors and corroborated by data collected for this work summarized in Section 6.4, the most performance critical mapping decisions are:

- allocating tasks to processing elements
- laying out data in memories
- assigning communication links to interconnect

As with traditional multiprocessor platforms, balancing the load of computation across processing elements has an impact on performance. As shown later in Section 6.3.2, applying a greedy method to resource constrained load balancing for a relatively small task graph targeting can result in a 19% worse solution to the optimal with respect to the model. With memory exposed architectures such as the IXP2xxx series, the choice of where to locate a particular piece of data also has an impact on performance. Locating a highly accessed piece of data in a faster, smaller memory can improve performance. Our results show that for loosely coupled task graphs (those in which most of the data that is shared between tasks are queues), memory aware task placement improves memory unaware task placement by 5% to 7%. The communication between tasks and data needs to also be mapped to interconnection architectural components. For example the IXP2xxx series, microengines are connected via next neighbor registers which enable low latency, high bandwidth

communication between adjacent microengines. Utilizing this architectural feature is a boon to the tasks communicating and also alleviates some of the contention on the shared memory that would also be used.

Each of these mapping decisions potential impacts the other. Task placement will decide whether a certain memory can be used for a shared datum. Conversely if a next neighbor register is tapped for a particular communication channel, the tasks writing to it or reading from it are restricted to certain processing elements. As all of these decisions are critical for performance, an integrated mapping approach is needed that can consider these axes simultaneously. These intertwined design decisions makes for a large and irregular design space. Automation is the key to traversing the space effectively.

4.2 Mapping Framework

There are many possible approaches to solving the mapping problem. The ideal mapping engine is fast and accurate. A static mapping framework should account for worst case traffic loads without sacrificing the overall performance. It is capable of arriving quickly at mappings that are efficient. In practice determining a good static mapping will inevitably require a significant amount of design space exploration. The mapping engine should allow the designer to tune how much of the design space needs to be explored. In other words, the designers should be able to direct the tool to spend more time to find a solution closer to the optimal one. Flexibility is

another important feature of a mapping framework. The ability to incorporate new application requirements, designer feedback, and even new architectural features is part of a ideal mapping solutions. This will permit the mapping engine to be applied to a variety applications and platforms while still incorporating designer insight.

The mapping framework should be based on an optimization engine capable of many of these features. As the mapping problem is generally an NP-complete problem (which we shown in Section 4.7), most tools do not solve it exactly. Heuristics are often employed to exploit the structure of the problem to arrive at good solutions quickly in practice. The strongest of these are "approximation algorithms" which arrive at a solution that is provably close to an optimal one [19]. Randomized algorithms attempt to traverse the design space by incorporating randomness in the search. For example simulated annealing samples the design space and picks the most promising regions to search more extensively. As regions are successively narrowed, a random variable determines if the algorithm backtracks and tries a different region. Genetic algorithms produce a set of feasible solutions to the mapping problem. It selects the best candidates and randomly switches parts of their solution in an effort to create better ones. Some of the mapping approaches based on these are covered in Section 4.3.

These solutions have been used in various contexts successfully, but lack the flexibility and the efficiency guarantees of an ideal mapping solution. We base our mapping approach on *integer linear programming* (ILP). ILP is a flexible method for describing

a design space and the cost function to optimize for. Decades of research have gone into powerful engines that can either solve these problems optimally, or provide a guarantee that a given solution is within a percentage of the optimal. The following section overviews the basic ideas of ILP and the common solution techniques.

4.2.1 Integer Linear Programming

Linear programming is a constraint based method of describing optimization problems. The feasible region of the solution is defined by a set of variables of real numbers and the intersection of constraints which are linear with respect to these variables. Since each constraint must be linear, the feasible region forms a convex polytope (i.e. a convex, n-dimensional polygon). The objective function of a linear programming problem describes the cost function to optimize inside of the feasible region. Linear programs have the desirable property that if a optimal solution exists, it is on one of the edges or points of the polytope. Finding the optimal point is then a matter of traversing the surface of the polytope until the optimal point is found. This problem can be solved in polynomial time.

Integer linear programming (ILP) problems are linear programming problems where the range of the variables is restricted to the integer domain. While this reduces the number of feasible solutions, it significantly complicates the problem by making points not on the surface of polytope potential optimal solutions. The decision problem of this optimization is NP-complete. Exact solvers typically employ

branch and bound methodology to arrive at a solution. To bound the solution space, consider that the bounds of the feasible solution are by linear constraints. If we consider integer variables as having a range of real numbers, the resulting problem is the *LP-Relaxation* of the ILP. If the optimal solution of LP-relaxation problem happens to be integer, this is optimal solution of the original ILP problem. If not, it is a bound on what is the best possible integer solution. Such a bound is still no indication of where the optimal solution exists in the polytope. To examine the interior of the polytope, the region is divided along one of the variables. If there is a single division, the solution must exist in one of the these two regions. These subproblems are solved recursively until an optimal integer solution is found on the subproblem. The best integer solution is recorded and used to prune other subproblems. By using the bounds generated by solving the LP relaxation, branches of subproblems can be eliminated without finding integer solutions.

Many software solutions exist that are geared to solve large instances of ILP problems quickly [8]. They employ a variety of solution techniques for deciding what variables to branch on, how to decompose the problem for solving smaller subproblems, and learning based on previous results. These are employed in solvers such as Mosek [5], CPLEX [1], and Xpress-NP [3].

4.2.2 Pseudo-Boolean Problems

Pseudo-boolean problems (or 0-1 integer linear programming problems) impose a further restriction on the range of the variables. Solvers are specialized to efficiently traverse a design space defined by 0-1 variables. The similarity of these problems to boolean satisfiable (SAT) makes them amenable to utilizing well engineered SAT solvers. While pseudo-boolean problems still have linear constraints, SAT problems are described using only boolean reasoning. In other words, a SAT problem is described using only boolean ANDs, ORs or NOTs. Without addition or subtraction or multiplication by constants, SAT is unable to efficiently capture problems described by linear constraints. Recently, SAT solvers have been extended to handle such constraints [23] [15]. These solvers can rival the performance of general purpose ILP solvers on these classes of problems.

4.3 Prior Mapping Work

Existing related automated mapping approaches are in somewhat overlapping classes. The first are existing frameworks for arriving at mappings that target multiprocessor platforms. These employ a variety of the methodologies discussed in Section 4.2. The second class of related work are those ILP approaches that solve similar problems to the one tackled in this work. The following section touches on many of these approaches and discusses the differences to this work.

4.3.1 Existing Multiprocessor Mapping Approaches

Mapping applications to multiprocessors has been examined before in the field of networking. Shangri-La [27] proposes an alternate flow from a DSL called Baker to the IXP2xxx series. Applications are partitioned and annotated with mapping information which is used by a runtime system to assign them to processing elements. To ensure packet delay guarantees are met, Kokku, et al. present an algorithm can be used for allocating processors to computational stages [41]. Scheduling mechanisms have also been examined specifically for SMP Click [18] [13]. Static assignment in an SMP system increases performance by increasing the amount of shared data locality on a given processor. Srinivasan et al. consider the scheduling problem for the Intel IXP1200 and present a theoretical framework in order to provide service guarantees to applications [62]. However, they do not consider practical resource constraints of the target architecture, nor do they test their methodology with real network applications. In contrast, our approach provides an efficient solution to the mapping problem, explicitly taking into account resource constraints of the multiprocessor.

Looking more broadly, the problem of task allocation and scheduling has be considered for decades by the scientific computing and supercomputing community [58] and general multiprocessor models [17][60]. With careful consideration of the problem structure, they created a variety of heuristics and approximations schemes that yield high-performance solutions when mapping to specific platforms. These solutions came at the expense of portability to other domains including application specific multi-

processors. The unique resource constraints of the embedded multiprocessing world limits the applicability of these approaches. However, there are many lessons to be learned from this community. For example, programming at the task level tends to lead to portability issues and makes race conditions difficult to handle. Conversely, parallel languages tend to remove programmer too far from task level considering multiprocessors are motivated by performance.

4.3.2 Existing Integer Linear Programming Approaches

ILP has proven to be a robust solution method to tackling mapping applications to embedded multiprocessors. The advantage of ILP is the natural flexibility to express diverse constraints and its potential to compute optimal solutions with reference to the problem model. Yang et al. have developed an ILP framework to minimize the use architectural resources considering computation allocation, targeting the IXP2400 [43]. In work done here at Berkeley, Jin, et al. consider the mapping problem of tasks to a soft multiprocessor instantiated on an FPGA [37]. Eisenring, et al. present a CoFrame, a modular and flexible framework for exploring architectural design spaces as well as mapping by using task graphs consisting of communication and computation [50]. Bender develop an ILP formulation [10] for task allocation schemes for heterogeneous multiprocessor platforms. The formulation arrives at a mapping of application tasks to hardware resources that optimizes a trade-off function between execution time, processor and communication costs. Wehmeyer, et al. also employ

ILP in a uniprocessor setup to optimize for memory placement of instructions and data across multiple exposed memories [69]. Methods for resource aware a compiler are presented by Palsberg and Naik, focusing on instruction scheduling and register allocation [56]. Hwang, et al. [30] presented an ILP model for resource-constrained scheduling and developed techniques to reduce the complexity of the constraint system.

We utilize ideas from many of these approaches when constructing our own framework. We use a similar set of applications as Yang [43] which also uses a task graph style of representing the application profiled with completion times. Their approach targets the IXP using an ILP framework, but does not consider the data to memory or topology of the architecture. Like Bender [10], we use an architectural model consisting of processing elements, memories, and interconnect. In a collaborative effort here at Berkeley with an ILP formulation targeting FPGA soft multiprocessors [37]. We developed core set of architectural constraints that are applicable across these two targets.

Missing from these works is an integrated consideration of task allocation and data layout onto multithreaded processing elements with a diverse memory hierarchy. This is the key design problem when traversing the implementation gap for an application specific multiprocessor. Our work examines this problem in the context of a complete design flow for ASMPs, not an incremental improvement to an existing one. Domain specific knowledge contains useful assumptions to construct models which can be

mapped quickly while still producing efficient implementations.

4.4 Application Model

For our application model we use a *task graph* that can be directly mapped to the architectural model. We define a task graph as a graph with nodes of computational blocks called *tasks* and the states which are read or written by them called *data*. A simple task graph depicting a two port forwarder is shown in Figure 4.1. Tasks are shown as white ovals while data are gray boxes. These nodes are connected via edges that represent communication links with directionality indicating reads and/or writes between nodes. A task graph may be mapped onto a corresponding architectural graph by covering tasks with processing elements, data with memories, and links with interconnect. For simplicity we require that connections occur only between tasks and data, not between data and data or between tasks and tasks. Any communication that is not rendezvous style communication between tasks will require some memory resource. As rendezvous is not a dominant form of communication in networking applications, a communication channel between two tasks in the application is modeled as datum with edges between the two tasks.

The restriction on communication leads to the following assumption: dependencies between tasks are assumed to be decoupled. In networking applications, communication between tasks is principally packets through queues. After a network device is well past its initialization, it is in "steady state". In a well balanced network ap-

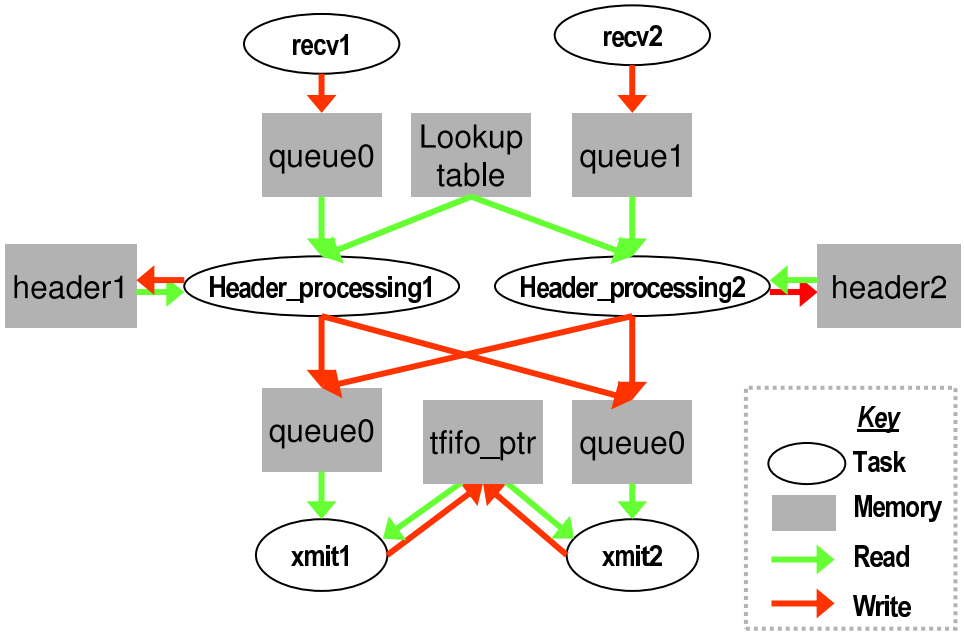

Figure 4.1: Task graph as an application model

plication that highly utilizes the platform, steady state implies most queues having some packets and every task running or ready to run. If the implementation is in steady state and tasks are waiting for packets to execute, then there are other bottlenecks which are limiting the performance of the system, not scheduling dependencies. We model steady state as periods of identical task invocations. Figure 4.2 shows an example of a system period visually. When running on hardware, tasks are usually allowed to run whenever the processor is idle, which would make Figure 4.2 have tasks running in during prior stage. Conversely, a task may occasionally have a longer execution time causing it to borrow from the next system period. Such effects can play a role in tuning the implementation for more performance, but they are difficult to capture using models used by automated tools. These second order effects are left to

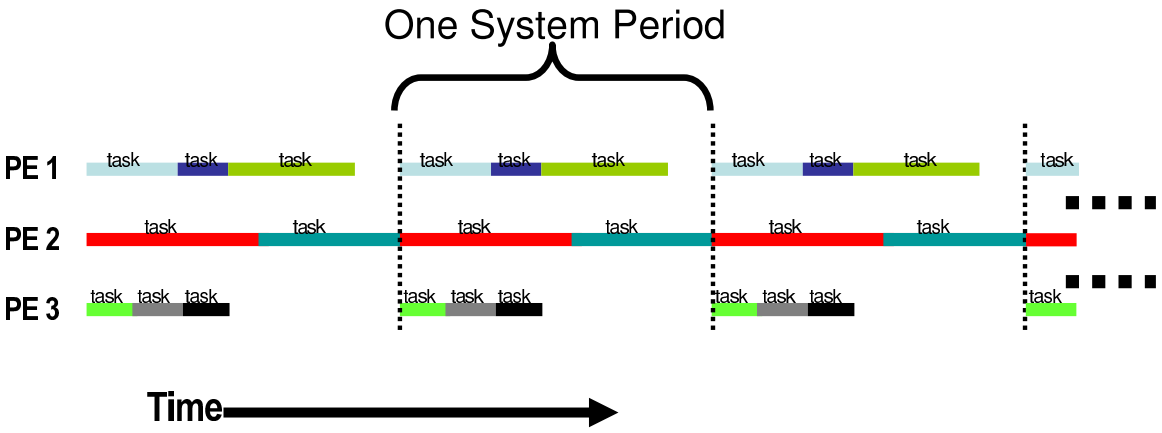

Figure 4.2: System period

the designer to express the implications to the mapping engine.

When considering the mapping of a task graph to an architecture, model of how much many resources it consumes and how often it consumes them is needed. For example from Figure 4.1, let the *header_processing1* task take 10 cycles to process a packet and *header_processing2* task take 5 cycles. Also, let 4 times as many packets ingress through port 2 than through port 1. *header_processing2* consumes more execution time in the implementation than the seemingly heavier weight task *header_processing2*. A model should factor in both rates of execution along with cycle times. To this end our execution model normalizes all task annotations to a single system execution period. In the preceding scenario if the system period was set to the arrival rate of packet to port one, the execution cycles consumed for *header_processing1* is 10 cycles and for *header_processing2* is 20 cycles.

The following section details the components of the task graph. The notation used

for describing the ILP problem will be slowly introduced in this section, but the full condensed description of the notation can be found in Section 4.6. The construction of the task graph from the application description is discussed in Section 5.3.

4.4.1 Tasks

In our application model, tasks represent loci of computation that may execute in parallel with respect to other tasks. The set of tasks T is all of the tasks present in a given application. Outgoing and incoming arcs denote communication with other parts of the application. Currently, the tasks do not have unique ports, but the model could be extended to incorporate them. The granularity of tasks is not fixed. An application may choose to represent computation in a finer or coarser granularity as need be.

In general a variety of proprieties may characterize a task. Typically the most important is the time needed to complete during one system period. This may be captured in a few ways. The most common are worst case execution time and average case execution time. For more accuracy on tasks with variable execution, the distribution of execution times can be captured and include variance, type of distribution, median value, etc. Patterns of execution time can also be useful. If every other invocation of a task takes half the time, designers may be able to utilize that when scheduling it. Patterns reflect changes in internal state given regular input, while a data dependent execution profile reflects how different data size, type, or

value affects the execution time. Execution time can be broken down into the cycles spent actively executing instructions and the cycles spent waiting for a long latency event (like a read from DRAM) to return. This breakdown can be especially useful for multithreaded processing elements common to many network processors that can swap in ready threads for waiting ones with little or no time overhead. Tasks may also be either preemptable or run to completion. Such a property has implications on how the code in task is written and how it is treated for performance.

For our modeling, we annotate a task $t \in T$ with the average execution cycles consumed e_t and the average time spent waiting for long latency events l_t given worst case input. Exposing the cycles consumed by active execution and the cycles spent waiting for latency events creates the potential to utilize hardware multithreading, a common application specific performance enhancement. The choice of focusing on the average case is motivated by the fact that worst case execution time in practice can be an order of magnitude greater than the average. Hosts on a network are resilient to the occasional corrupt or dropped packet, so a missed deadline that leads to a dropped packet is tolerable if it means higher total throughput for the majority of packets. Programmers design around this principle, optimizing for the average case and improving the worst case traffic profiles for their application.

Target Specific Modeling

Certain tasks characteristics can be exposed specifically fora particular target. For example, applications targeting the IXP1200 often benefit from the consideration of multiple implementations. The functionality of a task may be implemented in different ways. For example a lookup table task may be implemented with a larger route table to reduce the number of memory accesses. Considering multiple implementations proves to an important decision in the IXP1200. For all tasks T, there is a set of available implementations H. Since any given task $t \in T$ may only be able to use some $h \in H$, the viable implementation classes are described by the set $B \subseteq T \times H$. With this construction an implementation class h defines the execution cycles consumed e_h instead of a task. A task's e_t depends on the selection of an implementation class.

4.4.2 Data

Data nodes D in the task graph represent state that is needed for the execution of tasks. Data can be read from and written to, but does not perform any computation. Like tasks, data can be characterized in a variety of ways that is somewhat dependent on the architecture. The size of a datum can be measured in bits, bytes, or words. The actual space it will consume in the platform may depend on the tools ability to pack smaller variables into larger words. Otherwise one bit of state may use an entire word of space in memory. The other major characteristic of a datum is

the way it is accessed. A datum may be accessed by reading and/or writing it. The number of times of each during a given system period can be measured as the average or as the worst case. It can be modeled as different statistical functions (e.g. uniform or Gaussian) and be profiled for variance. As with execution time for tasks, accesses may exhibit patterns that can be exploited when mapping, including input dependent number of accesses between a task and a datum. Contention for a shared datum is another characteristic to model for data. Data with multiple writers must be protected with critical sections when modification requires more than a single atomic operation. A datum with many writers may leave many tasks often waiting for access, severely affecting system performance. These are often modeled with exponential growing cost with respect to the number of writers.

We model a datum $d \in D$ in terms of its size s_d measured in words as most compilers for network processors are unable consolidate smaller data into words. For a read from a datum $d \in D$ to a task $t \in T$, we capture the average number of read accesses given a worst case input load for a system period as a weight $w^R_{t,d}$. This can be visualized as a weight on the arch between the tasks and data. The average number of access under worst case load is appropriate for the same reasons that a task's execution time is well modeled as an average under worst case input. Conversely, a write from $t \in T$ to a datum $d \in D$ is modeled by a weight $w^W_{t,d}$ on a write arc between those two nodes. Read-modify-write access of data which cannot be described by a single atomic instruction are labeled in each task. These are inserted into the graph

as weighted read and write arcs with an extra weighting penalty. This construction is covered in detail later in Section 5.3.

4.4.3 Communication

Communication between tasks and data is represented as arcs in the task graph. Application communication can be principally characterized by bandwidth, directionality, and connectivity. Bandwidth is the amount data that is transfered in a certain amount of time. As with execution time and data access profiling, the bandwidth of a link can be characterized by average case or worst case. It may be modeled by a statistical function, a pattern, or data dependent profile. The communication is built using some protocol in the tasks, although it may be simple. The definition of the protocol can make the communication robust to data loss or to latency. The sensitivity to latency is protocol dependent. Many network applications will trade-off latency for throughput, but others like VOIP are sensitive to moderate network latency. Network designers in that case would opt for implementations with a facility to route low latency packets at the cost of throughput. Directionality and connectivity of a communication channel describe which application elements are communicating and in what direction.

We capture the directionality and the connectivity of communication links with edges between tasks and data in the task graph. The dominant form of communication between tasks within a networking application is with packets. Most network

protocols are robust to packet loss, so these links may be lossy. The actual sizing of the buffer for the communication link is application dependent since bigger buffers may result in fewer packets dropped, but also higher latency experienced. Therefore, we leave this issue to the application designer. From the application description, each communication link is modeled as an appropriately connected and sized datum in the task graph.

4.5 Architectural Model

Our architectural model is based on our own experience ASMPs in the networking area. We capture the architecture by modeling the platform as a directed graph in which the vertices represent processing elements and memories, and the edges represent available interconnect. Each processing element provides compute cycles while each memory has certain capacity for holding data and an average access time to write or read from it. To capture how computational blocks run on multithreaded cores, the model distinguishes between cycles used for execution and those spent with the processing element idle during long latency events such as memory accesses. Each local memory is connected solely to one microengine, while SRAM, DRAM, and scratchpad are connected to every microengine. This model maybe parameterized for each of the members in IXP2xxx network processor family by specifying number of cores and average access time and capacity of memories. The following section elucidates our model of each of these architectural components.

4.5.1 Processing elements

The set of processing elements P in our architectural graph are those nodes which support computation. Processing elements can differ in instruction sets, clock frequency, issue width, pipeline depth, etc. These factors coupled with the features of the compiler affect how fast a task can be executed and how much data memory it may consume. The time to access instruction memory and instruction store type (instruction cache vs. instruction store) are also key parameters to consider when mapping tasks to processing elements. Instruction store present programmers with hard limits as to the number of instructions and therefore the size of the tasks that can be mapped to an architecture.

The architectures focused on in this work have a uniform set of processing elements for data plane processing. Therefore, we assume uniformity in frequency and instruction set. Consideration for heterogeneous computing should be a straightforward extension to this approach. Execution time on a processing element is broken down into the number of cycles spent actively executing and the cycles waiting for long latency events to complete. To this end, we profile each task $t \in T$ on one of these uniform processing elements. We profile a task t for execution cycles consumed, e_t, and the idle cycles from latency events that cannot be altered by the mapping stage, known as fixed latency cycles, l_t. Tasks also experience latency accessing data assigned to memories by the mapping stage, called variable latency cycles, v_t. The completion time of a task t is then $e_t + l_t + v_t$.

Hardware multithreading allows for fast thread swaps that enables processing elements to stay utilized during a long latency events by swapping out the waiting task and swapping in a ready one. To model how tasks run on a multithreaded core, we consider a processing element to be in one of two modes:

- **Compute Bound** - Latency events are effectively masked by useful execution done by other tasks and task completion is limited by the speed of execution.
- **Latency Bound** - There are not enough execution cycles to mask latency events, so task completion is limited by the speed of the longest task.

To determine which of these two modes a given processing element is operating in, we calculate number of cycles needed to finish all the tasks on a processing element as the maximum of the sum of the execution cycles from all tasks assigned to that processing element and the total number of cycles needed to finish longest task, which is the sum of the execution cycles, fixed latency cycles, and the variable latency cycles. Consider the mapping of one task connected to two memory elements and one special purpose unit, shown in Figure 4.3. An average execution of task t involves a few blocks of sequential instructions interleaved with memory and special purpose hardware accesses, which we then model as e_t, l_t, and v_t as shown below the execution trace. Note that v_t will change if either d_1 or d_2 is assigned to a memory with a different access time.

Once tasks are allocated to processing elements, the total execution cycles con-

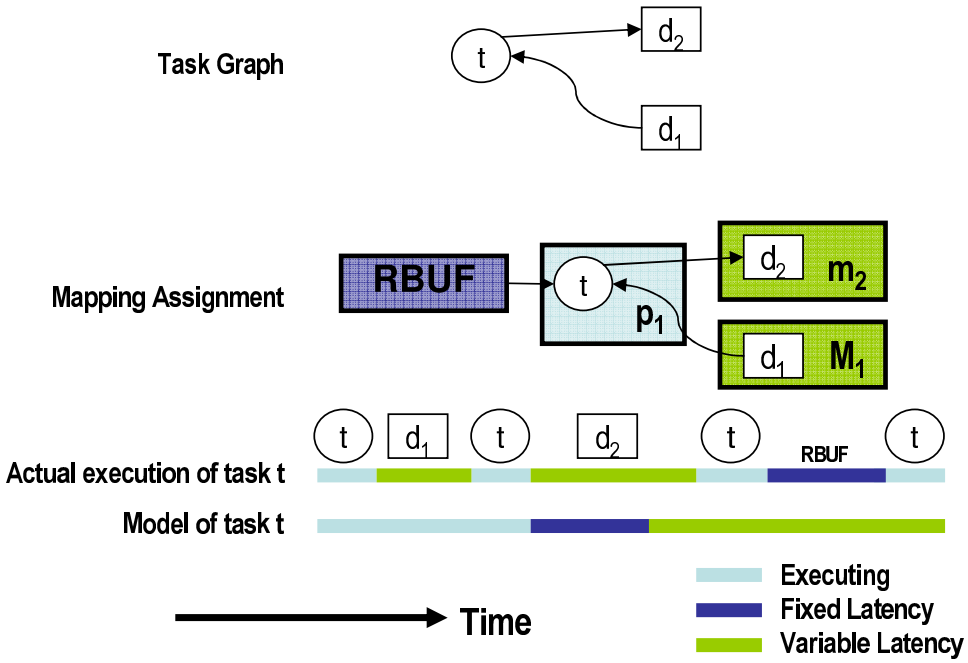

Figure 4.3: Our model of hardware multithreading

sumed and latency can be calculated. When a thread is actively running, it has exclusive control of the processing element. For a processing element $p \in P$ the total execution cycles consumed is $\sum_{t \in \pi_p} e_t$, where $\pi_p = \{t \in T | \text{task } t \text{ is assigned to processor } p\}$. Conversely, during long latency events, other tasks are able to actively run or wait for their own long latency events to complete. The total execution time needed to complete the tasks assigned to a processing element is at least $\max_{t \in \pi_p}(e_t + l_t + v_t)$. The total time needed to complete the execution of tasks is the maximum of the value of these two calculations. If the execution cycles higher, the processing element is acting in a compute bound mode. If there exists a task t such that $e_t + l_t + v_t$ is greater than the total execution time, the processing element is in a latency bound mode. An example

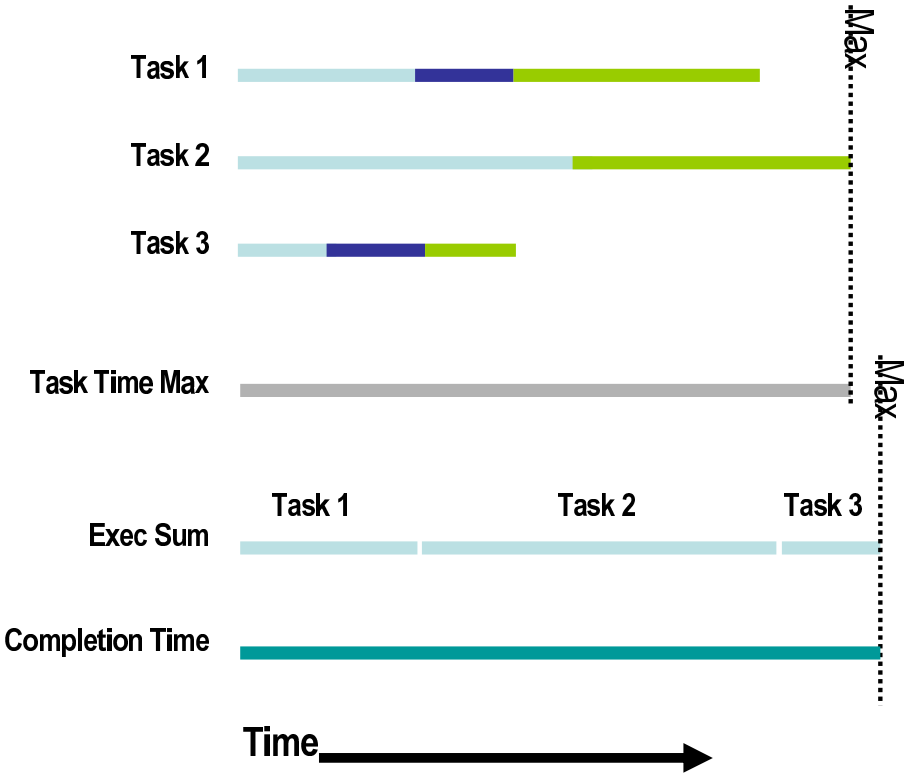

Figure 4.4: Our combining tasks on a multithreaded processing element

of how tasks are combined tasks is shown in Figure 4.4 where three tasks have been assigned to a single processing element. In this case, the execution cycles consumed sum is found to be greater than the completion time of any one task. Therefore the completion time is the former and is compute bound.

Processing elements in the network processor community often employ hardware multithreading which can provide significant performance benefits. An implementation class with state can only share instructions across the number of hardware threads n present on the processing element. Instructions for n sharing contexts using context relative addressing (see Section 2.3.1) can automatically distinguish between shared

states. Multiple tasks can share the i_h footprint. However task $n+1$ assigned to a single processing element cannot make use of this mechanism. It requires duplication of the instructions that refer to state to prevent improper state sharing. The instructions that are shareable only up to $n+1$ contexts are *quasi-shareable instructions* q_h. When $2n+1$ contexts are assigned, the instruction store must be tripled, and so on.

4.5.2 Memory

The set of memories M in the task graph are the nodes of the graph that can house data. Memory can be characterized by its capacity, which is the amount of data it can hold at any one time. The word size of a memory defines how densely data may be packed and what the minimum access size of the memory is. The access time itself is often measured to determined the cost of reading or writing data. Access time can be effected by the burst modes available which can offer performance advantages by accessing sequential memory locations. Access time can also be effected by the number of requesters. Contention models can be used for memories to capture the cost of accessing highly contented memories. Bandwidth of memory can also be a metric to consider when mapping data to it. Within a memory structure, designers often account for where data is located among memory banks. Bank accesses may be interleaved for higher utilization.

The architectures focused on in this work utilize custom distributed memories extensively. For a memory $m \in M$, we capture the capacity in words c_m. Since we

statically map an application to the architecture, all of the memory assigned to it must be less than c_m. We assume there is no mechanism for dynamically bringing data into and out of memory that is transparent to the mapping engine (i.e. memory cache). To access data housed in memory $m \in M$ these latency access k_m. Both writes and reads are modeled as symmetric access. The latency access cost is modeled as independent of the size of data. For shared memories, the number of cycles required to send the request for access and navigate the memory controller are the major contributors to a latency of a memory event, not the actual transfer of data.

For the IXP1200, memory resources are shared and have symmetric access from each of the microengines except for small register files. The register files are used for state for sequential blocks which leaves little room for communication variables. These variables instead must be located in globally shared memory. While the data placement in memory is still important, it can be considered separately from task allocation. The mapping problem can be visualized as the packing of separate tasks into unconnected processing element bins.

4.5.3 Interconnection

Topology of an architecture is defined by the interconnection of its architectural resources. Interconnection is the mechanism by with data is transmitted between these elements. Like memory, it is characterized by access time, which can be effected by features like bursting, contention, and connection protocol used. For our

model, interconnection is captured by topology. Access time penalties for interconnection are incorporated into memory latency costs (k_m) or fixed latency of a task (l_t). This is because for typical network processors there is little choice about the interconnection mechanism to be used when accessing a resource. Bursting and contention are considered second order effects and would significantly complicate the model if incorporated.

For the IXP1200, the topology of the architecture is somewhat trivial. The only memory local to a processing element is its register file. All other memory is global and has symmetric access latency from any processing element. Data placement in memory is still important, but can now be considered separately from task allocation.

4.6 Mapping Formulation

The following section formulates the mapping problem using the models described above. It can broken out into two sections: a core formulation, which uses a general model of network processors and a platform specific section, which incorporates those features of the architecture that are custom to it.

4.6.1 Core Formulation

To maximize for performance in the mapping step, we attempt to solve the following optimization problem: given a task graph and an architectural graph as previously

described, find a feasible mapping of tasks to processing elements, data to memories, and communication links to interconnect such that the maximum cycles needed by any processing element during an average period of tasks, called the *makespan*, is minimized. A summary of the constants are:

e_t	execution cycles consumed by task t
l_t	fixed latency cycles of task t
k_m	latency incurred by access memory m
s_d	space in memory consumed by data d in words
c_m	capacity of memory m in words
$w^R_{t,d}$	access weight of read from data d to task t
$w^W_{t,d}$	access weight of write from task t to data d
n_p	number of hardware threads processing element p

Variables in the formulation are in the form of a selection matrix, in which a one represents that an architectural element is covering an application element. More formally they are:

$X_{t,p} \in \{0,1\}$	$\forall\, t \in T, \forall\, p \in P$ task t assigned to processor p
$Y_{d,m} \in \{0,1\}$	$\forall\, d \in D, \forall\, m \in M$ data d assigned to memory m

$Z^R_{t,p,d,m} \in \{0,1\}$ $\quad \forall\, t \in T, \forall\, d \in D, \forall\, p \in P, \forall\, m \in M$
task t on processor p reads
from data d in memory m

$Z^W_{t,p,d,m} \in \{0,1\}$ $\quad \forall\, t \in T, \forall\, d \in D, \forall\, p \in P, \forall\, m \in M$
task t on processor p writes
to data d in memory m

$E_{cycle} \geq 0$ $\quad$ max execution time over
all processors (i.e. makespan)

These variables represent the universe of all possible mappings allowed by our model. We use the following constraints to confine the space to feasible solutions.

$$\sum_{p \in P} X_{t,p} = 1 \quad \forall\ t \in T \tag{4.1}$$

$$\sum_{m \in M} Y_{d,m} = 1 \quad \forall\ d \in D \tag{4.2}$$

$$\sum_{p \in P} \sum_{m \in M} Z^R_{t,p,d,m} = 1 \quad \forall\ d \in D, \quad t \in T \text{ s.t. } w^R_{t,d} > 0 \tag{4.3}$$

$$\sum_{p \in P} \sum_{m \in M} Z^W_{t,p,d,m} = 1q \quad \forall\ d \in D, \quad t \in T \text{ s.t. } w^W_{t,d} > 0 \tag{4.4}$$

$$\sum_{d \in D} s_d \cdot Y_{d,m} \leq c_m \quad \forall\, m \in M \tag{4.5}$$

$$\sum_{t \in T} e_t \cdot X_{t,p} \leq E_{cycle} \quad \forall\, p \in P \tag{4.6}$$

$$(e_t + l_t) \cdot X_{t,p} + \sum_{m \in M} \sum_{d \in D} (w^R_{t,d} Z^R_{t,p,d,m} + w^W_{t,d} Z^W_{t,p,d,m}) k_m \tag{4.7}$$
$$\leq E_{cycle} \quad \forall\, p \in P, \quad \forall\, t \in T$$

Constraints (4.1) and (4.2) ensures that each task and datum are covered by exactly one processor or memory, respectively. Constraints (4.3) and (4.4) enforces that each communication link in the task graph is covered by exactly one communication resource. Capacity constraints for each memory is described by constraint (4.5). The makespan by definition must be greater than or equal to the number of cycles consumed on each microengine in a given period, which is enforced by constraint (4.6). Constraint (4.7) forces the makespan to be greater than the completion time of any task. Note that the summation in this constraint captures the variable latency cycles v_t for task t. After we are constrained to feasible mappings, we minimize for makespan, E_{cycle}.

4.6.2 IXP1200 Specific Constraints

The biggest difference of the IXP1200 formulation from the core formulation is the omission of data and communication constraints. Since the architecture does not have a meaningful graph structure (see Sections 4.5.2 and 4.5.3)), capturing the application

as a graph and incorporating it into the mapping adds little value. Application model is better visualized as a set of *unconnected* tasks without dependencies or data.

If consideration of multiple implementations B of particular task t is important, this must be exploited in the mapping formulation. Since a task's execution and latency vary by implementation, a class H defines execution cycles consumed e_h and latency l_h. The set of implementations in an application is the product of classes and tasks or $B \subseteq T \times H$. B is the set of all (h, t) such that a class $h \in H$ exists as a potential option for implementation for a task $t \in T$.

A summary of the IXP1200 specific constants are:

e_h	execution cycles consumed by an implementation of class h
i_h	*shareable* instruction store footprint of an implementation of class h
q_h	*quasi-shareable* instruction store footprint of an implementation of class h
$\jmath_{limit}$	instruction limit for each processor

For the consideration of multiple possible implementations we add the following variables to our core formulation. As with the other variables they are in the form of boolean selection matrices.

$G_{h,t} \in \{0, 1\}$	$\forall\, h \in H, \forall\, t \in T$
	task t is using an implementation of class h

$$A_{h,p,k} \in \{0,1\} \qquad \forall\, h \in H, \forall\, p \in P, \forall\, k \in \{1, 1+n, 1+2n, \cdots 1+n \cdot \left\lceil \frac{|T|-n}{n} \right\rceil\},$$
at least k tasks using an implementation of class h on processor p

The following constraints that must be added to model the instruction store limitations of the IXP1200 architecture. A few of the core constraints must be modified slightly to account for consideration of multiple implementations of tasks.

$$\sum_{h \text{ s.t. } (h,t) \in B} G_{h,t} = 1 \quad \forall\, t \in T \tag{4.8}$$

$$\sum_{t \in T} \sum_{h \text{ s.t. } (h,t) \in B} e_h \cdot X_{t,p} \cdot G_{h,t} \leq E_{cycle} \quad \forall\, p \in P \tag{4.9}$$

$$A_{h,p,k} = 1 \Leftrightarrow \sum_{t \text{ s.t. } (h,t) \in B} G_{h,t} \cdot X_{t,p} \leq k \quad \forall\, h \in H, \tag{4.10}$$
$$\forall\, p \in P, \forall\, k \in \{1, 1+n_p, 1+2n_p, \cdots 1+n_p \cdot \left\lceil \frac{|T|-n_p}{n_p} \right\rceil\}$$

$$\sum_{h \in H} \sum_{t \text{ s.t. } (h,t) \in B} \sum_{k \in \{1,1+n,1+2n,\cdots 1+n \cdot \left\lceil \frac{|T|-n}{n} \right\rceil\}} q_h \cdot A_{h,p,k} + \tag{4.11}$$
$$\sum_{h \in H} \sum_{t \text{ s.t. } (h,t) \in B} i_h \cdot A_{h,p,1} \leq j_{limit} \quad \forall\, p \in P$$

In addition to assigning tasks to processing elements, an implementation must be selected for each tasks. Constraint (4.8) ensures that exactly one implementation is selected for each task. Since tasks execution time varies by implementation, core

constraint (4.6) must be modified slightly to include implementation selection as shown in constraint (4.9). Examples regarding how these constraints are applied are covered in the evaluation section including IP Forwarding in Section 6.3.1 and Diffserv in Section 6.3.2.

4.6.3 IXP2xxx Specific Constraints

Any feasible mapping should have the application's communication links covered by architectural communication resources. The following constraints describe the restrictions imposed by the topology of the IXP2xxx series. Microengine number j is denoted by ME_j, while its corresponding local memory is LM_j. The next neighbor register that Microengine j reads from is represented as NN_{j-1} while it writes to NN_j.

$$Z^R_{t,\mathrm{ME}_{j,d},\mathrm{LM}_k} = 0 \quad \forall d \in D, \forall t \in T, \tag{4.12}$$

$$\forall\, j \in \{1..|P|\},\, \forall\, k \in \{1..|P|\} \text{ where } j \neq k$$

$$Z^W_{t,\mathrm{ME}_{j,d},\mathrm{LM}_k} = 0 \quad \forall d \in D, \forall t \in T, \tag{4.13}$$

$$\forall\, j \in \{1..|P|\},\, \forall\, k \in \{1..|P|\} \text{ where } j \neq k$$

$$Z^R_{t,\mathrm{ME}_{j,d},\mathrm{NN}_k} = 0 \quad \forall d \in D, \forall t \in T, \tag{4.14}$$

$$\forall\, j \in \{2..|P|\}, \forall\, k \in \{1..|P|-1\} \text{ where } j \neq k+1$$

$$Z^W_{t,\mathrm{ME}_{j,d},\mathrm{NN}_k} = 0 \quad \forall d \in D, \forall t \in T, \tag{4.15}$$

$$\forall\, j \in \{1..|P|-1\},\, \forall\, k \in \{1..|P|-1\} \text{ where } j \neq k$$

$$\sum_{t \in T} X_{t,p} \leq n \quad \forall\, p \in P \tag{4.16}$$

Constraints (4.12) and (4.13) force data linked to a particular task to not be associated with a local memory not on the same microengine. Constraints (4.14) and (4.15) ensures that data being used as a producer-consumer link may only be assigned to a next neighbor register if the associated tasks exist on microengines before and after the register, respectively. Finally, we restrict the solver to finding solutions that do not require multiple tasks to be implemented by a single thread as described by constraint (4.16). Since an architecture like the IXP2800 has 128 hardware threads, most applications are not limited by the number of tasks. In fact in the construction of the task graphs, duplication of computational chains in the Click graph will be done to create more tasks to take advantage of extra thread. To utilize the hardware thread scheduler for swapping between tasks.

These constraints capture the basic features of the IXP2xxx necessary to arrive at high performance implementations, but we acknowledge that certain applications will require guidance from a designer to arrive at efficient or even feasible mappings. Such guidance could include binding a particular task to a certain microengine or ensure certain tasks appear on the same microengine together. As benefit of using ILP, this formulation can accommodate these or many other conditions like it without modification to the original constraints.

4.6.4 User Constraints

As with any model, some details which affect performance are inevitably omitted. While an ideal model exposes the important features of the general problem, certain instances of the problem may be greatly effected by detail not modeled. Designer insight has always been the key to solving this problem, and it must be harnessed for any performance oriented framework to succeed.

ILP is particularly adept at incorporating designer guidance, as the underlying engine assumes nothing about the structure of the problem. To guide the tool, a designer need only add a constraint with the rest of the problem indicating a partial solution or simply a restriction in the design space. The designer's only restriction to guiding the tool is that the constraint must be linear. This still permits a expressible facility for incorporating guidance. For example, if a designer had decided on t_1 and t_2 are to be assigned to processing element p_1, he adds the constraints:

$$X_{t_1,p_1} = 1 \tag{4.17}$$

$$X_{t_2,p_1} = 1 \tag{4.18}$$

Other constraints become trivially true and certain variables have become constants, but these do not conflict with the original set of constraints in anyway. For a more complex example, consider the case where a designer wants a task t_1 to be assigned to a processing element with at most 1 additional task to it. Let N be a sufficiently

large number:

$$N \cdot X_{t_1,p} + \sum_{t \in T \setminus \{t_1\}} X_{t,p} \leq N + 2 \quad \forall\, p \in P \tag{4.19}$$

4.7 Complexity Analysis

As discussed in Section 4.2 ILP provides enough expressiveness to capture architectural features such as multithreading, next neighbor registers, and quasi-sharable instructions, but is still able to solve the mapping problem exactly. In this section we show that the decision version of problem is in the complexity class of *NP-complete*. To this end we must prove two things: first that the problem is in NP and second that an NP-complete problem is polynomial-time reducible to this one.

Definition Let TASK-GRAPH-MAPPING as the problem formulated in Section 4.6. A task graph G_{tg} is given with nodes of tasks T and data D with directed edges that represent communication between the two. A architecture graph G_{arch} is also given with nodes of processing elements P and memories M with edges defining their interconnection. A solution to the mapping problem is finding a covering such that all $t \in T$ is covered by exactly one $p \in P$ and that all $d \in D$ is covered by exactly one $m \in M$ where the makespan E_{cycle} captured by equation (4.6) is not greater than a given number N. Furthermore, the constraints C in Section 4.6.1 must be satisfied for a feasible mapping.

Theorem 4.7.1 *TASK-GRAPH-MAPPING is NP-complete.*

Proof We must show that TASK-GRAPH-MAPPING is in NP and that an NP-complete problem is polynomial-time reducible to TASK-GRAPH-MAPPING.

Lemma 4.7.2 *TASK-GRAPH-MAPPING is in NP.*

Proof Given the inputs to the problem are the graphs G_{tg} and G_{arch} and the constraints C, a solution can be checked by evaluating each of the constraints once. The makespan E_{cycle} can be directly calculated from the properties of the graphs and the solution in question. This requires only $O(|G_{tg}| + |G_{arch}| + |C|)$ time which is polynomial in the size of the input.

We use bin packing as our known NP-complete problem that can be polynomial-time reduced to TASK-GRAPH-MAPPING.

Definition Let BIN-PACKING be the NP-complete bin packing decision problem. Given a set of items I and a set of bins B each with a capacity L, each item $i \in I$ must be placed in exactly one bin, where it will consume l_i units of capacity in the bin. The decision is whether all the items I can fit in the bins B.

Lemma 4.7.3 *BIN-PACKING is polynomial-time reducible to TASK-GRAPH-MAPPING.*

Proof We must find a reduction function f where a solution S_{bp} to a BIN-PACKING problem BP is valid if and only if $f(S_{bp})$ is a valid solution to a TASK-GRAPH-MAPPING problem TGM. We define f to evaluate S_{bp} using the following construction of TGM. A processing element $p \in P$ is created for each bin $b \in B$ (i.e. P is

isomorphic to B). A task $t \in T$ is constructed for each $i \in I$ where execution cycles consumed e_t is set to the value of i_l. The makespan bound N is set to the same value as L. Every $t \in T$ has no fixed latency cycles (i.e. $l_t = 0$). There is no data ($D = \{\}$) and no memory ($M = \{\}$). There are no edges in either G_{tg} or G_{arch}.

By this constructions bins are processing elements and items are tasks. Since there is no data and no fixed latency on any task, every processing element is operating in a compute bound mode. Therefore the makespan is determined by the sum of the e_ts of the tasks assigned to it. Furthermore since no data or communication requirements exist in the task graph, most of the rest of the constraints C are vacuously satisfied. If a set of items can fit into a set of bins in BP, then the same arrangement of tasks to processing elements must necessarily fit under a makespan of N. Conversely if a set of items has no way of fitting into the bins, there is no arrangement of tasks to processing elements that fits under a makespan of N. Thus BIN-PACKING is polynomial-time reducible to TASK-GRAPH-MAPPING.

Since Lemmas 4.7.3 and 4.7.2 hold, TASK-GRAPH-MAPPING must be an NP-complete problem ∎

Chapter 5

Application Transformations

Domain specific languages enable natural application capture but are also structural with uniform computation and communication semantics. These domain specific constructs can be leveraged to transform the original application description to a task graph representation. The application designer must augment the original application description and supply a library of elements annotated with profile information. With this augmented application description the task graph needed by the mapping engine can be created automatically using polynomial time algorithms. Higher performing implementations can be achieved by managing local data used across multiple Click elements. More throughput can also be achieved by exposing more of the application parallelism through retiming and replication of tasks. Manual techniques and heuristics may be used on the original application description to expose this additional parallelism.

5.1 Motivation

To facilitate an efficient mapping, a task graph exposes the computation, data, and communication of an application while abstracting away all other details. While this is effective representation for solving the mapping problem, the task graph is not the most natural way of capturing an application. To describe an application as a task graph, designers must consider their mental model of the application and extract the parallelism into concurrent tasks. The application semantics must be mapped faithfully to task graph semantics and the task graph annotated with profiling information. Designers must also take care to arrive at the appropriate granularity: one that will give freedom to the mapping engine while not burdening it with an unnecessarily large design space. Poor choices on any of these design decisions can mitigate the benefits of using a task graph and the architectural model proposed by this work. Describing an application at this level is better than at the low level languages that ship with many of these devices, but is still significantly removed from our ideal application entry environment.

By using domain specific languages application, designers enjoy high productivity through natural application capture. The structural nature of the description and restricted semantics also enables automatic task graph generation and application level optimizations. The semantics, granularity, and graph differ between the application representations, but algorithms and heuristics can traverse this remaining piece of the implementation gap. The following section covers first what must be added to

the original domain specific language description of the application to facilitate this automated translation. It describes the algorithms which consume this augmented description of the application and produce a task graph that may be efficiently mapped using the approach described in the previous chapter. We end this chapter with the manual techniques and heuristics used to find application level optimizations that are difficult to detect or perform with a low level application description.

5.2 Domain Specific Language Additions

While a domain specific language like Click is a great starting point for producing a structured, parallel representation of the application, a Click description of an application lacks information necessary to produce an efficient implementation. As the Click description is used to build the task graph (and later generate code), we make additions that mirror the constructs of the task graph. For example, Click elements are profiled for average execution cycles consumed and idle cycles during long latency events under worst case input load. This section covers the information added to the original application description which enables the transformation to task graphs. As an illustrative example for this section, we recall our simple two port forwarder shown in Figure 5.1.

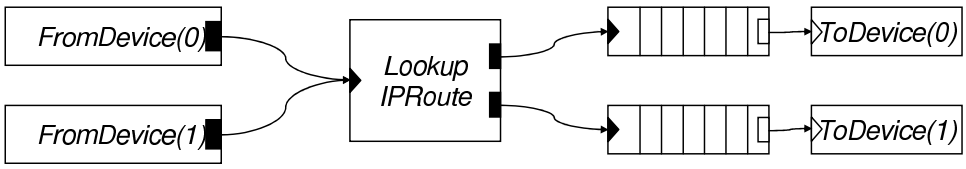

Figure 5.1: Simple two port forwarder

5.2.1 Packet Distribution

The percentage of packets that flow along each arch under a reasonable worst case load should be specified to accurately construct task graph edge weights and task profiles. Since only packets flow over arcs, each is annotated with the number of packets produced on a single execution or *firing* of its *controlling element*, which is the element connected to it that initiates the communication (for a push arc, the immediately upstream element and for a pull arc the immediately downstream element). The weight is relative to the controlling element and is independent of other arc percentages or element firing rates. Figure 5.2 shows an example annotation of the simple two port forwarder. The controlling element of the arc annotated with $0.75\frac{\text{packets}}{\text{firing}}$ is *LookupIP*. Input packets cause *LookupIP* to fire and 75% of them are forwarded along that arc. As with the task graph edges, we ignore temporary imbalances of packets between ports and elements with the assumption that the queues are sized to absorb such effects. By annotating the arcs of the Click graph, we attempt to capture the long-term packet distribution through the Click graph. It is on the onus of the application designer to specify a packet distribution that is

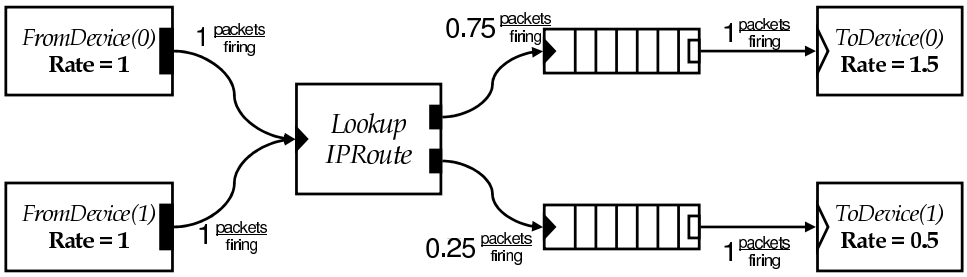

Figure 5.2: Simple 2 port forwarder described in Click

balanced.

Schedulable elements may operate at different rates with respect to a single system period. A designer fixes the logical duration of a system period and normalizes the firing rate of each schedulable element to it. In our example in Figure 5.2, the schedulable elements are the two *FromDevices* and the two *ToDevices*. The designer has chosen to normalize them to the rate at which packets are received on an ingressing port. By specifying that *ToDevice(0)* has a rate of 1.5, he expects it to transmit packets 50% faster than the input ports. Likewise port 1 transmits at half the rate of the ingressing packets.

5.2.2 Writing Library Elements

Programming in a domain specific language is the primary mechanism by which the programmer describes the application. However, there will always be new applications that will need elements of packet processing not contained in the existing library of Click elements. An application or library designer must specify the com-

putation of each of these elements. To promote uniformity and ease of programming, a template is used to describe an element. As with Click, the core packet processing is described in the *SimpleAction* function while a *Push* function captures the functionality if the element is a pushed a packet. As an example, we implement a static lookup element called *LookupIP* shown in Figure 5.3. *LookupIP* consumes a packet and performs a longest prefix match for the outgoing port of the packet. The packet is pushed to the element on the corresponding outgoing port of *LookupIP*. The lookup functionality of the element is in *LookupIPSimpleAction* while the push action is captured by *LookupIPPush*. Note that many local functions are omitted here for brevity. Keywords in this description will be used to construct the task graph and later to generate the code for the final implementation. The following section discusses the concepts and semantics of the keywords in this description.

The execution cycles consumed and the fixed latency cycles spent waiting for events should be specified with each implementation of an element. In *LookupIP*, they are specified in the first lines of *LookupIPSimpleAction*. To match their eventual use in a task graph, these two numerical values are found from running the element on an unloaded processing element using worst case traffic as input. Note that variable latency events are not included in this calculation, so *LookupIP*'s fixed latency does not include the access to the trie table since LookupTable may be placed according to the mapper.

Capturing the scope of variables is needed to properly create the data in the task

Function `LookIPSimpleAction`

Data: Packet descriptor p
Result: Returns next hop port number
; `/* Setup execution and latency meta data */`
ExecutionCycles = 100
LatencyCycles = 0
; `/* Describe computation */`
int NextHop
declaration (name=Header, type=PacketHeaderStruct, scope=local, size=12)
declaration (name=LookupTable, type=PacketHeaderStruct, scope=regional, size=10000)
load {**ExecutionCycles** = 0, **LatencyCycles** = 60} **begin**
    Header ← `LoadHeader(`*p*`)`
end
read (name=LookupTable, Number of times = 4)
read (name=Header, Number of times = 5)
write (name=Header, Number of times = 2)
NextHop ← `TrieTableLookup(`*Header.DestinationIPAddress, LookupTable*`)`
return NextHop;

Function `LookupIPPush`

Data: Packet descriptor p
Result: Passes packet descriptor to another function created by the code generator
int NextHop
NextHop ← `LookupIPSimpleAction(`*p*`)`
switch *NextHop* **do**
    generateOutputCase
end

Figure 5.3: *LookupIP* Element Description

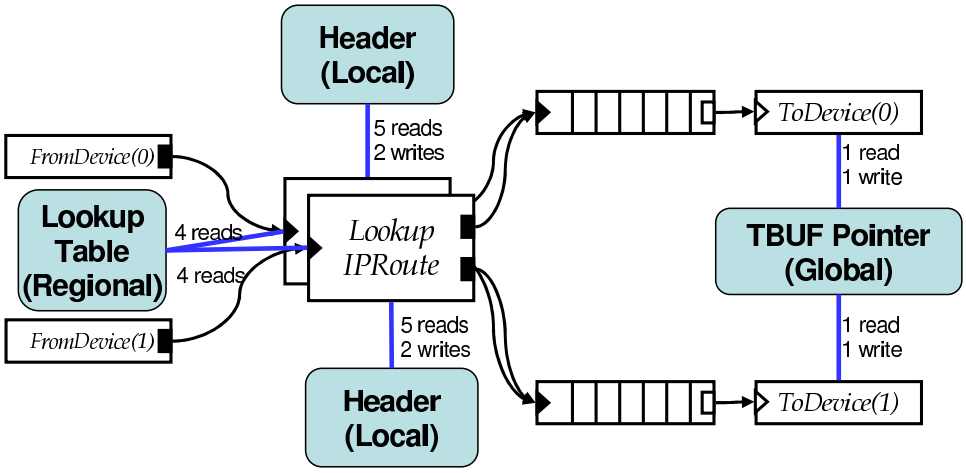

Figure 5.4: Examples of data scoping

graph. With typical parallel programming languages, variables exist in two types: local (can only be accessed by one thread) or shared (can be accessed by any thread). While the mechanisms for creating and sharing the variables varies widely, these two types are ubiquitous. Programmers are expected to map the topology of their application to these two types, forcing variables that are shared in a very restricted way to the same shared construct as a variable shared by all execution contexts.

In our experience, networking applications have other kinds of scoping that data employ. We create the following constructs to be used in the target specific library element descriptions for the different kinds of scoping used in networking:

- **Local** – Local variables are local to a particular instance of an element. These are usually temporary variables created for each new packet. *LookupIP* expects exclusive access to this copy of the header of the packet it is processing. When instantiated in the simple forwarder, a new header datum is constructed for

each instance of *LookupIP* as shown in Figure 5.4.

- **Regional** – Regional variables are shared between multiple instances of a single element. The computation of an elements may be duplicated by virtue of the fact that packets from different ports may be processed independently in any order. Regional data is common to all of these duplicated instances, but is still separate from other elements in the original application graph. *LookupIP* has a LookupTable datum which is shared among all instances of *LookupIP* as shown in Figure 5.4.

- **Global** – Global variables are shared among all elements of a certain type. This shared state allows for a set of elements of a single type to synchronize and share information. In our forwarder example, the *ToDevices* need a variable to coordinate their access to the shared buffer used for transmitting packets. As shown in Figure 5.4, the TBUF datum is a global variable to be used by all of the elements of the *ToDevice* type.

- **Universal** – Universal variables may be used by any element in the application. This special class of data can include packet buffers so that any element can create or destroy a packet.

After deciding on the scope of each of the variables in an element, the element writer places tags into the code of the element. These tags expose the memory accesses, sizes, and scope of each data to be seen by the mapper. The specific location

of certain tags also guides the code generator so that it may relate the result of the mapping assignment to the sequential compiler. Tags for variables are as followed:

- **Declaration** – Declarations of variables must be visible to create the appropriate data in the task graph. These tags include the variable's name, scope, and size as shown in Figure 5.3 for *Header* and *LookupTable*. The code generator also uses these tags to indicate the mapping assignments to the compiler.

- **Read** – A read tag indicates that the data is being read. This adds a read link entry into the library element or adds to the weight of an existing one.

- **Write** – Indicates that the data is being written. This adds a write link entry into the library element or adds to the weight of an existing one.

- **Load** – Certain variables must be initialized before they can be used. Such initialization is usually associated with local variables which must be reloaded or reinitialized with each new packet processed by the element. Loads are annotated with latency and execution cycles which add to the fixed latency and execution of the tasks that cover them. *Header* is a local variable that loads the words of a packet associated with the IP header from main memory. In a naive set up, this tag is not needed but it useful when optimizing local variables across elements which is discussed later in Section 5.4.1.

- **Save** – If variables that have **load** tags are modified during the execution of an element, it is usually appropriate to write that information back. Saves are

annotated with latency and execution cycles as well.

- **Exclusive** – While capturing all types of critical sections and synchronization mechanisms is impractical, it is beneficial to capture a commonly used type in networking. Exclusive access to a variable is used often in networking to coordinate concurrently executing tasks. By surrounding a body of code with an **exclusive** label for a variable, the code generator can insert a critical section which grants it exclusive access. The cost of accessing such a variable is best modeled by an exponential cost with respect to the number of simultaneous accessors. However in ILP contention modeling with exponentials (even with a coarse piecewise linear model) greatly hinders the performance of the solver. We choose instead to add edges with an additional fixed cost to the weight of an edge. This captures the simple observation that such accesses are more costly than non-exclusive reads and writes without forbidding variables from being shared in slower, global memory.

5.3 Task Graph Generation

Even with the additions to the domain specific language discussed in the previous section, an application description in Click is still not a application model that can be mapped to the architecture. The push-pull communication semantics of Click are more complex than that of the task graph's decoupled communication. Click

communication implies dependence between the execution of elements. Modeling this dependence would complicate the mapping formulation and increase the time to solution. Besides a communication semantic mismatch, the computation semantics of Click do not match with task graph semantics. In certain situations, multiple packets may be simultaneously processed in a single Click element while tasks typically process a single packet per system period. The granularity of the Click graph can also be problematic as designs often have many small elements. Therefore a transformation from an augmented Click graph should construct a task graph that respects the original computation and communication semantics while extracting parallelism that may be exploited by the architecture.

More specifically, Click provides the following construction for concurrency (first described in Section 2.5.1):

- Packets on different execution contexts may operate in *parallel*. Packets along different paths may be processed in any order or in parallel, provided the execution contexts running fire at the appropriate rate. Even a single element may process multiple packets in parallel, if they originate from different schedulable elements.

- Packets on a single execution context are guaranteed to stay *in-order*. On any given path of the Click graph, a packet is guaranteed not to pass or to be passed by another packet on the same path. This can be particularly important in protocols sensitive to ordering like TCP, in which many out of order packets

cause retransmissions and lower effective bandwidth.

We use these two assertions of Click to structure our task graph generation. First, parallelism is readily extracted by creating a tasks for each push chain or pull chain that begins with a schedulable element (see Section 2.5.1 for further explanation). Non-schedulable elements on the push or pull chains that originate from the schedulable element may be covered by more than one task, which replicates the computation of the element. To ensure that the in-order guarantee is met, each schedulable element path (i.e. each task) is executed by a single execution context. It cannot begin processing a new packet until it finishes with the current one. The execution contexts are free to operate concurrently thus exploiting this parallelism inherent in the structure of Click. Each task is constructed such that it is bounded by queues. We assume that in average operating conditions, internal queues will be non-empty, effectively decoupling any dependences between connected tasks. In an average system period, each task is able to fire and produces output such that all tasks my fire on the subsequent round as well. This construction is able to transform the push/pull semantics of Click to homogeneous dataflow of task graphs. It also transforms the in-order, path based parallelism to concurrent independent tasks.

In our simple forwarder, this construction creates four tasks shown in Figure 5.5. Each of the four schedulable elements (*FromDevice(0)*, *FromDevice(1)*, *ToDevice(0)*, *ToDevice(1)*) becomes a task. Since *LookupIP* is shared among two schedulable element push chains, this computation is duplicated across two tasks. All of the push

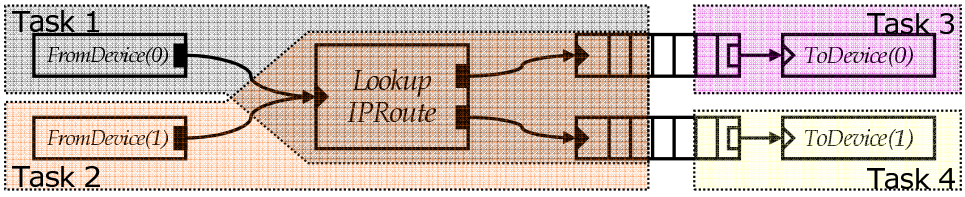

Figure 5.5: Covering Click elements with tasks

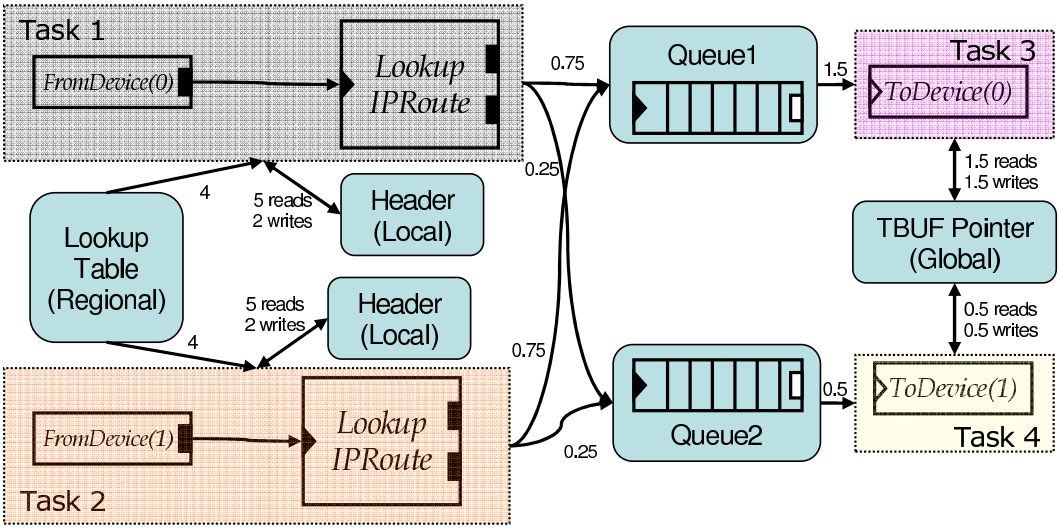

Figure 5.6: Task graph generated from a simple forwarder

and pull chains are implemented by tasks that may execute concurrently.

We glean data scope and size information for each of these tasks from the target specific library entries for Click elements covered by the task. Applying this methodology to our simple forwarder including the memory structures described in Figure 5.4, the result is shown in Figure 5.6.

Task parameters are calculated by starting with the firing rate of the covered schedulable element to determine how many times in a single system period a chain

executes. Since profiling is done on a per invocation basis for each Click element, this information must be normalized to account for different firing rates and imbalanced packet distribution. The execution cycles consumed by each element in a task is the product of the per invocation execution cycles consumed by the element, the relative edge weights of the path(s) leading to it, and the firing rate of the schedulable element. The execution cycles consumed by the task is then the sum of the elements it covers. The actual calculation is covered in detail in Section 5.3.3. This technique of accounting for different execution rates and unbalanced packet distribution is also applied to the fixed latency of a task, its outgoing and incoming edge weights, and its memory accesses.

Figure 5.6 shows the absolute edge weights calculated for each link in the task graph using this technique. The system period has been chosen to match the rate of the *FromDevice* elements. Each *FromDevice* is invoked once per system period and each copy of *LookupIPRoute* is also invoked once. The memory attached to them is accessed the same number of times by the tasks as by the original library entries. *ToDevice(0)* is invoked 1.5 times on average during a system period. This execution rate is multiplied by the relative arc weight of $1\frac{\text{packets}}{\text{firing}}$ to result in a task graph edge of weight 1.5. Similarly the library entry indicating a single read and a single write to the TBUF Pointer is modified by the firing rate giving a task graph edge weight of 1.5 reads and 1.5 writes.

Further parallelism maybe realized by pipelining a design with additional queues

between elements; however this may result in unintended latency through the application. We defer the discussions on optimizing Click graphs to later in this chapter. The following subsections build up the notation for an augmented Click graph and the generated task graph. The algorithms and their run times are then discussed.

5.3.1 Click Graph Notation

Before describing the algorithms which performs these applications, we must first define notation for describing the augmented Click graphs which will be the input to these algorithms. Consider the following definitions:

- Γ_{click} is the set of types as defined by the Click library such as *FromDevice* or *DecIPTTL*.
- Tuple $G_{\text{click}} = (V_{\text{cg}}, E_{\text{cg}})$ defines a Click graph.
- $\{V_{\text{cg}}, E_{\text{cg}}\}$ is a weighted directed graph with vertex set V_{cg} and edges E_{cg}.
- A vertex $v \in V_{\text{cg}}$ represents one instance of a Click element. An element $v \in V_{\text{cg}}$ is a tuple of parameters which define the element. v is defined by the tuple $(v_i, v_\tau, v_s, v_r, v_e, v_l) \in V$ where $V \subseteq \mathbb{Z} \times \Gamma_{\text{click}} \times \{\text{schedulable, non-schedulable}\} \times \mathbb{R}_+ \times \mathbb{R}_+ \times \mathbb{R}_+$. v_i is a unique identifier, v_τ is the type of element, v_s indicates whether it is a schedulable element, if it is schedulable v_r indicates its rate of execution, and v_e and v_l refer to its execution cycles consumed and fixed latency. Consider each of the components separately:

- $v_i \in \mathbb{Z}$ is the component of v that uniquely identifies v in the application.

- $v_\tau \in \Gamma_{\text{click}}$ component of v is the Click type of v such as *FromDevice* or *DecIPTTL*.

- $v_s \in \{\text{schedulable}, \text{non-schedulable}\}$ is the component of v that indicates whether an element is schedulable or not. Schedulable elements can initiate packet transfers without external invocation. This property is determined by the Click library.

- $v_r \in \mathbb{R}_+$ is the component of v that represents the execution rate of v. v_r indicates the number of times v executes in a system period. This rate is only applicable to schedulable elements as non-schedulable elements execution rates are determined by the schedulable elements that share a push chain or pull chain with them. For static scheduling, we again assume that dynamic effects may be reasonably modeled as a constant periodic invocation. For non-schedulable elements v, v_r is zero ($\forall v \in \{v' | v' \in V_{\text{cg}}, v'_s = \text{non-schedulable}\}.v_r = 0$).

- The edges $e \in E_{\text{cg}}$ model the dominant communication between the *Click* element instances. Only packets are transferred along the edges. e is a tuple $(e_{src}, e_{dst}, e_i, e_p, e_w) \in E$ where $E \subseteq V_{\text{cg}} \times V_{\text{cg}} \times \mathbb{Z} \times \{push, pull\} \times \mathbb{R}_+$. e is a directed edge, so e_{src} is the source node of e, e_{dst} is the sink node, e_i is the unique identifier, e_p indicates whether it is a push or pull edge, and e_w is the weight assigned to the edge.

- $e_i \in \mathbb{Z}$ uniquely identifies the edge e in the application.
- $e_p \in \{\text{push, pull}\}$ indicates whether the packet transfer along the edge $e = (v, v', e_i, e_p, e_w)$ is initiated from the sending element v (*push*) or from the receiving element v' (*pull*). From this property, we can define an *active edge*: The edge $e = (v, v', e_i, e_p, e_w)$ is an active edge of vertex v if it is a *push* edge and it is an active edge of v' if it is a *pull* edge.
- $e_w \in \mathbb{R}_+$ is the edge weight which represents the communication frequency for the edge e. This value represents the number of packets that traverse the link on average per invocation of the edge's active element. For a push edge this is the source node, while for a pull edge this is the destination node.

For each *Click* graph an activity graph $G_{\text{activity}} = \{V_{\text{cg}}, E^a\}$ can be constructed by inverting the direction of *pull* edges:

$$\text{Let } -e = (e_{src}, v_{dst}, e_i, e_p, e_w) \quad \forall\, e = (e_{src}, v_{dst}, e_i, e_p, e_w)$$

$$E^a = \{-e | e \in E_{\text{cg}}, e_p = \text{pull}\} \cup \{e | e \in E_{\text{cg}}, e_p = \text{push}\}$$

Figure 5.7 shows the activity graph for the simple forwarder. Schedulable elements are always the sources of this directed graph while queues are always the sinks.

5.3.2 Task Graph Notation

For the purposes of describing the algorithms of task graph generation, we condense and slightly modify the task graph notation from its original formulation in

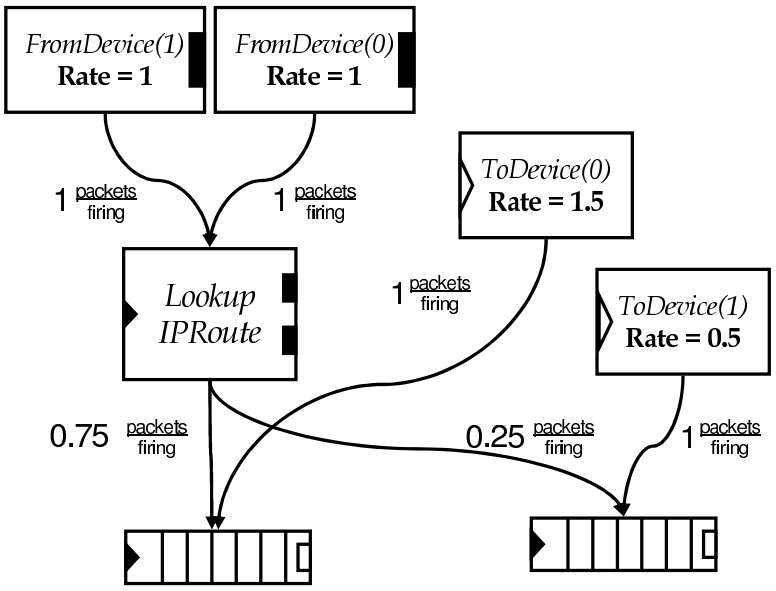

Figure 5.7: Activity graph of the simple forwarder

Section 4.6. Note that while the notation changes slightly, the task graph in the ILP formulation is directly derivable from this form as the following notation only adds information. First we introduce two new sets to aid the transformation to a task graph:

- Γ_{data} is the set of types for data elements. Types of task graph data include *Header, LookupTable,* and *TBUF Pointer* and are not to be confused with the traditional computer science definition of data types.

- S_{data} is the set of data scopes discussed in Section 5.2.2.

  $S_{\text{data}} = \{\text{LOCAL, REGIONAL, GLOBAL, UNIVERSAL}\}$.

As stated earlier, a task graph G_{tg} is a weighted directed graph. It can be captured by the tuple $(T_{\text{tg}}, D_{\text{tg}}, E_{\text{tg}})$. The components of this tuple are:

- T_{tg} is the set of tasks in the task graph. Each task $t \in T_{\text{tg}}$ is captured by the tuple $(t_i, t_e, t_l) \in \mathbb{Z} \times \mathbb{R}_+ \times \mathbb{R}_+$, where t_i is the task identifier, t_e is the execution cycles consumed, and t_l is the fixed latency in cycles.

- D_{tg} – the set of data. Data is described by the tuple $(d_i, d_s, d_z, d_\tau) \in D_{\text{tg}}$ where $D_{\text{tg}} \subseteq \mathbb{Z} \times S_{\text{data}} \times \mathbb{Z} \times \Gamma_{\text{data}}$. d_i is the datum's unique identifier, d_s is the scope, d_z is the size, and d_τ is the type.

- E_{tg} – the set of edges between these two sets. We define an edge as either a read tuple $(e_d, e_t, e_w) \in D_{\text{tg}} \times T_{\text{tg}} \times \mathbb{R}_+$ or a write tuple $(e_t, e_d, e_w) \in T_{\text{tg}} \times D_{\text{tg}} \times \mathbb{R}_+$ (i.e. $E_{\text{tg}} \subseteq T_{\text{tg}} \times D_{\text{tg}} \times \mathbb{R}_+ \cup D_{\text{tg}} \times T_{\text{tg}} \times \mathbb{R}_+$. Therefore tasks can only connect to data and data can only connect to tasks. This eliminates situations such as two data communicating which is not well defined or two task communicating without a buffer between them.

5.3.3 Algorithms

A few key algorithms are utilized to transform the domain specific language description to a task graph. The following section covers these algorithms using the notation from above.

Partial Execution Rate Calculation

The goal of the transformation algorithms is to cover all Click elements by at least one task. The transformation must instantiate data as indicated by the target

specific library of elements and convert the per invocation profiling information to absolute (with respect to cycles) task parameters. To solve this, we first calculate the invocation rates of each Click element with respect to a schedulable element. We call this value the *partial execution rate* ($\psi^v_{v'}$) of an element v' with respect to a schedulable element v. Our algorithm is presented in Figure 5.8 which takes a Click graph and one schedulable element as input and returns the set of partial execution rates with respect to that schedulable element. After initializing the set of returned values, the activity graph of the input Click graph is constructed and then topologically sorted. This gives a complete ordering of the elements such that all elements upstream to a given element appear before it in the sorted array. Each element of the Click graph is then processed in this order. The partial execution rate of an element is the sum of the partial execution rates (ψ^v) of its activity graph direct predecessors (V^{dp}) multiplied by the number of packets sent across the edge that connects them (e_w).

The partial execution rates algorithm relies on a topological sort of the directed application graph. In Click it is perfectly legal to have a cycle and in some cases it is quite useful. Consider the example shown in Figure 5.9. For those ingressing packets that have their time-to-live decremented to zero (i.e. those packets that have routed across more than their prescribed number of hops), an Internet Control Message Protocol (ICMP) [57] error packet is created with a "Time Exceeded Message". There is no unbound processing loop here as the ICMP error packet created by *ICMPError* will not fail the time-to-live test. This creates a problem if each of these arcs in this

Data: Click graph $G_{\text{click}} = \{V_{\text{cg}}, E_{\text{cg}}\}$ and the schedulable element v^{in}
Result: The set of partial execution rates $\Psi^{v^{in}}$ calculated with respect to schedulable element v^{in}
`/* Initialize partial execution rates */`
foreach $v \in V_{cg}$ **do**
  if $v_i = v_i^{in}$ **then** $\Psi_v^{v^{in}} \leftarrow v_r^{in}$
  else $\Psi_v^{v^{in}} \leftarrow 0$
end
`/* Construct activity graph */`
$E^a \leftarrow \{(e_{src}, e_{dst}, e_i, e_p, e_w) | (e_{dst}, e_{src}, e_i, e_p, e_w) \in E_{\text{cg}}, e_p =$ pull$\} \cup \{(e_{src}, e_{dst}, e_i, e_p, e_w) | (e_{src}, e_{dst}, e_i, e_p, e_w) \in E_{\text{cg}}, e_p =$ push$\}$
$G_{\text{activity}} \leftarrow \{V_{\text{cg}}, E^a\}$
`/* Sort elements in activity graph */`
VArray $\leftarrow$ `TopologicalSort(`$G_{activity}$`)`
for $j \leftarrow 1$ **to** $|VArray|$ **do**
  $v \leftarrow$ VArray$[j]$
  $V^{dp} \leftarrow \{v' \in V_{\text{cg}} | (v', v, e_i, e_p, e_w) \in E^a\}$
  $\psi_v^{v^{in}} \leftarrow \sum_{v' \in V^{dp}} \psi_{v'}^{v^{in}} \cdot e_w$ where e_i is the component of $(v', v, e_i, e_p, e_w) \in E^a$
end

Figure 5.8: Compute Partial Execution Rates Algorithm

loop has a non-zero weight. The semantics of this would be that some packets would continue around the loop infinitely. If the product of all the arcs is less than one, this is a diminishing percentage of the packets, but still creates a problem for the topological sorting employed by the clustering algorithm. Therefore we require that any loop in the original Click graph must contain at least one zero weighted edge which the topological sorter considers absent. In practice this is a reasonable restriction as such situations usually involve error packets. They are critical for correction functionality, but can be ignored when optimizing the fast path of a network application. If the Click loop has a non-negligible impact on performance, manual adjustment of the task may be required, which would be part of one of the first feedback iterations (see Section 3.2.7) in the overall design flow.

Constructing the activity graph requires traversing each edge and node of the original Click graph or $O(|V| + |E|)$. The non-zero weighted edges form a directed acyclic graph (DAG) which topological sorting has known runtime of $O(|V| + |E|)$. Each element is then visited once using the order of the sorted array. The calculation per element involves visiting each of input edge for its edge weight and the partial execution rate of the source node. Since each edge is only visited once by its sink node, the runtime is again $O(|V| + |E|)$. The runtime of the total algorithm is consequently $O(|V| + |E|)$.

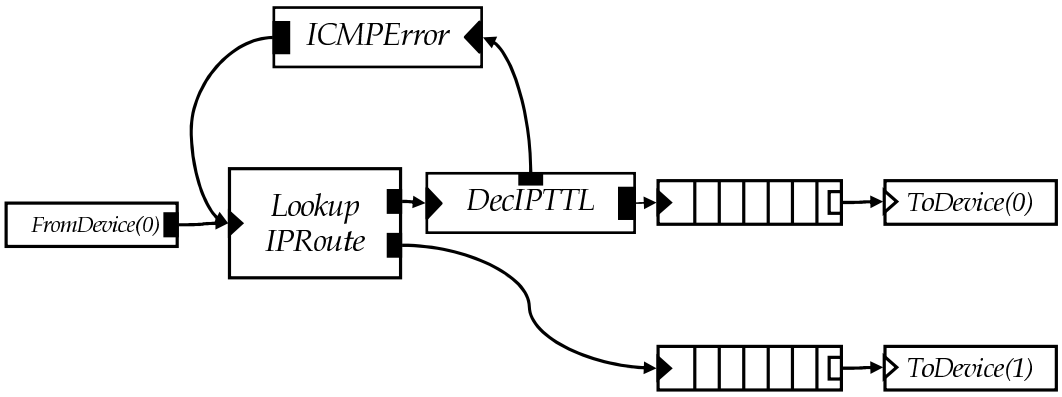

Figure 5.9: Example of a cyclic Click graph

Clustering Algorithm

Figure 5.10 describes the clustering algorithm. The algorithm traverses the Click graph several times to produce a task graph. The outer loop steps through each of the schedulable elements and creates a task that corresponds to it. This creates a one-to-one correspondence between tasks and schedulable elements. For each schedulable element, the partial execution rate of each element is calculated with respect to it. Elements are on the push or pull chain connected to the schedulable element if they have a non-zero partial execution rate. Each of these elements are stepped through, first associating it with the current task. Its implementation is then looked up in the target specific library using *LibraryLookupElement*. This function takes a Click element as input and based on the type returns a Click element with its profiling information included. Implicitly the architectural target is also an input to this function such that it produces profiling information of element implementations specific

to the target. The profiled execution cycles consumed and latency cycles are multiplied by the partial execution rate and added into the existing totals for the task. To add data to the task graph, the library is checked again for the data elements associated with the current element using *LibraryLookupData*. Since data elements may be shared among other tasks, the graph is first searched for an existing instance of it based on the type and the scoping rules of the element. This is implemented by the function *FindData* which is covered shortly. If a matching data element is not found, the data is created based on the library data returned by *LibraryLookupData*. The current task is connected to the existing or new datum as described by its library entry profile as returned by *LibraryLookupLink*.

As an example of how the algorithm works, consider how the algorithm takes the simple forwarder from Figure 5.1 and creates the task graph in Figure 5.6. Each schedulable element (both *ToDevices* and both *FromDevices* are iterated through in the outer loop. Starting with *FromDevice(0)* (abbreviated $fd0$), the elements' partial execution rates are calculated ($\psi_{fd0}^{fd0} = 1, \psi_{lu}^{fd0} = 1, \psi_{q0}^{fd0} = 0.5, \psi_{q1}^{fd0} = 0.5$, all others are zero). Note that partial execution rate propagation does not traverse across queues because they are the sinks of the activity graph. The new task covers the non-zero elements (*FromDevice(0), LookupIP*, and the two queues). The *LookupTable* and *Header* data is read from the *LookupIP* library entry and added to the task graph. The task graph edges are added between these and the queue data added according to the weights in the library entry and the partial execution rates. The

Data: Click graph $G_{\text{click}} = (V_{\text{cg}}, E_{\text{cg}})$
Result: Task graph $G_{\text{tg}} = (T_{\text{tg}}, D_{\text{tg}}, E_{\text{tg}}, w_{\text{tg}})$
`/* initialize various elements of` G_{click} `*/`
$T_{\text{tg}} \leftarrow \emptyset$
$D_{\text{tg}} \leftarrow \emptyset$
$E_{\text{tg}} \leftarrow \emptyset$
`/* loop through each schedulable element */`
foreach $v \in \{v'' \in V_{cg} | v''_s = schedulable\}$ **do**
  $T \leftarrow T \cup$ a new t
  $\Psi^v \leftarrow$ `CalculatePartialExecutionRates(`G_{click}, v`) //see Figure 5.8`
  foreach $v' \in \{v'' \in V_{cg} | \psi^v_{v''} \neq 0\}$ **do**
    associate v' with t
    $v_{\text{lib}} \leftarrow$ `LibraryLookupElement(`v'`)`
    $t_e \leftarrow t_e + v^{lib}_e \cdot \psi^v_{v'}$
    $t_l \leftarrow t_l + v^{lib}_l \cdot \psi^v_{v'}$
    $D^{\text{lib}} \leftarrow$ `LibraryLookupData(`v'`)`
    foreach $d^{lib} \in D^{lib}$ **do**
      $d' \leftarrow$ `FindData(`G_{tg}, d^{lib}`) //see Figure 5.11`
      if d' *is not NULL* **then** $d \leftarrow d'$
      else
        create a new d based on d^{lib}
        $D \leftarrow D \cup d$
      end
      `/* Add links based on library entry */`
      foreach $e^{lib} \in$ `LibraryLookupLinks(`v_{lib}, d_{lib}`)` **do**
        create new task graph edge e based on e^{lib}
        $w_e \leftarrow e^{\text{lib}}_w \cdot \psi^v_{v'}$
        $E_{\text{tg}} \leftarrow E_{\text{tg}} \cup e$
      end
    end
  end
end

Figure 5.10: Clustering Algorithm

other elements proceed similarly. When *LookupIP* is processed during *FromDevice(1)* task, the *FindData* function returns the *LookupTable* datum as it is regional and it already exists in the graph.

CalculatePartialExecutionRates is an $O(|V| + |E|)$ algorithm which is called with every schedulable element. The none-zero partial execution rate elements are iterated over which is $O(|V|)$. And for each of these the algorithm checks a constant number of data elements and searches the graph for them. There are $O(|V|)$ data elements which must be checked by *FindData*. The final complexity of the clustering algorithm is $O(|V||E| + |V|^3)$. In practice this run time is negligible with respect to the time spent in the mapping problem.

Find Data Algorithm

Finding the data in a task graph is not just a simple graph traversal because of memory scoping. When searching for a matching datum, the match should be found no further than the scope of the input datum. The *FindData* algorithm implements this functionality as shown in Figure 5.11. Along with the datum itself, the current element being added to the task is an input so that the scope of the element can be evaluated. It returns either the matching datum from the task graph or a NULL indicating that the matching data does not exist. For *local* scoped data, the function immediately returns as local data is not shared with any other element. For data of other scopes, each task that has already been added to the task graph is stepped

Data: The partially completed task graph $G_{\text{tg}} = (T_{\text{tg}}, D_{\text{tg}}, E_{\text{tg}}, W_{\text{tg}})$ and the datum to be found d^{in} and the element $v^{in} \in V_{\text{cg}}$ to which the data is associated
Result: Either the matching datum from the original Click graph d^{out} or NULL if there is no matching data
`/* Local variables are new for each element */`
if $d_s^{in} = LOCAL$ **then return** NULL
`/* Otherwise search through other tasks already created */`
foreach $t \in T_{tg}$ **do**
  $D^t \leftarrow \{d | d \in D_{\text{tg}}, (t, d, e_w) \in E \text{ or } (d, t, e_w) \in E\}$
  foreach $d' \in D^t$ **do**
    if $d'_\tau = d_\tau^{in}$ **then**
      let v be the element in the Click graph using d'
      if $(d_s^{in} = REGIONAL)$ *and* $(v_i^{in} = v_i)$ **then**
        return d_{tg}
      end
      if $(d_s^{in} = GLOBAL)$ *and* $(v_\tau^{in} = v_\tau)$ **then**
        return d_{tg}
      end
      if $d_s^{in} = UNIVERSAL$ **then**
        return d_{tg}
      end
    end
  end
end
return NULL

Figure 5.11: Find Data Algorithm

through. Each datum of each constructed task is examined the for a type match with the input datum. If it matches, the scope of the input data and the relationship between the input Click element and the Click element using the existing datum. *Regional* data match requires the Click element instances match, *global* data matches if the Click element types match, and *universal* data matches for any datum type match.

5.4 Optimizations

Domain specific languages provide structural information about the application that is difficult to glean from the set of sequential code eventually generated. This provides this design flow the opportunity for high level optimizations. Exposure of data with scoping information along with the semantics of the application domain known permits the software management of data. By combining local data used as working copy of data in global memory, software managed interprocedural caching can be accomplished. By eliminating memory accesses, there are fewer latency events in the resulting task graph and consequently better performing design. Replication of tasks and breaking up existing tasks makes the granularity of the application finer. A finer grained application provides more opportunities for the mapping engine to find a good implementation, but increases the size of the design space. Clustering provides a useful way of grouping tasks to reduce the design space intelligently, but this coarse description of the application can be intelligently refined for more performance.

Software caching can be handled automatically using the structural information of the application and meta-data in the target specific library. We have handled the replication and retiming of tasks manually but we have also found that it can be automated as well using heuristics. The following section discusses these techniques and the heuristics.

5.4.1 Software Caching

Software caching is a software technique for locating copies of data in global memory in local resources. Caching for multiprocessors can be handled with hardware, but for a large number of processors this proves difficult cite. Hardware schemes retain the transparency of uniprocessor computing, but incur overhead due in large part to cache coherence protocols which adds significant logic to on-chip real estate and increases power consumption. Cache miss rates and on-chip buses needed for cache coherence can account for a significant percentage of the power consumption on chip [52]. Hardware support for cache coherency can be beneficial, but utilizing the structure of software is a boon for large scale multiprocessors [28] [7].

Domain specific languages impose structure onto the application designer that can be exploited at compile time. As with conventional programming languages, data size and scope is known statically. Using Click with our target specific library, inter-element (i.e. interprocedural) access to memory can be reasoned about explicitly. A common scenario in a network application is that a data is explicitly loaded from main memory into local memory and if modified it is explicitly saved back. This is the case with many elements in the Click library. For example, Figure 5.12 shows the data indicated by library elements overlaid on a Click application. *CheckIPHeader*, *IPMirror*, *DecIPTTL* each load the header for different purposes. *CheckIPHeader* verifies the checksum and various flags of the packet header, *IPMirror* swaps the source and destination IP addresses, and *DecIPTTL* decrements the time to live.

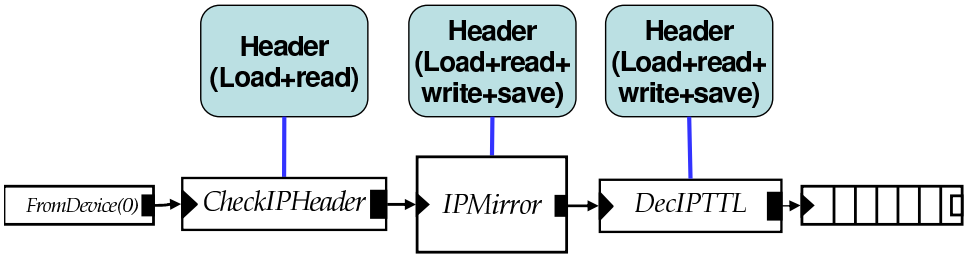

Figure 5.12: Example of a memory access pattern for a local variable

The local variable *Header* for each of these elements represents the local copy of the words of the packet in main memory that are the header. The initialization routine for each element loads the packet from main memory. *IPMirror* and *DecIPTTL* both modify *Header* so they each save their copy back to main memory.

Local data such as *Header* has the property that it is scoped to a single context of execution for one element. During the execution of a push or pull chain of elements (i.e. a task), each element has exclusive control with respect to that chain over data. Therefore local data may be safely shared among elements within a task. Such sharing can reduce the amount of memory needed for a task or eliminate redundant code. As the application tasks are eventually generated into a set of sequential programs (see Section 3.2.6), it may be possible for a traditional compiler to find and exploit this optimization. However, it is difficult to reason about these variables inside and across the functions. In practice, we have found that compilers for these systems are unable to find or make such optimizations.

Considering that the elements in Figure 5.12 are implemented on the same con-

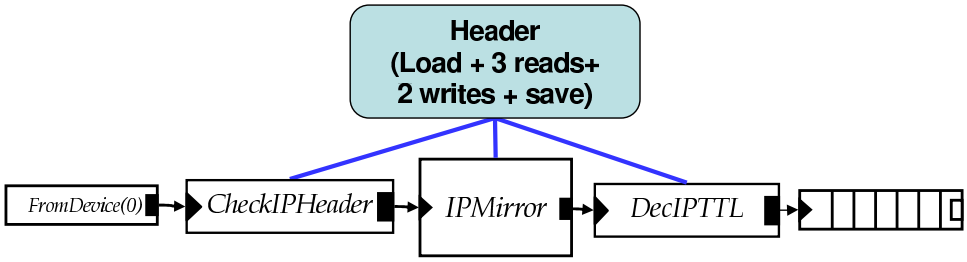

Figure 5.13: Example of optimization of a local variable

text of execution, there is obvious redundancy. *Header* needs to be loaded only once at the beginning of the chain and saved back only once at the end. The elements themselves can share the same local copy by virtue of the fact that they are covered by the same task and therefore implemented on a single context of execution. The optimized memory access pattern is shown pictorially in Figure 5.13. Note by contrast, if elements are mapped used pipelining instead of clustering, each element could be operating concurrently with elements in its chain. In this scenario, it is not possible to share local variables with other elements since it may incur race conditions on data or they may be working on different packets.

The algorithm for finding the loaders of a particular datum is described by recursive function in Figure 5.14. It is initially called with inputs of the original Click graph, the schedulable element that defines the push/pull chain (i.e. task), and the datum to be used as the software managed cache. Also the set of elements that will do the load is passed by reference initially as the empty set. The function operates by first recursively checking if the upstream pull elements have already done the loading

for the datum for *all* paths leading to this element. If any path does not load the datum, the current element is searched for a connection to it. By restricting loads and saves to be the responsibility of elements that use the datum, elements do not need to contain load and save directives for data they would otherwise have no knowledge of. Furthermore, locating the loads and saves in elements that touch the data should minimize the number of unnecessary loads and saves. If a load was missing on some entry path to this element and the element failed to load the datum, the elements active output edges (output push edges) are exercised with the same function. This ensures that an element using the datum downstream loads it before using it.

Software caching requires additions to the element descriptions that employ local variables. Previously those elements which did not modify the variable could safely throw away their stored copy. But with software caching, elements upstream may have modified a local datum but not yet written it back because it is used later in the context of execution. The elements that did not modify the datum may now have to write back the result. In the *LookupIP* simple action, this requires the additional save code at the end of *LookupIPSimpleAction* as shown in Figure 5.16.

Software caching for memory exposed multiprocessors is not new. Hot Pages [51] is a software solution for the Raw Machine [67]. It is a compile time approach that utilizes knowledge of the state of virtual pages to remove cache-tag lookups. The lookups are replaced with register comparisons with previously loaded virtual pages. The target has multi-banked memory whose structure is considered directly. By

Function `GetLoaders(`G_{click}, v^{in}, d^{in}, $V_{cg}^{loaders}$`)`

Data: Original Click graph $G_{\text{click}} = (V_{\text{cg}}, E_{\text{cg}})$, an element $v^{in} \in V_{\text{cg}}$, and a datum d^{in}

Result: The function returns a boolean value indicating whether a load has covered all upstream paths. Also if the current element should load the datum d^{in}, the function adds the it to the set of Click elements $V_{\text{cg}}^{\text{loaders}}$, which is passed by reference

static isLoaded ← true `//initialize each node to not be a loader`
static visited ← false `//static variable to prevent path explosion`
if *visited* **then return** isLoaded
visited ←1
`/* Check direct predecessor elements for previously done loads */`
$V^{dp} \leftarrow \{v' | v' \in V_{\text{cg}}, (v', v, e_i, e_p, e_w) \in E \, e_p = \text{pull}\}$
if $\bigwedge_{v' \in V^{dp}}$ `GetLoaders(`G_{click}, v', d^{in}, $V_{cg}^{loaders}$`)` **then** isLoaded ← true
else
  `/* Check if the datum is used by element */`
  foreach $d^{lib} \in$ `LibraryLookupData(`v'`)` **do**
    if d^{in} *is based on* d^{lib} **then**
      $V_{\text{cg}}^{\text{loaders}} \leftarrow V_{\text{cg}}^{\text{loaders}} \cup v'$
      isLoaded ← true
    end
  end
end
`/* Call the function on direct successors if the element (or all paths upstream) failed to load it */`
if *isLoaded* = *false* **then**
  $V^{ds} \leftarrow \{v' | v' \in V_{\text{cg}}, (v, v', e_i, e_p, e_w) \in E, e_p = \text{push}\}$
  foreach $v' \in V^{ds}$ **do**
    `GetLoaders(`G_{click}, v', d^{in}, $V_{cg}^{loaders}$`)`
  end
end
return isLoaded

Figure 5.14: Function for determining loaders for software caching

Function `GetSavers(`G_{click}, v^{in}, d^{in}, V_{cg}^{savers}`)`

Data: Original Click graph $G_{\text{click}} = (V_{\text{cg}}, E_{\text{cg}})$, an element $v^{in} \in V_{\text{cg}}$, and a datum d^{in}

Result: The function returns a boolean value indicating whether a save has covered all downstream paths. Also if the current element should save the datum d^{in}, the function adds the it to the set of Click elements $V_{\text{cg}}^{\text{savers}}$, which is passed by reference

static isSaved ← true `//initialize each node to not be a saver`
static visited ← false `//static variable to prevent path explosion`
if *visited* **then return** isSaved
visited ←1
`/* Check direct successor elements for previously done saves */`
$V^{ds} \leftarrow \{v' | v' \in V_{\text{cg}}(v, v', e_i, e_p, e_w) \in E, e_p = \text{push}\}$
if $\bigwedge_{v' \in V^{dp}}$ `GetSavers(`G_{click}, v', d^{in}, V_{cg}^{savers}`)` **then** isSaved ← true
else
  `/* Check if the datum is used by the current element */`
  foreach $d^{lib} \in$ `LibraryLookupData(`v'`)` **do**
    if d^{in} *is based on* d^{lib} **then**
      $V_{\text{cg}}^{\text{savers}} \leftarrow V_{\text{cg}}^{\text{savers}} \cup v'$
      isSaved ← true
    end
  end
end
`/* Call the function on direct predecessors if the element (or all paths downstream) failed to save it */`
if *isSaved* = *false* **then**
  $V^{dp} \leftarrow \{v' \in V_{\text{cg}} | (v, v', e_i, e_p, e_w) \in E, e_p = \text{pull}\}$
  foreach $v' \in V^{dp}$ **do**
    `GetSavers(`G_{click}, v', d^{in}, V_{cg}^{savers}`)`
  end
end
return isSaved

Figure 5.15: Function for determining savers for software caching

Function `LookIPSimpleAction`

Data: Packet descriptor p
Result: Returns next hop port number
; `/* Setup execution and latency meta data */`
ExecutionCycles = 100
LatencyCycles = 0
; `/* Describe computation */`
int NextHop
declaration (name=Header, type=PacketHeaderStruct, scope=local, size=12)
declaration (name=LookupTable, type=PacketHeaderStruct, scope=regional, size=10000)
load {**ExecutionCycles** = 0, **LatencyCycles** = 60} **begin**
    Header ← `LoadHeader(`*p*`)`
end
read (name=LookupTable, Number of times = 4)
NextHop ← `TrieTableLookup(`*Header.DestinationIPAddress, LookupTable*`)`
save {**ExecutionCycles** = 0, **LatencyCycles** = 60} **begin**
    `SaveHeader(`*p, Header*`)`
end
return NextHop;

Figure 5.16: Element description adjusted to support software caching

allocating data across banks, the memory bandwidth of the Raw architecture can be maximized.

5.4.2 Retiming

Application tasks can be split temporally to expose more parallelism. While the mapping engine has no knowledge of such internal task boundaries, the domain specific language describes the application at a finer granularity and includes the communication semantics. At this application level, designers may safely change the application description to expose more temporal parallelism to the mapping engine. If a set of Click elements are connected in a non-branching chain (i.e. there is no element with a fanout of more than 1), these elements may operate on different packets in parallel that are moving down the chain. By pipelining the application, each stage may run concurrently increase the ability to utilize parallel platform. In a system period, a single task that is pipelined may be broken into multiple pieces. This does not reduce the amount of execution in one period as each piece must still process one packet, but it gives the mapping engine more freedom to spread the computation over multiple processing elements. In networking pipelining can be especially useful when there is a long chain of computation naturally attached to a given port. We employ pipelining in each of the IXP2xxx forwarding as discussed in Section 6.3 to decouple the receiving functionality from header processing.

Pipelining in Click can be accomplished by modifying the application description.

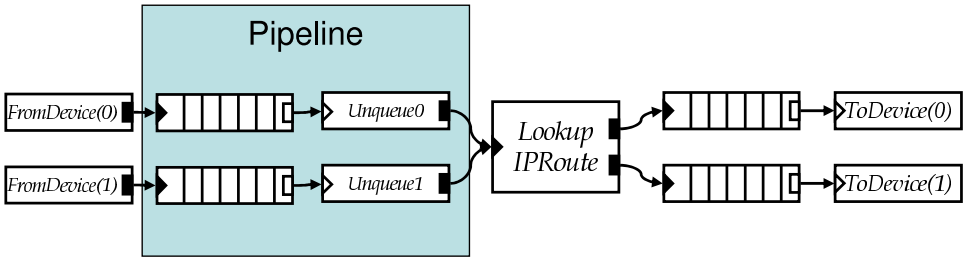

Figure 5.17: Simple forwarder with pipelining

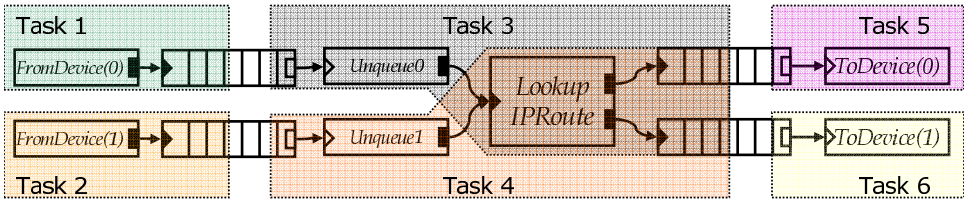

Figure 5.18: Tasks generated from a pipelined simple forwarder

By interrupting a chain of elements with a *Queue* and *Unqueue* pair, a single chain may be split into two. The new *Unqueue* element serves as the schedulable element driving the new chain. If the pipelined Click edge is a push edge, the pair is inserted as *Queue* first then *Unqueue*. This pipeline stage has a push input and a push output and therefore does not require a change to any of the other elements. Conversely a pull chain pipeline is inserted as first an *Unqueue* and then a *Queue* so that the pipeline has both a pull input and a pull output. Figure 5.17 shows a possible retiming of the simple forwarder. When transformed to a task graph, two additional tasks are created as shown in Figure 5.18. The resulting task graph has the same functionality but with more tasks, exposes more temporal parallelism.

Pipelining is not always beneficial. First, network designers carefully size buffers

to absorb the burstiness of network traffic while ensuring there is not undue delay in routing the packets. Designers also take care to size buffers such that they may be efficiently supported by the memory target. This carefully sizing and placement in the application can be upset by the introduction of more queues. Adding pipelining either increases the memory consumption of the application or changes buffer distribution consequently altering the applications behavior to bursty traffic. Second, introducing new pipeline stages adds a new enqueue and dequeue to each packet. Latency bound stages may not benefit from the offloading of computation if it comes at the cost of an even greater number of latency cycles. Finally, pipelining disrupts software caching described in Section 5.4.1. Elements on a push chain that would otherwise be able to share a local datum when split by a pipeline stage must do multiple loads and saves. While retiming gives more freedom to the mapping engine, it comes at the cost of additional latency.

5.4.3 Replication

Application tasks can be split spatially. By allowing a task to process multiple packets concurrently, allows multiple instances of the same task to run within a system period. If there are two instances of a task, they can each process one packet per system period. After normalizing the invocation rates to other other tasks in the system, they can also be modeled as two tasks processing a packet every other system period or half a packet each system period. The execution cycles consumed and the

latency cycles spent by the task are rated accordingly. By exposing spatial parallelism, a single task can be considered as multiple smaller tasks giving the mapping engine more freedom to find a good design.

As with retiming, the task graph itself does not contain enough information to safely replicate a task. A replication algorithm that operates on a task graph would not be able to determine when to connect to existing data and when to create new ones. The domain specific language description coupled with the target specific library contains scoping information thus enabling safe replication. To implement replication at the application level, the designer or heuristic need only added more schedulable elements with the same connections as the schedulable element assigned to the task to be replicated. The computational chain of non-schedulable elements will be automatically replicated by the clustering algorithm during transformation. Local variables are created by the same algorithm while the non-local variables are connected to the new task automatically. An example of replication of a Click graph and the subsequent task extraction of the retimed simple forwarder is shown in Figure 5.19. Each schedulable element of the retimed simple forwarder is duplicated, consequently doubling the number of tasks. Note that the new tasks necessarily have exactly the same input and output queues. Thanks to packet independence, replication is an extremely useful technique which most of the applications for evaluation employ. As discussed in Section 6.4, we have found replication can garner 59% to 122% performance improvement.

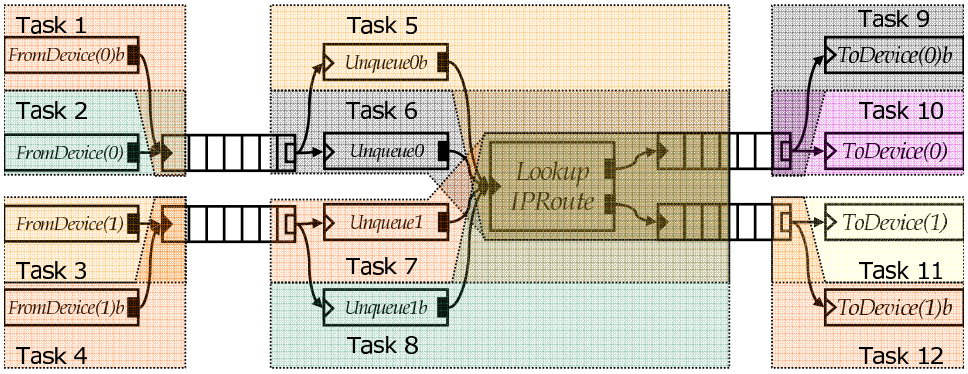

Figure 5.19: Simple forwarder with replication

Replication of network applications has certain limitations. To exploit spatial parallelism, the target must be able to swap between multiple concurrent replications of the same task or run them on different processing elements. For architectures with a fixed number of hardware threads, there is an upper-bound on the amount of replication that may occur. Furthermore, with each replication more local data is created that must be accommodated. Since all tasks may be running at the same time, all of the new local data must be accommodated with more memory. Replication may also lead to the violation of the in-order packet processing assumption. Since many protocols are sensitive to packet reordering in the network, routers must avoid allowing packets on the same flow to overtake each other. Fortunately many common basic units of packet processing have relatively consistent processing rates on packets within the same flow. For example, *LookupIP* has an unknown number of lookups through the trie table structure for any given packet. But packets within the same flow have the same destination address, and will consequently exercise the trie table

in the same way. Such processing regularity and the fact packets are serialized at each queue (replicated tasks share the same input and output queues) leads to a wide applicability of replication.

5.4.4 Automated Optimizations

Prior sections described the an automated cache management and manual optimization using retiming and replication technique. As discussed in the prior section, retiming and replication can have adverse effects on the implementation. In a collaborative effort here at Berkeley, a heuristic approach were developed to retime and replicate the original application description [66]. By adding pipeline stages and replication to those tasks that are the bottlenecks, the original unpipelined design can be improved upon automatically. This heuristic approach relies on a fast feedback to iterate over many possible replications and retimings. As the exact mapping engine can take minutes to a single instance, we use a bin-packing heuristic in which tasks are items with sizes equal to the execution cycles consumed and processing elements are bins. The heuristic that arrives at an allocation of tasks to processing elements. Such heuristics arrive at sub-optimal results, but perform well enough on loosely coupled designs to explore the replication space.

However these heuristics perform poorly when replicating a task results in two tasks accessing a regional datum. As the bin packing algorithm does not group items before allocating them to processing elements, the replicated tasks are often

Table 5.1: Heuristic solution for retiming and replication

Application	**Click Graph**	**Solver**	**Makespan**	**Runtime**
IPv4	Original	WFD	967	0.2s
IPv4	Original	Exact	907	18.9s
IPv4	Replicated	Exact	291	738s
IPv4	Original	Proposed Heuristic	**290**	**4s**
NAT	Original	WFD	1645	0.2s
NAT	Original	Exact	1505	1.3s
NAT	Replicated	Exact	543	755s
NAT	Original	Proposed Heuristic	**289**	**5s**

placed on different processing elements. Consequently the datum must be placed in a memory shared between multiple processing elements which usually has a significantly higher access latency. A solution to this problem is to introduce a different style of replication. It differs from the manual replication steps mentioned above in that all replicated instances of the task are grouped as a single item in the bin packing problem which is referred to as *single item replication*. A task using single item replication has all of its instances placed on a single processing element enabling the use of local memory for shared data. Replicating a task and representing all instances with a single item does not change the total size of the item, but lowers the latency contribution to the makespan. It also changes the resource requirements of an item regarding the architectural constraints. It increases the number of threads consumed by an item when allocating it to a bin (processing element).

Initial results for this approach have been promising, producing replicated versions of the applications that rival the manually replicated and retimed applications

as shown in 5.1. Like the greedy heuristic applied to DiffServ in Section 6.3.2, exact solutions are almost 10% better than the *worst fit decreasing* (WFD) heuristic. Applying replication to bottlenecks while keeping replicated tasks grouped through single item replication creates improved makespan at a orders of magnitude less time spent on the problem. The NAT implementation is even better than the manually replicated implementation. However this is in model only as over replicating the task which reads and writes to a regional data in globally shared memory creates more latency than is captured by the model. As the mapping framework relies on a good starting point from the designer, this approach requires more investigation for the heuristics appropriate to optimize network applications.

Chapter 6

Evaluation

Using a commercial network processor and the tools presented in the previous chapter, we demonstrate that this automated design flow is a productive alternate to producing efficient implementations. We also show that memory aware mapping increases overall application performance compared to memory unaware mappings. To this end, we implement a set of representative applications using our design flow selecting applications that range in size and complexity. When possible we hand mapped or hand coded implementations in alternative programming environments for comparisons. We also perform replication of the task graph to increase performance to within 17% of hand mapped designs. We find that memory aware mapping improves on memory unaware mapping by 5% to 7% and that it is necessary for reaping benefits from replication. Replication can improve on the original designs by 59% to 122%.

6.1 Metrics

Performance of network processing is measured by the rate at which it processes packets or data. Packet processing is measured in packets per second (pps) while data processing is measured in megabits per second (Mbps). For a given traffic load, this is measured in two different ways: *throughput* which is the maximum ingress rate at which the router operates without dropping packets and *forwarding rate* which is the maximum rate at which packets egress from the device. Throughput can measured by increasing the ingress rate until the application begins to drop packets. The last drop free ingress rate is the throughput. Forwarding rate is measured by applying *line rate* (the maximum rate of packets for a given port type and packet size) to the input and measuring the egressing bandwidth. These measurements are based on a particular traffic load applied to the application. The traffic loads applied vary by size and processing type as appropriate for the application and are meant to reflect real world.

6.2 Test Benches

6.2.1 Simulators

We use cycle accurate simulators to exercise our tool flow. For the IXP1200 and the IXP2xxx series simulation environment that ships with these products. While the two environments are separate products, they have many similar features: cycle

accurate execution of instructions on processing elements, memory access, and interconnect latencies. The simulator models media access channels (MACs) of various kinds of ports. The environment also features a test infrastructure that allows users to build packets from a variety of protocols and to specify parameters such as size, rate, packet header fields, and the packet's payload. The packets created are valid packets that resemble real traffic.

6.2.2 Profiling

To obtain average execution cycles per task, the application was tested with worst case input traffic. An instance of each task class and implementation was run on a PE by itself in a functionally correct configuration so that it could be profiled with the appropriate traffic. We note that different configurations may cause a task to perform differently, but those effects have not yet been substantial. For shareable and quasi-shareable instructions, the application was complied with varying task configurations. Data size was determined by examining each of the elements when assigned to memory by the compiler. We have seen at most a 15% change in execution cycles consumed between tasks compiled alone and in the presence of other tasks.

6.2.3 Solvers

To solve the formulations we have tried randomized rounding, pseudo-boolean solvers, and general purpose ILP solvers. With the IXP1200 target, we have the most

success with the pseudo-boolean solver, Galena [16] and Ampl/CPLEX 10.1 [1]. The solvers were run on a Linux desktop with 2GHz AMD Athlon with 512KB of cache and 1.5 GB of memory. For Ampl/CPLEX we utilize the concept of an optimality gap to trade off solution time for provably better results. Typically, the closer the solver gets to the optimal solution (or proving that it has the optimal solution), the more time it takes for a unit of change in the upper and lower boundaries. Quite often the solver may find the optimal solution relatively quickly but spends longer proving that it is. The optimality gap represents the difference between the best known solution and the bound on what the best solution could be. By setting the optimality gap to 5% for the IXP2400 designs presented in this chapter, the large amount of time spent on achieving a small *potential* improvement is removed.

6.3 Applications

Unfortunately there does not exist a set of universally accepted benchmarks for network processors, much less a set written in a domain specific language. We selected a representative set of networking applications and built them in our design flow and when possible alternative design environments. They offer a range of complexity and scale. IPv4 forwarding is a high performance, low complexity application while the webswitch is a feature rich application difficult to implement on a network processor due to its complexity.

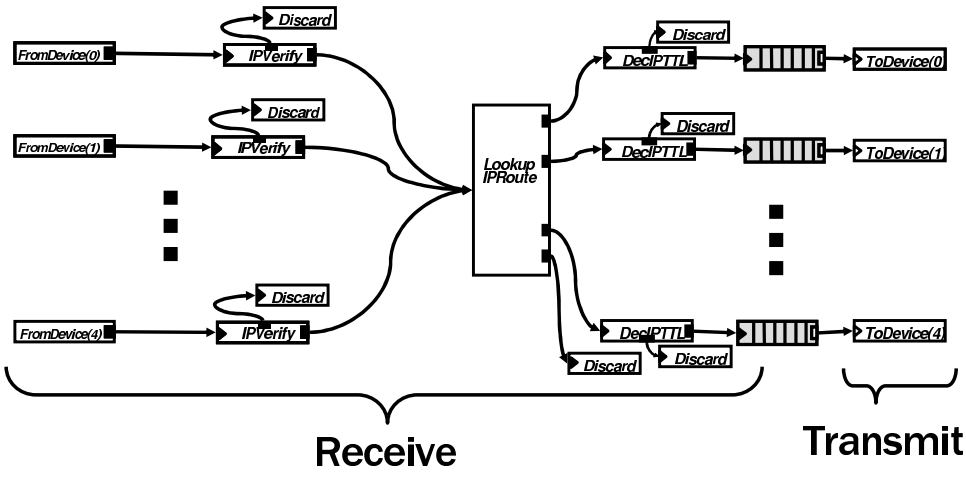

Figure 6.1: Click description of the dataplane of the IP forwarding

6.3.1 IP Forwarding

IP forwarding is a common application benchmark for networking that is discussed in Section 2.2.1. While relatively simple, the dataplane of the application is readily comparable with other frameworks and implementations. Figure 6.1 displays the Click description of the application pictorially. Packets move from left to right through the diagram. They ingress through *FromDevice* and are immediately queued. Packets are dequeued and continue through a header verification process, a next hop lookup based on a 4 level trie table, and then have their time-to-live counters decremented. Finally they are queued and then egress through *ToDevice*.

Implementation on IXP1200

We implemented a 16 port Fast Ethernet (16x100Mbps) IPv4 router based on a 16 port version of the application shown in consisting of 16 *receive* tasks and 16

Table 6.1: IPv4 forwarding task characteristics

Class	Number of Tasks	Execution Cycles	Shareable Instructions	Quasi-shareable instructions
Receive	16	337	801	0
Transmit (Impl 1)	16	160	348	0
Transmit (Impl 2)	16	140	5	285

transmit tasks with characteristics shown in Table 6.1. A *transmit* task has local state that may be can be coded to access that data in two ways. The task can use either shareable instructions or quasi-shareable instructions to reference its shared variables (see Section 2.3.1 for an explanation on shareable versus quasi-shareable instructions). For this application, we target a version of the IXP1200 with an instruction store of 1K words per microengine.

The instruction store constraints preclude the two task classes from coexisting on any PE. The problem then degenerates into bin packing, the only wrinkle being that Transmit class tasks may exist in either implementation. Implementation 2 is preferred for all Transmit tasks since it is faster of the two and fits within instruction store limits. The resulting configuration shown in Table 6.2 (RX=Receive and TX=Transmit). The automatically mapped implementation is exactly the same partition that was arrived at from hand tuning. We compare the automatically generated design to a hand coded design written in Microengine C as shown in Figure 6.2. The modularity of the framework of the automated design incurs some overhead, making the header process take longer than the microengine C. As packets get larger, they both become transmit limited.

Table 6.2: IPv4 mapping to the IXP1200

Application	PE1	PE2	PE3	PE4	PE5	PE6
IPv4 - Auto	4 RX	4 RX	4 RX	4 RX	8 TX	8 TX
IPv4 - Hand	4 RX	4 RX	4 RX	4 RX	8 TX	8 TX

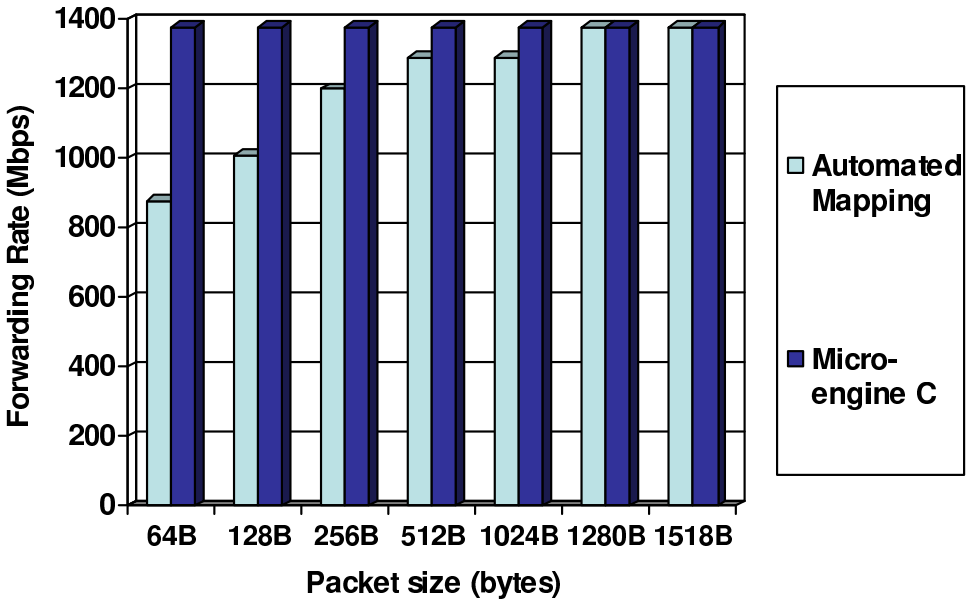

Figure 6.2: IXP1200 IPv4 forwarding rates vs. different packet sizes for each approach

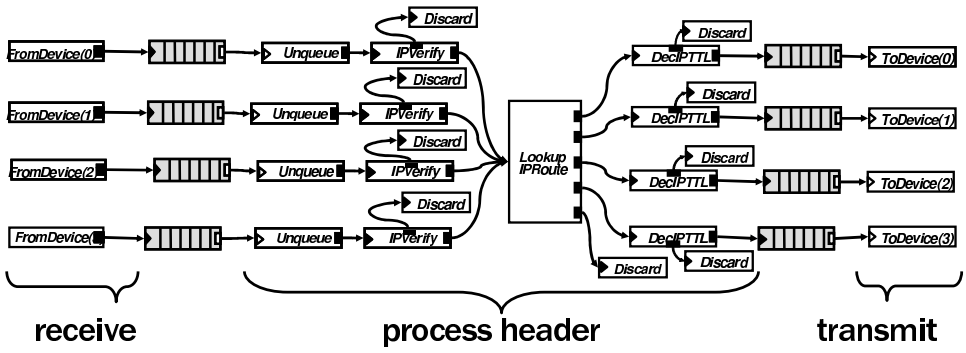

Figure 6.3: Retimed IP forwarder for the IXP2400

Implementation IXP2400

The same basic functionality is in the IXP2400 implementation. Instead of receiving and transmitting to many 100Mbps Fast Ethernet ports, the IXP2400 design implemented uses 4 Gigabit Ethernet ports. The 4 port version of the Click graph implemented on the IXP2400 reveals a receive bottleneck as more packets stream in through fewer ports. Since Receive tasks do not share any local data with downstream, pipelining the design by inserting a *Queue*/*Unqueue* pair is an effective way of increasing the forwarding rate as shown in Figure 6.3. There are three task classes as the *Process Header* tasks have been separated from the *FromDevices*.

Since the IXP2400 has an exposed memory architecture that may influence the placement of data along with the task, the tool generates the task graph shown in Figure 6.4. Ovals represents tasks while boxes are data. The receive tasks are at the top of the graph, one for each port which feed to process header tasks in the middle. The transmit tasks are located at the bottom. The task execution and latency weights

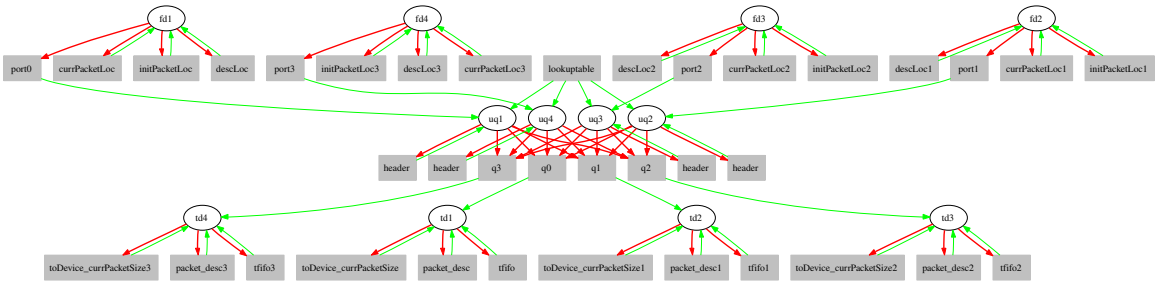

Figure 6.4: 4 port IP forwarder task graph

Table 6.3: 4 port forwarder task characteristics

Element	Number	Execution Cycles	Fixed Latency
Receive	4	143	353
Process Header	4	267	220
Transmit	4	148	402

shown in Figure 6.3 are computed from the profiled library of elements. The mapping engine uses this information to produce the set of assignments indicated by Figure 6.5. The white boxes enclosing ovals represent processing elements covering tasks and filled boxes encompassing gray rectangles as memories covering data. Note the use of two next neighbor registers for queues.

This implementation can be improved upon by exposing more of the application level parallelism. In this case, we can create more tasks to run in a system period. By creating four copies of the schedulable elements, new computational paths form through the Click graph, the tool constructs more tasks to allow for more threads and microengines to be working on that stage in parallel sketched in Figure 6.6. Some of the memory resources are shared differently based on their use in the application.

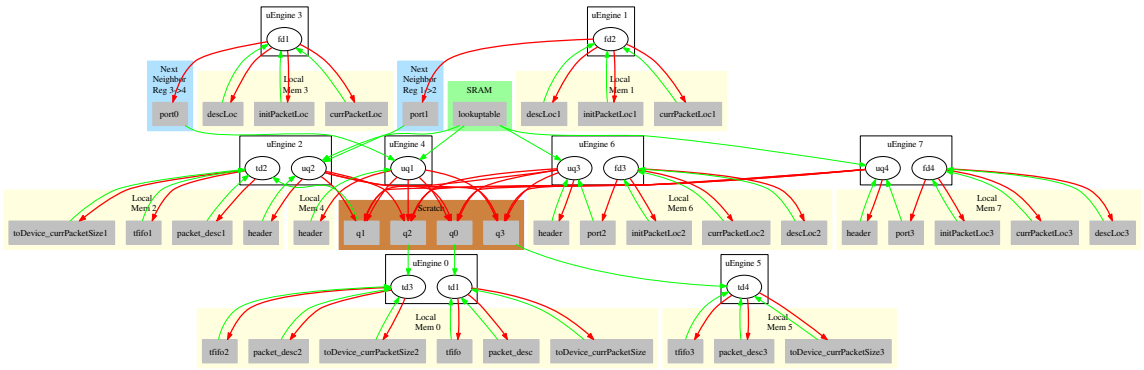

Figure 6.5: 4 port IP forwarder mapped to the IXP2400

For example, the state corresponding to receiving a packet from a single port must be shared among all receive tasks bound to that port. This results in a set of highly connected receive tasks for each port. The memory aware mapping clusters these highly connected tasks as shown in Figure 6.7, successfully exploiting the exposure of more parallelism. The only data to be located in non-local memories are LookupTable and queues.

To assess the utility of considering memory simultaneously with task allocation, both approaches are mapped with a memory unaware mapping formulation. The constraints are the same except for the objective function which no longer considers the cost of accessing data in memory. Data must be assigned to a feasible memory, but the formulation is unaware of the memory's contribution to the makespan. To improve trivial solutions where all data is assigned to large global memory like DRAM, we run a second phase of optimization over each datum. If there is room in a fast memory that is appropriately scoped for the current task allocation, the datum is moved there.

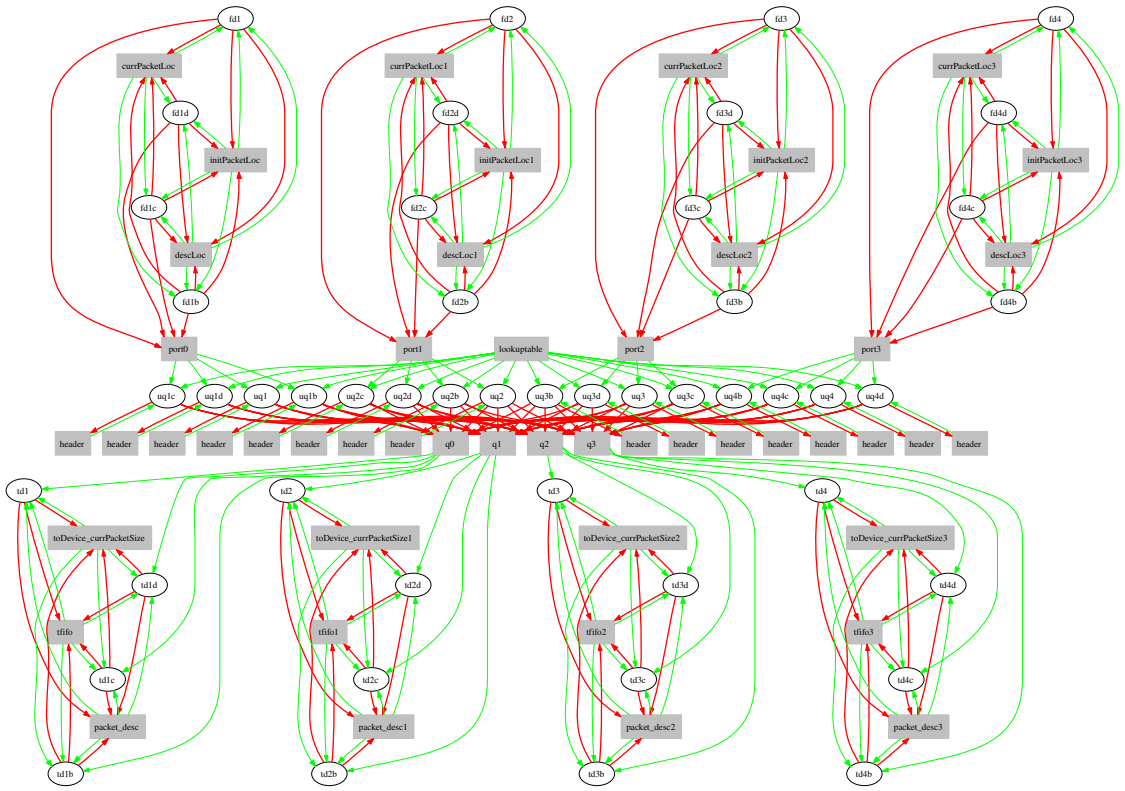

Figure 6.6: 4 port IP forwarder task graph with replication

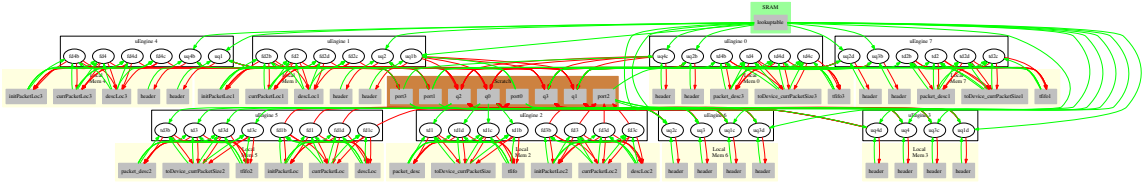

Figure 6.7: 4 port IP forwarder mapped to the IXP2400

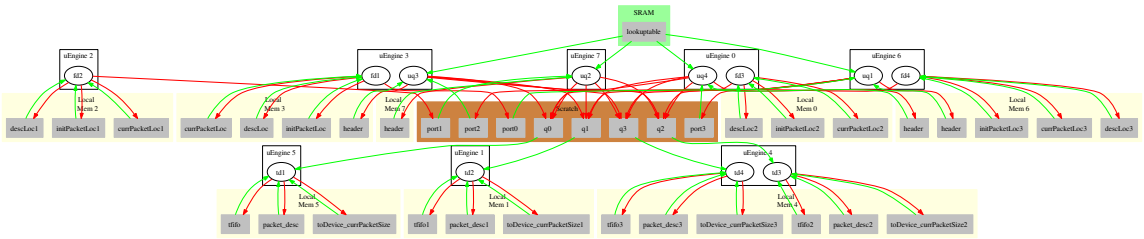

Figure 6.8: 4 port IP forwarder mapped to the IXP2400 without considering memory

The memory unaware mapping of the unreplicated IP forwarder is shown in Figure 6.8. Special memory structures are underutilized in this configuration as tasks are not placed to make use of them. We applied the memory unaware optimization to the replicated IP forwarder too. Since the memory unaware formulation ignored the tightly connected nature of these tasks, the resulting implementation is *worse* than the original non-replicated version. Commonly used data that were located in local memory are forced into slower global memory. The performance benefit of increased threads cannot make up for the slow down of using slower memory. This design would thus never be a solution considered viable by a designer.

Each of the IP forwarding implementations was tested using a variety of packet sizes each of which exercised all levels of the trie table (a worst case lookup). The forwarding rates of each are given in Figure 6.9 including an implementation hand assembled from Microblocks (see Section 2.4.4). For the packet size the application was tuned for (64B) memory aware mapping outperformed the memory unaware mapping by 7%. However as the packet sized moved away from that which was profiled, the memory aware mapping no longer outperformed the unaware. The change in the ratio

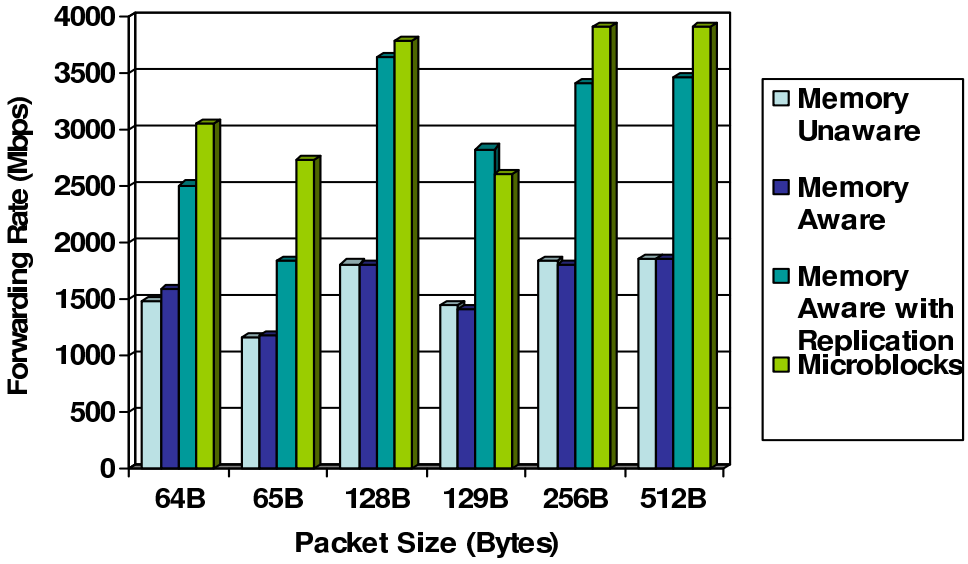

Figure 6.9: IPv4 forwarding rates vs. different packet sizes for each approach

of execution times was immediately significant as the near doubling of the execution cycles and latency of the receive and transmit tasks relative to process header tasks made 64B tuned mapping suboptimal. This happens on the seemingly incrementally change to 65B packets because the minimum packet chunk that can be moved from the MAC to the IXP is 64B (in this configuration). So one additional byte in the packet results in nearly twice the work for receive and transmit. The manually replicated design performs within 17% of the hand mapped design from the performance oriented framework of Microblocks.

Table 6.4: DiffServ task characteristics

Class	Number of Tasks	Execution Cycles	Shareable Instructions
Receive	4	99	462
Lookups	4	134	218
DSBlock	4	320	1800
Transmit	4	296	985

6.3.2 Diffserv

Differentiated services is a quality of service application discussed in Section 2.2.3. After attempting different numbers of ports implementations, we implement a 4 port version of an interior node. We implement it on the 2K instruction store version of the IXP1200 with the Click diagram shown in Figure 6.10. Packets are processed in the same fashion as with the forwarder with the addition of the *DiffServ* element. This element classifies each packet based on a field in the packet header indicating its class. Each class is uniquely shaped through bandwidth rating and random early detection (RED) queue management. If the Assured Forwarding 2 class exceeds a certain bandwidth, packets from the flow are demoted to Assured Forwarding 3. These flows are then prioritized for egress on a particular port by the *DeficitRoundRobin* element. Each flow is allocated a certain percentage of the egress bandwidth except Best Effort which is only scheduled if no other packet it ready.

The corresponding task properties are presented in Table 6.4. Since there are only 4 tasks in each class and the IXP1200 has 4 hardware threads per PE, the quasi-shareable instructions are omitted and all instruction store used is represented by

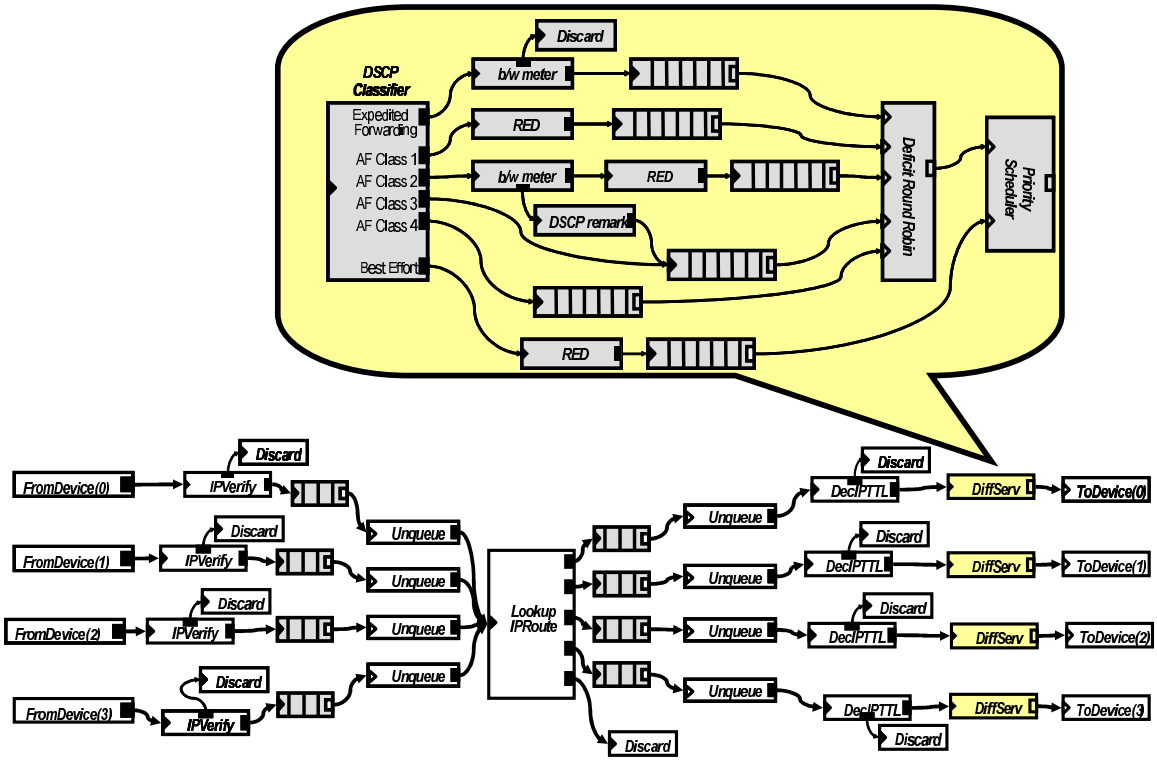

Figure 6.10: DiffServ Click graph

the sharable component. Even targeting an architecture where task mapping may be considered independently of data assignment, this relatively small application benefits from using an exact solution method over a general heuristic. One effective method on similar problems is a greedy heuristic (called *worst fit decreasing* (WFD)) in which tasks are ordered by their weight and then assigned in that order to processing element. Each task is mapped to the least weighted microengine at that time. While this works provably well for an ordinary item to bin minimization problem, the instruction store constraints on microengines reduces its utility.

Consider the mapping produced by the greedy algorithm shown in Figure 6.11. The DSBlock are the first tasks to be placed as they are the heaviest weighted. The microengines are empty at the time of their assignment so they are each assigned one. Instruction store consumed by DSBlock prevents receive or transmit tasks to be assigned to any of those four microengines. All eight of those tasks are confined to the two remaining microengines, pushing the makespan up to 790 cycles. With our formulation, the mapping engine returns the optimal mapping with respect to the model shown in Figure 6.12. By using an exact method the makespan improves to 640 cycles, a 19% improvement.

The mapping found by hand and by the automated tool flow are shown in Table 6.5. On a representative mix of 64 byte packet classes, the hand map design slightly outperforms the automated design as shown in Figure 6.13. The ingressing bandwidth of each traffic flow is held constant. The Assured Forwarding classes ingress

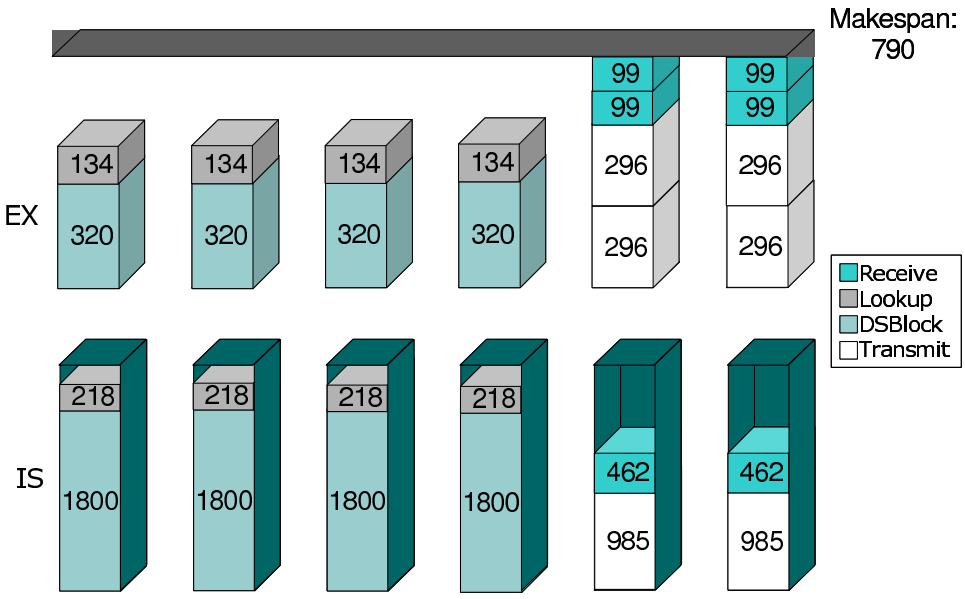

Figure 6.11: DiffServ mapped with a greedy heuristic

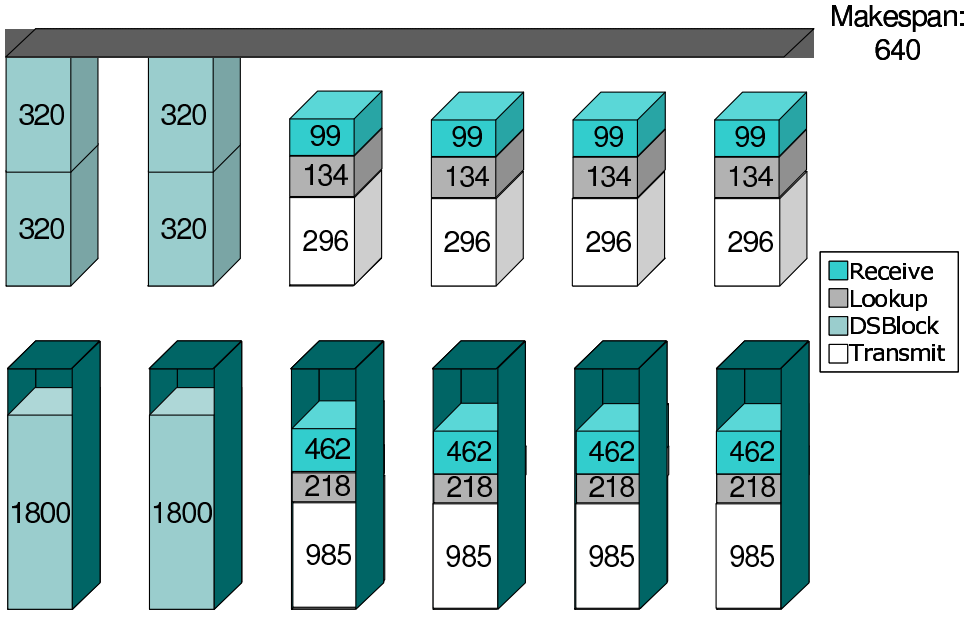

Figure 6.12: DiffServ mapped optimally

Table 6.5: DiffServ mapped to the IXP1200

Application	PE1	PE2	PE3	PE4	PE5	PE6
DiffServ - Hand	4 RX	4 LU	2 DS	2 DS	2 TX	2 TX
DiffServ - Auto	1 RX 1 LU 1 TX	1 RX 1 LU 1 TX	2 DS	2 DS	1 RX 1 LU 1 TX	1 RX 1 LU 1 TX

rates sum to 50% of the line rate and the Best Effort class is 10%. The Expedited Forwarding packet class is varied from from 0 Mbps to 160 Mbps (or 0-40%). Expedited Forwarding is the highest priority traffic flow but it is metered to not consume more than 100 Mbps of the egressing bandwidth. Ideally, the Expedited Forwarding egress rate increases linearly with the increase ingress rate until it is 100 Mbps. After that point the egress bandwidth is ideally flat at 100 Mbps. Regardless of the ingress bandwidth of Expedited Forwarding, the ideal router would hold the other traffic flows at the same egress rate since they are maintaining the same ingress rate. If packets must be dropped, they should first be dropped from the Best Effort traffic class which is the lowest priority class.

Both the hand mapped and the automatically generated design approximate this behavior as shown in Figure 6.13. Furthermore, the egress bandwidth of each packet class is held close to its ideal behavior. The automatically generated design is within 2% the egress bandwidth of the hand-tuned design for each packet class except for Best Effort. At points Best Effort transmits at only one-third the data-rate of the hand-tuned. This is because there is a strict priority scheduler between Best Effort traffic and all other traffic. To account for the discrepancies consider again the two mappings

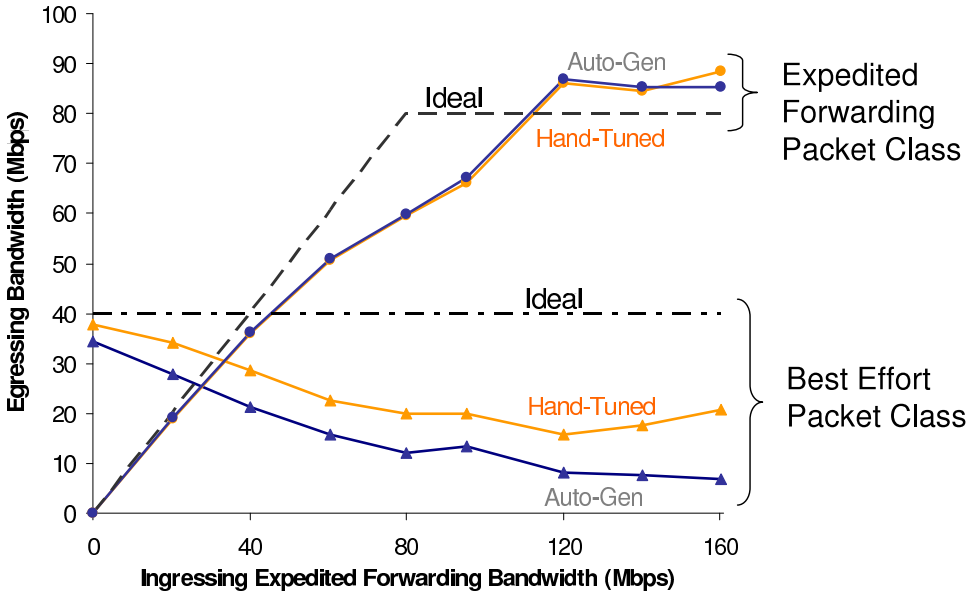

Figure 6.13: DiffServ results comparisons

shown in Table 6.5. The two implementations have the same primary makespan which is determined by the DiffServ Block but profiling them reveals the completion times of Transmit tasks are longer in the automated mapping. In the average case, the DiffServ Block is the bottleneck of the system. But since its execution is dependent on the class of the packet being processed, there may be many system periods in a row where the transmit task is the bottleneck. As Best Effort packets are strictly lower priority, they are not serviced until all backlogged packets from other packet classes are egressed. In this way, the mis-characterization of the Transmit tasks completion times is reflected directly in the Best Effort packets dropped.

The reason Transmit has been mis-characterized is because computation cycles consumed by polling events are ignored in the model. In the automatically gener-

ated mapping, the mapping engine evenly distributes computation after the DiffServ Blocks are placed. This is optimal with respect to the model, but fails to account for the fact that Transmit, the second most compute intensive element, is paired with two elements which spend much of their time polling for new packets. Pairing Transmit with two light weight tasks which poll frequently is worse for its completion time than simply pairing with one other heavy weight task (namely another transmit task). Overall the automated partition is within 5% of the hand mapped aggregate bandwidth for all data points, and was generated in less than a second while the hand-tuned design took days to arrive at.

An astute designer would realize that the model does not incorporate execution cycles from polling. To guide the tool to a better solution, he could constrain the design space to not allow more than one task to exist on the same microengine. To accomplish this the designer inserts an instance of the following constraint for each Transmit task to the mapping engine. With this assistance the tool arrives at the same implementation as the hand mapped solution.

$$N \cdot X_{\text{transmit},p} + \sum_{t \in T \setminus \{transmit\}} X_{t,p} \leq N + 2 \quad \forall\, p \in P \tag{6.1}$$

6.3.3 Network Address Translation

Network address translation is a different functional extension to forwarding described in Section 2.2.2 which we implement on the IXP2400. A Click description

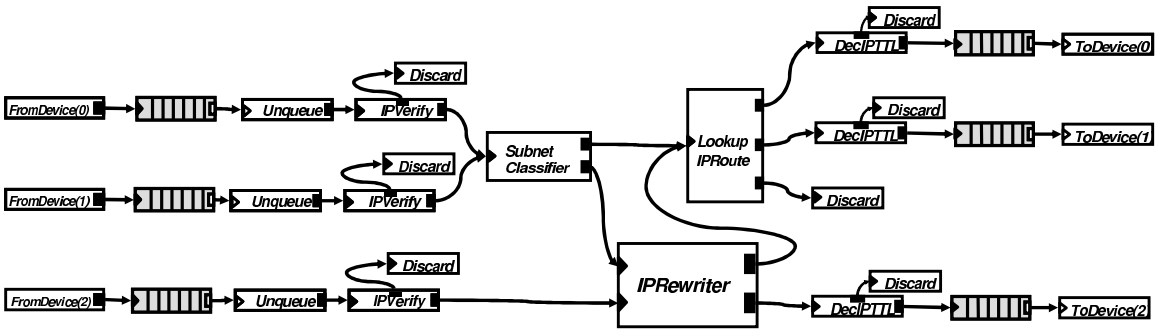

Figure 6.14: NAT Click graph

of a NAT router is in Figure 6.14. Like the forwarder, packets ingress from the left and egress to the right. In addition to the forwarding elements, NAT functionality is incorporate through a *Classifier* and an *IPRewriter*. Ports are divided into two networks: the first two ports are bound to the LAN while the last port is bound to the WAN. Hosts on the LAN share a single WAN address. For those packets that arrive from the LAN, *Classifier* determines the packets staying within the LAN and those going outside it. For a packet destine for a host on the WAN, *IPRewriter* rewrites the source address to the shared WAN address and the source port to a unique port on the NAT router. *IPRewriter* contains a hash table of the state of all active connections.

Figure 6.15 shows the task graph generated from the NAT Click description. It has a similar structure to the forwarder with additions for global data for the hash table and local data for examining the packet header more extensively along with extra execution cycles consumed and latency. The additional computation is reflected in the calculated execution and latency cycles as indicated in Figure 6.6, which is calculated based on the elements profiled using 64 byte packets part of existing flows. The two

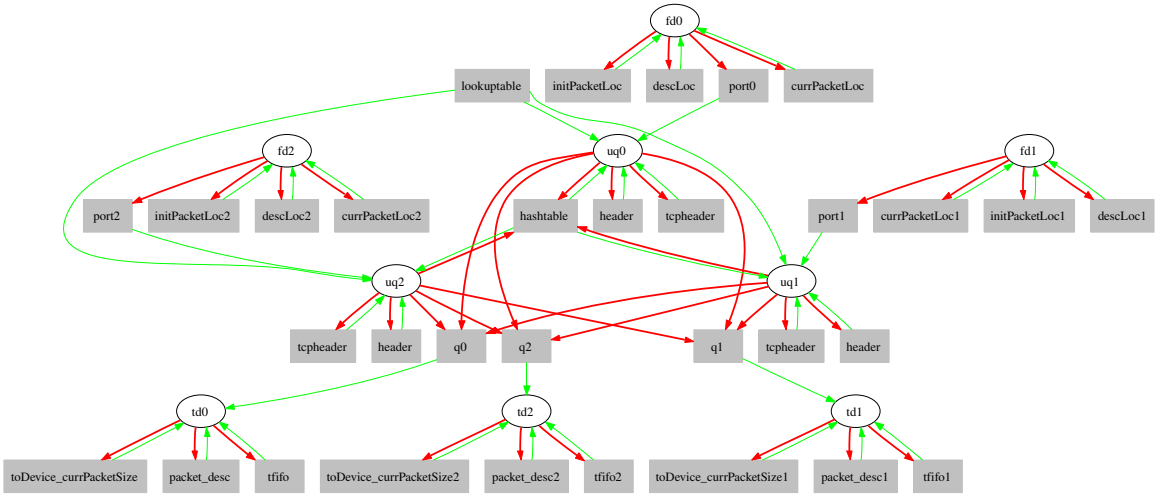

Figure 6.15: NAT task graph

Table 6.6: NAT task characteristics

Element	Number	Execution Cycles	Fixed Latency
Receive	3	143	353
Process Header (LAN)	2	475	387
Process Header (WAN)	1	685	359
Transmit	3	148	402

profiles of the process header task represent the average computation seen by packets originating from the LAN and those from the WAN.

This NAT description was targeted for the IXP2400. Again, it can be seen that task placement done such that non-global memories are utilized for communication channels as shown in Figure 6.16. We also replicated each of the tasks in this application to improve the performance of the implementation. The task graph that results from the manual replication is shown in Figure 6.17. As with IP forwarding, the memory unaware mapping is worse than the original because of data relocated to

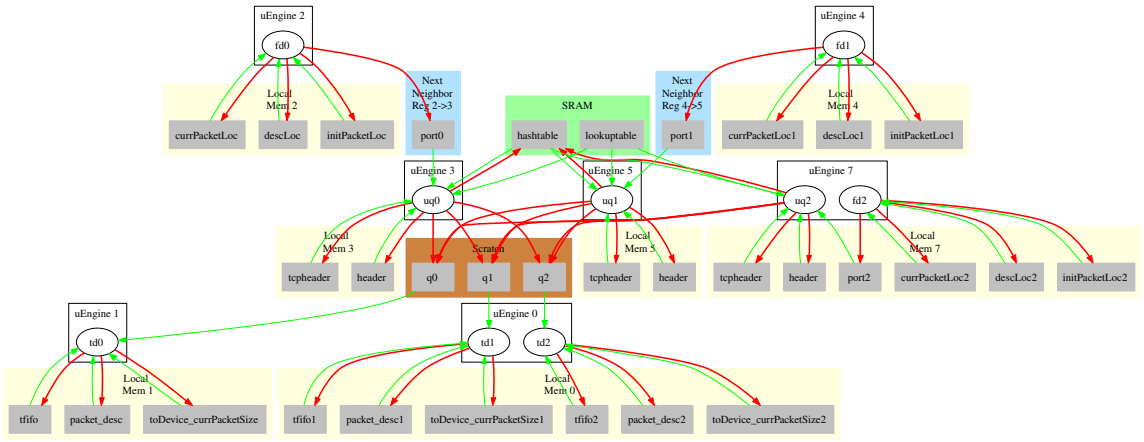

Figure 6.16: NAT task graph

global memory. The memory aware mapping is shown in Figure 6.18.

As with the forwarder we compared with a memory unaware mapping engine shown in Figure 6.19. As with the forwarder, local memories and next neighbor registers are not exploited. Tasks are placed such that when the data are pulled closer to the processing elements, they are unable to use these structures due to a mismatch of application and architecture connections. The results in Figure 6.20 are similar to that of IP forwarding. For the tuned packet, the memory aware mapping has a 7% higher forwarding rate than the unaware. As with IP forwarding, mapping the application without memory consideration creates a design worse than the original unreplicated design. The application must be mapped with memory awareness for the replication to achieve a 122% increase in forwarding rate.

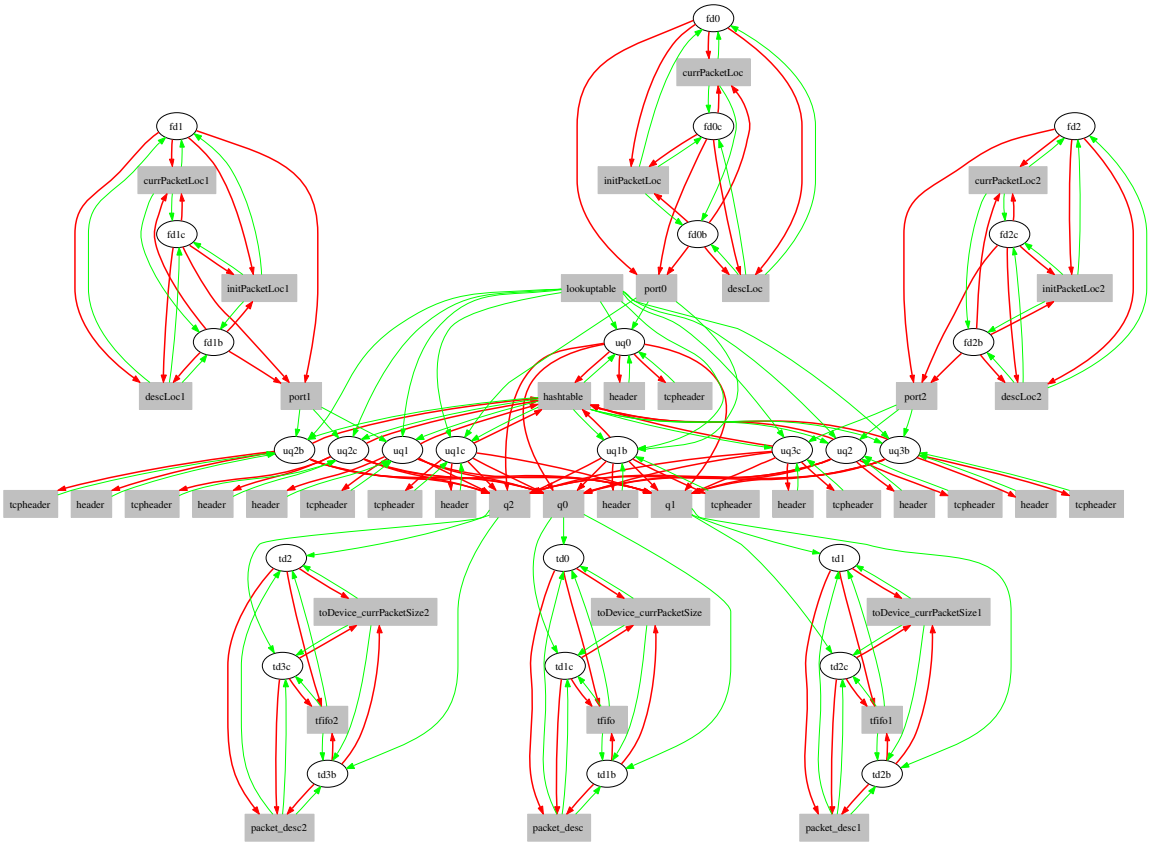

Figure 6.17: NAT task graph with replication

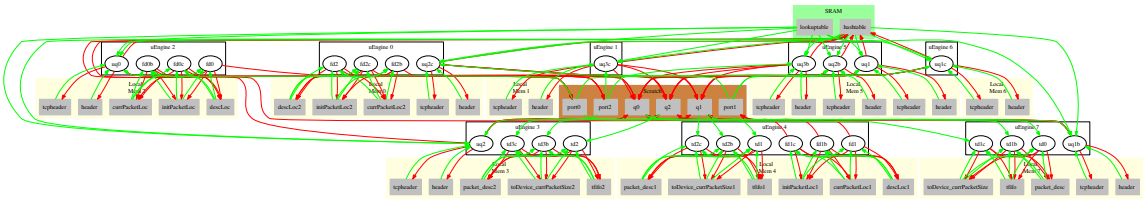

Figure 6.18: NAT forwarder with replication mapped

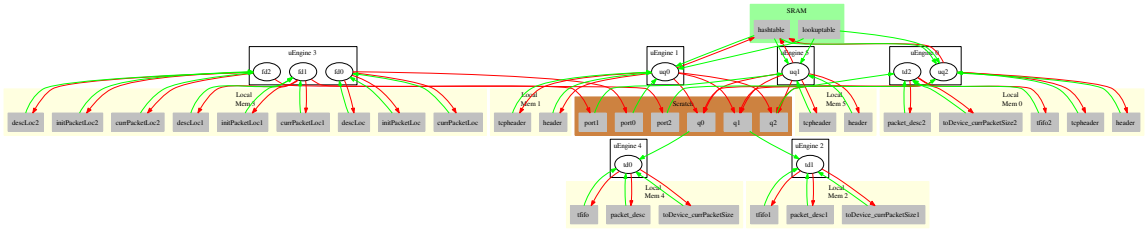

Figure 6.19: NAT forwarder mapped to the IXP2400 without considering memory

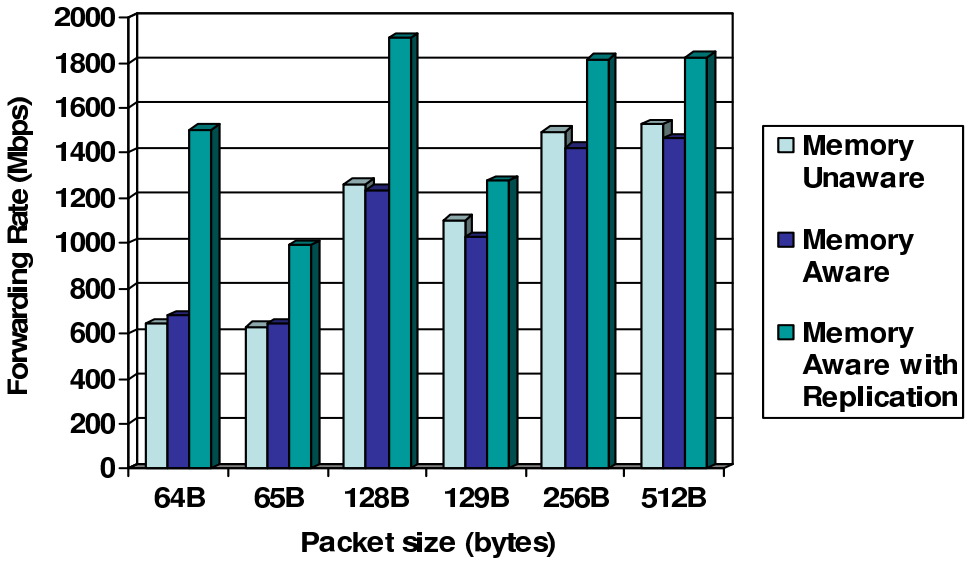

Figure 6.20: NAT forwarding rates vs. different packet sizes for each approach

Comparison to other models

To get a more unbiased comparison to other productive approaches to crossing the implementation, three graduate students used three productivity oriented environments to implement a new application. They reimplemented NAT in TejaNP, PacLang, and the approach present in this work. As with other functionally rich applications, it was not practical in terms of time to implement the application in performance oriented frameworks. Even with Microblocks, a relatively modular approach, the design time would have dominated by adding and modifying library elements. After normalizing out commercial versus research robustness of tools, they arrived at the results shown in Table 6.7. Our approach was able to arrive a more efficient solution in less time. From the estimated breakdown of design time, the biggest differential was spent in the design and implementation phase. Producing an implementation requires designers add functionality to the existing library of elements and map the application to the architecture. Each approach required debugging the synthesized code and each defined the element bodies with C like languages. The time saving comes from the techniques for mapping. In TejaNP programmers must assign each of the tasks and data to architectural resources manually. PacLang allows the user to script of the mapping the application of the architecture, but only our approach provides an automated solution to implementation. Designers can focus their efforts on writing new elements and incorporating feedback into the application description or into guiding the mapping engine.

Table 6.7: NAT implementation comparison of programming environments

	TejaNP	PacLang	Our approach
General Software Architecture	7 hours	0 hours	0 hours
Implementation and Debugging	18 hours	12 hours	5 hours
Optimizing	10 hours	6 hours	5 hours
Total design time	35 hours	18 hours	10 hours
Forwarding rate	1100 Mbps	670 Mbps	1400 Mbps

6.3.4 Web Switch

Web Switching (see Section 2.2.4) is the most complex application attempted in this framework. It uses the widest variety of Click elements and those used are more complicated than any of the other examples. The common processing for a packet from a web client is to first ingress from the WAN. If it is a syn packet indicating it is the start of a new flow, it is duplicated by the *Tee* element. One copy is forwarded to the layer 7 lookup element to be held until the first data packet arrives. The other copy is rewritten into an ack packet to continue the TCP handshake without involving the web server. Subsequent packets contain application data the first of which is scanned for a particular string to determine the appropriate web server. This checks the packet body for a substring in the URL of the packet. It determines which server will be establishing a connection with that packet. *TCPRewriter* keeps the state of the connection, rewriting the packet for a session with a computer behind the web switch. *TCPRewriter* is similar to the *IPRewriter* element except that it rewrites the fields of the TCP header and remaps the sequence numbers for each side of the connection. By managing the TCP handshake and keeping more information

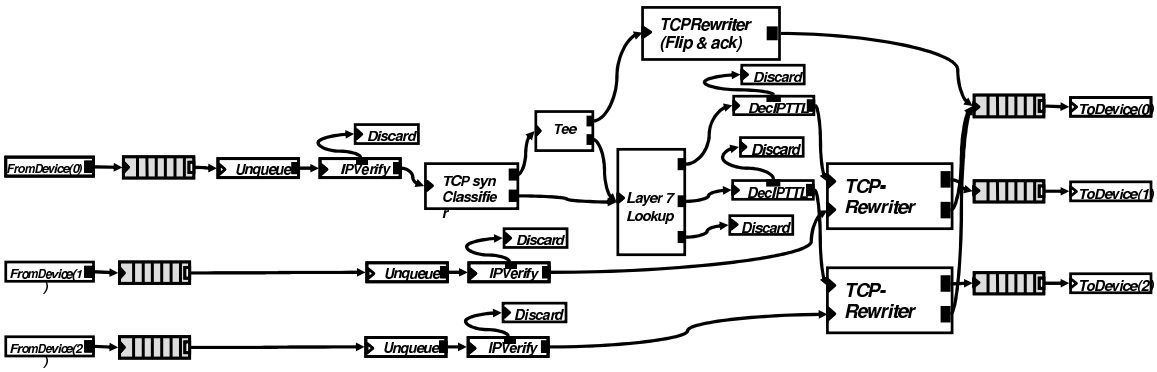

Figure 6.21: Web switch Click graph

about the connection state, the web switch implements even more of the transport layer of the network stack.

The number of elements covered by a single task and the relative complexity of each of the elements leads to a task graph shown in Figure 6.22 that has heavily weighted tasks and many data. When mapped using a memory aware placement the result is shown in Figure 6.23, while an unaware mapping is shown in 6.24. The transformations automatically add the data from the connection state shared between *Layer 7 Lookup* and *TCPRewriter* and for the TCP header memory. The memory aware mapper utilizes a next neighbor register to decrease the completion time of the task with *Layer 7 Lookup* which is the bottleneck in the system.

By avoiding the of data to global memory has, the memory aware mapping improves on the unaware by 7% on the packet size it was tuned for (64 byte) as shown in Figure 6.25. For this benchmark, the memory aware mapping retains its advantage for more packet sizes, as the header processing tasks remain the bottleneck until much

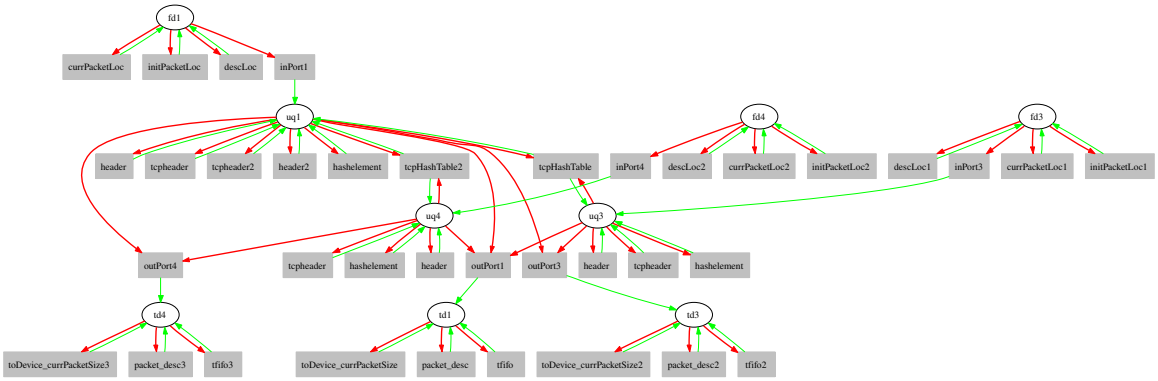

Figure 6.22: Web switch task graph

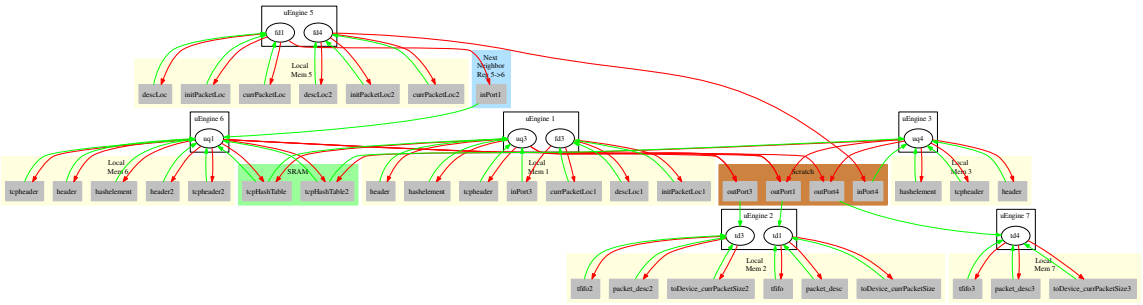

Figure 6.23: Memory aware mapping of the web switch

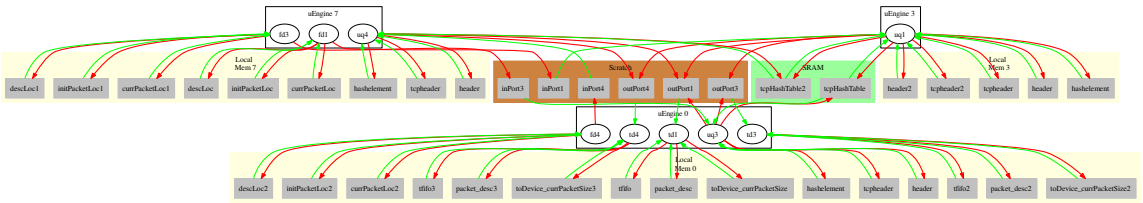

Figure 6.24: Memory unaware mapping of the web switch

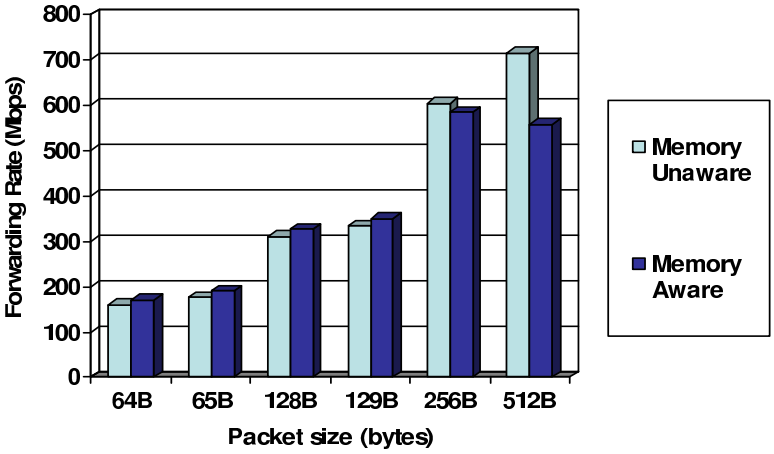

Figure 6.25: Forwarding rate of the web switch

larger packet sizes.

6.4 Summary

The results of this section show that for multiple platforms and representative applications that this design flow is a productive alternative to producing efficient implementations for a commercial application specific multiprocessor. First we found that less time was needed to arrive at an implementation using our design flow than others. While we estimate that using a domain specific language is an order of magnitude more productive than using low-level approaches, we also compared favorably to state of the art approaches that emphasize productivity. Based on the implementation time of network address translation and IP forwarding, application designers are 1.80-3.5x faster at arriving at an implementation for network address transla-

Table 6.8: Runtime summary

Application	Target	Number of Ports	Number of Tasks	Number of Data	Solve time (s)	Opt. Gap (%)
IPv4	IXP1200	16	32	0	< 1	0
DiffServ	IXP1200	4	16	0	< 1	0
IPv4	IXP2400	4	12	37	18.9	4.9
IPv4 MR	IXP2400	4	48	49	738	4.12
NAT	IXP2400	3	9	32	4.79	1.3
NAT MR	IXP2400	3	27	44	557	4.3
Web Switch	IXP2400	3	9	37	3.05	3.9

tion. Across benchmarks our approach was within 17% of hand mapped designs one of which was developed from a performance oriented commercial framework. Furthermore our design flow enabled the construction of a web switch which would not practical to implement efficiently in the other design flows given time constraints. While the automation framework takes time to find a solution provably close to optimal, large examples take only 10 minutes to solve with a 5% optimality gap. Table 6.8 summaries the run-times of the benchmarks. *MR* indicates an application that we have manually replicated.

The results also indicate exact task mapping in the presence of resource constraints even for small applications such as a 4 port DiffServ can provide a 7-19% reduction of makespan over greedy heuristics. Considering memory placement and topology when allocating tasks to processors proved indispensable for applications with tightly connected tasks. If IPv4 forwarding and NAT replication to achieve higher performance, the benefits are lost without memory aware mapping. Applications with loosely coupled tasks benefit by 5-7% increases in forwarding rate. We found the advantage of

memory aware mapping was tied to the packet size that the profiling was performed on. A change in packet size creates different ratios in completion times altering their contribution to the makespan. In particular receive and transmit tasks demand more computational cycles per packet, while header processing remains the same.

Designer guidance aids the DiffServ implementation, allowing the mapping engine to overcome its limitations and find the hand mapped solution. Application level optimizations are important in increasing the forwarding rate of these designs. Software caching is employed in each of these designs and retiming is used to decouple receiving from header processing. Replication is the primary mechanism by which we increased performance of an application. When memory aware mapping is employed, performance increased by 59-122% over the original unreplicated design.

Chapter 7

Conclusions

Application specific multiprocessors are flexible, high performance platforms whose adoption has been inhibited by programming difficulties. Designers have assumed that to properly utilize these devices that they must be programmed at the lowest level. However, the approach described in this work shows that the potential of application specific multiprocessors can be tapped into using a more natural and productive programming approach. By starting from a domain specific language and automatically producing efficient implementations, these platforms can be used for more complex applications by a larger set of application designers. It has positive implications for teams that would use it, but does not radically restructure the design process. We believe the framework proposed is applicable across other application domains. Future work includes adding dynamic support, applying new application level heuristics, and improving the mapping engine time to a solution.

7.1 A Powerful New Design Flow Can Be Achieved

Application specific multiprocessors are capable of achieving high performance implementations by incorporating custom silicon that can utilized by the target application domain. Through special purpose hardware, multiple often multithreaded processing elements, distributed heterogeneous memories, and custom peripherals, applications can be efficiently implemented on these devices. While ASMPs provide a significant performance advantage over general purpose approaches, architects use programmable cores to retain flexibility to accommodate many applications. These advantages should make ASMPs the preferred option for electronic system designers implementing high performance applications, but their adoption has been hindered by the challenge of programming them.

Ideally, programmers would be able to describe their applications in a natural way such as a domain specific language. Describing an application in a domain specific language can be an orders of magnitude more productive than traditional low level approaches such as C or assembly. This productivity boost is a product of component libraries, communication and computation semantics, visualization tools, and test suites tailored to a given application domain. Using lines of code as a proxy for productivity, the largest designs in this work are no more than 200 lines of DSL code, but their C representations are typically around 5,000 lines of code. Despite this distinct productivity advantage, many electronic system designers and programmers assume that to exploit the unique features of ASMPs, they must program at the

lowest level. This work has sought to test this assumption.

We theorized that a productive design flow could be constructed that starts from a domain specific language and efficiently targets an application specific multiprocessor. To this end, we structured our reasoning based on key problems we identified that must be solved to cross the implementation gap as discussed in Section 1.2.1. Prior approaches have improved on low level programming techniques, but have been unable to solve these key problems and provide a complete solution to crossing the implementation gap. To show that a productive, efficient design flow can be constructed, we demonstrate that the following subproblems can be solved with techniques presented in this work.

Extract parallelism from a natural application description

Parallelism must be captured from an application such that it can be exploited on the architecture. *All* of the approaches examined in this work that target ASMPs start from a design entry point that exposes the parallelism of the application explicitly such that it can be used directly for the remaining design flow. The basic form of this is describing the application in a low level C variant, in which the application parallelism is already broken out into multiple sequential C programs. But even productivity oriented approaches create custom design entry languages to ensure that application parallelism can be found directly. The application entry language for Shangri-La (Section 3.3.3), Baker, is written specifically to get the designer to ex-

pose parallelism useful to the remaining flow. TejaNP (Section 2.4.3), designers must describe applications based on state machines. By avoiding existing DSLs for application entry, these design flows implicitly assume that DSL descriptions not tailored for parallelism extraction will produce inferior results.

We test this theory by starting with Click (Section 2.5.1), a popular domain specific language, designed for expressibility and extensibility and *not* for targeting ASMPs. Instead of using the application description directly to provide the parallelism for the design flow, we use an intermediate representation called a task graph (see Section 4.4) which decouples the expression of the application's functionality and the application's parallelism to be exploited by the target. A task graph is a model of the application which exposes the parallelism that may be directly exploited by the architecture. For network processing, a task graph is analogous to the architectural model, exposing the computation, the data, and the communication features of the application. These features may be directly mapped to the corresponding architectural model of processing elements, memories, and the topology. Unlike the custom built design entry languages, task graphs allow for a domain specific language to be platform independent, while still allowing for effective parallelism representation.

Transforming an Click description to a task graph requires only polynomial time algorithms as discussed in Section 5.3. Furthermore we find the utility of starting with a domain specific language representation of the application provides opportunities for optimization difficult to capture at lower levels. In Section 5.4, we described

methods for improving the performance of applications through manual and heuristic techniques. The most beneficial of these produces a task graph capable of performing up to 122% better than the original.

Construct an effective model of the architecture

Network processors have a variety of architectural features to improve the performance of network applications as discussed in Section 2.3. To accelerate applications, programmers may consider a host of architectural details including the speed and instruction sets of the processing elements, the topology of the architecture, the memory structure including size and access time, word sizes, instruction store limits, special purpose hardware, thread scheduling, bus protocols, etc. Prior approaches have tended to one of two extremes. Either the approach exposes all of these features burdening designers with the task of managing them as with low level C variants as discussed in Section 2.4.2. Or the approach abstracts away too many details to utilize these systems effectively, relying on operating systems or hardware to manage these issues. Such transparent management techniques have yet to succeed on network processors as the demands of managing exposed memories, many processing elements, and special purpose hardware have so far proven too great. SMP-Click covered in Section 3.3.1 relies on an SMP Linux platform to manage issues such as load balancing and data location. This makes the approach inapplicable to network processors which have unique topologies, exposed memories, multithreaded processing elements,

and no operating systems running on the dataplane processing elements.

In this research we have built upon prior approaches and tested architectural models formulated mathematically such that they may be mapped to automatically. They are based on our own experience with these architectures and empirical tests of efficiency and automated mapping speed. Instead of relying on programmers or operating systems to manage all of the details, we show that a balanced architectural model does exist that can produce efficient mappings quickly. Presented in this work is an architectural model that allows designers to arrive at solutions within 17% of hand mapped performance oriented designs automatically.

Efficiently map the application to the architecture

After applications have been extracted for parallelism and architectural models constructed, designers must map their applications to these architectures. The mapping should utilize the specific features of the architectures by evenly distributing computation across processing elements. If the architecture has exposed distributed memories, the mapper must take care to locate program data in appropriate memories. Since data placement is often effected by task allocation, the ideal solver should consider each of these problems simultaneously taking care to account for communication bandwidth and special resource constraints. Each of these are performance critical design decisions that may need to be explored extensively in this large irregular design space to arrive at a good solution. While there has been a significant body

of work examining more general problems of application to architecture mapping (see Section 4.3), there was not a solution appropriate for ASMPs. Performance critical architectural constructs such as multithreading and distributed memories are not easily accommodated in existing frameworks. Probably for this reason, existing solutions for network processors rely heavily on designers to derive the mapping. TejaNP requires manual mapping of each task and data. PacLang requires scripting and even Shangri-La espouses a semi-automated approach to finding mapping annotations.

To see if an automated mapping engine could be built to incorporate ASMP architectural constructs along with designer guidance while still having fast runtimes, we used the flexible framework of integer linear programming. We found ILP to be an excellent vehicle for experimenting with different mapping formulations, architectural models, or application models. It was just expressive enough to capture the salient features of the architecture as described in Section 4.5, while still being producing mappings fast. Our final mapping formulation produces efficient results in seconds for most examples and takes only minutes for the largest applications tried. Integer linear programming can capture a variety of specific architectural features. We have used it for quasi-sharable instructions, next neighbor registers, and multithreading, all which are critical to arrive at feasible, high performance solutions. As designers we incorporated feedback directly, pruning the design space explored by introducing custom constraints along side those described in the formulation. Based on the derived mapping, the task graph is divided into the individual programs to be run

the processing elements. A compiler specific to each processor finally produces the executable.

By creating a design flow that focuses on solving the fundamental problems to crossing the implementation gap, we succeed in improving on existing productivity oriented approaches for programming network processors. For a functional rich application we show that the proposed framework takes less time to arrive at and has a higher forwarding rate. The overhead of modularity of this framework also proves to minimal as designs produced perform within 17% of hand mapped designs. As architectures continue to increase the amount of memory with limited scope to architectures, we believe the approach will increase in utility. While other approaches have tackled aspects of improving these design decisions, the work presented here considers the entire flow from an natural description of an application to an implementation on an ASMP.

The proposed design approach improves on the existing design flow described in Section 1.2.2 by automating the manual step of traversing of the implementation gap. Designers will be able to produce implementations faster, increase the number of feedback cycles to the application design, and have fewer errors in the generated implementation. The use of this approach is not likely to change the team structure as there are still many design tasks that justify two separate teams. This is examined in more detail in Appendix B.

7.2 The Design Flow Can Be Applied to a Wider Range of Applications

This work has focused on network processing as it is a mature field with parallel applications and multiprocessor architectures. As performance improvement through frequency scaling continues to diminish, more domains will adopt single chip multiprocessors. New applications will arrive to take advantage of the increased computational power, creating new instances of the implementation gap to cross. We believe the key problems for each application domain will still center around

- extracting parallelism from the application,
- developing a model of the architecture,
- and formulating the mapping problem so that it is fast and produces efficient solutions.

Different applications domains will likely change how each of these are approached. They will have alternative styles of concurrency, different architectural features to model, and different performance requirements. But while some changes in the stages of the proposed framework will be necessary for new application domains, we believe that the framework's structure will be generally applicable to many domains. By creating stages for each of the key problems to crossing the implementation gap, the framework should be applicable to other application areas.

7.3 Future Work is Needed to Extend this Approach to Larger Problems and Other Domains

While this work has demonstrated that more productive paths to implementations exist for ASMPs, deficiencies of this approach have been revealed during the course of this research. Static consideration of applications proves to somewhat limiting in networking. For data dependent execution, considering a single value for task completion time is restrictive. Changes in the traffic profile on a large enough time scale will cause the carefully mapped design to perform suboptimally. A static approach such as the one presented here will not be able to accommodate this. A dynamically adapting system incorporating along with lighter weight techniques for arriving at a new mapping should be more robust for implementations subject to varying loads.

Integer linear programming has been an excellent platform for trying different models, for utilizing guidance, and quickly finding solutions for the examples tried in this thesis. However, its runtime will be an issue scaling to larger designs. As we have experimented with finer granularity and consequently more tasks in the course of this work, runtimes have become prohibitively high. While we have spent time improving the encoding of the formulation, a longer term solution to the mapping problem will involve a more customized mapping engine. The models, constraint based programming, and branch and bound techniques create a solid framework for building this solution, but as long as a general purpose ILP engine is the core mapping

engine, the scale of the problem size will be limited. We believe the time needed to arrive at an efficient solution can be reduced through intelligent heuristics and tuning the general purpose solver used for this problem. Decompositions techniques, variable ordering, and improved lower and upper bounds should allow the ILP engine to scale to larger problems. Scaling to huge instances of applications will likely require some hierarchical solution.

This framework on should also be applied on other application areas as they will present new challenges to crossing the implementation gap. More application level optimizations are possible to further aid the application development process. For instance, if the designer has profiles of traffic or information about the variance in completion time for elements, queue placement and sizing may be more automated. Application level optimizations can further improve the productivity of application designers while improving performance through domain specific knowledge. However, the underlying compiler technology for the specialized cores often employed in ASMPs must also improve so that application level optimizations may be reaped.

Bibliography

[1] ILOG CPLEX. http://www.ilog.com/products/cplex/.

[2] OpenMP. http://www.openmp.org/.

[3] Xpress-NP. http://www.dashoptimization.com/.

[4] *Building ASIPs: The Mescal Methodology*, chapter Inclusively Identifying the Architectural Space: Network Processing Design Space, pages 73–82. Springer, 2005.

[5] Mosek optimization toolbox, 2005. http://www.mosek.com.

[6] "Gartner Dataquest 2002/2003; I.B.S. 2002". Custom edac study.

[7] A. Agarwal, R. Bianchini, D. Chaiken, F. T. Chong, K. L. Johnson, D. Kranz, J. D. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):430–444, 1999.

[8] A. Atamtürk and Martin W.P. Savelsbergh. Integer programming software systems. *Annals of Operations Research*, 140:67–124, 2005.

[9] F. Baker. Requirements for ip version 4 routers. RFC 1812 (Proposed Standard), June 1995. Updated by RFC 2644.

[10] A. Bender. MILP based task mapping for heterogeneous multiprocessor systems. In *EURO-DAC '96/EURO-VHDL '96*, pages 190–197, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[11] Steve Benford and Jacob Palme. A standard for OSI group communication. *Comput. Netw. ISDN Syst.*, 25(8):933–946, 1993.

[12] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated service. RFC 2475 (Informational), dec 1998. Updated by RFC 3260.

[13] Giorgio Calarco, Carla Raffaelli, Giovanni Schembra, and Giovanni Tusa. Comparative analysis of SMP Click scheduling techniques. In *QoS-IP*, pages 379–389, 2005.

[14] Manohar Castelino, Ravi Gunturi, and David Meng. Ixp2400 intel network processor mpls forwarding benchmark full disclosure report for gigabit ethernet. Network Processing Forum, May 2003.

[15] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver.

In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 830–835, New York, NY, USA, 2003. ACM Press.

[16] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *Proc. 40th Design Automation Conference (DAC)*, pages 830–835. ACM Press, June 2003.

[17] C Chekuri. Approximation algorithms for scheduling problems. Technical Report CS-TR-98-1611, Computer Science Department, Stanford University, August 1998.

[18] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor pc router. In *Proceedings of USENIX '01*, pages 333–346, June 2001.

[19] E. Coffman, J. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: Combinatorial analysis, 1998.

[20] K. G. Coffman and Andrew M. Odlyzko. Internet growth: Is there a "moore's law" for data traffic?

[21] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. Technical report, University of Cambridge Computer Laboratory, 2004.

[22] EZchip. *EZchip Introduces First Members of Its NP-2 Network Processor Family*, March 2004.

[23] Niklas En and N. Srensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling, and Computation*, 2, 2006.

[24] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[25] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003.

[26] Sharon Grace. Spending in u.s. telecommunications industry rises 8.9% in 2005 reaching $856.9 billion, February 2005.

[27] H. Vin, et al. A programming environment for packet-processing systems: Design considerations. In *Proceedings of Third Workshop on Network Processors and Applications (NP3)*, February 2004.

[28] J. Hennessy, M. Heinrich, and A. Gupta. Cache-coherent distributed shared memory: Perspectives on its development and future challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):418–429, 1999.

[29] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[30] C-T Hwang, J-H Lee, and Y-C Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):464–475, April 1991.

[31] Intel. *Intel IXA SDK ACE Programming Framework Developer's Guide*, June 2001.

[32] Intel. *Intel IXP1200 Network Processor Family: Hardware Reference Manual*, December 2001.

[33] Intel. *Intel IXP2400 Network Processor Hardware Reference Manual*, November 2003.

[34] Intel. *Intel C compiler for Intel network processors*, June 2004.

[35] Intel. *Intel IXP2800 Network Processor Hardware Reference Manual*, May 2004.

[36] Intel. *Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*, November 2004.

[37] Yujia Jin, Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. An auto-

mated exploration framework for FPGA-based soft multiprocessor systems. In *CODES-05*, pages 273–278, September 2005.

[38] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[39] Eddie Kohler, Robert Morris, Benjie Chen, and John Jannotti andM. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[40] Eddie Kohler, Robert Morris, and Massimiliano Poletto. Modular components for network address translation. In *Proceedings of OPENARCH '02*, pages 39–50, June 2002.

[41] R. Kokku, U. Shevade, N. Shah, H. Vin, and M. Dahlin. Adaptive processor allocation in packet processing systems. Technical report, May 2004.

[42] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *FPGA*, pages 21–30, 2006.

[43] L. Yang, et al. Resouce mapping and scheduling for heterogeneous network processor systems. In *ANCS '05: Proceedings of 2005 ACM Symposium on Architectures for Networking and Communications systems*, pages 19–27. ACM Press, 2005.

[44] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. Comparing models of computation. In *ICCAD*, pages 234–241, 1996.

[45] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[46] The Mathworks. *MATLAB 7*, May 2004.

[47] The Mathworks. *Simulink 6*, May 2004.

[48] Kent R. (Kent Richard) McCord and Steven Daniel. Eppinger. Managing the integration problem in concurrent engineering. Working papers 3594-93., Massachusetts Institute of Technology (MIT), Sloan School of Management, April 2003. available at http://ideas.repec.org/p/mit/sloanp/2484.html.

[49] David Meng, Ravi Gunturi, and Manohar Castelino. Ixp2800 intel network processor ip forwarding benchmark full disclosure report for oc192-pos. Network Processing Forum, October 2003.

[50] Eisenring Michael, Zitzler Eckart, and Thiele Lothar. CoFrame: A modular co-design framework for heterogeneous distributed systems. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, October 1999.

[51] C. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot pages: Software caching for raw microprocessors, 1999.

[52] Trevor N. Mudge. Power: A first class design constraint for future architecture and automation. In *HiPC*, pages 215–224, 2000.

[53] J S Krishna Murthy and Rajeevalochan Ramaswamy. Wipro vpls solution on intel ixp2400 network processor. White paper, 2004.

[54] National Instruments. *LabVIEW*, August 2005.

[55] NEC. *16-Bit D/A Converter with Built-in Digital Filter for Audio*, September 1996. PD63210.

[56] Jens Palsberg and Mayur Naik. ILP-based resource-aware compilation. In *Multiprocessor Systems-on-Chips*, chapter 12. Elsevier, 2004.

[57] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. Updated by RFC 950.

[58] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, USA, 1989.

[59] Christian Sauer, Matthias Gries, and Sören Sonntag. Modular domain-specific implementation and exploration framework for embedded software platforms. In *DAC '05*, pages 254–259, New York, NY, USA, 2005. ACM Press.

[60] H Shachnai and T Tamir. Polynomial time approximation schemes for class-constrained packing problems. In *Proceedings of Workshop on Approximation Algorithms*, pages 238–249, 2000.

[61] Niraj Shah, William Plishker, Kaushik Ravindran, and Kurt Keutzer. NP-Click: A productive software development approach for network processors. *IEEE Micro*, 24(5):45–54, September 2004.

[62] A. Srinivasan and et al. *Network Processor Design: Issues and Practices*, volume 2, chapter Multiprocessor Scheduling in Processor-based Router Platforms: Issues and Ideas, pages 75–100. Academic Press, Inc., Orlando, FL, USA, 2003.

[63] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), jan 2001.

[64] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE Transaction on Engineering Management*, EM-28(3):71–74, August 1981.

[65] Teja. *Teja C: A C based programming language for multiprocessor architectures*, October 2003.

[66] Martin Trautmann. Improving application design for embedded multiprocessors with automatic design space exploration. "Diplom"-thesis, Universität Karlsruhe (TH), Germany, August 2006.

[67] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[68] Charles Waltner. *A New Era for Communications Begins with CRS-1*, May 2004.

[69] Lars Wehmeyer, Urs Helmig, and Peter Marwedel. Compiler-optimized usage of partitioned memories. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 114–120, New York, NY, USA, 2004. ACM Press.

[70] Ben Wun, Jeremy Buhler, and Patrick Crowley. Exploiting coarse-grained parallelism to accelerate protein motif finding with a network processor. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 173–184, 2005.

Appendix A

Acronyms

ASIC - Application Specific Integrated Circuit - Silicon customized for an application or application domain that is designed and fabricated for a particular company. that may implement arbitrary logic.

ASMP - Application Specific Multiprocessor - A single chip multiprocessor customized for an application domain. A subset of ASSPs

ASSP - Application Specific Standard Part - Silicon customized for an application or application domain that is sold as an off-the-shelf part.

DSL - Domain Specific Language - A design language tailored to an application domain.

FPGA - Field Programmable Gate Array - Off-the-shelf reconfigurable silicon that is based on interconnected lookup tables

ILP - Integer Linear Programming - A constraint based method for describing a design

space with a set of linear inequalities over variables confined to the integer domain.

IPv4 - Internet Protocol version 4 - The common protocol for network layer packet communication.

LAN - Local Area Network - A small network of computers that would typically service an home or an office.

NAT - Network Address Translation - An edge application that enable communication between networks with conflicting address spaces.

NRE - Non-recurring Engineering - The one time cost of designing a new product.

OSI - Open Systems Interconnection - A standard which describes a typical network stack.

PHB - Per Hop Behavior - The operations performed in the interior of a network to a packet.

TCP - Transmission Control Protocol - A network protocol that gaurantees in order, lossless delivery.

UDP - User Datagram Protocol - A lossy network protocol

WAN - Wide Area Network - A large network that can support many hosts.

Appendix B

Design Structure Matrix

Design Structure Matrices (DSMs) are a structural way of laying out a design process for analysis [64]. Design *tasks* are placed on the edges of a two dimensional matrix and an entry in the matrix represents dependency or communication between tasks. The information in the DSM includes the dependencies and the concurrencies of the design process such that they can be observed directly. Figure B.1 shows examples of the various relationships that can be expressed by a DSM. **A** and **B** represent two tasks in a design process. If there are no tick marks in the boxes common to the rows or columns of each task, then there is no dependency between the two tasks, so they may execute in parallel. A tick mark in the lower left half of the matrix indicates that **B** is dependent on **A** and cannot begin until **A** completes. A tick mark in the upper right part of the matrix represents a feedback relationship meaning that an iteration will occur from **B** to **A**. The presence of two tick marks

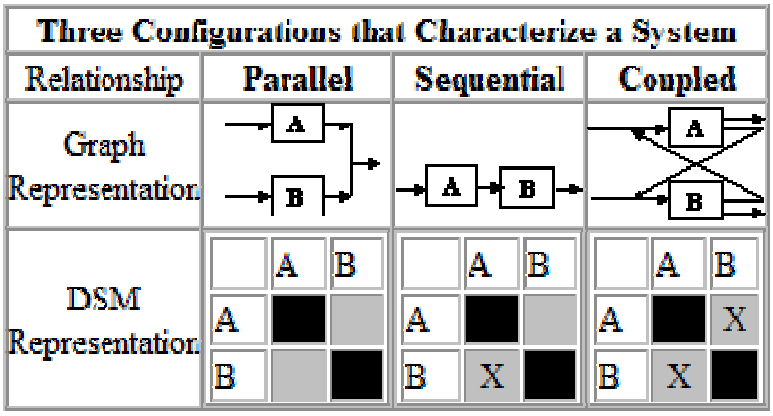

Figure B.1: An introduction to design structure matrices

indicates the tasks are coupled. They feedback to each other to produce the desired result.

DSMs can be used to derive a good team structure for a design process [48]. Small teams are effective at completing tightly coupled tasks as the amount of coordination overhead for a small team is low. Large teams are able to complete more tasks, but they require more coordination overhead between the team members and therefore operate less efficiently than a small team. When determining team structure to implement a set of design tasks, the goal is to create small teams with little overlap. We apply this idea to the design process discussed in Section 1.2.2. Based on our experience with such design processes, to arrive at an implementation using the design flow in Figure 1.5 the following tasks must be done:

- Define inputs and outputs of the application. This describes the type of data that will be received and what form the output will take.

- Define functionality of the application. Designers must decide on behavior of the application on any given set of valid inputs.

- Sketch kernels. If the application has performance sensitive computation, designers should specify how that computation should be performed.

- Set performance goals. Based on estimates from the kernels and the demands of the application domain, designers have performance expectations for the application.

- Generate test vectors. When the implementation is complete, a set of inputs and what the outputs should be when the inputs are applied are supplied by the application designers for validation.

- Choose target. The platform that will implement this application that will satisfy performance needs at the lowest cost must be decided on.

- Create testbench. Technical infrastructure with the chosen platform must be designed to inject the test vectors into the target platform and validate the results.

- Map application onto target. For application specific multiprocessors designers must settle on a mapping of the application to it.

- Code application on target. Using the programming environment programmers

must implement the functionality of the application based on the mapping assignments.

- Profile implementation. Using the testbench and the implementation designers can evaluate the performance of the application with the current implementation. This information is fed back to various parts of the process to improve or refine the result.

These tasks and their dependencies are captured in a DSM in Figure B.2 along with the common team structure used to perform them as described earlier. While the team coverage is well suited to the tasks, there are many feed-forward and feedback points between the two teams. This inter-team communication is indicative of the long process of converting the original application description to a form that may be implemented on the target.

By employing our proposed approach, we believe the design flow would change by removing the "Code application on target" and "Map application onto target" steps. These would be replaced by:

- Generate Implementation. The mapping step is automated so that the algorithm team may generate mappings with or without the assistance of the implementation team.

- Code new elements. The entire application is not coded monolithically. Instead programmers need only write library elements that do not exist for the current

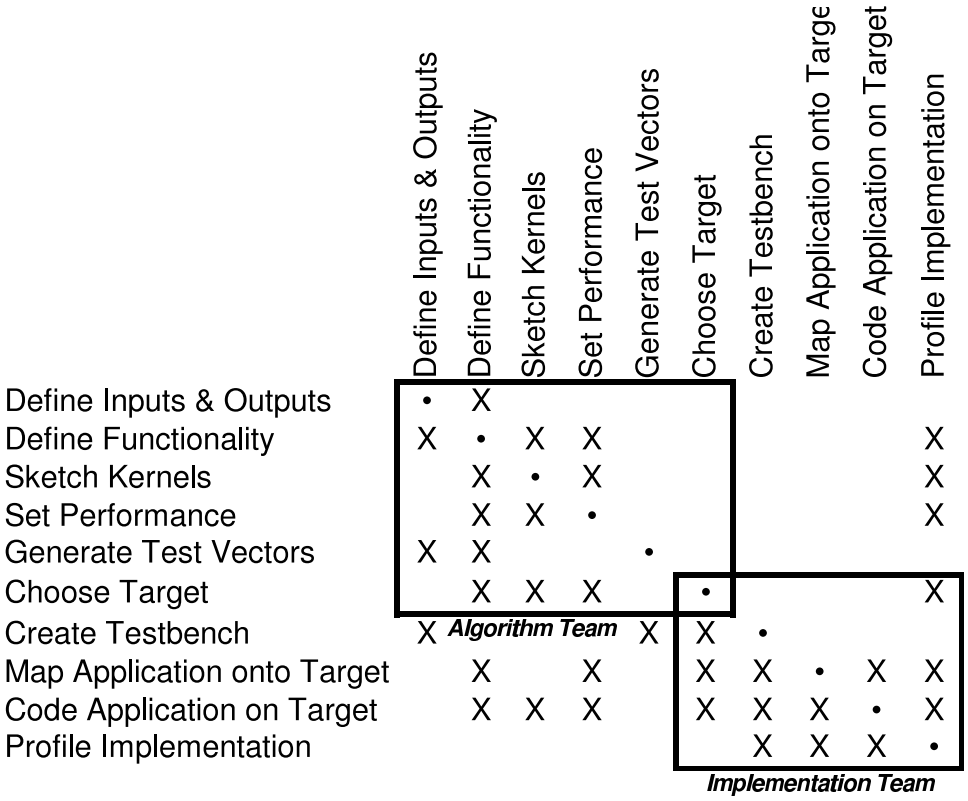

Figure B.2: The design structure matrix of the existing design flow

application.

- Refine Mapping. The implementation team will have insights into the platform that the tool will not, so this step allows for that designer guidance.

By removing the two of the original design tasks, the inter-team communication has been reduced. Using a domain specific language with our tool flow enables the application team to generate implementations and make immediate adjustments based on profiling. The net result is a faster design flow less encumbered by inter-team communication as shown in Figure B.3. However, introducing the proposed tool to the process does not eliminate the need for two teams. There are enough other activities that are tightly coupled to merit a two team structure instead of one large team to do the entire implementation. For that to happen, the process of creating new target specific library elements and new test benches must also be automatically generated. Also profiling would need some way of backward annotating into the original application description to guide application changes. Each of these represents a difficult set of challenges to a fully integrated design flow.

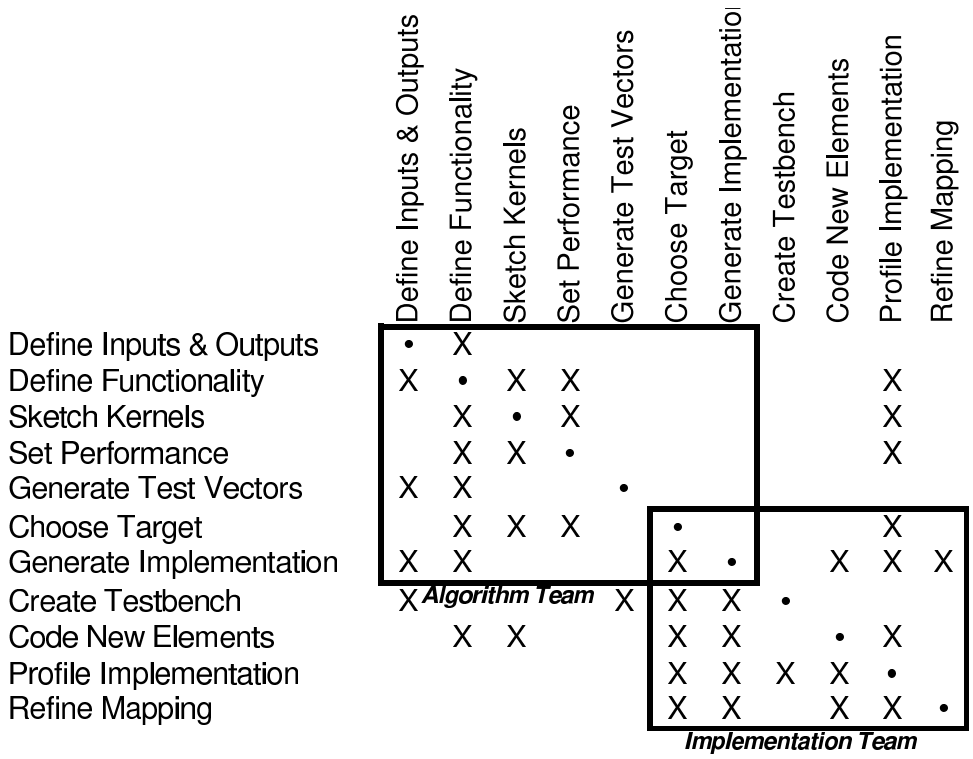

Figure B.3: The design structure matrix of the design flow enabled by this work