FUNCTION/ARCHITECTURE OPTIMIZATION AND CO-DESIGN OF EMBEDDED SYSTEMS

by

Bassam Tabbara

Memorandum No. UCB/ERL M00/21

17 May 2000

FUNCTION/ARCHITECTURE OPTIMIZATION AND CO-DESIGN OF EMBEDDED SYSTEMS

by

Bassam Tabbara

Memorandum No. UCB/ERL M00/21

17 May 2000

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Function/Architecture Optimization and Co-design of Embedded Systems

by

Bassam Tabbara

B.S. (University of California at Riverside) 1994
M.S. (University of California at Berkeley) 1998

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 at BERKELEY

Committee in charge:

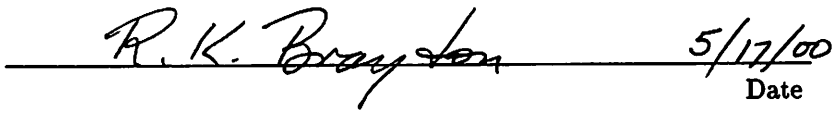
Professor Alberto L Sangiovanni-Vincentelli, Chair
Professor Robert K Brayton
Professor A Richard Newton
Professor Shmuel S Oren

Spring 2000

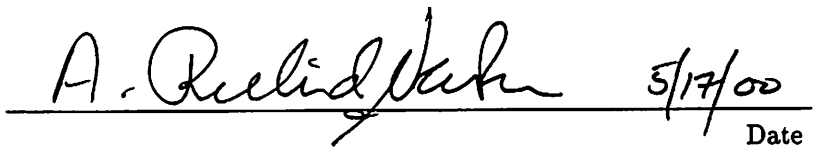
The dissertation of Bassam Tabbara is approved:


5/15/00

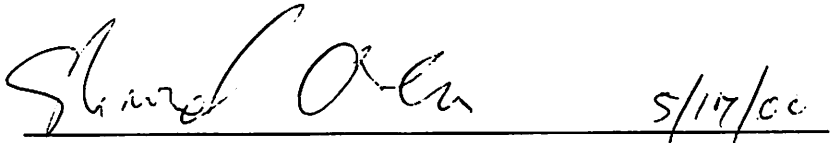
Chair Date


5/17/00

Date


5/17/00

Date


5/17/00

Date

University of California at Berkeley

Spring 2000

Function/Architecture Optimization and Co-design of Embedded Systems

by

Bassam Tabbara

Abstract

Function/Architecture Optimization and Co-design of Embedded Systems

by

Bassam Tabbara

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Alberto L Sangiovanni-Vincentelli, Chair

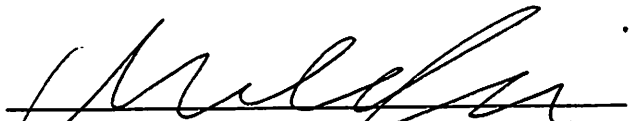
Embedded systems are very prevalent in today's society and promise to be even more pervasive and found in many of the things we interact with in our daily lives in the future. These various and abundant applications not only require that the system be reliable and cost-effective, they impose constraints on the hardware and the software components. Invariably, the system must be *efficient* that is speed of execution of the software, and performance of the hardware must be adequate; and it must be *small* in size if it is to fit seamlessly in common objects; in other words code size of the software must be small, and the area of the silicon for hardware must be within bounds.

Current co-design methodologies of control dominated embedded systems focus on design at the hardware (HW) and software (SW) component abstraction level, and deal primarily with improving design productivity. Such approaches have a very limited view of the design problem and thus suffer from biased trade-off evaluation and limited optimization

opportunities resulting in inefficient HW and SW synthesis of the various reactive system tasks.

Function Architecture Co-design is a new paradigm we proposed in recent years for the design and implementation of embedded systems. I present my work in developing a function/architecture *optimization* and *co-design* formal methodology and framework for control-dominated embedded systems that incorporates both data flow and control optimizations performed on a suitable novel intermediate *design task representation* in order to not only **enhance productivity** of the designer and system developer, but also **improve quality** of the final synthesis outcome.

I discuss here the function/architecture co-design methodology, focusing on design representation, optimization, and synthesis. I show that performing data flow and control optimizations at the high abstraction level can lead to significant size and performance improvements in both the synthesized hardware and software.



Professor Alberto L Sangiovanni-Vincentelli
Dissertation Committee Chair

To my parents,

the guiding light of my life.

Contents

List of Figures

List of Tables

Acknowledgements

I want to thank my advisor Alberto for his supervision and support, Alberto's insight into the system level design field has shaped much of my personal views on the subject. I am deeply indebted to both Bob and Richard for their excellent input on this work, and general advice, and to Professor Oren for his feedback.

I appreciate the help of all the professors who have nurtured my research career over the years, in particular Jan Rabaey from UC Berkeley, Jing Wang, Yu-Chin Hsu, and Ping Liang from UC Riverside, and Tim Usher from CSU San Bernardino. I am very grateful to my brother Abdallah for the joint work and the numerous lively discussions we have had in system level design, and synthesis for Deep-Sub Micron (DSM). My thanks to the students of EECS249 who I mentored in Fall 1999 for some of the experimentation, and useful discussions that also made it into this work in some form.

Special gratitude and recognition go out to my mentors Felice Balarin and Luciano Lavagno from Cadence, and to my good friends Attila Jurecska from Synopsys, Alan Taylor, Lisa Guerra, and Dipankar Talukdar from Conexant, and Vojin Zivojnovic from Axys Design Automation.

I'd like to thank the people who have made it all possible: The University of California public education system in particular the Berkeley and Riverside campuses, the Semiconductor Research Corporation (SRC) that has supported my research under the Graduate Fellowship Program, and industry supporters, in particular Cadence Design Systems, and Conexant who have provided me with much needed feedback about my work and research endeavors.

Finally, I would also like to acknowledge the Max-Planck-Institut fuer Informatik, Saarbruecken for granting me a LEDA research license, and Microsoft Research for the MSBN license.

Part I

Introduction and Background

Chapter 1

Introduction

Embedded systems are prevalent in today's society and promise to be even more pervasive and found in many of the things we interact with in our daily lives in the near future. Applications vary from today's airplane jet or car controllers, and communication devices like cellular phones and pagers to the future's autonomous kitchen appliances, and intelligent vehicles. The mega trend in semiconductor industry is that the Internet and e-commerce will change our lives and impact the semiconductor industry even further; the consumer e-commerce estimated at 31 Billion U.S. dollars in 1999 is expected to rise to 400 Billion in 2003 [48]. Much of the economic growth in the semiconductor industry, estimated at 58 percent annual improvement in usable transistors and a greater than 20 percent annual decrease in cost, stems from this consumerization of electronics which has created entirely new markets as functionality and affordability continue to improve [104]. Some of the main drivers of the consumer markets are the digital TV and the cellular phone. For the latter, Dataquest forecasts that while 230 million handsets were shipped in 1999, 550 million are

expected to in 2003. Looking at the semiconductor market growth (1997 - 2002), the top 10 semiconductor applications are the Internet phone, DTV, xDSL modem, DVD player, cable modem, automobile GPS, hand-held PC/Companion PC, chip cards, LAN switch, and workstation; most are consumer and communication applications driving the market. Electronics are also penetrating the automobile market in increasing numbers[1]. For example, the electronic content of an automobile in 1998 was $852, including $153 of semiconductor components. By 2002, the electronic content will increase to $1,105 with a semiconductor content of $222. At the high-end, an S-class Mercedes can have $750 of semiconductor content and up to 100 micro-controllers. In this automobile domain, electronics are replacing mechanical systems, augmenting mechanical systems and adding increased functionality to vehicles. Therefore, demand for improved processing of information is bound to increase in wireless intelligent transport systems.

These various commercial applications not only require that the system be reliable and cost-effective, they impose constraints on the hardware and the software components of the system. Invariably, the system must be *efficient*, that is speed of execution of the software, and performance of the hardware must satisfy budgeted constraints, and the system must be *small* in size if it is to fit seamlessly in common objects; code size of the software must be small, and the area of the silicon for hardware must be within bounds. Designing such an embedded system involves complex trade-offs among the traditional design metrics such as size, power, and performance, in addition to other metrics specific to the embedded domain such as cost and reliability. Sometimes metrics particular to the target application itself are most important, for example in a car the exhaust emission and

[1]Data collected by Dataquest; paraphrased from Grenier in [48]

fuel consumption are often of primary concern in the design of the drive controller.

1.1 Motivation

Embedded applications are demanding more and more "intelligence" on board either in the form of information *processing* power, or information *translation* for man-machine interfaces such as touch, speech, and vision based input schemes [62]. This means that the division between control and data is getting even more blurred in this domain. No more is this apparent than in future wireless applications such as the ones being developed at the Berkeley Wireless Research Center [14], and the Gigascale Silicon Research Center [45]. Such systems have a large set of constraints and demand considerable analysis. Figure 1.1 presented by Jan Rabaey at a recent GSRC workshop [96] outlines the design challenges of the *PicoRadio*, a distributed energy efficient multi-hop wide-scope communication infrastructure.

The increasing complexity of designs in general and embedded systems in particular, as demonstrated by BWRC's PicoRadio, is responsible for the widening of the gap between complexity and designer productivity as shown in Figure 1.2 from data collected by SEMATECH in recent years [101]. Design productivity, however, should not be improved at the expense of quality; application demands and market pressures not only require that the product design cycle be fast and correct-in-the-first-time, the final implementation must be reliable, cost-effective, and of good merit. Quality of a product (size, performance, energy consumption, reliability) is also a crucial factor in the success or failure in a competitive market.

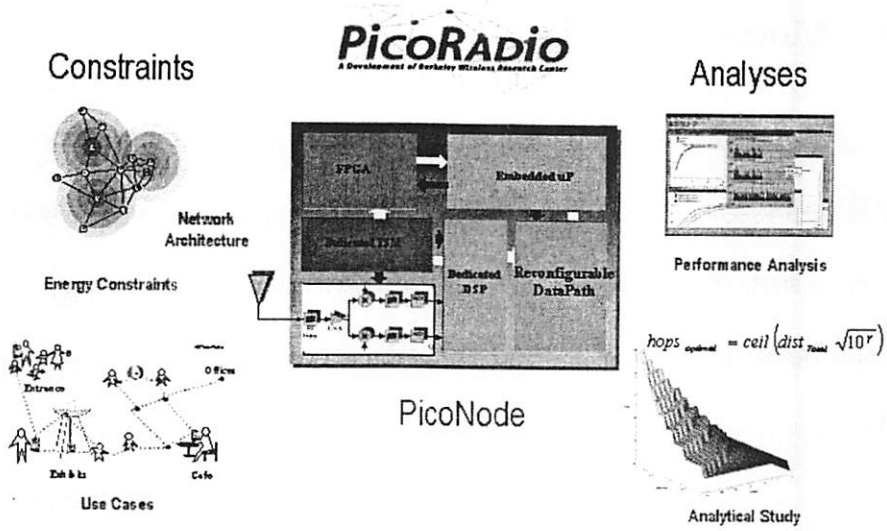


Figure 1.1: Future Designs: The PicoRadio and Its Design Challenges *(Courtesy Jan Rabaey, BWRC)*

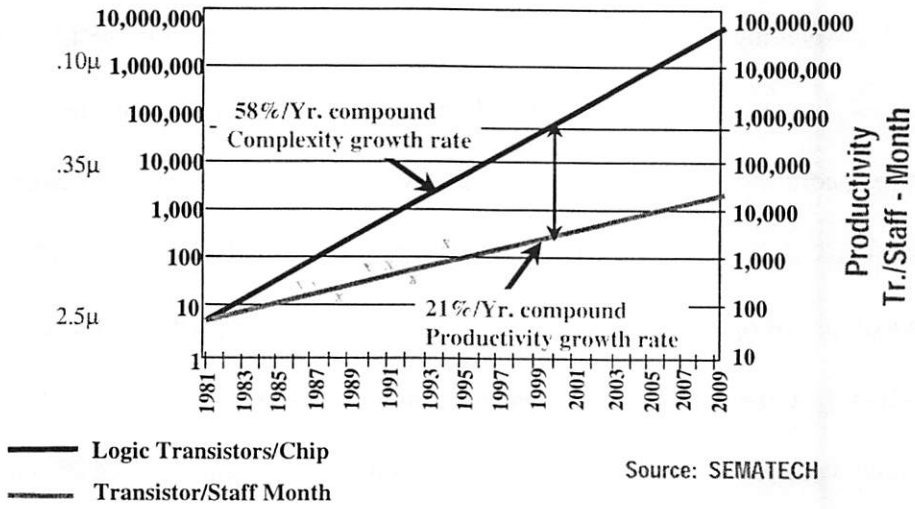


Figure 1.2: New Design Start Crisis: The Productivity Gap

Design complexity is on the rise, and we seem to be approaching physical reliability limits in conventional technologies. Our recourse, in my opinion, must be a renewed emphasis on design methodology. The costs of high-tech fabrication are rising quickly and only companies that truly understand the increasingly critical role of design will be able to effectively compete in the coming era [104]. To support my opinion, I ask you, the reader, to consider the following. If we assume our goal in design is to *keep the fabrication lines going* [89] with *quality* products then I argue that there is a direct analogy between chip fab lines and vehicle assembly lines. I make this analogy not only because of the similarity in terms of a fabrication and assembly line as well as the addition of features (in cars and ICs) over generations but because the car industry is often credited with the *mass production* revolution at the turn of the 20th century, as well as the widespread successful implementation of the concept of *interchangeable parts*[2]; two notions that I see a congruence among EDA visionaries from industry and academia on the need to incorporate into design; these perceptions usually go under the guise of: design re-use, and component-based design.

The vehicle making industry has been active since before the 1900's, a good half century before the invention of the transistor, so we would do well to learn from its experiences and trials. Abernathy in the late 1970's, right about the time when the car industry was quite mature and to take the analogy further as mature as the IC industry is now at the turn of the 21st century[3], performed a study on the productivity dilemma then faced by car manufacturers. Abernathy argues, by analyzing car engine and part design, manufacture and assembly plant data from 1900 till the mid 1970's from U.S., European, and Asian man-

[2]Eli Whitney (1765-1825) is the originator of the idea during the industrial revolution of the 18th century [47].

[3]Roughly speaking, without comparing growth rates

ufacturers, that the design and manufacturing processes have been improved throughout that period by one of the following four innovation functions [1]:

1. Introducing new process capabilities,

2. organizing the process,

3. integrating an existing process, and

4. improving the overall process as a system.

Methodology in embedded system design is an evolving process. Metaphorically speaking, methodology is the gene carrier or, more accurately, methodology embodies the memes[4] in the Dawkins terminology, information carriers of the evolving ideas [105]). We cannot focus on just churning out chips using conventional design techniques; methodology improvement in the form I stated above (new approaches, re-organization, and new integration methods) must be on-going if we are to ensure success in our design activities, keeping in mind that our goal is not just to produce (survive in the Dawkins metaphor), but rather be able to generate quality products (good offspring in the biological metaphor). To take the evolution metaphor and this line of reasoning further, Sabel and Zeitlin argue in [99], after analyzing the early evolution of western industrialization in the 1800's and 1900's, that adjustment proceeds in economy by *adaptation* rather than "natural selection" since we humans are sentient beings, and using our wits can devise the (direct or indirect) means to meet our needs. We developers and designers must also adapt our design methodology in a strategic fashion so that it tracks, and maximizes the benefit from, the improvements

[4]From the Greek root "Mimeme" for *imitation*

in IC processing and manufacturing if we are to subsist in the increasingly complex design world.

1.2 Research Overview and Objective

Current co-design methodologies of control dominated hardware software systems focus on design analysis, validation, optimization, and synthesis at the hardware (HW) and software (SW) component abstraction level. Such approaches have a very limited view of the design problem and thus suffer from biased trade-off evaluation, limited optimization capability and consequently inefficient HW and SW synthesis of the various reactive system tasks.

Typical hardware and software co-synthesis methodologies of control dominated embedded systems focus primarily on improving productivity in the complex design process. In most cases they rely on a *low level* Control Data Flow directed acyclic Graph (CDFG) internal reactive task *representation.* This representation varies in form from one tool to the next but being at this low abstraction level, this representation limits control and data flow analysis and optimization to just manipulating computations along the paths of the CDFG without considering optimizations across such paths in the task as a whole [115]. In essence, the representation allows *micro*-architectural trade-offs to be evaluated, and leaves *macro*-architecture and its interplay with the function largely unexplored.

Function Architecture Co-design is a new paradigm we proposed in recent years [5] for the design and implementation of embedded systems. I elaborate in this document on my research in, and development of, a function/architecture analysis, optimization,

and co-design framework for control-dominated systems that incorporates both *data flow* and *control* analyses and optimizations, performed on a novel *high level* implementation-independent task *representation*, in order to improve synthesis quality in addition to designer productivity.

The approach is applicable to any co-synthesis tool; I have incorporated this function/architecture optimization and co-design approach in the public domain co-design environment Polis [94]. The data collected shows that performing such optimizations leads to considerable size and performance enhancements in both the synthesized hardware and software. It is my hope that the methodology introduced in this work, as well as my implementation of the majority of these concepts can set the groundwork for a new breed of tools that raise the abstraction level of the current day tools, and that this dissertation can serve to document the issues that must be considered in the methodology, the algorithms that can be leveraged, and the techniques and approaches that are likely to become the pillars of future co-design flows [100].

1.3 Dissertation Contribution

My dissertation builds on significant bodies of work in the optimization, high level synthesis, and software compilation domains among others. My main theoretical and methodological contribution is the proposal and realization of a novel **formal function/architecture optimization and co-design methodology and framework** that captures the notions of *abstraction*, *decomposition*, *refinement*, and *architecture-driven targeted and constrained synthesis*. The design method revolves around a theoretical formal

analysis and optimization framework. The tangible contribution is the **mutually reinforcing optimization and co-design strategy** for control-dominated embedded system design which encompasses several new architecture-independent and architecture-dependent guided (constrained) optimization for synthesis algorithms.

My innovations in the co-design process, presented and discussed in the dissertation, can therefore be divided into three major inter-dependent categories[5]:

1. *Function/Architecture Organization and Representation* of the embedded system design process enveloping specification, analysis, and co-design.

2. *Introduction of Optimization* comprised of both data flow and control analysis, improvement, and specialization *into Co-design* at all levels of architectural constraint abstraction, and design refinement thus bolstering the function/architecture co-design approach.

3. *Improvement and (Forward and Backward) Vertical Integration of the Overall Co-design Process*: where conventional hardware/software co-design techniques are augmented, integrated and consolidated with the function/architecture optimization and co-design method.

In the next Chapters, I will first lay the foundation for the function/architecture representation, then discuss how I have introduced control and data flow optimization into the: functional (architecture-independent), architecture constrained functional, and finally the *macro-*, and *micro-* architectural abstraction levels. Before presenting results, and concluding, I establish vertical integration of the trade-off analysis and optimization method

[5]Borrowing the spirit of Abernathy's classification [1]

with the previously known co-design techniques to form the improved function/architecture optimization and co-design framework for embedded system design set forth by this dissertation.

1.4 Dissertation Roadmap

In Part I of the dissertation I provide background, and then elaborate on the research problem, my formulation, methodology and solution in Part II. The function/architecture optimization and co-design flow, experimental results, and conclusions are presented in Part III, as well as my conclusions, and an outline of opportunities I see for future research and development.

Chapter 2

System Level Design of Embedded Systems

2.1 The Application Domain, and Design Tools

Embedded systems are informally defined as a collection of programmable parts surrounded by ASICs and other standard components, that interact continuously with an environment through sensors and actuators [5]. In today's world there is a wide proliferation of such electronic devices in everything from tea kettles to life-critical systems. Up till very recently, embedded systems have been designed in an *ad hoc* fashion based on manual interference and guidance. With increasing complexity, formal methodologies that incorporate HW/SW trade-off analysis and evaluation, and validation at the highest possible abstraction level have become essential. Obviously, an overhead is incurred in this top-down process: quality of the final output is typically traded-off with increased pro-

ductivity, but as I will show, this overhead can be again managed and put within bounds if the methodology includes constraint-driven optimizations; the subject of the upcoming Chapters.

2.2 Embedded System Design

The broad areas and concerns of the embedded system design methodology are shown in Figure 2.1. I will cover and discuss in this work mainly the following major areas: Design Representation, Evaluation (of the optimization, trade-off, and co-design), and Synthesis.

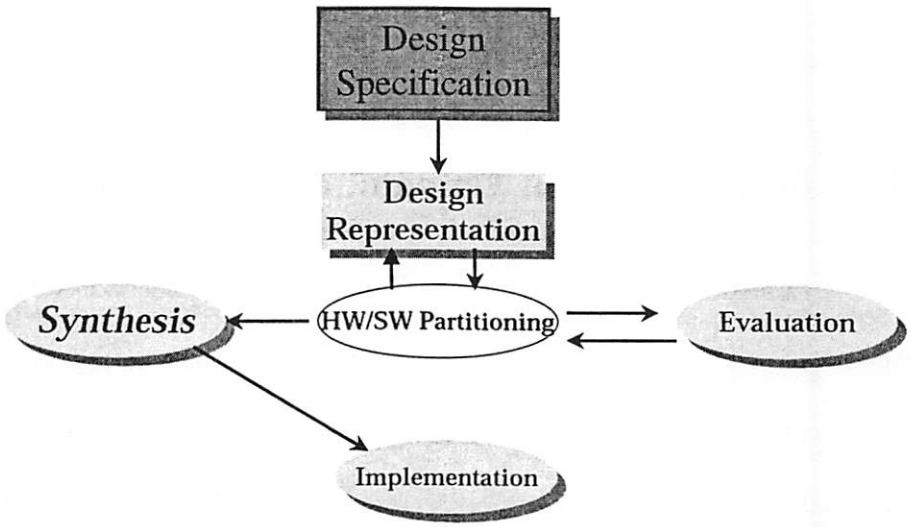


Figure 2.1: Major Roles in Embedded System Design

Hardware/Software Co-design (HSC) is a recent field that emerged out of the pressures of the always shrinking time-to-market and the ever-increasing demand for more "intelligence" on the embedded device. *Optimization*, however, has for the most part been

neglected in the field in general; it is only addressed as part of *synthesis*; through logic synthesis for hardware [102], and the recent introduction of software synthesis techniques [7]. In this work, I focus on a synthesis-driven top-down co-design approach where optimization starting from the highest abstraction level and constrained (bottom-up guidance of) synthesis can boost productivity with an attention to quality.

2.3 System Level Design Validation of Embedded Systems

Design validation in general currently consumes a significant percentage of the design team and takes months of simulation time. This validation strain is bound to increase as the complexity of designs increases. In embedded system design this problem is compounded by the presence of numerous heterogeneous interconnected functional blocks that are implemented in hardware or software. In my proposed embedded system design and development methodology I will need to *rapidly* and *relatively accurately* evaluate trade-off decisions, and co-design alternatives that result from optimization at the high level without the need to fully synthesize the design at every iteration, so the need arises for developing a suitable validation technique in order to be able to measure the consequences of the trade-off decisions at the high level on the final implementation.

Embedded systems include hardware and software components cooperating together to achieve a common goal, like implementing a cellular phone, controlling an engine, and so on. Their validation according to the current design practice requires *performance simulation* of both hardware and software, in order to assess the overall performance of the system and to check the correctness of the interfaces. Hardware in an embedded target

architecture is usually needed for performance while software is used for flexibility. It is often quite desirable to be able to specify the design functionality and constraints at a high level, and then synthesize the hardware, software, and the necessary interfaces. Verifying the sometimes complex interaction between this mix of components is then the major task that follows synthesis. Our aim is to validate an embedded system composed of hardware and software components that are mixed together. In order to accomplish this goal our immediate concern becomes modeling the combined hardware and software system functionality, and its target architecture in a manner that requires trading off several aspects of the validation approach. These aspects are:

1. *Accuracy*: captures accuracy of our estimation and reliability of this information at the current design abstraction level.

2. *Throughput*: determines how fast our simulation is i.e. how many simulation cycles per CPU second the method can run; meaning how rapidly the method can generate a response feedback to the query made to it.

3. *Convenience*: measures how much user intervention is required, how automated the simulation process is, and how easy it is to generate test benches for the design.

The requirements of the validation environment to be used for performance evaluation in the embedded system domain include [5]:

- Fast co-simulation of mixed hardware/software implementations, with fairly accurate synchronization between the two components,

- evaluation of different processor families and configurations; with different speed, memory size, and I/O characteristics, and
- co-simulation of heterogeneous systems, in which data processing is performed simultaneously with reactive processing, i.e., in which regular streams of data can be interrupted by urgent events. Even though our main focus is on reactive, control-dominated systems, we should to allow the designer to freely mix the representation, validation, and synthesis methods that best fit each particular sub-task.

2.4 Co-simulation Validation Framework

The basic idea behind my co-simulation validation framework is the use of VHDL ([56]) to model all the system components to be synthesized, regardless of their future implementation, starting from an initial specification written in a formal language with a precise semantics, which can be analyzed, optimized, and mapped both to software and hardware. Figure 2.2 shows where this approach lies in the Accuracy vs. Throughput trade-off curve. On the accuracy extreme lies platform emulation where code runs on the actual processor interfaced to the real hardware. Functional timeless simulation is on the throughput extreme, while in the middle is HW/SW Co-debug that relies on Bus Functional Models (BFM) of the processor interfaced through an API, and typically an IPC communication mechanism to the software that will run on the target; the BFM is then interfaced to the HW HDL [50]. My approach is intended for high level performance co-simulation so it falls in the area between functional, and HW/SW Co-debug.

Software and hardware models are thus executed in the *same simulation environ-*

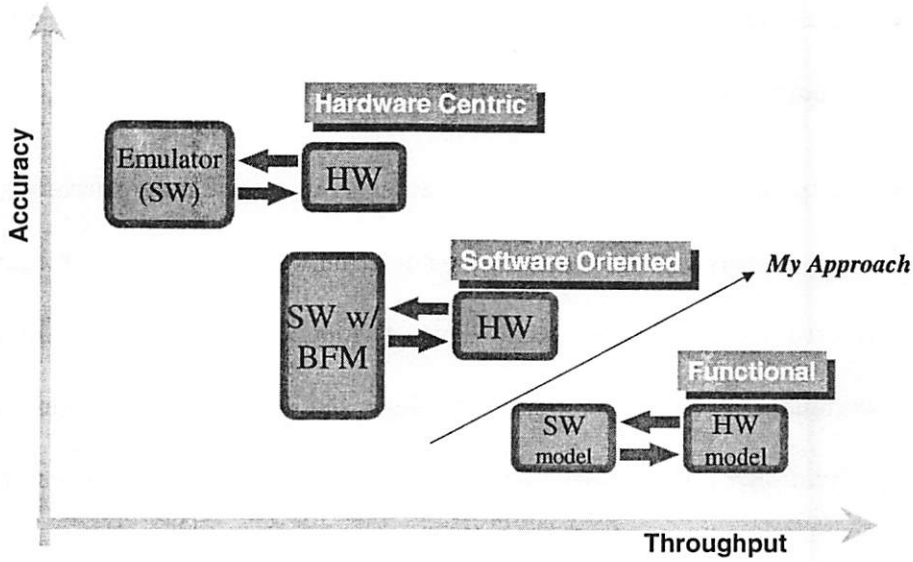


Figure 2.2: Accuracy and Throughput Trade-offs in HW/SW Co-simulation

ment, and for the software partition, the simulation code is *very similar to the code that will run on the target processor*. Different implementation choices are reflected in different simulation times required to execute each task (one clock cycle for hardware, a number of cycles depending on the selected target processor for software) and in different execution constraints (concurrency for hardware, mutual exclusion for software). Efficient synchronized execution of each domain is a key element of any co-simulation methodology aimed at evaluating potential bottlenecks associated with a given hardware/software partition [5], so the method also models the interfaces between among the HW, and SW tasks.

2.4.1 High-level Co-simulation Using VHDL

My approach to co-simulation is based on the decomposition of the system into three classes of component models:

1. Software tasks to be executed on some processor under the control of a Real-Time Operating System (RTOS). The RTOS which handles communication within the processor (i.e. scheduling and I/O), and with the rest of the system is also modeled.

2. hardware tasks communicating via a standardized protocol with the rest of the system, as well as

3. existing pieces of hardware (software) IP, modeled in behavioral or RTL VHDL (C Foreign Language Interface (FLI) in VHDL).

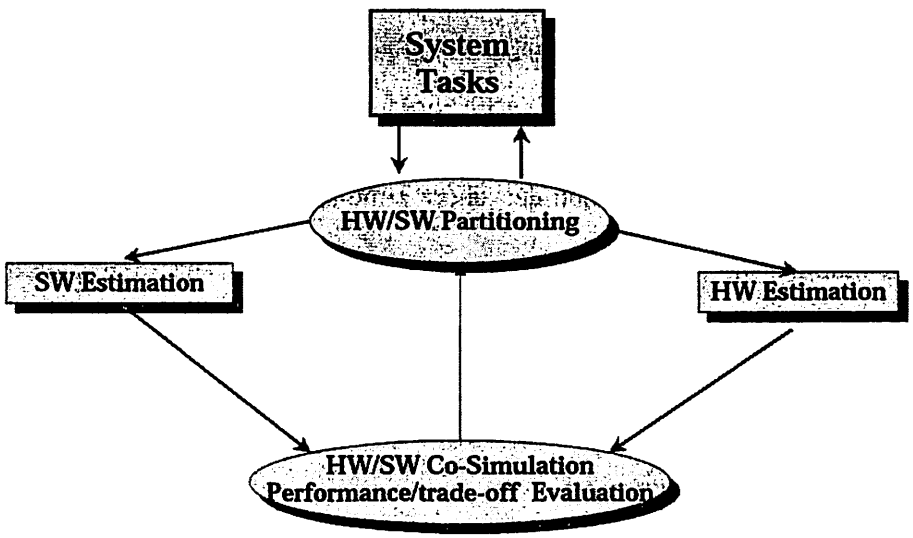


Figure 2.3: Estimation-based High Level HW/SW Co-simulation

Figure 2.3, outlines the basic idea of my estimation based approach. I build (automatically synthesize) a VHDL model for each task, the RTOS scheduler, and the interfaces. Data flow functions are permitted in order to perform computations, and are usually described as VHDL or C routines, and modeled as VHDL function calls or through the C Foreign Language Interface (FLI) of VHDL respectively. Complete details on this perfor-

mance co-simulation framework are provided in [113].

2.5 Function Architecture Co-design Methodology

System level designs are typically composed of heterogeneous hardware (digital as well as analog) and software components. Current design methods follow the traditional hardware or software block development approach, and are inadequate for the future's system level (board or IC) design demands. These outdated methods simply do not scale in the large, and most often have a pre-conceived conventional hardware and software implementation and interface scheme. This narrow and limited view leads to costly design iteration, until a desirable design implementation is found, in today's system level design dynamics that typically include: short time-to-market, rising cost of design, validation, and manufacture, in addition to keen competitiveness that demands quality products. I have developed on a novel top-down (synthesis), bottom-up (constraint-driven) system level design methodology. The Function Architecture Co-design Methodology put forth in this work[1] is shown in Figure 2.4.

The methodology revolves around three founding concepts:

1. *Decomposition*: In a top-down flow and heterogeneous design target architecture, design space is large. In order to successfully find an optimal match between the application function and its constraint metrics (e.g. power, performance, size), we use the "separation of concerns" approach where the function is decomposed into units, and the system constraints are broken up as well onto these system sub-units [100].

[1]Expanded from our initial introduction of this approach in [5] to include trade-off analysis and optimization at the high level

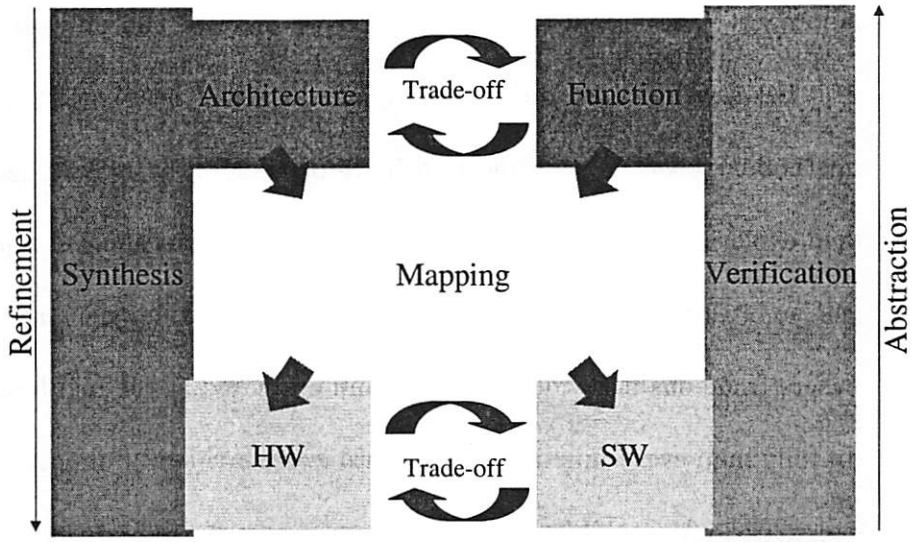


Figure 2.4: Function Architecture Co-design Methodology

2. *Abstraction and Successive Refinement*: After the design function and architectural constraints are decomposed, the co-design process begins where an "educated" function/architecture formal trade-off method is applied leading to the "best" mapping of the function onto the architecture and to the desired goal. This is achieved by co-design and trade-off evaluation starting from the highest abstraction level, and successive refinement of the function, guided by architecture model abstraction, down to the lower levels. Successive refinement is the process of adding more detail to the earlier abstraction level.

3. *Target Architectural Exploration and Estimation*: This is the **bottom-up** part where the synthesis target architecture is analyzed, characterized, and estimated in order to derive the *architectural constraints* that, together with *user-specified constraints*, will drive the trade-off and optimization decisions at the high level in the top-down

approach. The formal trade-off method we describe can only succeed and converge if it has a well-defined understanding of where it is heading. No system level design methodology can be expected to achieve optimality by making design optimizations and exploring suitable implementation choices without adequate modeling of the target architecture. Estimation is therefore a crucial component in system-level design where trade-offs need to be explored and evaluated at the high level without needing to fully map every function variant onto every architectural option.

The methodology for embedded system architectural exploration we proposed in [5] is shown in Figure 2.5. Deriving the correct selection of an architecture at an early stage of the design using constraints helps in reducing the number of iterations of the design process, and decreasing the time to market. Moreover, working at the high level adds flexibility, and the design can be easily re-targeted if new target platforms become available.

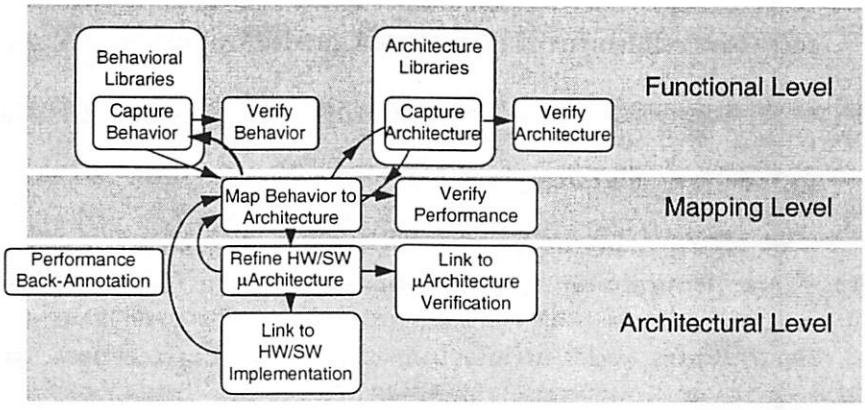


Figure 2.5: Methodology for Embedded System Architecture Exploration (from [5])

My co-design and synthesis approach is composed of the following main steps described in the subsections to follow:

1. Function Architecture *Co-design* and Trade-off

2. *Mapping* Function on the Architecture

2.5.1 Function Architecture Co-design

If we can fully synthesize the architecture or if the target architecture itself is *parameterizable* at the *macro* and/or *micro* levels (e.g. type and number of functional units such as multiplies, adders, or Multiply-Accumulates (MACS) for the macro-level, and size and access modes of the register file, cache, and memory for the micro-level) then we can perform this function/architecture co-design step. On the other hand if fully synthesizing or partially modifying the architecture is not an option then the architecture is fixed, and our only recourse is to perform architecture independent function optimization and then map the function onto this architecture in the "best" manner possible. Of course, the latter restriction seriously curtails our ability to optimize the function to suit the architecture and function/architecture trade-off aspects are limited severely. The reader must not interpret my words to mean that I am advocating full flexibility, that will most likely make the problem intractable, on the contrary my approach revolves around constraints; trade-off evaluation needs guidance, but we must have alternatives to consider.

2.5.2 Mapping the Function onto the Architecture

After the functional decomposition, and the architectural organization are fixed, the second step in the function/architecture co-design methodology is *mapping* the function onto the chosen architecture. The architectural organization can be a pre-designed collection of components with various degrees of *flexibilities* as in the notion of *platforms* presented in [42] by Ferrari and Sangiovanni-Vincentelli, and Chang et. al. in [22]. The various tasks to be implemented are "assigned" to the different portions of the architectural organization. Previous "hardware/software co-design" approaches have relied solely on this mapping step to achieve "optimality" where the hardware and software tasks are optimized *separately using different techniques* and then mapped to the target. While the approach is valid, it does not leverage any "co-design" in the full sense of the word as we propose in subsection 2.5.1 but rather "co-creation" hence the connotation of being at a low implementation level as opposed to a design inception level where tasks are allowed to be "free" at the highest abstraction level and not bound to a specific implementation.

2.5.3 Function/Architecture Co-design versus Hardware/Software Co-design

As I alluded to earlier, I believe the term most commonly used today to describe the embedded system design methodology is a misnomer. The community in this field, aside from notable exceptions (such as [42]), has in general referred to hardware/software trade-off evaluation and implementation, shown at the very last stage of function/architecture co-design in Figure 2.4, as "Co-design for Embedded Systems". In fact some even go as far as

calling hardware/software co-debug as "co-design". I believe that a very limited set of trade-offs and optimizations can be done at this low level since most of the crucial function and architecture decisions would have been made, and the design problem quite over-simplified. Seems to me that choosing to move a SW task to HW because of performance considerations for example is no different than choosing to use a more powerful processor (where HW can be thought of in some sense as such an additional more powerful processor).

It is my hope that the upcoming Chapters in this dissertation will demonstrate why I take issue with current misleading hardware/software co-design approaches, emphasize the message of function architecture optimization and co-design, and demonstrate how we can achieve our goal of *matching* the *optimal* function to the *best* architecture.

2.6 Reactive System Co-synthesis

As I stated earlier, current co-design techniques have a serious shortcoming: they are employed at a relatively low abstraction level. In this Section I show one of the major manifestations of this: poor synthesis output quality. Since software and hardware co-synthesis strategies for *control-dominated* applications are mainly concerned with the efficient (fast and compact) implementation of a *reactive* decision process [5], data flow aspects are usually neglected; it is generally assumed that software compilers and hardware Register Transfer Level (RTL) compilers will address these optimizations. The typical synthesis process, as shown in Figure 2.6, starts with design capture, followed by modeling and representation using finite state machines extended with operations and data computations referred to here as Extended Finite State Machines (EFSMs). The EFSM of each system

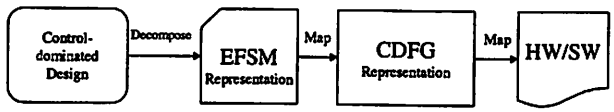


Figure 2.6: Reactive System Co-synthesis

component module is then mapped in this flow onto a Control Data Flow directed acyclic Graph (CDFG) which is then used to generate reactive hardware or software. A transition of the EFSM is performed by executing a path in the CDFG when the task is invoked.

While the CDFG is ideal for representing the reactive tasks to be synthesized since it can be used for both early size/speed estimation as well as synthesis of the hardware, and code generation of the software, this representation hides much of the control flow *across invocations* of the reactive module, and consequently data effects cannot be fully propagated and adequately evaluated in a analysis based on the CDFG representation. This fact limits data flow optimizations, as well as control optimizations that are data dependent, to optimizations restricted to paths in the CDFG DAG without considering the optimizations across such paths. I illustrate this point using the simple example of Figure 2.7. The example shows an EFSM with a constant propagation opportunity that would save a needless addition operation. The `a = 5` operation of `S0` and the `a = a + 1` of `S1` can be combined into a single `a = 6` assignment operation in `S1`, provided that `a` is an *internal variable* whose value is available to[2] the environment only *after* state `S2`. This optimization cannot be easily identified in the low level CDFG representation using conventional analysis techniques since it is distributed across two invocations of the reactive task (first for state `S0` and second for state `S1`).

[2]Observable by

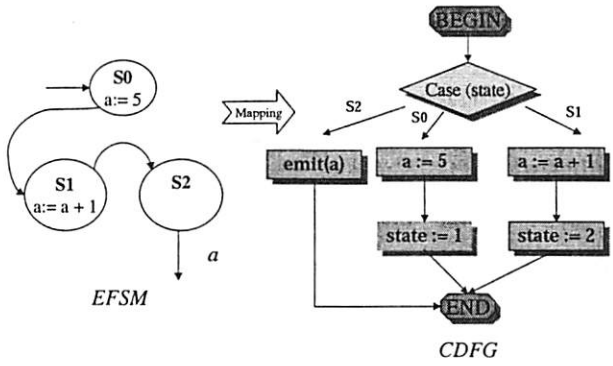


Figure 2.7: Data Flow Optimization and the CDFG Representation

The **key point** to emphasize here is that it cannot be expected of hardware and software compilers (as "conventional wisdom" leads one to believe) to *statically* discover such an optimization that involves a *run-time* decision in the task CDFG representation. The very simple example of Figure 2.7 shows that the *implementation level* CDFG representation puts a roadblock in the tracks of analysis techniques aimed at identifying such potential optimizations. The CDFG has a shortcoming when it comes to representing optimization opportunities that lie across task invocations. I aim to develop an equivalent task representation that is better able to capture such opportunities and to present them to optimization techniques that can identify and exploit these prospects.

I introduce in the next Chapter this design representation of each system task that is able to capture the EFSM description, and is at the same time suitable for performing data flow and control optimizations starting at this high design description level. I show in Chapter 9 that performing data flow and control optimizations at the design representation level will directly reflect positively on the size and performance of *both* the synthesized *hardware* and *software*. It will then become apparent that performing optimization at an

abstraction level that is higher than the one on which current approaches operate has a definite advantage in the output quality. Our optimization and co-design approach is divided into 2 phases:

1. Architecture-Independent Phase: Task function is considered solely and control and data flow analysis is performed, followed by optimization. The optimizations here are useful for both size and performance improvement since they involve removing redundant information and computations.

2. Architecture-Dependent Phase: Optimizations in this stage rely on architectural information to perform additional *guided* optimizations tuned to the target platform and typically involve some estimation-based trade-off between the different design metrics measuring implementation cost (such as size and performance).

Part II

Function/Architecture

Optimization and Co-design

Chapter 3

Design Representation

We in the HW/SW Co-design field have been searching for some time now for a unifying Hardware/Software Design Representation. Fellow researchers have developed computational models especially suited for data processing applications (such as SDF [86] or DDF [30]), or proposed a unified model for control and data flow modeling (such as [49]), I am however targeting heterogeneous control-dominated embedded systems, and would like to capture function, architecture and be able to manipulate, co-design, and trade with both aspects. Figure 3.1 outlines this goal.

At the functional level, the search for a system level unifying design representation has recently become the focus of the SLDL committee [10]; while in target architectures there has been a shift towards flexible hardware architectures that can support a variety of applications via programmability and reconfigurability as exemplified by the position paper of Ferrari and Sangiovanni-Vincentelli [42]. Sangiovanni-Vincentelli also gives an abstract definition of a *platform* and its interplay with the application it is intended to support as

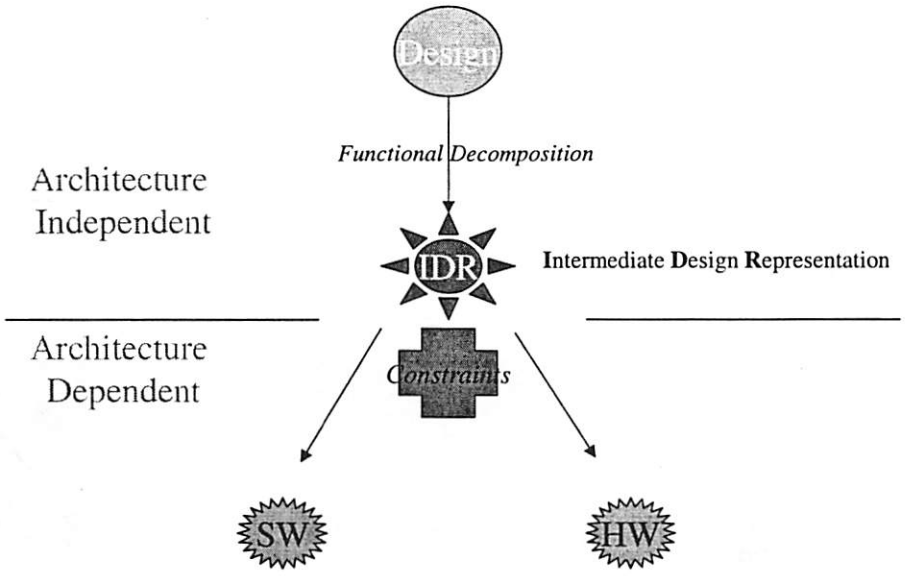


Figure 3.1: Unifying Intermediate Design Representation for Co-design

shown in Figure 3.2.

Figure 3.2 reflects the search for a *common abstraction* that both the application (the "function" in our context here), and the architecture can "agree" on and trade-off to find the best platform instance for one or more intended application instances. This recent change in focus also reflects the fact that we in the embedded domain are starting to feel manufacture *cost pressures* where the *demand* is sometimes not sufficient for a large volume production of a specific application device. This pressure is forcing the use of *generic parameterizable* programmable and flexible platforms which can be used to target several application domains whose aggregation can create a reasonable profitable Integrated Circuit (IC) production volume.

My work focuses on "software-like" techniques for programmable components not only because embedded software typically represents 50-90 percent of the design functional-

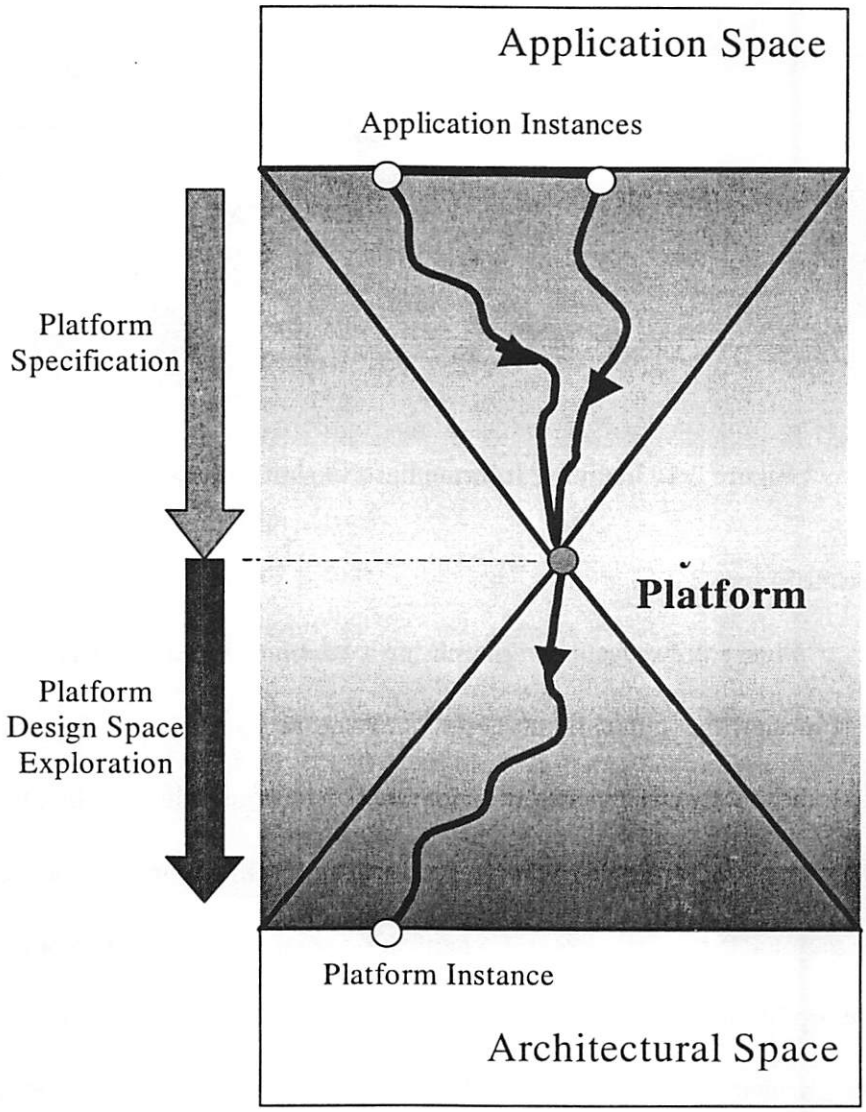


Figure 3.2: Applications and Platforms (from [42])

ity[1] but also because the most successful *mass produced* product ([11]) that the electronics industry has been involved with is the micro-processor which represents the extreme in *generality* for the target architecture. In the embedded domain, however, many constraints (performance, power, area, cost, supply and demand) forbid the reliance on fully programmable solutions. Productivity has its benefits, but it usually has a toll on creativity as well [1]. Embedded design is as much an art as it is a science, and we cannot afford to neglect designer talent i.e. *quality* in favor of production numbers i.e. *quantity*. Therefore in order to capture the best of both worlds, productivity and innovative forte, I believe that we should be looking for a *flexible specialization* or *customization* platform as opposed to a mass produced one [99]. We need an *adequate intermediate representation* that captures the developer's *intent*[2] in order to be able to design for such a platform. We in the EDA community must also develop a **formal design methodology** able to offer both flexibility and efficiency, and tool support for *system level design* that will assist designers in creating, and relieve them of the mundane tasks that stifle inventiveness.

Designer *creativity* in product enhancement such as the addition of features while preserving efficiency at a low development cost is a major product differentiator. Without newfangled ideas and features in product generations the basis of comparison with the *competition* will shift to price, and *profit* margins will fall. This is especially true for fabless semiconductor design companies, and in fact Dataquest estimates that by 2005 the fragmentation and conversion of fabbed semiconductor companies to a fabless/foundry structure will be commonplace, and that the majority of top semiconductor companies will get rev-

[1] Quote taken from Coware Inc., SystemC Initiative

[2] As defined by the work on *Intentional Programming* of Simonyi [105]

enue from 3 or less products. The two factors, cost and competition, actually feed on each other. The shift to a foundry model is in response to the increasing cost of fabs. With the rise of the foundry model, the barriers to entry decrease resulting in the proliferation of fabless semiconductor companies, thus creating intense competition at the chip design level [48]. I believe that the fluid embedded market with its seemingly never ending imaginative applications has been a technology improvement driver for many years, and must remain so if we are to prevent market stagnation exemplified by the saturation that the PC commodity market is experiencing currently; while PCs composed the largest segment of the semiconductor market they are quickly being replaced and surpassed by the consumer and communication applications.

This Chapter sets forth a novel unifying design representation for control-dominated heterogeneous systems. While I do not claim this is *the* representation able to address *all* our needs in embedded design, I believe that it is a sizable step in the right direction. I will show that this particular representation has opened doors and new means for function/architecture optimization and co-design, the embedded design approach proposed in this dissertation, for rapid productive and effective discriminative design.

3.1 Background

3.1.1 Models of Computation

State-oriented models [88] are common in control-dominated systems. They describe the function of a task by a set of states and a set of transitions between them. Examples of such models of computation include:

1. Petri net models [87]: are a graphical language for modeling design systems. They have proven useful for describing work flows and assembly lines, but they quickly become incomprehensible for large complex systems.

2. Finite State Machine (FSM) models: consist of a set of states, and a set of transitions connecting these states, along with a set of outputs. In *Mealy* machines outputs depend on both state and current inputs for the transition, while *Moore* machines associate outputs with the states themselves.

3. Hierarchical Concurrent FSM models (such as Harel's StateCharts [53]): decompose a single FSM state into (possibly concurrent) sub-states, and thus have support for both hierarchy and concurrency.

3.1.2 Polis Semantics and Model of Computation

I use the Polis co-design environment for reactive embedded systems ([28]) in order to synthesize software and hardware, and analyze their performance. The underlying semantic *Model of Computation (MOC)* in Polis is based on Co-design Finite State Machines (CFSMs) and a system is described as a network of CFSMs. Each CFSM is an Extended FSM (EFSM) where the extensions add support for data handling and asynchronous communication. In particular, a CFSM has

- a *finite state machine part* that contains a set of inputs, outputs, and states, a transition and output relations.

- a *data computation part* in the form of references in the transition relation to external, instantaneous (combinational) functions.

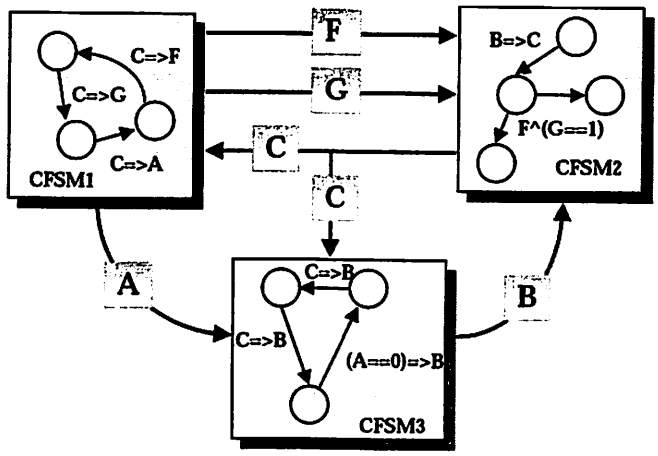


Figure 3.3: System Representation in Polis: A Network of CFSMs

- a *locally synchronous behavior*: each CFSM executes a transition by producing a single output reaction based on a single, snap-shot input assignment in zero time. This sequence of sensing and acting is synchronous from *its own perspective.*

- a *globally asynchronous behavior*: each CFSM reads inputs, executes a transition, and produces outputs in an unbounded but finite amount of time as seen by the rest of the system. This is an asynchronous interaction from the *system perspective.*

These semantics, along with a scheduling mechanism to coordinate the CFSMs, provides for a *Globally (at the system level) Asynchronous and Locally (at the CFSM level) Synchronous (GALS)* communication model. Each element of a network of CFSMs describes a component of the system to be modeled, and defines the partitioning and scheduling granularity. The design representation as a network of CFSMs is shown in Figure 3.3. Communication between CFSMs is not by means of shared variables (as in the classical composition of Finite State Machines), but by means of *events.* Events are a communication primitive

that implements synchronized read, unsynchronized write over a *depth-1* single-entry (lossy) buffer. Synchronized read is needed to capture the "reactiveness" of the CFSM tasks i.e. a task only executes when events are present on its input ports, while unsynchronized write is necessary with a bounded queue (size 1 for simplicity) so that the sender can continue execution without waiting for the receiver [5]. This communication mechanism is both powerful enough to represent practical design specifications, and efficiently implementable within hardware, software, and between the two domains.

3.1.3 System Specification Language

I target heterogeneous control-dominated embedded system applications, so I assume the design process starts initially with a functional decomposition that captures the design intent as a network of Finite State Machine modules extended with data computation (EFSMs) as described in [28], and [125]. Each module behavior is conveyed using graphical entry or an FSM-based reactive language (for example Esterel) front-end. I have chosen this specification approach because, in my opinion, it is the ***least intrusive*** upon the **separation of functional, and architectural concerns**. While other modeling approaches may have more powerful input description languages such as the C/C++ approach of CoWare [126], I believe that it is a mistake to add implementation detail (e.g. complex types, pointers, "threads" etc ...) into the high abstraction level unless the power of such mechanisms is curtailed, and used mainly for increased expressiveness; the work of Lavagno and Sentovich in developing ECL [74], and previously Boussinot et. al. with Reactive-C [20] is commendable in this regard. Low-level details must be left for later refinement stages in function/architecture co-design, otherwise they will most definitely limit the scope of the

potential co-design trade-offs that can be investigated at the system level. While several researchers in the co-design field may not wholly agree with my previous statements, I am quite adamant about this issue as is reflected in this work.

Esterel as "Front-end" for Functional Specification

Esterel [17] is a *synchronous programming language* developed for specifying reactive real-time systems. Synchronous programming languages are based on the perfect synchrony assumption meaning that computations take zero execution time, as a consequence timing constraints cannot be modeled [91]. Esterel, however, is ideal for use as a "front-end" for single task function specification since it contains reactive constructs that support pre-emption. Esterel is one of the input languages in the Polis co-design environment, and I will use it in this work to specify the function for each task, and use its compiler to translate the reactive constructs into an EFSM. This EFSM is then used as the task functional description that needs to be realized. Constructs in Esterel allow ease of description using such formalisms as *concurrency* and *sequencing* at any nesting depth without restrictions [16]. A very simple example in Esterel is shown in Figure 3.4[3].

The front-end compiler takes this reactive, expressive and compact[4] EFSM description[5] and builds an abstract description of the automaton for the described module, after verifying that there are no causality cycles (e.g. instantaneous loops), non-determinism, or non-reactivity in the description, and generates the *Object Code (OC)* portable format that captures the EFSM.

[3]Should be read from left to right

[4]Write Things Once (WTO) has been a goal in the design of Esterel.

[5]The *power* is that of an EFSM essentially (assuming all variables used in "host-language" function calls are bounded).

```
module simple:
input  inp : integer;
output outp : integer;
var
a := 0 : integer  ,
b := 0 : integer  ,
c := 0 : integer  ,
x := 1 : integer  ,
y := 1 : integer
in
  await inp;
a := ?inp ;
loop
a := b + c ;
   loop
     await tick ;
     x := x + y;
     x := x + y;
     a := b + c ;
     a := x ;
     if y = 1 then
       emit outp(a) ;
     else
       emit outp(b) ;
     end;
   end;
   a := x ;
end;
end.
```

Figure 3.4: Simple Esterel Design Example

Reactive VHDL "Front-end"

Esterel however is not very *transparent* to the user; the semantics (i.e. meaning) of the constructs may not always be *apparent* to a designer (particularly a hardware designer used to RTL-like descriptions), so in our work we have also defined a "reactive" subset of VHDL in the spirit of Synchronous VHDL by Baker [3], but even much simpler since we only attempt to capture the CFSM reactive model of computation semantics. By restricting the expressiveness of VHDL, we have identified an FSMD VHDL policy of use that is sufficient for the purposes of describing a set of interconnected CFSM tasks. Below is a small example that outlines the basics of this policy.

Example 3.1.1 Simple Reactive VHDL Example

```
-- System I/O
Entity system is
port( inp :in integer; ...);
End;

-- Overall Composition
Architecture CFSM of system is
--types/constants
```

```
--communication signals
Begin
        Task1:
        Process(inp, ...)
        --local types/constants/variables
        type S_Type is (S1, S2, ...);
        variable State, Next_State : S_Type := S1;
        ...
        Begin
        -- sample inputs
        ...
        -- update state
        State := Next_State;
        ...
        Case State is
                when S1 => if (...) then ...; Next_State := S2;
                when S2 => ...
                ...
        End Case;
        End Process

        Task2:
        ...

End Architecture
```

We interpret such a description to mean that a system is an *Entity* with *in*put and *out*put *ports*, and an *Architecture* that models the CFSM network as a collection of CFSM *processes* communicating through *signals*. The CFSM module represented by a process in the overall system *reacts* to the list of signals in its sensitivity list. The behavior it executes at each *invocation* is that of a Finite State Machine with Datapath (FSMD)[6]. This interpretation is consistent with the VHDL modeling I described earlier in Section 2.3. The description, however, is as verbose as the EFSM itself, and does not offer any of the benefits of *compactness* and *readability* that the Esterel front-end offers; a potential exponential reduction in the FSM description.

[6]The workhorse of VHDL descriptions

While the front-end does not play a crucial role in this work since I focus mostly on the intermediate representation and the "back-end" optimizations, I will assume when the front-end does matter for the sake of clarity that an Esterel-like front-end is used (LUSTRE [21] for example is another real-time language based on the same strong synchronicity hypothesis but geared for a data-flow style of specification).

3.2 Novel Intermediate Design Representation

The models I have introduced earlier are intended to capture the model of computation, and not for optimization and synthesis. I made the point in Section 2.6 that most of the current optimization and synthesis techniques are performed at the low abstraction level of a DAG representing the various task execution flows. I have developed a novel implementation-independent task representation referred to as *Function Flow Graph (FFG)* equivalent to the EFSM representation, and is a specialization of the classical CFG from the software domain. The FFG is *Input/Output (I/O) scheduled*, that is the representation preserves the task's I/O semantics. The FFG describes the task's function as well as its interaction with the environment i.e. when it reads inputs, and emits outputs.

The representation is therefore able to capture the behavior and I/O semantics of the task, and is well suited for control and data flow analysis and optimization techniques that, as we will see in the next Chapter, serve to optimize the function while preserving the externally observable[7] behavior of the task. The representation fits in the co-design framework in the manner shown in Figure 3.5. The design is initially decomposed into a

[7]By the environment (which includes other tasks in the system)

set of communicating EFSMs, with a model of computation governing the composition. The CFSM model of computation is one such model that can be used for this; no CFSM semantics are present in the FFG model itself since our intent in this modeling is to be able to "map" the FFG onto various computation models, and the CFSM model is only one of these. After decomposition, the task semantics "in the large" are captured in the FFG i.e. the FFG is able to describe the control and data flow much as the CFG does for the case of a software program. The difference here is that the FFG is intended to capture *reactive behavior* so it has the notion, a restriction of the CFG, of *Input/Output semantics* that need to be preserved in the representation in addition to those of the regular data dependencies between various computations.

The FFG as we will see has a further restriction on these operations that prevents side-effects. This is the initial abstraction level at which we can perform various optimizations. The next abstraction level *refines* the previous with additional constraints; one such consideration is the imposition of *EFSM semantics* onto the model. This essentially means applying the scheduling that the front-end computed based on the user's functional specification when building the EFSM onto the (initially unscheduled) FFG to obtain an *Attributed Function Flow Graph (AFFG)* which elaborates on the FFG through the addition of attributes thus extending the classical CFG; this is shown graphically in Figure 3.5. The reader can therefore see how the architectural/application/domain constraints shape the initial representation into one tuned for the particular target. We will be concerned here with the control-dominated domain and will apply EFSM semantics to shape the FFG, but the FFG model itself is quite general and can conceivably be used in other forms as well.

Next I describe the Function Flow Graph (FFG) in detail. I delay describing the AFFG to Chapter 5.

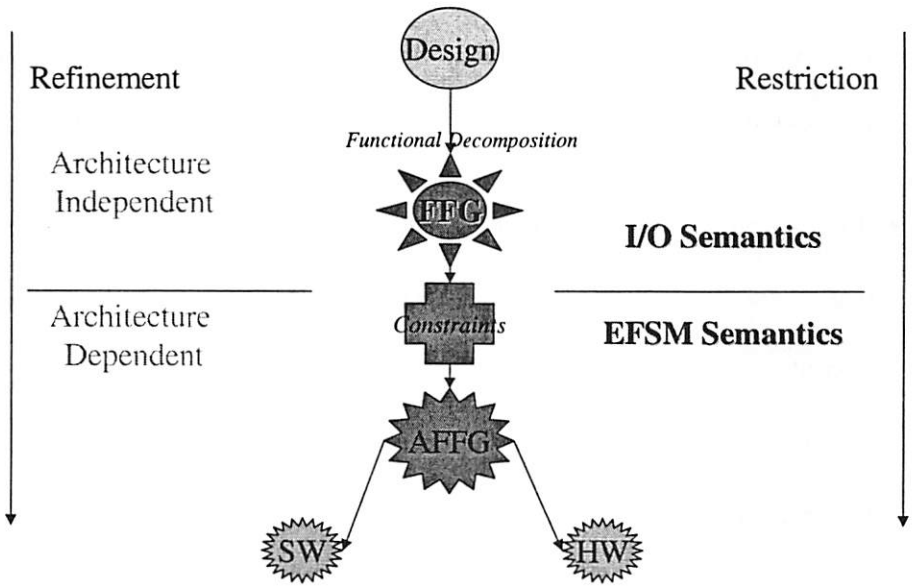


Figure 3.5: Our Proposed Unifying Task Representation for Function/Architecture Co-design

3.2.1 Functional Flow Graph

Definition 3.2.1 *A Function Flow Graph (FFG) is a triple* $G = (V, E, N_0)$ *where [114]:*

- V *is a finite set of nodes*
- $E = \{(x, y)\}$, *a subset of* $V \times V$, *where* (x, y) *is an edge from* x *to* y *where* x *belongs to* $Pred(y)$, *the set of predecessor nodes of* y.
- $N_0 \in V$ *is the* begin *or* start *node*[8]. *A single out edge connects* N_0 *to the next FFG node(s) (which represents the EFSM initial state(s)).*

[8]This is the *header*; the standard linear programming trick of creating a super-source is quite useful in the context of iterative solution methods (see Chapter 4).

- *Operations are* associated *with each node N.*

- *Operations consist of* **TEST**s *performed on the EFSM inputs and internal variables, and* **ASSIGN**s *of* computations *on the input alphabet (inputs/internal variables) to the EFSM output alphabet (outputs and internal (state) variables).*

- **TEST**s *and* computations *that read one or more EFSM inputs are marked as* input dependent *computations, while* **ASSIGN**s *to EFSM outputs are marked as* observable output *assignments. These operations define the task's interaction with the external environment referred to in the sequel as* **Input/Output (I/O) Semantics** *of the task represented by the FFG.*

The task behavior is captured by the *front-end* and represented as an FFG. Every set of operations is a node, a branch (conditional or a jump) necessitates the presence of a target node for the branch. The FFG representation is based on the classical Control Flow Graph[9] (CFG) where both control and data flow can be represented. Edges in the FFG represent control flow between the nodes. FFG nodes are *multi-entry* (except for N_0 which is single-entry), *single exit* nodes *in general.* Although the front-end will create a sequence of operations within each FFG node; this *initial* order of execution of operations is *not fixed* in the FFG representation. Any order of operations in an FFG node is acceptable as long as it is *valid* as defined in Definition 3.2.2 that follows.

Definition 3.2.2 *An execution order o for operations* within an FFG node *is* valid *if:*

- *The* data dependence *of the input alphabet of operations is preserved between the* initial

[9]Cyclic Graph

order i generated by the front-end from the user's EFSM description, and the order o, and

- *the task* I/O semantics *are preserved i.e. the input sampling, and output emission remains the same between the initial order i and the order o. In other words, if each possible sequence of input sampling and output emission is called an* I/O trace, *then orders i and o must be identical with respect to* all *possible I/O traces.*

The FFG is the task representation I use for design analysis and optimization. It is the abstract data structure on which the task control flow analysis is performed, and data flow information is gathered. I focus in this Chapter on the representation itself, and will refrain here from presenting an elaborate discussion of how analysis and optimization algorithms determine and preserve such things as data dependence and I/O semantics; that is the subject of the Chapters to follow.

3.2.2 C-Like Interchange Format

The *concrete syntax* or textual interchange format of the FFG is called the C-Like Intermediate Format (CLIF). The format consists of an unordered (in the sense of data independence outlined earlier) list of **TEST** and **ASSIGN** operations:

- **TEST** instruction

  $[if(condition)]$ *goto label*

- **ASSIGN** instruction

  $dest = \text{unop}(src1)$

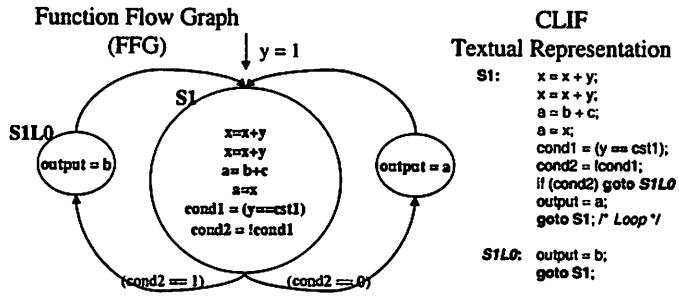


Figure 3.6: Simple FFG and Its CLIF Representation

$dest = src1$ binop $src2$

$dest = \text{func}(arg1, arg2, ...)$

The braces around the conditional check in the **TEST** instruction ($[if$ ...]) are intended to indicate that **TEST**s can be broken into 2 parts: conditionals, and jumps. FFG operations do not have any aliasing, that is there are *no side effects* (no pointers); they either involve **ASSIGN**ing to the target a result of a computation performed on one or two source operands, typically referred to in the software compilation domain as *quadruples*, or an assignment from a stateless arithmetic or Boolean function, or they consist of **TEST**ing a variable and performing a resulting action. All variables i.e. sources and destinations are assumed to be "registers" in the abstract sense, meaning that a variable *holds* its value until it is re-assigned a different value. This semantic of an operation in CLIF, i.e. an operation in an FFG node, is consistent with that of imperative programming languages where variables retain their value unless explicitly modified, and is needed for monitoring *data flow*, and for supporting data flow analysis to be described in Chapter 4. This operation semantic has no bearing on the actual final operation scheduling and register allocation but is rather a *simplifying abstraction* at the functional level.

The control transfer statement from a CLIF basic block to another is the `goto` statement. **Labels** of the form `identifier:` indicate the start of a CLIF basic block which is a *goto target.* Each CLIF block corresponds to a single FFG node. The format has C syntax, and supports all the unary and binary arithmetic, Boolean, and relational operators of C. A simple FFG graph, along with its equivalent CLIF textual representation is shown in Figure 3.6. The Figure shows an FFG with three nodes and their associated operations corresponding to the CLIF description on the side. The Figure also exhibits typical opportunities for data flow and control optimizations such as eliminating the `a = b + c` operation since it is useless (`a` is re-defined before the result of the said operation is used), and performing dead operation elimination on the `(cond2 == 1)` branch since `y` is always equal to `1` upon entry to node `S1`, consequently `cond2` is `0`. While the Figure shows "local" optimizations for simplicity, our goal is to address global versions of these types of optimizations.

3.2.3 FFG Structural Forms

In this Chapter we think of the FFG as a CFG that captures the task behavior (or "program description") and introduce only one additional consideration, that of identifying (and keeping track of) the externally observable input/output behavior of the task. The FFG however is more than a "program description". It is generated by the front-end from an initial EFSM description, so the FFG is essentially the EFSM *cast* as a classical CFG. Each EFSM *state* is represented in the FFG as a *collection of nodes* that model the computation, and transition behavior of the input EFSM. The FFG can be built from the EFSM by the front-end into either of two distinct structural forms: Tree form, and shared DAG form. The

following sub-sections describe these 2 forms in detail; a formal definition of the relationship between state attributes and the FFG in its 2 forms will be introduced in Chapter 5 (see Definition 5.5.1).

FFG in Tree Form

The EFSM captured as an FFG is shown in Figure 3.7 in *Tree* form for a simple example. In the Tree form, a single FFG node, out of the collection of nodes capturing an EFSM state, is the *start node* for the state, and the rest are *unique* conditional or jump *target nodes*. All the FFG *target nodes* that lie **within** a state are single entry and single exit nodes thus forming the so-called "tree". *Loops* in this Tree form, exist only between states, more accurately from an FFG *Tree leaf* node of some state to the *start node* of some (other or same) state.

In the Figure, nodes labeled with an `N` refer to FFG nodes while states are shown with a label of `S`. In fact what is displayed in the Figure is an Attributed FFG since the states are *attributes* associated with the FFG nodes, and therefore this a refinement of the FFG description, more on this later (in Chapter 5)[10]. The number of nodes n in the Tree FFG built from the EFSM generated by the front end is:

$$n = 1 + \sum_{i \in [1..N]} (1 + 2 * c_i)$$

where:

N = number of EFSM states

c_i = number of TESTs (i.e. conditionals) in EFSM state i

including the initial header node (N_0), a *single* FFG node as a *place holder* for state entry,

[10] As I stated earlier, the fact that nodes are partitioned into states is not relevant if we are thinking of the "functional behavior", and not associating any state semantics.

and two nodes for the `True` and `False` valuations for *each* conditional in the state.

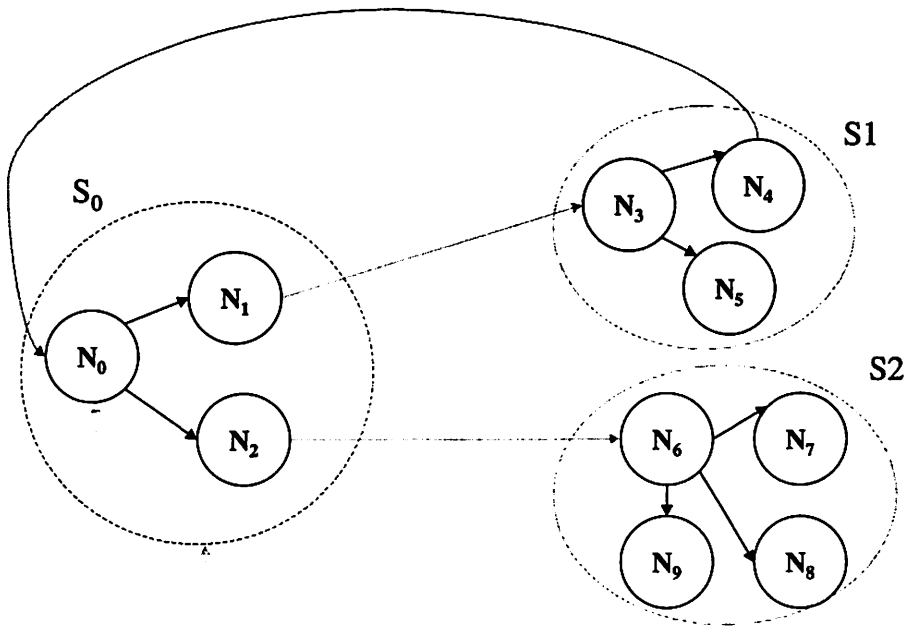


Figure 3.7: EFSM in FFG *Tree* Form: A Simple Example

FFG in Shared DAG Form

A *DAG form* can also be built by the front-end where some FFG nodes that perform a common functionality are "shared" *within* and *between* states as shown in the simple example of Figure 3.8. Nodes that are shared form a Directed Acyclic Graph of computations (therefore the name); these nodes can be shared between different computation paths within a state, or among different states. In this form, in addition to the state start nodes, a single FFG *DAG start node* denotes the start of a DAG, and FFG nodes may be multiple entry single exit in general. *Loops* in this form (as in the Tree form) can only occur between states i.e. from an FFG "leaf" node to a *state start node.* The number of nodes of the FFG in the DAG form is considerably smaller than that of the Tree form because of the node

sharing. The node sharing step is performed by the front-end (e.g. Esterel) while building the FFG where same computation expressions are shared in calls to a generated DAG (see [81]).

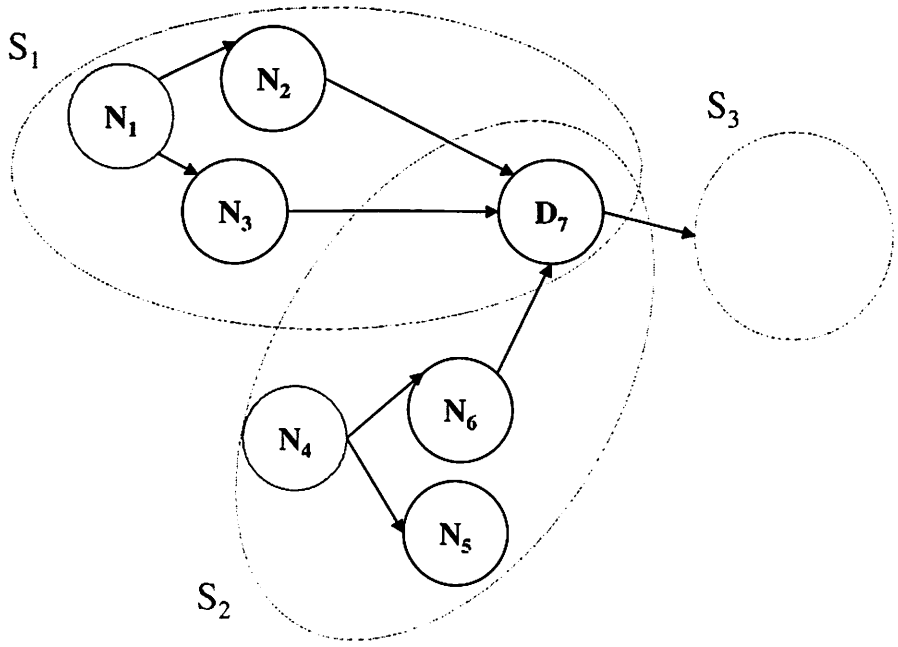


Figure 3.8: EFSM in FFG *Shared DAG* Form: A Simple Example

3.3 Proof of Concept

The skeptical reader may not be convinced by the benefit of high level optimizations and despite my earlier motivation in Section 2.6 this reader is still doubtful. So, in this section I would like to go back to the *very* simple example of Figure 3.4 that has some useless operations that open the potential for data flow and control optimizations, to put to rest any suspicions about the usefulness of design representation level optimizations. Figure 3.9 shows the *CLIF* textual representation outputted from the Esterel front-end (useless

statements are highlighted in **bold face**). To answer the question of whether the standard compilation process (of software code) can capture all the potential optimizations, I show in Figure 3.10 that follows a comparison result of two synthesis and compilation flows using gcc *with the highest optimization setting* for software code size with and without the FFG representation and the high level optimization approach.

```
Input inp;
Output outp;
int a, b, c;
int x, y;
int  cst0 = 0   ;
int  cst1 = 1   ;
int cond1,cond2,cond3;
int cond4;

        goto S1;
S0:     goto S0;
S1:     a = cst0;
        b = cst0;
        c = cst0;
        x = cst1;
        y = cst1   ;
        goto S2;
S2:     cond1 = inp;
        cond2 = !cond1;
        if (cond2) goto S2L0;
        a = inp   ;
        a = b + c ;
        goto S3;
S2L0: goto S2;

S3 :    x = x + y   ;
        x = x + y   ;
        a = b + c   ;
        a = x  ;
        cond3 = ( y == cst1 );
        cond4 = !cond3;
        if ( cond4 ) goto S3L1;
        outp = a   ;
        goto S3   ; /* Loop */
S3L1:   outp = b;
        goto S3;
```

Figure 3.9: Simple Design Example in CLIF

SIMPLE	gcc (bytes)	gcc –O3 (bytes)	gcc opt. (%)
EFSM→CDFG→obj	100	82.7	17.3
EFSM→FFG→CDFG→obj	89.9	75.8	*14.1*
% Difference	10.1	*6.9*	*3.2*

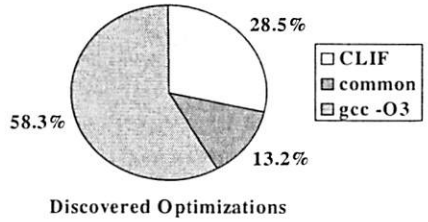


Figure 3.10: Compilation Code Size Result With and Without FFG Level Optimizations

It should be clear to the reader that indeed the compiler is working at a lower level of abstraction and is not aware of the *complete* picture of the control flow; about 30 % of discovered optimizations in the simple design example shown are due purely to high level optimizations. This is again why the FFG representation and optimization at the high level is quite beneficial.

Chapter 4

Function Optimizations

Optimization is the process of improving a particular task or progress in order to find the "best" way for performing the requirement. Let us first consider a simple example of the following task (from real life, and quite a common occurrence at that): filling a tub with hot water. The person performing this task has at his/her disposal two faucets: hot and cold. If we assume that hot water is a valuable resource that cannot be wasted; the optimization problem simply stated is that of finding the proper opening gauge for the hot and cold water to achieve the task with minimal hot water expenditure. As you can imagine, the problem becomes more interesting when the person has a time constraint for filling the tub, or possibly an externally imposed schedule on the opening and closing of the faucets. Of course, I argue that the best way to solve this is by using an intelligent controller to solve this constrained optimization problem. Note that I assume throughout this work that the environment imposes a known (finite) set of constraints over the lifetime of the controller and a static analysis can be done once *before* the function and architecture are fixed. This

may not be the case in general for a constantly changing environment. In this latter case techniques such as *introspection* [64], where the controller needs to reconfigure itself based on the environmental demands through *machine learning* [89], need to be applied.

4.1 Optimization Methodology

In order to solve any decision problem "optimally", an *optimization model* must be developed. Optimization models have been used for centuries since their purpose is quite appealing [90]. A model is a representation of the problem that captures its essence. It is most often the case that we are attempting to model a non-isolated process; in this work, in fact, I am concerned with reactive processes, that are running continuously and *interacting* with their environment. I therefore represent the environment as a set of *constraints* to be imposed on the process we are modeling. Optimization models can get large, and quite complex. There are typically a considerable number of constraints to deal with, however the problem can be managed by *relaxing* the constraints that are not tight, and optimizing (i.e. getting the "best" possible solution) for the remaining constraints. This process is adequate as long as we "keep an eye" on the relaxed constraints always making sure that they are within bounds (or "budgets", if you will, are not exceeded). This is a well known technique in constrained optimization where we optimize considering the *tight* constraints, while meeting the lax ones.

The notion of relaxation is one of the main strategies in the solution process based on the FFG model that I exploit in this work as captured by the *dual* concepts of: abstraction and successive refinement. The former is intended to state the notion of *scoping* that is at

any one stage in any analysis we can throw away irrelevant detail by extracting its essence, and working at this coarse level. The latter is the process by which we consecutively add more detail thus moving from one abstraction level to the next more detailed one. A closely aligned concept is that of the separation of concerns where orthogonal issues are dealt with through separate abstraction and refinement processes thus simplifying the procedure as a whole. We have to make a lot of decisions when optimizing and synthesizing with constraints, by knowing the proper *ordering* of optimization steps, and using the *separation of concerns*, followed by the aforementioned *abstraction* and *successive refinement*, a good deal of the problem complexity can be handled adequately and in an *optimal manner.*

In order to implement task optimizations on the FFG we need to identify operations and variables that can be eliminated using a static and conservative procedure. To that end I have developed an optimizer that examines the FFG in order to statically collect data flow and control information of the task under analysis using an underlying *data flow analysis framework* so that potential optimization opportunities can be identified. This formal framework is presented in the next Section.

4.2 Mathematical Framework for Control and Data Flow Analysis

Optimizations to be performed on the design representation involve solving a class of problems each of which can be dealt with in a similar fashion. A solution is derived from static information captured from the behavioral description itself. Of course it is impossible to determine the exact execution of this behavior and its result before runtime since this

clearly subsumes the halting problem [82]. These problems, called *global data flow analysis problems* therefore involve *static* determination and collection of information distributed throughout the design task [57]. Kildall ([65]) was the first to express this class of problems in a general lattice theoretic framework. Kam and Ullman later generalized this framework in [58]. In the subsections to follow I provide background and explain what the data flow analysis framework is all about. To make the discussion relevant to us immediately, I talk about the instances of frameworks that use the FFG (see Definition 3.2.1) as the control flow graph, and use adequate terminology, by adapting the definitions of the classical sources (cited where appropriate), and supplying my own set of theorems, and proofs that bolster the various FFG optimization algorithms, and finally lead to the *FFG Optimization Algorithm*, and its proof of optimality.

4.2.1 Data Flow Analysis (DFA) Framework

I limit myself here to describing frameworks that are intended to model *forward propagation* problems, that is problems that gather and manipulate information in accordance with the flow of control in the behavior (from the "start" node towards its successors in the FFG). For the rest of this document, I only consider such frameworks since I deal almost exclusively with the iterative approach to solving data flow problems, and in this case a backward flow can be modeled by simply reversing the flow and making some minor syntactic adjustments to the notions, so "conceptually" I can discuss here one type of framework and use notations consistent with (and adequate for) it.

Our ultimate goal in optimization is to be able to manipulate *information*, propagating it throughout the behavior, and making some observations about it, in order to

simplify the process of generating this information from the input when requested at the outputs. Let us call this set of *values* for a particular problem instance, or *information* in general I. Here, I put forth the following propositions[1] about our endeavor in optimization (and later on with some refinement in constrained optimization as well).

Proposition 4.2.1 Boundedness of Static Information *A behavior has within it a certain maximum amount of* static information. *This information can be* collected *in an* additive *fashion from the given behavior.*

Proposition 4.2.2 Information Preservation Law *Information must not be created or destroyed by the process of information gathering or optimization. Information gathering or optimization algorithms that obey this law are referred to as* preservative. *Preservative optimization algorithms can only add or remove* redundant *information.*

Proposition 4.2.3 Information Manipulation Safety Requirement *Transformations are said to be* safe *if they preserve the input alphabet (consisting of all the inputs and state variables) to output alphabet (consisting of outputs and state variables) traces of the FFG. The language of the optimized machine must be contained in that of the original machine, and the behavior of the two machines must be* indistinguishable *based on exhaustive I/O trace comparison.*

Proposition 4.2.4 Information Validity Assertion *A solution to an* information-gathering problem *is* under-determinate *if it fails to report as strong an assertion as possible as opposed to an* over-determinate *or* aggressive *solution that has strong (possibly invalid) assertions [82]. An under-determinate solution is always said to be* valid.

[1] Axioms

In one sentence[2]: **Our goal in analysis and optimization is to gather information in a *preservative* fashion about the behavior of a task then apply *preservative* and *safe* transformations in order to find a *valid* solution. The solution we seek is the *best possible statically determined valid solution.***

Let me begin the formal discussion by defining a semilattice (adapted from [58]).

Definition 4.2.1 *A* meet semilattice *is a set L with a binary* meet *operation $\wedge$ such that for all $a, b, c \in L$:*

$a \wedge a \equiv a$ (idempotent)

$a \wedge b \equiv b \wedge a$ (commutative)

$a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c$ (associative)

The meet can be extended to arbitrary finite sets where:

$\bigwedge_{1 \leq i \leq n} x_i = x_1 \wedge x_2 \wedge ... \wedge x_n$

We can also define an order relation on the set of information or values in the lattice L denoted by I where $a > b$ means that a is "bigger" than b, or equivalently $b < a$ means that b is "smaller" than a; we also have the notion of "equality" as shown in the following definition.

Definition 4.2.2 *Given a meet semilattice L and elements, $a, b \in L$, we say that:*

$a \geq b$ *if and only if* $a \wedge b = b$

$a > b$ *if and only if* $a \wedge b = b$ *and* $a \neq b$

Definition 4.2.3 *A meet semilattice L is said to have a* top element[3] $\top \in I$ *if for all $x \in L$:*

[2]The mission statement

[3]This is the *one element* of the lattice.

$$\forall x \in L, \top \wedge x = x$$

We assume that every meet semilattice has such a $\top$ *element.*

Definition 4.2.4 *A* join *semilattice is a set* L *with a binary* join *operation* $\vee$ *such that the* duals *of the meet semilattice properties carry under this join [18].*

Definition 4.2.5 *A join semilattice* L *is said to have a* bottom element[4] $\bot \in I$ *if for all* $x \in L$*:*

$$\bot \vee x = x \; \forall x \in L$$

We assume that every join semilattice has such a $\bot$ *element.*

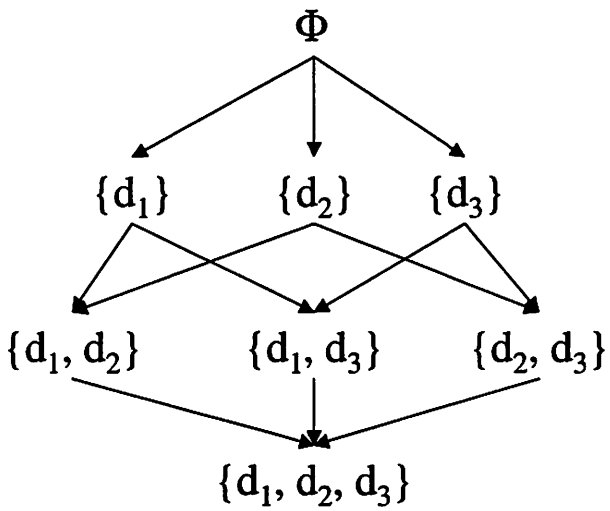


Figure 4.1: Lattice of Subsets of Definitions (from [2])

To demonstrate the meaning of the definitions of values and the lattice order, Figure 4.1, shows the lattice L of subsets of definitions where I is the set of all definitions in the behavior. In other words the lattice L is the *power set lattice* on the set of downward-

[4]This is the *zero element* of the lattice.

exposed definitions in the behavior[5]. In order to take meets over finite sets I also need the notion of a *bounded semilattice* defined as follows.

Definition 4.2.6 *Given a meet semilattice L, a sequence of elements x_1, x_2, ..., x_n in L forms a* chain *if $x_i > x_{i+1}$ for $1 \leq i < n$. L is said to be* bounded *if for each $x \in L$ there exists a constant b s.t. each chain beginning with x has length at most b.*

The FFG behavior transforms information as it is traversed by performing computations on input data; this is more eloquently stated in the following definition.

Definition 4.2.7 *An function f on the* information $I_n \subseteq I$ labeling *a node $n \in N$ of G is called a* **transfer function** *if it is a mapping from $Reach(n) \rightarrow Pass(n)$, where I is the set of values (or information) of a problem instance ordered by a semilattice L, $Reach(n)$ is the set of information entering node $n \in N$, and $Pass(n)$ is the set of information leaving the same node. The set of transfer functions on I of L for each node in the flow graph G is denoted by F.*

The set F depends on the problem we are solving (thus we have an instance of the framework). It represents moving information from the input of a node (at the *meet* of all in-edges) in the graph G to the output of the node (at the *split* of all out-edges). The *meet* or *join* (as the case may be) is typically referred to as the **confluence** operator.

I am now ready to introduce the notion of data flow frameworks. I ask the reader to bear with me as I introduce these definitions, an example will follow shortly in Section 4.2.3, and serve to make these notions concrete. Note that I will mostly focus on *meet* semilattices

[5]This is a simple and intuitive *join* semilattice.

unless otherwise stated; it should be understood that the dual properties hold for *join* semilattices.

Definition 4.2.8 *A meet data flow analysis framework is a quadruple* $D = (G, L, \wedge, F)$ *where* G *is the behavior flow graph under consideration (FFG is one such instance) and* L *is a semilattice with meet* $\wedge$*, and* F *is a function space associated with* L*.*

Definition 4.2.9 *A particular (problem) instance of a data flow analysis framework* D *is a pair* $Instance = (G, M)$ *where* $M : N \rightarrow F$ *is a function that maps each node* N *in* V *of* G *to a function in* F *on the node label semilattice* L *of the framework* D*.*

4.2.2 Monotone Data Flow Analysis (MDFA) Framework

I need some definitions to start with.

Definition 4.2.10 Function Properties *Given a bounded semilattice* L *with meet* $\wedge$*, a set of functions* F *on* L *is said to be* monotone *if the following conditions are met:*

1. *Each* $f \in F$ *satisfies the following* monotonicity *condition:*

   $\forall x, y \in L, f(x \wedge y) \leq f(x) \wedge f(y)$

2. *There exists an identity function* i *in* F *s.t.*

   $\forall x \in L, i(x) = x$

3. F *is* closed *under composition (o) that is for any two functions* f *and* g *in* F*:*

   $f, g \in F \Rightarrow fog(x) \in F$ *where* $\forall x \in L, fog(x) = f(g(x))$

I can now make the following observation that makes the meaning of *monotonicity* more intuitive (from [58]).

Property 4.2.1 Monotonicity Equivalence Property *Given a semilattice L, let f be a function on L, then* $[\forall x, y \in L, f(x \wedge y) \leq f(x) \wedge f(y)] - [\forall x, y \in L, x \leq y \Rightarrow f(x) \leq f(y)]$

Proof. The key is to realize that $x \wedge y \leq x$, and $x \wedge y \leq y$. The proof follows easily in *either direction* from the corresponding given monotonicity condition.

Definition 4.2.11 MDFA *A meet* monotone data flow analysis *framework is a quadruple $D = (G, L, \wedge, F)$ where G is the behavior flow graph under consideration (FFG in our case) and L is a bounded semilattice with meet $\wedge$, and F is a monotone function space associated with L.*

The framework can be used to manipulate the data flow information by interpreting the node labels on nodes N in V of the control flow graph G as elements of an algebraic structure L. We also have a variant of Definition 4.2.9 for the monotone data flow framework.

Definition 4.2.12 MDFA Instance *A particular (problem) instance of a monotone data flow analysis framework is a pair $Instance = (G, M)$ where $M : N \rightarrow F$ is a function that maps each node N in V of G to a function in F on the node label semilattice L of the framework D.*

Condition 1 of Definition 4.2.10 is what distinguishes a monotone framework from a DFA. Kildall in [65], uses MDFA frameworks that have the additional property of *distributivity* on the set F as described in the definition that follows.

Definition 4.2.13 Distributivity Property *The set F is* distributive *if it satisfies the following condition:*

$$\forall x, y, \in L, \forall f \in F, f(x \wedge y) = f(x) \wedge f(y)$$

From Definition 4.2.13 and condition 1 of Definition 4.2.10 we see that $f \in F$ is distributive $\Rightarrow$ f is monotone, so the monotone framework is more general; there are problems that do not fit in the DFA framework but do fit in the MDFA model. In particular constant propagation is one such data flow problem [58]. A data flow framework for a problem instance therefore involves a flow graph, a semilattice of values (information), and a set of functions from the semilattice to itself. Properties of these components (e.g. reducibility of the flow graph, monotonicity of the function space etc...) affect [82]:

1. Existence of an *exact* or *approximate* solution,

2. applicability of methods for arriving at this solution, and

3. complexity of the method used.

Point 1 is answered in the subsection to follow, while issues 2 and 3 are discussed in the next. In a marked difference from most papers and texts on the subject, I present the notions as they relate to join problems initially and then indicate the duals for meet problems[6]. Note that in the following, I talk of *fixpoints* as the approximate solutions (since I will eventually focus on the iterative method) the word should be taken in general to mean *intermediate result* that approximates the *true solution.*

[6]Most texts use the intuitive reaching definitions (join) problem as I do, but then they explain all about meets, this is quite confusing and I hope to circumvent it here.

4.2.3 Solutions: *Exact* and *Approximate*

Definition 4.2.14 *A* minimum fixpoint (**MFP**) *is a fixpoint which is smaller than or equal to any other fixpoint.*

Definition 4.2.15 *We define the* join over all paths *denoted by* **JOP** *to be as follows. Let* $PATH(n)$ *denote the set of paths from the initial node* N_0 *to node* $n \in N$ *in the flow graph* G*, then the* **JOP** *at node* n *is [58]:*

$\bigvee_{p \in PATH(n)} f_p(\perp)$*, where* f_p *represents information collection for the problem instance along path* p*.*

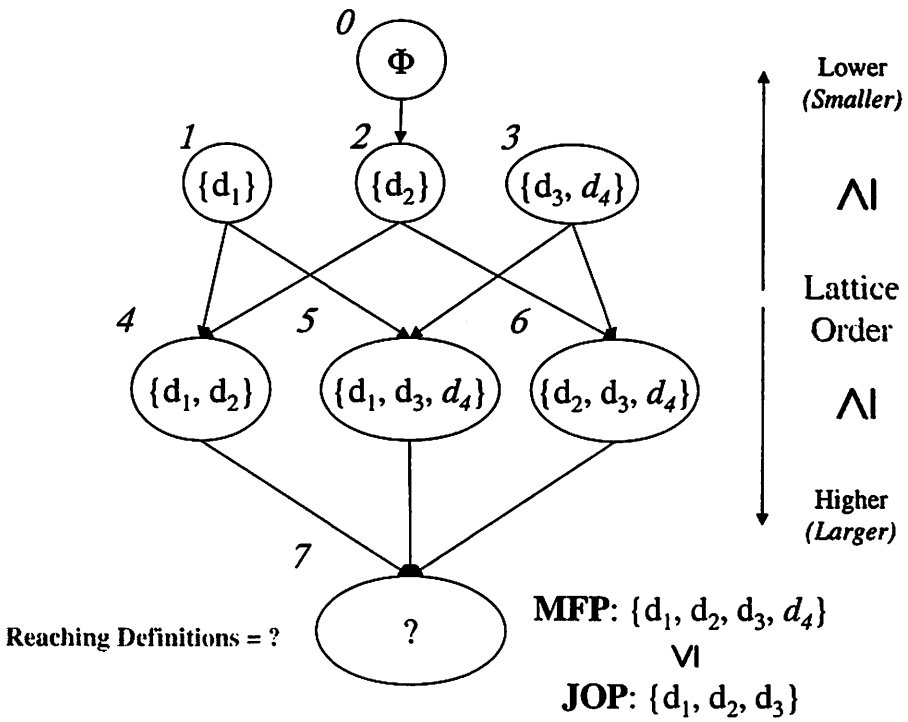


Figure 4.2: Reaching Definitions: MFP vs. JOP

To explain the previous definitions let us consider again the example of Aho [2] on *Reaching Definitions* as shown in Figure 4.2. Assume we are trying to find the reaching definitions at node 7 in the Figure. During an iteration (or a search if you will) we might get the solution (the **MFP** of this iteration) of $\{d_1, d_2, d_3, d_4\}$ where our analysis procedure has not realized yet that the definition d_4 is *useless* because, if we assume Figure 4.1 is the *optimal* information representation we seek, the definition is killed before reaching node 7. This solution is *greater* that the **JOP**. The notion of *greater* or *bigger* is that of the lattice order $\geq$ where: $\{d_1, d_2, d_3, d_4\} \geq \{d_1, d_2, d_3\}$

A *safe* solution to the *Reaching Definitions* problem would therefore include every definition in the solution since this *over-estimate* will result in fewer optimization opportunities but will *not* invalidate the result; however we will always use the **MFP** as the largest solution we report, and keep looking for *smaller* solutions and hopefully reach the **JOP**.

Lemma 4.2.1 *The* **MFP** *will always be a* safe approximation *to the* **JOP**. *Acceptable solutions are* at least as good as *the* **MFP**.

Proof. $\forall$ fixpoints x, $x \geq$ **MFP** by Definition 4.2.14. Therefore since the **JOP** is reached by a sequence of steps that refine the **MFP**[7], **JOP** $\leq$ **MFP**, and the **MFP** is always *safe*.

The **MFP** is the approximation normally discovered during an iteration [82]. In general the "larger" or "higher" a solution is in the join semilattice the safer it is. The "smaller" or "lower" a safe acceptable solution is in a join semilattice the better a solution

[7] We may never reach it by the method

it is[8] (i.e. it has *more* data flow information). It should be noted here that the initial solution for iteration will *not* usually be a safe solution. For example, in Reaching Definitions the initial guess is that no definition reaches any node (i.e. we start off with $\emptyset$ for the reaching definitions subsets at the nodes) which is an unsafe solution.

Definition 4.2.16 *The* optimal solution X_{opt} *we are looking for in join problems is:*

$$\mathbf{MFP} \geq X_{opt} \geq \mathbf{JOP}.$$

Similarly we can define the following for *meet* problems.

Definition 4.2.17 MFP Approximate Solution *A* maximum fixpoint *(also denoted by* **MFP** *and distinguished by the context) is a fixpoint which is greater than or equal to any other fixpoint.*

Definition 4.2.18 MOP Exact Solution *We define the* meet over all paths *denoted by* **MOP** *to be as follows. Let $PATH(n)$ denote the set of paths from the initial node N_0 to node $n \in N$ in the flow graph G, then the* **MOP** *at node n is [58]:*

$$\bigwedge_{P \in PATH(n)} f_P(\top)$$

Definition 4.2.19 Optimality Definition *The* optimal solution X_{opt} *we are looking for for meet problems is.* $\mathbf{MFP} \leq X_{opt} \leq \mathbf{MOP}$.

Of course we'd like X_{opt} to be *as large as possible* that is as far from the **MFP** and as close to the **MOP** as possible. Ideally we would like to iterate to the true solution which we hope to be the **MOP** however there are three potential problems that may prevent us from accomplishing this (adapted from [82]):

[8] Dual holds for meet semilattices

1. The true solution may not be expressible in lattice form: This is the case where the **MOP** does not exist since approximations may have already occurred in the modeling itself. This is quite common in *aliasing* ([9]) where it is very hard to make safe assertions. In our FFG representation we assume no aliasing, and thus free the designer from worrying about this implementation detail ([89]) when describing the intended function, so this does not come into play (see Section 4.7.1 for future enhancements to the FFG).

2. The true solution may not be fixpoint computable: This is the case where the **MOP** exists but the DFA framework instance itself cannot handle the situation adequately (such as Constant Propagation in distributive frameworks [57]). In this case a more elaborate DFA framework will need to be developed at a higher computation cost in the analysis steps.

3. The true solution (**MOP**) takes too much (even infinite) computation: Aliasing falls into this category where possibly the number of indirect referencing can be limited but a large amount of computation is required for the analysis stage.

4.2.4 Iterative Algorithm for MDFA Instances

I have chosen in this work to use the iterative algorithm [2], which is typically the algorithm used in optimizing software compilers[9], for finding the "optimal" solution of the framework formulation because it is general and applies to any graph G without imposing any restrictions. Other approaches such as the elimination iterative approach,

[9]The algorithm lends itself to efficient implementation.

and the interval search approach require that the graph be *reducible.* Those approaches can achieve in theory a speedup over the iterative scheme; in practice however it has been shown by Kennedy [59] that typically this is not the case. Figure 4.3 shows a non-reducible flow graph. I informally define *reducibility* in the following definition.

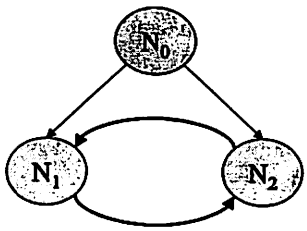


Figure 4.3: An Irreducible Flow Graph

Definition 4.2.20 Reducible Flow Graph *A flow graph G is said to be* reducible *if entry to every loop in G is through the loop's header only.*

While compilers for a given language are aware of the control structures in the input language (FORTRAN, C etc...) and can possibly make assumptions on the flow graph, I do not make any assumptions on the FFG *structure* in the embedded domain since we could potentially have a *multitude* and a *variety* of input languages, and I do not wish to impose any demands on the front-end for handling non-reducible flow graphs if they exist (such as node splitting techniques [2] to make the graph reducible). As mentioned earlier I opt here to use a *conceptually simple and general DFA engine* in order to permit ease of future expansion and building of algorithms and heuristics, and to port this work to other endeavors (such as co-design or IP assembly [111]) that can benefit from such techniques. The algorithms in our simple DFA framework will of course *run faster* if the flow graph is

reducible than non. The iterative algorithm for general data flow analysis frameworks is shown next. While I present a *meet* forward flow algorithm, a *join* algorithm is quite similar with $\wedge$ replaced by $\vee$, and $\top$ replaced by $\bot$.

Algorithm 4.2.1 (Iterative Algorithm for General Frameworks)

Iterative Algorithm for General Frameworks(G, F)

begin

foreach *node* $n \in G$[10] **do**

$Reach(n) = f_n(\top)$; /* *Initialize* */

end foreach

while $\exists m \in G$ *such that* $Reach(m)$ *changed* **do**

foreach *node* $n \in DFS(G)$ **do**

$Reach(n) = \bigwedge_{P \in Pred(n)} Pass(P)$; /* *Confluence Equation* */

$Pass(n) = f_n(Reach(n))$; /* *Transfer Equation* */

end foreach

end while

end

The algorithm is along the lines of its lattice theoretic foundation. For each node in the flow graph we iteratively use the function $f \in F$ on the semilattice L of the problem information I until we converge to a solution. The natural question that arises: Does this algorithm ever halt ? Does the **MFP** improved with each iteration ever stabilize to X_{opt} ? The next Theorem answers these question in the affirmative. Note that $DFS(G)$ stands for a depth first ordered version of graph G; for the time being we can neglect this and

[10]Shorthand for $n \in V$ of G

assume we are using some *order with repetition* to visit the nodes $n \in G$ that abides by the condition to follow (taken from [58]). In the sequel I denote the algorithm result at node n after step m by $A^{(m)}[n]$.

Assumption 4.2.1 Node Visit Order *The node order in Algorithm 4.2.1 is such that if* $\exists$ *a node* $n \in V - \{N_0\}$ *where* $A[n] \neq \bigwedge_{p \in Pred(n)} f_p(A[p])$ *after we have visited node* n_s *in the sequence, then* $\exists$ *integer* $t > s$ *such that* $n_t = n$. *From the previous statement: if after visiting node* n_s, $A[n] = \bigwedge_{p \in Pred(n)} f_p(A[p])$ $\forall n \neq N_0$ *then the sequence eventually ends.*

Theorem 4.2.1 Halting and Optimality Theorem *Algorithm 4.2.1 halts and will do so in no more than* n, *where* n *is the number of nodes in* G, *iterations (of the* **while** *loop) resulting in the* best *attainable solution.*

Proof. *Adapted from [57] and [58]* By induction on m, the number of steps applied in Algorithm 4.2.1:

$A^{(m+1)}[q] \leq A^{(m)}[q]$, $\forall$ nodes $q \in G$

According to the condition on the sequence of nodes being visited in Assumption 4.2.1, after we apply the k-th step of Algorithm 4.2.1 either $\exists$ an integer j s.t. $A^{(k+j+1)}[q] < A^{(k+j)}[q]$ for some node $q \in G$ or the sequence will halt. Since L **is bounded** and the **FFG** G **has a finite number of nodes** n the sequence is guaranteed to end and the algorithm will eventually halt. The result after the algorithm halts is the *best* or *optimal* solution X_{opt} in the sense outlined earlier in Definition 4.2.19.

Theorem 4.2.1 states that the algorithm halts, and in the **worst case can take** $O(n^2)$ **steps to converge**. Intuitively stated, the Theorem says that the "effect" of the

information coming from any node n's predecessors is felt at n, in the worst case if information travels through all the nodes to get to n, essentially that ***all the other nodes*** are on a valid information flow path (worst case happens when every node is connected to every other node in the flow graph). But we can improve the **average complexity** of this algorithm. Since it has been found ([2]) that most of the useful information reaching a node n gets there typically by an acyclic path while the ***loops kill information***, the performance of the algorithm can be improved by finding a good visit order of the nodes $n \in G$. As the intuitive analysis earlier leads us the believe, studies (such as [54]) have shown that **depth first search** provides an efficient ordering of the nodes of the flow graph where $d+2$, iterations of the *while loop* in Algorithm 4.2.1 is sufficient for typical data flow problems, d being the graph *depth* which is the maximum number of retreating edges in a DFS spanning tree. I will not dwell on this the interested reader should consult the references; for our purposes here I will always order the FFG graph G a priori in DFS order, and I can only add my own experience with control-dominated embedded system designs (where loops are not typically deeply nested)[11] to Knuth's experience that d is on average 3 or less [69]. Another process that we need in Algorithm 4.2.1 is the computation of the *transfer functions* f_n at each node n, next I describe how we can compute these functions.

Assumption 4.2.2 Information Transfer Functions *In the MDFA frameworks we deal with, we assume that any* information processing *function can be expressed as follows:*

$$\forall n \in G,\ f_n(Reach(n)) = (Reach(n) - Kill(n)) \cup Gen(n),\ \textit{where:}$$

$Reach(n)$ *is the information* reaching *node* n,

$Kill(n)$ *is the information* killed or invalidated *in node* n, *and*

[11] Using the Esterel front-end for specification for example

Gen(n) is the information generated *in node n*

In summary, the algorithm applied to MDFA frameworks results in a unique solution X_{opt}, which is the **MFP** of a set of data flow equations where: $X_{opt} \leq MOP$. While control flow can be inferred from the structure of the FFG flow graph, data flow information can be collected by setting up and solving a system of equations that relate data at various points in the FFG module behavioral description using *set operations*, and in fact most of the information that we need can be collected and operated on using *bit-vectors* [2]. In the context of the FFG and the problems we solve, M in Definition 4.2.9 is an algorithm for extracting information from the operations (i.e. the *quadruples* see Chapter 3) associated with each node N, and grouping it into a set which becomes the node label for the corresponding node in the data flow problem instance graph G; f at each node N can be fully represented by the pair of equations describing information flow while the meet operator for forward flow problems is set intersection $\bigcap$.

4.3 The FFG Data Flow and Control Optimization Algorithm

Our optimization *objective* at this architecture[12]-independent level can be stated as follows

Proposition 4.3.1 *Architecture-Independent Optimization Objective The* optimization objective *at the* architecture-independent *level is* redundant (useless) information elimination

[12]Implementation

in the FFG. In other words, our goal is to represent the information in an optimized **FFG** *that has a minimal number of nodes and operations associated with these nodes.*

The *FFG Data Flow and Control Optimization Algorithm* can be stated as follows.

Algorithm 4.3.1 (FFG Optimization Algorithm)

FFG Optimization Algorithm(G)

begin

while changes to the FFG **do**

Variable Definitions and Uses

FFG Build

- *Build Graph, Symbol Table, Instruction Table*
- *DFS order*
- *Node Reaching Definitions and Reached Uses*

Reachability Analysis

- *Unreachable Node Elimination*

Normalization

Available Elimination

- *Available Computation Determination*
- *Available Computation Elimination*

False Branch Pruning

Copy Propagation

Dead Operation Elimination

end while

end

It should be noted that while some of the aforementioned optimization steps can be cast in a *distributive* data flow information gathering framework, the overall FFG optimization flow subsumes the constant propagation problem (combination of *copy propagation* followed by *normalization*) and therefore can only be formalized using a **monotone data flow framework**. A more rigorous analysis of the framework properties is presented in Section 4.4. Before delving into the properties however, let us first see what these steps are all about.

The reader should be aware that while most of these techniques are variants of the classical ones they have all been **specialized** to our domain where **reactive semantics** apply, and all are *safe* in the sense that they follow the **Information Manipulation Safety Requirement** set forth in Proposition 4.2.3. In copy propagation for example, *input* sources are never propagated since the *sampling of inputs* is governed by the MOC, and is not something we are looking into at this level of abstraction (we will refine this in Chapter 5). Computations that depend on inputs also have special handling, and output assignments are also always preserved[13]. In essence all *interactions with the environment* are handled differently that regular operations. In the following discussion I will assume this issue is clear and explain the basic operation of the steps.

4.3.1 Variable Definitions and Uses

In this step (performed at every iteration) we update the variable definitions and uses and store the updated information in the symbol and instruction tables. A "definition"

[13] An output is most likely never "used" if we were to apply the pure semantics of the DFA analysis, but we should never throw such an assignment away.

is where a variable is defined, a "use" is where a variable is used for example:

Example 4.3.1

`a = b + c` *defines* `a`, and *uses* `b`, and `c`.

Uses and definitions are the *currency* used and manipulated in the data flow analysis.

4.3.2 FFG Build

In this step we build the FFG flow graph (initially from the behavior description, and at every iteration to *update* the information within each node and remove the unreachable nodes), as well as an updated *symbol table* containing all the variables in the FFG, and an *instruction table* containing all the FFG (i.e. CLIF) statements, auxiliary data structures. These data structures provide a higher visibility for variables and FFG statements, and therefore speed up, and simplify the implementation of the algorithms to be introduced shortly. In addition to the analysis leading to building the FFG control structure, operations associated with each node are broken up into a canonical representation of *quadruples* (see Chapter 3, and Section 3.2.2 in particular).

We then perform a DFS ordering of the nodes in the FFG. Next we solve the *Reaching Definitions* problem which is a *forward flow* join problem as discussed earlier in this Chapter, followed by the *Reached Uses* problem. The former analysis is used to detect uses of variables before their definition in which case the initial value is used[14]. The latter is a *reverse flow problem* useful for detecting useless assignments; those assignments with no reached uses. This is shown in Figure 4.4. The *use* of `a` is *reached* from the assignment

[14]The Front-end flags the error when a variable in the specification is used but not defined.

statement `a = b + c` if there is *some* path from the assignment to the use along which `a` is not redefined [76].

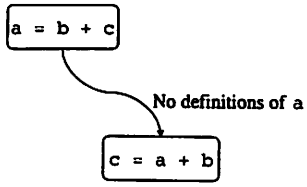


Figure 4.4: Reached Uses

4.3.3 Reachability Analysis

Definitions and uses are also the currency used in the *control flow analysis.* Labels (of nodes) are defined and used as shown in the following example:

Example 4.3.2

`if (condition) then goto S1L1;` *defines* the node label `S1L1`.

...

`S1L1: a = b + c;` *uses* the node label `S1L1`.

...

This simple example gives the reader some intuition on how reachability is expressed in the definition/use currency: A label not defined is unreachable, whereas a label defined but never used means we should flag trouble ahead. Reachability is therefore a very simple process performed as part of building and analyzing the FFG. I have just described an *incremental* method for performing reachability based on the definition of labels. Alternatively *reaching definitions* can be used to see if the definition of the initial FFG label (N_0)

reaches any one point in the flow, if it doesn't then that point is declared unreachable[15]. Unreachable node elimination involves marking the corresponding node for removal from the flow graph, as well as all its associated operations, and definitions. Of course all the labels reachable (i.e. defined) from this node only will be marked for deletion as well.

4.3.4 Normalization

Normalization is the process of making the representation *canonical* and making all computations *visible* by identifying computations and storing them in a *hashing table* and associating a unique definition with the operation. This step is very useful for the following analysis and optimization steps since it simplifies the process of detecting similar operations. The procedure is shown in the simple example that follows:

Example 4.3.3

Assume we have:

```
a = b + c;

d = f * a;

x = c + b;
```

It gets transformed after this step to:

$T_{(b+c)}$ `= b + c;`

`a =` $T_{(b+c)}$`;`

$T_{(a*f)}$ `= a * f;`

`d =` $T_{(a*f)}$`;`

$T_{(b+c)}$ `= c + b; /*` This is the same $T_{(b+c)}$ `*/`

[15]Current implementation in toolset

x = $T_{(b+c)}$;

Note the canonical ordering of source operands enforced (based on string hashing) to identify similar operations. This step also performs simple *algebraic simplification* for things like addition to 0 or multiplication by 1 and 0 etc... Such optimizations are crucial not only to the reduction of "silly" operations but also to the identification of other optimizations in the steps to follow. **Constant Folding** is also performed in this procedure.

4.3.5 Available Elimination

The *Available Computation Determination* is an implementation of the available expression problem ([57]) that uses the previous step (normalization and variable definition and uses) to identify available computations at each FFG node's input *meet*. This is a forward flow problem and therefore has the same time complexity as Algorithm 4.2.1. Figure 4.5 shows how this analysis is performed and what it means.

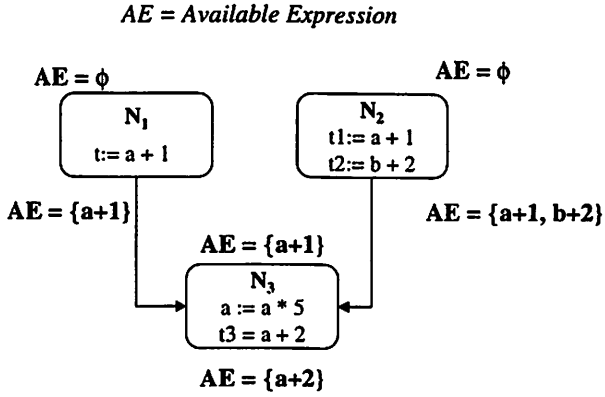


Figure 4.5: Available Expr

If we examine node N_3 in the Figure we c. ..e that $\bigwedge_{P \in Pred(N_3)} Pass(P)$ is $\{a+1, b+2\} \wedge \{a+1\} = \{a+1\}$ and therefore a+1 is *available* to node N_3 i.e. $Reach(N_3) =$

$\{a+1\}$. Now, $Gen(N_3) = \{a*5, a+2\}$ while $Kill(N_3) = \{a*5, a+1\}$ since a is redefined in the first statement of N_3, therefore $Pass(N_3) = \{a+2\}$. Once the analysis step is complete we perform an ***elimination*** step which is a ***linear*** scan of the FFG nodes that throws away redundant computations (i.e. since they are available, we need not compute them again).

4.3.6 False Branch Pruning

Conceptually, this step is intended to remove false branches that get exposed as data flow analysis is performed. The example shown in Figure 4.6 demonstrates this.

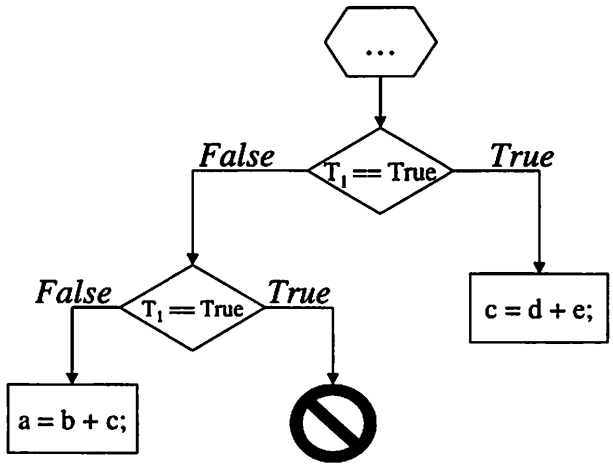


Figure 4.6: False Branch Pruning

The second `true` branch for the operation T_1 `==` `True` is never exercised and can therefore be ***pruned.*** Purely control optimization approaches (such as BDD approaches applied in [29]) cannot perform such an optimization since it depends on data semantics, not to mention that these "data-value blind" approaches cannot hope to *uncover* this optimization to begin with since no data flow analysis and optimization is done. While this transformation can conceivably be implemented with a dedicated analysis that checks avail-

able computations at each branch, practically I implement this transformation with a simple *linear in the FFG nodes i.e. $O(n)$ where n is the number of FFG nodes* technique based on *redundancy addition* as shown in Figure 4.7. I add *redundant assignments* at the head of the true branch and false branch targets. While I have not *performed* the branch pruning, the following steps namely copy propagation and dead elimination will eliminate the useless branches for me as we will see shortly.

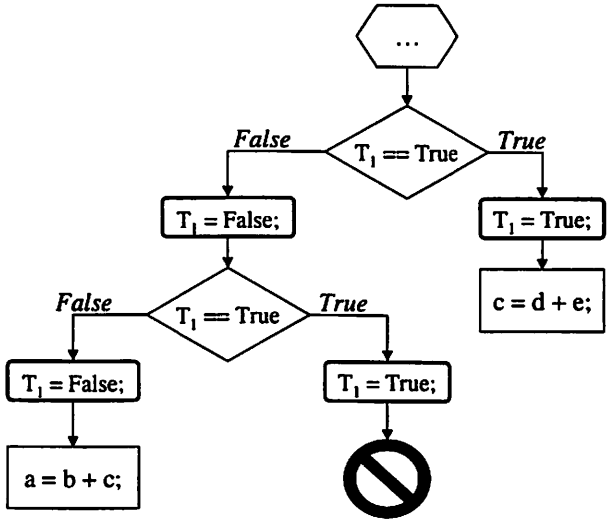


Figure 4.7: Redundancy Addition

4.3.7 Copy Propagation

Copy propagation is the process of *substituting equivalent* variables and is implemented as a linear scan of the FFG nodes[16], it is useful for eliminating "intermediate" computations and thus eliminating those variables as well. Copy propagation can also simplify the operations by substituting constant values for variables thus opening the potential

[16]Of course it relies, as most everything else, on determination of definitions and uses which is an $O(n^2)$ process

for further simplifications. For example if we consider Example 4.3.3 again, after this step it will be as follows, possibly optimizing away the variable a:

Example 4.3.4

...

a = $T_{(b+c)}$;

$T_{(a*f)}$ = $T_{(b+c)}$ * f;

...

In the context of control for example we have seen the false branch pruning implementation, after this step we will have a situation like:

Example 4.3.5

$T_{(false==true)} = (false == true)$;

if ($T_{(false==true)}$) goto label;

...

which evaluates to:

if (false) goto label; /* will be removed by dead elimination step */

4.3.8 Dead Operation Elimination

The best name for this step is *redundancy elimination.* Here a *live variable analysis* is performed on the operation destinations. All the useless operations are removed after the analysis; dead control is removed as well.

Definition 4.3.1 Live Variable *A variable is* live *at any point p in the FFG behavior if there is a subsequent* use *before a* redefinition.

4.3.9 Optimizing Function and Procedure Calls

Procedures are not supported in the FFG operations since typically procedures are meant for passing *indirect* variables and aliasing is not supported currently in the design representation. *Single-output functions* are supported in the FFG (as an operation), and in fact this is another **specialization** that all the above analysis and optimization algorithms share: the handling of such functions. At this architecture independent level we make no assumption on the functions, and treat them in a *safe* manner. In Chapter 5 we will see how we can optimize more given the knowledge or assumption that these functions are *stateless arithmetic or relational* functions. In the architecture-independent optimization phase we can only "propagate" simplifications to the arguments of such functions, but we cannot move computations in or out of the function boundary, nor can we move the function call around because we fear changing the behavior semantics. In general we will assume that functions written in the host language (e.g. "C" in the case of our Esterel or Reactive VHDL front-end) will be optimized (the body is optimized, and possibly the complete function is inlined) for synthesis by the target compiler. This is an adequate assumption since these functions are not reactive but only perform a (usually single) computation function, and the optimizer can do a good job in the optimization.

4.4 Properties of The FFG Optimization Algorithm

Now that I have outlined the main steps in the optimization algorithm, let us go back and attend to stating and proving some properties of the algorithm.

Theorem 4.4.1 Information Preservation and Safety Theorem *The FFG Optimization Algorithm* preserves *the information embodied in the FFG behavior, and is* safe *(Refer to Propositions 4.2.2 and 4.2.3 for the meaning of information preservation and safety).*

Proof. *Sketch* Each step in the FFG Optimization Algorithm collects information, or adds/eliminates *redundant* information and is therefore information preserving and safe *by construction*[17]. The *sequence* of the above steps is *information additive* and is therefore information preserving and safe.

The next assumption states that *heuristically* speaking the order of steps we have applied is a reasonable, and more forcefully an *optimal* and *efficient* order.

Proposition 4.4.1 Optimal Order of Steps *The* order *of steps as shown in the* FFG Optimization Algorithm *is* optimal *and* efficient *since every step uses information from the previous one, and follows it "naturally".*

So the previous proposition states that since each step *facilitates* the job of the next, and since in fact each step is *dependent* on the previous one then their composition is a *reasonable application order.* Any other order would not have the required effect, and would make the iteration take longer to converge. Let us now turn our attention to this composition of the steps or the *FFG Redundancy Identification and Removal Function* to identify properties of this function and consequently give formally the solution properties of the overall flow[18] which is the **FFG Optimization Algorithm.** To start off I need to

[17] Of course each such step can be formalized in terms of its transfer functions $f \in F$ for the problem instance, and a more rigorous proof can be called here to validate the "correct by construction" claim. The inquisitive reader can refer to reference [57] for such a sample proof of available expressions.

[18] Of the ordered composition of the algorithmic steps

state and then prove the following lemma.

Lemma 4.4.1 Monotonicity Lemma *Let λ be a semilattice of all the information captured in an FFG behavior denoted by 2^I, and $s_1, s_2, ..., s_n$ be functions*[19] *on λ.* $[\forall x, y \in \lambda, \forall 1 \leq i \leq n, s_i(x \wedge y) \leq s_i(x) \wedge s_i(y)] \Rightarrow \forall x, y \in \lambda\ [s_1 s_2 ... s_n(x \wedge y) \leq s_1 s_2 ... s_n(x) \wedge s_1 s_2 ... s_n(y)]$

Proof. Given $\forall i\ s_i(x \wedge y) \leq s_i(x) \wedge s_i(y)$; Let us assume that $s_i ... s_n(x \wedge y) \leq s_i ... s_n(x) \wedge s_i ... s_n(y)$ then by the Monotonicity Equivalence Property (Property 4.2.1) $s_{i-1}(s_i ... s_n(x \wedge y)) \leq s_{i-1}(s_i ... s_n(x) \wedge s_i ... s_n(y))$, which leads to (from our assumption) $s_{i-1}(s_i ... s_n(x) \wedge s_i ... s_n(y)) \leq s_{i-1} s_i ... s_n(x) \wedge s_{i-1} s_i ... s_n(y)$. The lemma follows by backward induction on i.

My intent in the *Monotonicity Lemma* is to introduce the lattice λ that "pulls together" the individual lattices of the separate optimization steps[20] in the FFG Optimization Algorithm, in order to be able to speak of this composition in a theoretical fashion considering an FFG G and a single λ to capture the behavior and its information. Simply stated each of those steps can be applied on a *restriction* of λ where the algorithm in the respective step is only concerned with a *subset* or informally a part of the information embodied in λ. While the lemma is stated in an abstract fashion, concretely what I aim to do is associate each s_i with algorithmic step i in the **ordered composition with repetition** of the FFG Optimization Algorithm. To revisit the Optimal Order of Steps proposition, the reader should note that while we can conceptually think of the notion of a function f_{ideal} that performs *Total Redundancy Identification and Removal* on λ building an algorithm

[19] Transformations or Optimizations

[20] A la *Abstract Interpretation* of Cousot in [34]

to perform such a function is undecidable, so we resort to decomposing the problem into several steps, and then say something about the *optimality* of the composition function or algorithm A where $\forall m \in G\ A[m] = s_1 s_2 ... s_n[m]$. Of course the quality of the solution A with respect to f_{ideal} is *inferior* but we have no way of actual comparison since A is the "best" algorithm, in the sense of decomposing the problem given a set of analysis and transformation steps that form the fundamental building blocks, we can build.

Theorem 4.4.2 Halting Theorem *The* FFG Optimization Algorithm *halts.*

Proof. By the halting proof of Theorem 4.2.1 each optimization step halts. Since λ is bounded and G is finite, and the FFG Optimization Algorithm is preservative[21] by Theorem 4.4.1, the *ordered composition* of the optimization steps also halts.

Intuitively from Theorem 4.4.2 (again since λ is bounded and G is finite, and the steps do not add any new information) we can see that the final optimization step (a function on λ) may have to wait for n iterations where n is the number of FFG nodes until it gets the proper information to do its job. By the *Monotonicity Lemma*, this gives an **upper bound complexity of $O(n)$ for the external iteration of the FFG Optimization Algorithm.**

Theorem 4.4.3 Safety Theorem *The solution from application of the* FFG Optimization Algorithm *is* safe[22].

[21] Safety of the algorithm guarantees that we are conserving the I/O semantics *by construction*, but has no real bearing on information preservation.

[22] This is not to be confused with *safety* of the *algorithm*; please refer to appropriate definitions.

Proof. Each step s_i in the FFG Optimization Algorithm is a *monotone* function on λ and after application i of a transformation step in the sequence of such steps we have:

$$f_{ideal}(x \wedge y) \leq s_i(s_{i-1}...s_1(x \wedge y)) \leq s_i s_{i-1}...s_1(x) \wedge s_i s_{i-1}...s_1(y)$$

So we have a safe solution after each step. The final solution (when the algorithm halts) is also safe.

Theorem 4.4.4 Closeness and Optimality Theorem *The solution of the* FFG Optimization Algorithm *is* as close as possible *to the ideal solution and* optimal *for the* given order *of the optimization steps.*

Proof. When the algorithm halts after applying m such optimization steps we have:

$$s_m(s_{m-1}...s_1(x \wedge y)) \leq s_m s_{m-1}...s_1(x) \wedge s_m s_{m-1}...s_1(y)$$

which states that the solution after each step is always smaller[23] with respect to the lattice order. So, by monotonicity of the steps s_i, the solution we obtain from the algorithm is *optimal* for a *given order* of application of the optimization steps, and there is a sense by which we can say we have reached the *best possible* solution (given this step application ordering) when the FFG Optimization Algorithm halts.

Theorem 4.4.2 and the *Monotonicity Lemma* lead us to deduce that the outer iteration is linear in the FFG nodes but (unlike Algorithm 4.2.1) there is no clear indication as to how we can *monitor* this change and determine when the algorithm has converged (and

[23] contracting

therefore when we can terminate the iteration) instead of looping for all the nodes $n \in G$, and try to do better (on average of course) than the $O(n^3)$ overall complexity. To address this issue, I propose the following *stopping criterion* for the FFG optimization algorithm.

Corollary 4.4.1 Iteration Stopping Criterion Heuristic *If we iterate until all the information from the first step in the sequence* reaches *the final step and there are no changes in the* operation count *in the FFG, then most likely there is no further significant change in the solution, and the algorithm can be* stopped. *It is our hope that indeed this is the point at which the algorithm has* converged.

Proof. Stopping Heuristic Validity Follows by generalizing the proof of Theorem 4.2.1 since at every iteration we compute a valid and safe solution (by Theorem 4.4.3), and therefore we can terminate the process when the stopping criterion is satisfied obtaining a safe and valid solution; we say the the FFG has been *improved* by redundancy identification and elimination.

Of course the *worst case* time complexity is the same as the FFG Optimization Algorithm ($O(n^3)$) in the FFG node number n, but the benefit of this algorithm is that on average it is more likely to finish (and quite likely converge to the *best* solution we are seeking) in a constant amount of time spent in the external loop; an overall $O(n^2)$ complexity. $O(n^2)$ in the FFG nodes n is $O(s^2)$ in the EFSM states s. This complexity is what the low level synthesis algorithms spend to *generate* SW code or a HW netlist *without low-level optimizations*, so it is quite desirable since it would not constitute an overhead. It "helps" synthesis run faster in addition to improving quality.

I conclude this section by presenting the FFG Improvement Algorithm[24] based on the *Iteration Stopping Criterion*; this is the ***fast approximate variant*** of the FFG Optimization Algorithm. `OUTER_ITER` is a constant set to the number of optimization steps in the flow. The algorithm measures change by monitoring the number of operations in the FFG: If no redundancy is removed after we wait for information to propagate from the first to the last step (i.e. `OUTER_ITER` times) the algorithm completes.

Algorithm 4.4.1 (FFG Improvement Algorithm)

FFG Improvement Algorithm(`G, OUTER_ITER`)

begin

count = OUTER_ITER; / Initialize Stopping Countdown */*

do

icount = FFG.operation_count();

Variable Definitions and Uses

FFG Build

Reachability Analysis

Normalization

Available Elimination

False Branch Pruning

Copy Propagation

Dead Operation Elimination

If *(icount == FFG.operation_count())* **then**

count = count - 1;

[24]Current default of the non-interactive version in the toolset

```
        else
            count = OUTER_ITER; /* Reset Stopping Countdown */
        end if
    while (count != 0)
end
```

Finally I'd like to conclude this section by pointing out that other less effective but very fast stopping criteria can be devised, such as the following ($O(n^2)$ complexity, but on average is constant based on the analysis of Algorithm 4.2.1).

Proposition 4.4.2 Very Fast Stopping Criterion *If the operation count of the FFG does not change during an iteration (see Algorithm 4.4.1) the iteration process can be stopped.*

The criterion is based on making `OUTER_ITER = 1`; we therefore have a continuum of trade-offs between quality and optimization time complexity overhead. We can for example make `OUTER_ITER` be proportional to n where `OUTER_ITER` $= k * n$, $k < 1$.

4.5 Tree vs. Shared DAG Form of the FFG

I have described in Chapter 3 how the given EFSM behavior can be represented in the *Tree* or Shared DAG forms of the FFG. In this Chapter I have presented a complexity analysis of the optimization flow that depends on the number of nodes n in the FFG. The Shared DAG form of the FFG identifies common computations and pulls them together and therefore has a much lower node count in practice than the Tree form where all the

computations are explicit in Trees. Time and space complexity is therefore lower in the DAG form, and the algorithm completes much faster with lower memory expenditure. The trade-off however is that the *size of the nodes* (basic blocks in software compilation terms) is much *smaller* in the DAG form than the Tree form. This fact makes optimizations *less effective* for each state; a shared node will have multiple edges coming into it leading to more potential for *Kill*-ing data, conceptually speaking, since the operations in the shared node may depend on several execution contexts. So, we can expect to get a considerable speed-up if we use shared DAGs, but a lower output quality.

As we will see in Chapter 5 *Operation Motion* is one optimization technique we can use to *smooth out* the difference in output quality between these two extremes and make it more of a continuum, at the expense of some extra processing in the analysis and optimization phase. In the sequel I will typically (unless otherwise stated) assume that we are working with the Tree form, and are not particularly concerned about the extra time/space overhead in the required processing.

4.6 The Backdrop: Related Work in Optimization

In this Section, I position my work with respect to the bodies of research it is built upon. Previous work in control optimization is mostly based on BDD [25] optimization techniques for Control Flow Graphs (CFGs) such as [7]. The limitation of these optimizations is that they neglect the data and are in fact, as I mentioned earlier, "data value blind"; control optimizations that can result from data analysis (such as dead operation elimination, and copy propagation for example) are not available to such techniques. The

two most relevant bodies of work to my research are:

1. High Level Synthesis (HLS) for *silicon compilation*, and

2. Code optimization techniques for *software compilation.*

High level synthesis for silicon compilation has been an active research area in the past two decades. The focus of such techniques however has been mostly on approaches for scheduling, allocation, and binding of the specification (usually a Hardware Description Language (HDL)) to the hardware implementation such as [46]. General optimization techniques, for example common sub-expression extraction, and constant folding, are applied in a local fashion. Bergamaschi recently proposed a design representation, Behavioral Network Graph (BNG), for unifying the domains of HLS and Logic Synthesis [13]. His work recognized the need for an internal design representation on which to perform data path and control optimization before logic synthesis of hardware. The BNG, however, is at an even lower abstraction level than the CDF DAG used for synthesis in the HW/SW Co-design field and is therefore not suitable for our need of a unifying design representation for software and hardware at the high abstraction level in the embedded optimization and co-design domain, as I have argued in Chapter 3.

The literature is rich in data flow optimization techniques, most notably classical optimization techniques of [65], [57], and recent work by [32], [2], and [118]. Most of that work, however, has focused on *hand-written code* optimization and assembly generation for a specific component processor and instruction set. As we have seen earlier, in my work I specialize the aforementioned silicon and software compilation techniques to the embedded systems domain since these techniques can be applied on any internal design

representation, no matter what the abstraction level, and need not be restricted to the final stages of software assembly code generation, or hardware synthesis.

4.7 Future Directions

4.7.1 FFG Extensions and Theoretical Implications

The FFG currently supports only the *integer* data type (*Boolean* type is represented as a zero/non-zero integer). In the future the FFG should handle more data types such as *floats* to support more elaborate mathematical operations as well as *aggregates* (i.e. `structs` in SW and `records` in HW) to support such things that require a collection of information (e.g. messages). Of course, as I stated earlier, this will require more complexity in the analysis not just within similar type operations but more importantly between such data types. In fact such analysis for complex data types may require the formulation of a more elaborate DFA framework since an analysis might need both forward and reverse flow information collection [2]. I touch on a variant (a simplified version) of this data type optimization issue again in Chapter 5 where we use architectural information to optimize the *bit widths* for the *integers* I have been assuming so far here.

As I mentioned earlier *aliasing*, that is allowing pointers in the FFG behavior, can also be supported in the future. Of course strict limitations must be imposed on the handling of such primitives. The FFG/CLIF design representation has some pointer support currently, and the analysis is expanded to handle such cases, but the front-end does not exercise this yet (so I neglected to elaborate on these facilities), since our goal, typically, in the function specification is to minimize the level of implementation detail. I envision

pointer operations as realization constraints that can be imposed later as a refinement where implementation details[25] start to be specified.

4.7.2 Optimization Heuristics

Aside from optimizations resulting from the extension of the FFG, and *architecture dependent* optimizations I discuss in Chapter 5, there is a tremendous opportunity here for devising new *improvement heuristics*. One such heuristic that I came across in a real design in my experimentation is *graph isomorphism* checks. Of course graph isomorphism is an unresolved problem, in the sense that no efficient algorithm for it has yet been found [63]. One potential method is to rely on BDD-like techniques where a canonical ordering of the variables is enforced thus simplifying the problem, but as I touched on earlier there is a large class of data dependent cases that cannot be identified in the BDD approach, and a simple (not exhaustive) heuristic implementation of isomorphic graph subset identification in the FFG and optimization would be quite useful.

These *graph matching* techniques [39], however, would require decomposing the behavior captured in the FFG to a finer granularity than **statements**. A possibly more adequate technique proposed by Vahid in [124] is that of searching for *similar forms* in an encoded string. I believe this latter technique might be formulated more readily than the former in the form of variable definitions and uses; I leave this investigation for the future.

[25] For execution performance enhancement for example

4.7.3 Formal Verification and Abstract Interpretation

I think it is worthwhile to make the point that the data flow and control analysis I have been discussing is (informally speaking) analogous to the issues in formal verification; for example we typically answer questions about the *existence* and the *validity* of statements in the behavior. In fact while formal verification typically works at the "bit" level and for control mostly while the data is abstracted away (as in [55]), here we are manipulating data as well and making assertions that deal with both data and control. Cousot in [34] realized (and provided a formal argument) that since a data flow problem seeks to uncover behavior information, the semilattice L of the framework is embeddable as a *sublattice*[26] of the lattice of possible behavioral assertions. The operations on frameworks can then be interpreted as operations on assertions in an *abstract semantics* for the behavior [82]. Determination of behavior properties by abstract interpretation is **formally equivalent** to the algebraic framework presented in this Chapter [82] (also see [80]).

While this brief discussion solidifies the *correct by construction* claim of the lattice theoretic framework, in the future, I'd still like to explore the *abstract interpretation* framework further and see what that line of thought has to offer in terms of optimization as well as formal verification avenues to answer queries about the behavior of a task represented by the FFG.

In this Chapter, I have presented several new variations on the classical algorithms that leverage the FFG representation and are particularly useful in the embedded systems domain. I also presented the **FFG Optimization Algorithm** as it relates to *function*

[26] A *sublattice* of a lattice is a subset that contains for any two elements their join and their meet.

optimization. In the Chapters to follow, I introduce new *constraint-driven* algorithms for function/architecture co-design inspired from the classical notions and algorithms I introduced in this Chapter. These algorithms will be integrated with the function optimization algorithms to form the refinement stages of the **Function/Architecture Optimization Framework**: macro-architecture and then micro-architecture aware optimization and co-design algorithms.

Chapter 5

Function/Architecture Optimizations

A conceptual figure of function/architecture co-design is shown in Figure 5.1. The idea is to employ a suitable constrained task function representation that enables trade-off and co-design at several abstraction levels between the function and architecture *before* mapping the function onto the target architecture.

5.1 Function/Architecture Representation: AFFG

I use the **Attributed Function Flow Graph (AFFG)** to represent architectural constraints impressed upon the functional behavior of an EFSM task. The AFFG is created from the FFG by imposing architectural constraints or information given by the user or obtained through analysis (e.g. through inference or profiling) in the manner shown in Figure 5.2. The AFFG is, informally speaking, a "schedule-aware" FFG since it typically

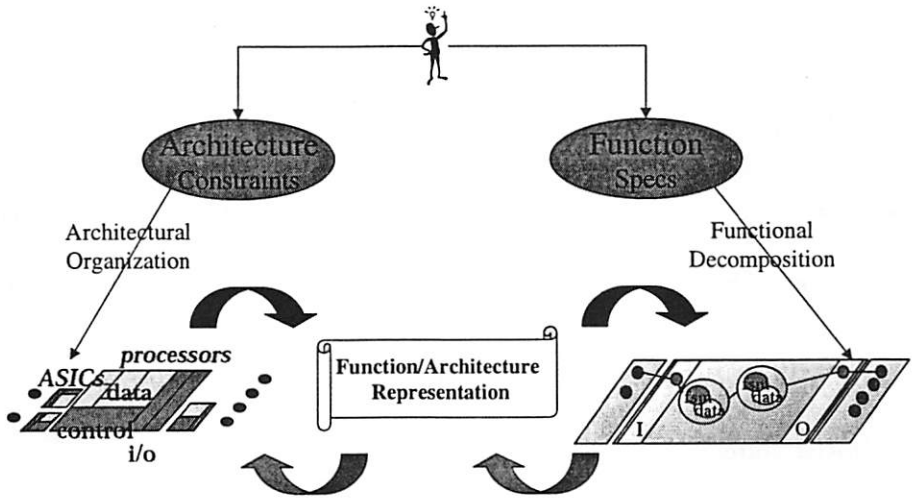


Figure 5.1: Function Architecture Co-design

has at the very least the *state schedule* attribute. Of course we will add more attributes as we refine the AFFG[1], but the known schedule is one of the key enabler attributes for the architecture specialized optimization algorithms. The AFFG definition follows.

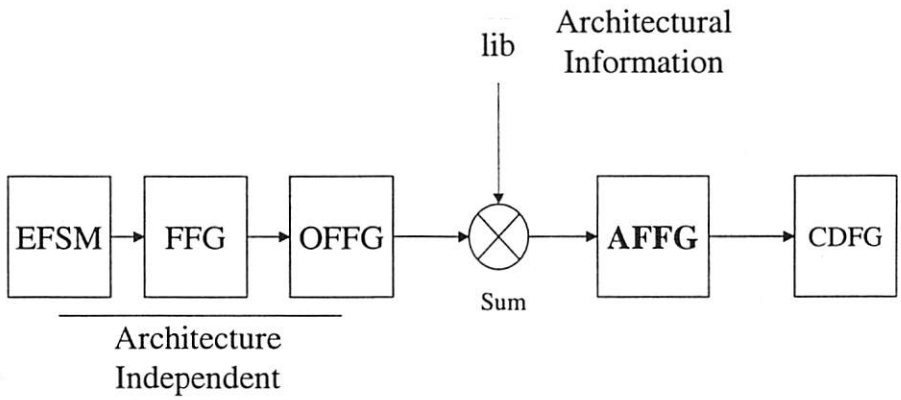


Figure 5.2: Architecture Dependent Representation

Definition 5.1.1 *An Attributed Function Flow Graph (AFFG) is a triple* $G = (V, E, N_0)$

[1]Attributes that document the refinement are also maintained e.g. source-level debugging information (preserved for inputs and outputs in the current toolset).

where:

- V *is a finite set of nodes*
- $E = \{(x,y)\}$, *a subset of* $V \times V$, *where* (x,y) *is an edge from* x *to* y *where* $x \in Pred(y)$, *the set of predecessor nodes of* y, *and* $y \in Succ(x)$, *the set of successors of* x.
- $N_0 \in V$ *is the start node (header that leads to the node(s) corresponding to the EFSM initial state(s)).*
- Operations *are associated with each node* N. *The only order imposed on these operations is that of data dependency (to preserve functionality).*
- Attributes *are associated with nodes and operations within nodes. We assume that at least a* state attribute *is obtained from the font-end.*

The optimization algorithms presented in this work are therefore broken into 2 levels:

- Architecture Independent level: The FFG is analyzed and optimized as a collection of operations as in the classical software optimization approaches except that I/O is preserved (operations with inputs and outputs have specialized handling). For the lack of additional knowledge we assume all the operations form a *sequence* ordered by the front-end[2]. This optimization processes has been discussed in detail in Chapter 4.
- Architecture Dependent level: The I/O schedule and the state assignment are taken into account, and nodes (and operations inside them) within states are optimized,

[2]That is, we manipulate operations as in the classical optimization approaches for *sequential* code generation; of course this need not be the case since we need only preserve the I/O semantics but this is the **simplest** way to deal with this issue without any guidance on what to do

using the guides and constraints from these and other attributes overlayed onto the FFG, followed by an allocation of registers and operations step. This stage is performed on the AFFG, the state scheduled *refinement* of the FFG, and is the topic of this Chapter.

5.2 Function Architecture Co-design in the *Macro*-Architecture

In Chapter 4, I discussed *safe* optimizations that can be performed by considering the FFG behavior restricted only by preservation of the I/O semantics. In this Chapter I consider the AFFG which has a set of *attributes* associated with the nodes and possibly also with operations within those nodes. These attributes are either solicited from the user like the state schedule, that is the *schedule* for the EFSM task execution, obtained from the "front-end" (e.g. Esterel or Reactive VHDL as I have described in Chapter 3) and used to qualify the AFFG nodes for example, or are obtained by inference or profiling such as the cost of operations like addition and multiplication, and the visit probability of AFFG node collections. These attributes, as we will see shortly, are the "costs" that *guide* or *constrain* the optimization algorithms performed at this level. In the next Sections I present my work in developing techniques for performing this constraint-driven optimization for function/architecture co-design. My optimization objective is outlined in the following proposition.

Proposition 5.2.1 *Architecture-Dependent Optimization Objective The* objective *at the* architecture-dependent *level is to* optimize *the* **AFFG** *task representation for* speed of execution *first and* size *(area of hardware, code size of software) second given a set of* ar-

chitectural constraints[3]. *Since we have more information about the target application and architecture we can also invoke* constraint-driven redundant (useless) information elimination[4] *which will improve both our primary (speed) and secondary (size) objectives.*

5.3 Operation Motion in the AFFG

The reader may be wondering why I have not mentioned the famous Common Subexpression Elimination (CSE) as yet. I have presented its (not so famous) cousin Available Expression Elimination (AEE); the difference between the two is that CSE has a reverse flow. The objective in CSE is to move similar operations from blocks to their predecessors if possible. So, in fact, this problem is better addressed in the most general term of **Operation Motion** in the context of the AFFG representation. Operation Motion is analogous to *Code Motion* from the software domain a technique for improving the *execution speed* or efficiency of a program by avoiding unnecessary re-computations of a value at runtime [67]. The primary goal of code motion is to minimize the number of computations on every program path. I present here a technique for task runtime response improvement based on the code motion (code hoisting) concept from the software (high level synthesis) domain. *Relaxed Operation Motion* is a simple yet powerful approach for performing safe and useful operation motion from heavily executed portions of a design task to less visited segments [116]. I introduce in this Section my algorithm, how it differs from other code motion approaches, and its application to the embedded systems domain. Investigation results, to be presented in Chapter 9, indicate that cost-guided operation motion has the potential to

[3]Additional domain information if you will
[4]Tuned; architecture/application driven

improve task response time significantly.

5.3.1 Related Work

There is a body of work on code motion (hoisting) from the software (high level synthesis) domain(s). The goal of code motion is to avoid unnecessary re-computations at runtime [2]. The creation of temporary variables (i.e. registers) to hold these computations typically improves runtime for most target architectures. Code must be relocated to valid program points and this movement must be safe, in the sense that it must not change what the program flow is intended to compute. The main strategy for code motion is that of moving candidate operations (loop-invariants in particular) as early as possible in the program as in [85] and [68]. In practice code movement to the earliest program points can create pressure on the target architecture resources for example because of the limited number of registers resulting in "spills" to main memory[5]. A more practical approach involves also performing temporary lifetime minimization as in the work of Knoop [67]. Knoop's technique is the best-in-class method for performing code motion since it involves unidirectional data flow analysis with respect to the program flow[6] where reducible programs (see Chapter 4) can be dealt with in $O(nlog(n))$ bit-vector steps (see [2]) where n is the number of statements in the program in contrast to $O(n^2)$ complexity for previously known approaches.

Hailperin in [51] extended Knoop's approach to incorporate cost into the code motion process. However, the cost metric he uses is based on individual operations (i.e.

[5]Traversing the memory hierarchy to get a value is typically slower than accessing a register
[6]i.e. forward or reverse

*, +, ...) and does not account for the frequency of execution of the program portions. The tactic's goal is to position the instructions in (safe) positions in the program where the context possibly permits simplification of the particular operation under consideration through constant folding, or operation strength reduction for example. Castelluccia et. al. [27] used, in my opinion, a more adequate runtime cost to optimize protocols but the techniques they applied where based mainly on node re-ordering at the CDFG for synthesis level and did not incorporate examination of data operations, I use similar cost guidance in my operation motion technique as discussed in the next Section.

5.3.2 This Work's Contribution and Overview

My work incorporates cost into operation motion. The cost I introduce and focus on mainly in this Chapter for illustration purposes is obtained from a task level static analysis used to identify the most frequently visited segments of the task behavioral description. Alternatively other target implementation costs can be used such as the task average or worst-case execution time; the latter costs will be discussed in Section 5.3.5. My code motion step *itself*, in addition to being cost-driven, is also faster than other approaches; it has a time complexity of $O(n)$ in the number of statements in the description. To achieve this simplicity, however, I give up slightly on size where for a brief period after this step the space needed is $O(n^2)$. I also rely on code motion being part of a general optimization framework; in particular I need to use reachable variable definitions, and reached uses of variables, an analysis that has a time complexity of $O(n^2)$. So, the framework's complexity ($O(n^2)$) is what dominates the overall complexity. Of course, for this increase in complexity we can get much better optimization results than [67] since code motion is applied to *all*

candidate operations simultaneously and is tempered by other data flow and control analysis and optimization steps that improve the final result even further. As will be described shortly, the approach is therefore much simpler (conceptually and in practice) than other approaches as it tackles operation motion *broadly*, and still performs the job adequately as part of the comprehensive multi-step data flow and control optimization approach described in Chapter 4.

My approach, dubbed *Relaxed Operation Motion (ROM)* because it accomplishes its objective *indirectly*, is also specialized to the embedded system domain where we are constrained by I/O schedule preservation, but where we can benefit from the user's insight by soliciting assistance in the cost estimation mechanism since embedded systems have a predictable (or pre-conceived typical) behavior. I have implemented my approach in the function/architecture co-design framework that uses the Polis synthesis engines for output generation.

5.3.3 Illustrative Example

In order to illustrate my approach I have adapted Knoop's "motivating example" from [67] by making it reactive through the addition of inputs and outputs, and a loop from the final node S10 back to S1 so that the system would be running continuously. This is shown in Figure 5.3. Variables a, b, c are declared internal, and initialized in S1 to a sampled input value, x, y, and z are declared as outputs and therefore *fixed* to their respective states since we always preserve I/O traces before and after the optimization. As in [67], our goal is to eliminate the redundant needless runtime re-evaluation of the $a + b$ operation. I will focus my discussion on the nodes S8 and S9 since as we will see shortly

they are *costly* compare to neighboring nodes, and I will try to *relocate* the aforementioned addition operation to other less costly nodes.

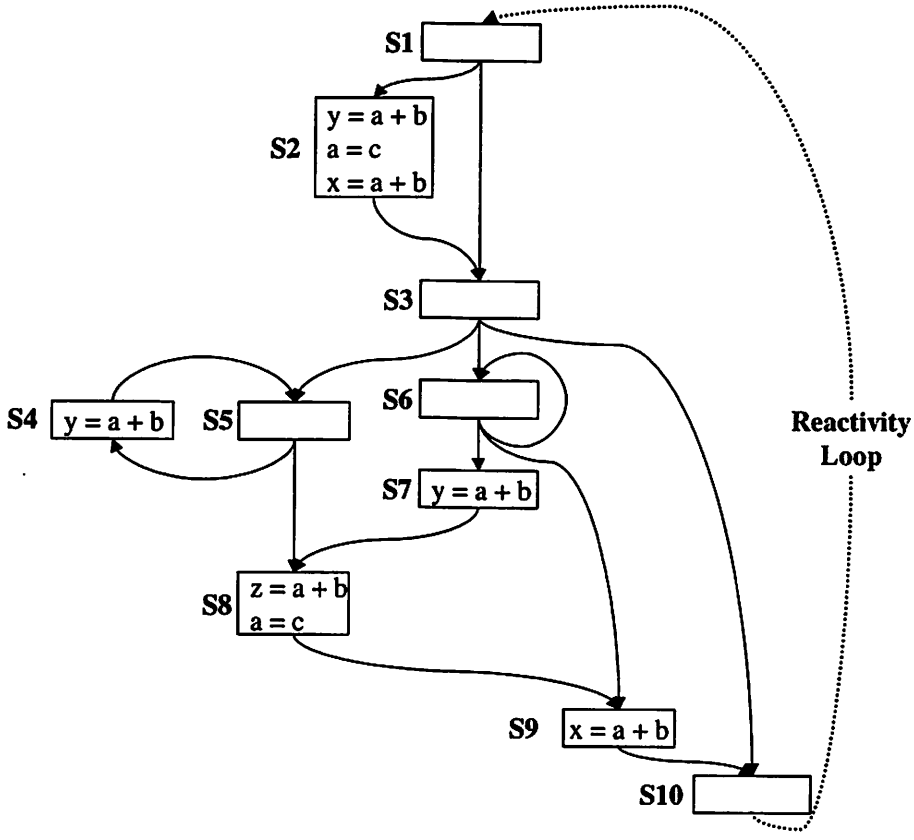


Figure 5.3: Illustrative Example (from [67])

5.3.4 Cost-guided Relaxed Operation Motion (ROM)

The operation motion approach is incorporated as part of the comprehensive optimization approach (ROM's overall complexity is $O(n^2)$ in the AFFG nodes) and consists of 4 main steps performed in order.

Algorithm 5.3.1 (Relaxed Operation Motion)

Relaxed Operation Motion(G)

begin

(a) **Data Flow and Control Optimization***: is a sequence of steps that optimize the FFG (AFFG if cost-guided) representation as described in Chapter 4.*

(b) **Reverse Sweep***: is the optimization step that I am mainly addressing in this Section where code is relocated from one or more (A)FFG nodes to others. This step can either follow the* as early as possible approach, *or be* cost-guided. *It consists of "indirect" operation motion through:*

i. Dead Operation Addition*: where operations are added to all (or to selected based on cost) FFG (AFFG) nodes.*

ii. Normalization and Available Operation Elimination*: This optimization step effectively "moves"*[7] *the* operation motion candidates *from the targeted FFG (or AFFG) nodes to other less costly nodes as a result of the preceding step.*

iii. Dead Operation Elimination*: removes the useless additions performed in step (b)i.*

(c) **Forward Sweep***: is optional. It tries to* minimize the lifetime *of temporaries (see Section 5.5.3) by pushing them as close as possible to their use.*

(d) **Final Optimization Pass***: performs the final clean up.*

end

The $O(n^2)$ size explosion could happen in step *(b)i* since for every (A)FFG node we can potentially add all the statements in the other nodes. Of course the *directed* or cost-guided approach does not incur as large a penalty. Also, the complexity of our approach compared to the best in class (similar for small n but difference is felt of course for large n) should be weighed against the fact that we need to perform the use/definition computation (see Chapter 4) anyway for other optimizations, and operation motion is not typically

[7] Evidently, on its own it just *eliminates* them from targeted nodes.

performed in isolation of those operations if the goal is overall improvement in behavior (information processing). If that fact is taken into account then indeed my approach is *linear* (i.e. $O(n)$) since in that case ROM would effectively be dominated by step *(b)i*. This, of course, makes sense since we trade-off size (i.e. we need memory) for speed. It can therefore be seen that my approach comes "naturally" in an optimization framework, which permits me to use relatively simple techniques to accomplish the task. The result after the first optimization pass and the dead addition step is shown in Figure 5.4. Figure 5.5 shows the result after available elimination, and the final pass. **Forward sweep is not applied here.**

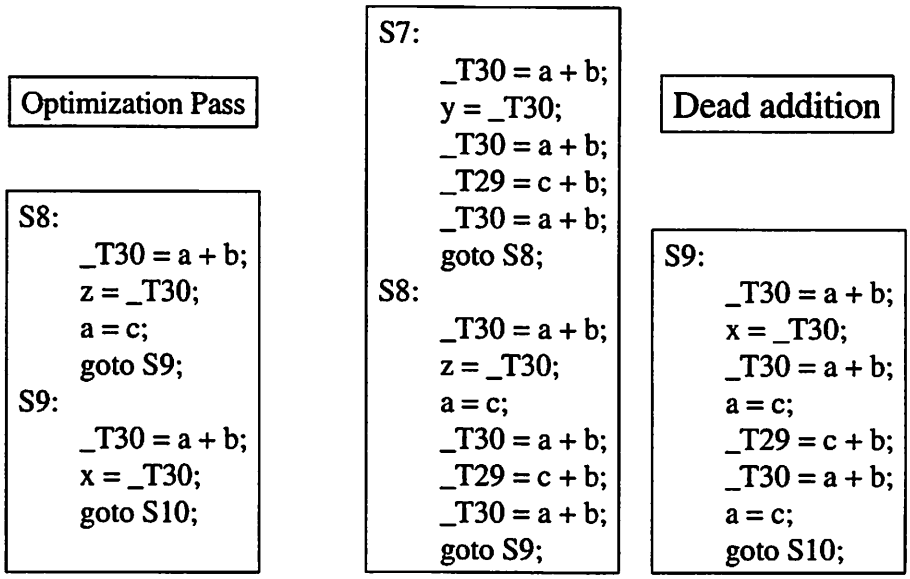


Figure 5.4: Result After Dead Addition

Note that in the final result the redundant computations in S8 and S9 are indeed relocated up to S1 (earliest, again forward sweep not applied). The final result is 60% better

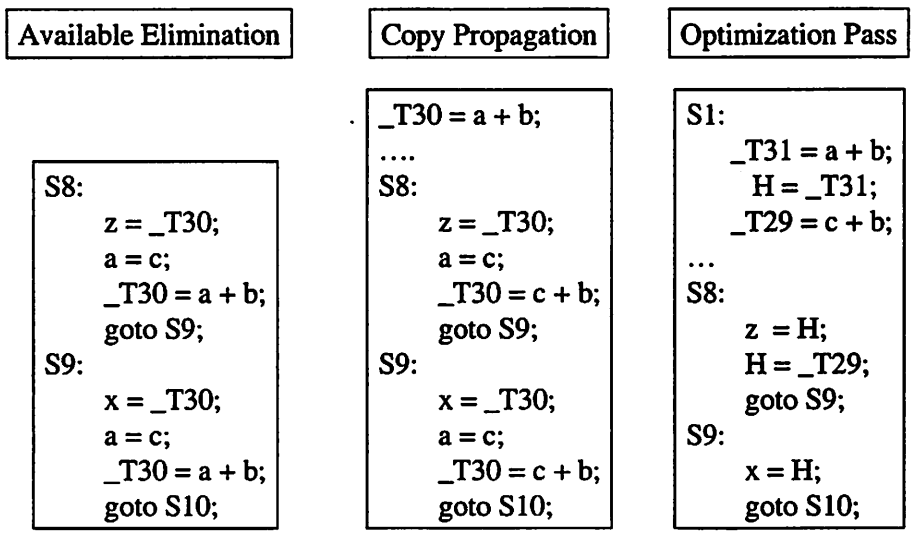


Figure 5.5: Result After Available Elimination

in terms of operation count[8] than Knoop's *Lazy Code Motion* of [67] if we count the remaining addition operations after ROM is performed. The improvement comes about because ROM is part of a comprehensive optimization framework that consists of ***normalization***, ***copy propagation***, and ***dead elimination*** to name a few (crucially) useful steps (see Chapter 4). If we examine the final output shown in Figure 5.5 we can see that normalization for example identified that we only need have two *operations* in the FFG behavior: `a + b`, and `b + c` only. All other computations can be referred back to these through the use of *copies* and *assignments* (`H` in the final result of Figure 5.5). Knoop's technique is developed in isolation and is therefore slightly faster (with less memory requirements) if compared on the face of it; but if we are using ROM in the context of an optimization flow all the LCM's advantages disappear. ROM is the clear winning strategy; if Knoop's technique is augmented with other analyses and optimizations in order to improve its result then ROM

[8]Computations are replaced by assignments from the registers.

becomes, in that case, the *simpler and faster* of the two approaches (with a small memory trade-off).

5.3.5 Cost Estimation

We can have several cost metrics to guide ROM, in this Chapter I will use the frequency of execution of behavior portions as such a metric, I will later present another metric based on the worst-case execution (see Chapter 9 for results).

The expected number of times a certain part of a program is executed can be determined once each branch probability in the program is known [70]. It can be shown that the number of times each basic block (and by analogy each (A)FFG node) is executed can be calculated by solving a system of n linear equations, where n is the number of basic blocks ([2]), if the probabilities of control passing from one block to the next is given [120]. This of course is a generalization of branch prediction, which only determines the most probable outcome of a branch [27]. The probabilities of all the **TEST** outcomes in the task are requested from the designer in an interactive fashion before the estimation and subsequent optimization takes place. A *Markov process* can be used to model and then compute statically the probabilistic control flow execution as described in [128] where it is also shown that this method is quite close to extensive profiling (assumed to be the "exact" metric). Of course, the advantage is that this estimation approach involves much less effort than profiling and is quite applicable in the embedded system domain where tasks are expected to perform a specific functionality and typically the designer has a good idea of where most of the execution takes place.

Bayesian Belief Networks

In order to identify the most frequently visited portions of the task's AFFG, I use an approach similar to Markov processes but based on *Bayesian Belief Networks (BBNs)* via the MSBN inference engine (Bayesian Belief Net construction and evaluation tool) that was available to me[9] from Microsoft Research [84]. The MSBN tool uses a version of the proposed *Bayes Net Interchange Format* for representing belief networks.

Definition 5.3.1 *A* Bayesian Belief Network (BBN) *is a Directed Acyclic Graph (DAG) consisting of nodes, each with an associated Node Probability Table (NPT). Nodes represent discrete random variables, while arcs connecting these nodes represent causal influences (adapted from [93]).*

The key feature of BBNs is that they enable us to model and reason about uncertainty. The advantage of BBNs is that they allow the capture of both subjective probabilities based on user knowledge or experience, and probabilities based on statistical data in a unified framework [19]. BBNs also permit the automatic construction of a network from a database of (possibly growing) experimental evidence, although the latter kind of tools should still be considered as promising on-going research [19]. Figure 5.6 shows an example of a BBN used to determine which is a more likely location for my graduation party: indoors or outdoors.

The example BBN consists of 3 decision nodes:

- `Attendance` which has values of: $(large, small)$,
- `Weather` which has values of: $(sunny, rainy, cloudy)$, and

[9]Research License

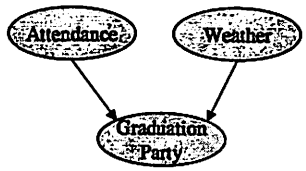


Figure 5.6: Bayesian Belief Network for Graduation Party Location

Graduation Party	Indoors	Outdoors
$(large, sunny)$	0	1
$(large, rainy)$	0.6	0.4
$(large, cloudy)$	0.3	0.7
$(small, sunny)$	0.2	0.8
$(small, rainy)$	0	1
$(small, cloudy)$	0.4	0.6

Table 5.1: `Graduation Party` Node Probability Table (NPT)

- `Graduation Party` which has values of: $(indoors, outdoors)$.

The NPT for the `Graduation Party` node is shown in Table 5.1. This is the input model that I have decided on given my *subjective knowledge* that an indoor location is costly, rain may force me to have the party indoors, and that an outdoor party is usually inexpensive and can hold a large number of people. My decision is also influenced by other factors like: people may not wish to stay out in the sun, and may prefer a small gathering indoors (in the case of small attendance).

With this model, we can now use the BBN evaluation procedure to determine the likelihood that the party will be outdoors. If the weather and attendance conditions have a uniform distribution, and without any additional observations about the weather and/or attendance the probability of having the party outdoors evaluates to **0.75**. If however we have the additional observation (from the weather forecast for example) that the weather will

be *cloudy*, and that attendance (from the RSVPs for example) is *small* then the probability if an outdoors party drops to **0.6**. Of course, this is just a very simple example where we have a single NPT, the evaluation procedure becomes more involved and more useful once we have multiple levels of decision in the BBN.

Visit Frequency Estimation Using BBNs

In order to compute the probabilities of execution in the task AFFG, I represented the state transition relation consisting of current state, next state, and conditionals as shown in the screen-shot of Figure 5.7. For the lack of any specific knowledge, I initially assign equal probabilities to all the reachable states and then iterate the probability computation until a fixpoint is reached. This is a **heuristic** intended to identify the effect of the behavior structure on the state visit probabilities. The heuristic can *always* be applied especially in cases when the user does not wish to explicitly enter specific node probabilities, possibly because of the lack of such knowledge, but would still like to perform optimizations in *potentially* heavily executed portions of the behavior, such as *loops* where *loop-invariant operation motion* can be performed for example, or *re-convergence segments* i.e. segments of the behavior that are visited from various execution contexts where *operation "replication"* on the different predecessor segments can result in runtime speedup, using guided ROM.

The reader should note that we need to perform the iteration on the probability computation in our **indirect relaxed iterative** solution method since the Bayesian network is a static rather than a dynamic modeling tool (i.e. it has no concept of "future" or *eventual* network state, *feedback cycles* cannot be constructed). Alternatively a **direct** solu-

tion method can be used, with the aid of a Markov process model[10] that represents a *linear system of equations* which can be solved using a matrix solver[11], to find the solution. We chose to use here the indirect method based on Bayesian networks because the approach does not require the direct solution of a large system of linear equations thus offering a computational advantage, since all we really need is a *rough relative visit frequency* of the states. This approach is also appealing because of the suitable level of interactivity with the user since the method operates directly on the state transition network, and allows dynamic update of observation evidence.

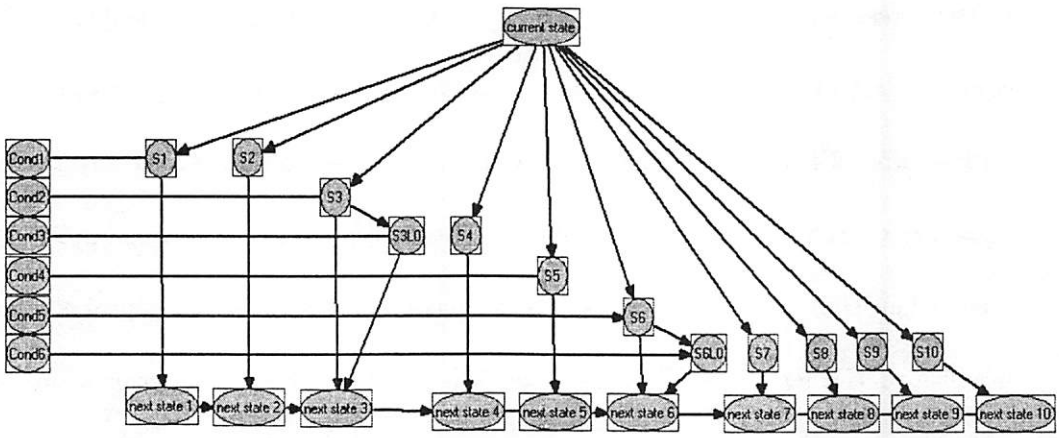


Figure 5.7: Belief Network for the Knoop Example *(Courtesy Microsoft Research)*

The frequency of execution results for the reactive version of Knoop (i.e. loop from `S10` back to `S1` added) with uniform probability of the conditionals (i.e. $P(True) = P(False) = 0.5$) are shown in Table 5.2. The costs of `S8`, `S9`, (operation motion candidates) and `S7` (target of operation motion) are highlighted thus outlining the potential for cost-guided ROM if the state visit probability metric is used, assuming that indeed these hy-

[10]There is a minor syntactic difference here where probabilities in a Markov process are on the arcs between states instead of within as in the Bayesian Net.

[11]For sparse matrices preferably for examples with a large number of states and conditionals

State	Probability
S10	0.15
S1	0.15
S3	0.15
S5	0.15
S9	**0.1**
S8	**0.09**
S2	0.07
S4	0.07
S6	0.046
S7	**0.024**

Table 5.2: Frequency of Execution Distribution for Uniform Conditions

pothetical probabilities are valid, or that, heuristically speaking, this gathered information is a "hint" of potential improvements based on the behavior (AFFG) structure knowledge.

Other Cost Metrics

I have focused here on a *state visit cost* approach at the **task-level** for guiding the operation motion. Other metrics to guide specializations of this approach could be addressing the *average response time* by focusing on the *common path* in the CDFG for synthesis (see Chapter 7) where typically the 80/20 rule applies (80% of the execution is in 20% of the behavior) [27]. Another cost guidance could be the *worst-case* response metric which can be identified at the CDFG level (longest structural path for example), or through analysis at the task level, where *abstract interpretation* ([34]) is applied, as in the work of Balarin [4].

Cost-guided Relaxed Operation Motion (ROM) Flow

The (constrained optimization) ROM flow is shown in Figure 5.8. On the left, we see the "interactive" profiling component where the user's guidance is requested, and an inference engine (MSBN for example, or a solution technique based on Markov analysis) determines the cost. These costs are then used (right side of Figure) in the function/architecture optimization framework to *change* the functional description at the macro-architectural organization level (since this is a transformation that operates *across* states) by leveraging the ROM technique I have developed.

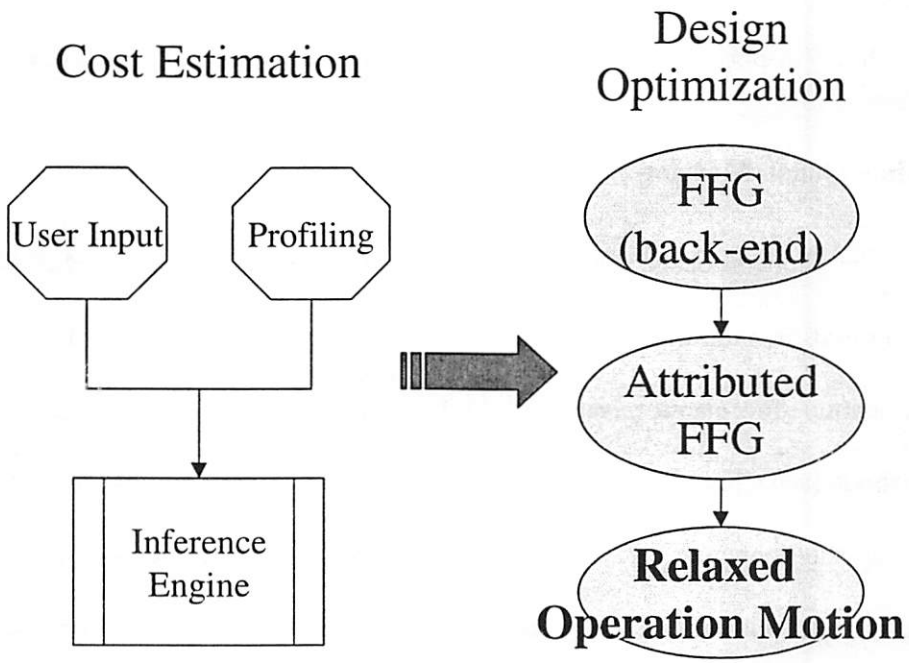


Figure 5.8: Optimization with ROM

5.4 Other Constraint-Driven Optimization Techniques

There are several works on optimization techniques given application or architectural constraints such as the work of [27] in protocol optimization. The techniques (like node restructuring to optimize runtime or size) are *control-oriented* and typically require low-level knowledge about the performance of the function once decomposed and mapped onto the target architecture itself (at the very least the *macro*-architecture and quite often the *micro*-architecture as well). The Polis synthesis back-end provides for such optimizations, and I will elaborate on this further in Section 7.2.1. I should note also that, in the spirit of *platforms* shown in Figure 3.2, an *abstraction* of these constraints can be quite useful in making it possible to apply such techniques at the (A)FFG task representation level itself where there can possibly be a *high value* outcome of such trade-offs much like the ROM approach has been able to do; I get into these aspects at the *macro*-architectural level in the next Section, and postpone the *micro*-architectural level aspects for later.

5.5 Optimizing the Function to be Mapped onto the Macro-Architecture

Let us now consider additional function optimizations that can be performed given some architectural information *before mapping* the function onto the architecture.

5.5.1 Copy Propagation

In Chapter 4, I have mentioned how I had to *specialize* the global optimization techniques I have discussed earlier to be *safe* and preserve the I/O semantics of the EFSM.

However, we can relax this somewhat at this stage since are now given more information about the EFSM schedule. This schedule has been overlayed on top of the FFG resulting in the AFFG. So the question is: What more optimizations can we do given this additional knowledge ? To answer this question let us first examine the restrictions we had imposed on inputs and outputs in the FFG task using the following simple example.

Example 5.5.1

input in;

...

S1:

t_1 = in;

a = t_1 + t_2;

b = t_1 + t_3;

The operation t_1 = in remains as is, and in *cannot* be propagated down, thus potentially saving the temporary t_1, because in is an input and cannot be "touched" if we are to preserve the I/O semantics that is for all we know changing the example to be as follows,

Example 5.5.2

input

...

S1:

t_1 = in; /* Will be removed by dead elimination */

a = in + t_2;

```
b = in + t3;
```

might in fact change the intended behavior since we now have *two samplings* of the input in instead of one sampling i.e. we are sampling it at two *different* times and have no guarantee that in does not change between these 2 samplings. We can now however **relax** these strict I/O assumptions since we are already aware of the EFSM execution schedule. To take this one step further we are also aware of the *macro*-architecture model we are targeting[12]: the *reactive CFSM*. So in the case of Example 5.5.1 in fact we *can* transform it into Example 5.5.2 since the CFSM samples its inputs at the start of the state (we are using the AFFG associated meaning to the labels, whereas we had not done so previously in the FFG), and holds this value until the next state (see Chapter 3). So, as long as we remain true to the reactive CFSM semantics we can perform copy propagation along *each state's computation path* and in fact come up with the *AFFG Optimization Algorithm* that *preserves input and output semantics at the state boundary.*

Algorithm 5.5.1 (AFFG Optimization Algorithm)

AFFG Optimization Algorithm(G)

begin

while changes to the AFFG **do**

...

Relaxed Copy Propagation

...

end while

end

[12]For Polis MOC, other engines would require adequate specializations as well

The *Relaxed Copy Propagation* step is performed as follows:

Algorithm 5.5.2 (Relaxed Copy Propagation)

Relaxed Copy Propagation(G)

begin

foreach state $s \in G$ **do**

Copy $= \emptyset$;

map_propagate_copy(G, s, Copy);

end foreach

end

Algorithm 5.5.3 (map_propagate_copy)

map_propagate_copy(G, n, Copy)

begin

copy_propagate(n, Copy); /* *perform propagation* */

Copy.*append*(find_copies(n)); /* *Scan all the operations in n* */

foreach $m \in Succ(n)$ **do** /* *DFS for speedup* */

if $m \in States(G)$ **then**

return; /* *Stop processing along this computation chain* */

else

map_propagate_copy(G, m, Copy); /* *recur* */

end if

end foreach

end

The main distinction between algorithms on the AFFG and those I presented earlier for the FFG is that here we have a concept of a *state*: We know where it *starts*, where it *ends*, and all the *nodes* that *belong* to this state along its various *transitions*; this is stated formally in Definition 5.5.1 that follows. The reader should note that the *ROM algorithm* described earlier in the Chapter does *not* need to be "relaxed" since it was already operating at the *state boundary* level[13], however, it does benefit as part of the overall optimization scheme that incorporates the improved copy propagation (normalization and dead elimination) since the number of operations are reduced (speed aspect) and new candidate opportunities are exposed (quality aspect).

5.5.2 Scheduling

The major differentiation between my work, and typical high level synthesis approaches (using HDLs for example) is the fact that I do not try to do all of: decomposition of the user's design description, scheduling, and resource allocation *automatically* where the user can only provide *guidance constraints*. I have separated this process into a front-end that captures the *designer intent* by building an EFSM description of the design where the *schedule* is solicited from the user either *directly* as in the case of the Reactive VHDL specification, or *indirectly* as in the case of the exponentially more expressive Esterel specification mechanism (see Chapter 3). A back-end then optimizes this function and co-designs it with architectural and application constraints.

[13]Of course the ROM algorithm is quite general, and it could be applied at a finer granularity level if we associated cost to the AFFG individual nodes as opposed to states which are collections of nodes. I think the reader should be convinced (by now!) of the value of techniques that can be refined (tuned) no matter what the functional (architectural) abstraction level is.

State Scheduling

Once the AFFG is built, we *already have a valid state execution schedule* of the task's behavior execution. In particular this is the **one schedule that the user specified**[14]. The state scheduling step is available *by construction* solicited in some sense from the user, and made concrete by the front-end; so we need not perform a prohibitively expensive analysis to find a *valid* schedule that may not be what the user had in mind[15].

Definition 5.5.1 State in the AFFG *A* state *s in the* AFFG *G consists of a collection of nodes $n \in G$. The set of states is denoted by S_G. Node $N_s \in G$ labeled with state s is the* state start node. *A path p from N_s to a node N_t where $t \in S_G$ is said to be a* transition path *from s to t.*

If the AFFG G is in Tree form *then states* partition *the set of nodes V of G:*

- *Each node in G must belong to state i.e. $\forall N \in V$ of G, $\exists s, s \in S_G$ such that $N \in s$, and*
- *a node in G cannot belong to two states i.e. for any two states $r, s \in S_G$, and a node $N \in V$ of G, $N \in r \Rightarrow N \notin s$.*

If the AFFG G is in Shared DAG form *then states divide V of G into* intersecting sets of nodes *such that each set (collection of nodes) corresponds to a* single *state $s \in S_G$. In this case:*

- DAG nodes *must belong to* at least two *sets of nodes, while*

[14] Assuming the front-end is expressive enough to capture user intent, as well as semantically correct to build a proper EFSM for the control design. This is indeed the case for Esterel, while in the Reactive VHDL front-end the user is asked to explicitly build the states.

[15] Typically in high level synthesis the user provides resource allocation constraints to guide this process which makes the analysis intractable; constrained scheduling is NP-hard.

- *the rest of the nodes i.e. "Tree" and "state start" nodes must belong to a* single *node set.*

A state in the AFFG is therefore represented by a start label (a node in the AFFG), and a collection of nodes that represent the computations performed along any *state transition* to the next state as shown in Figure 5.9. For example state S_0 in the figure is associated with the *state start node* N_0; S_0 consists of a collection of AFFG nodes, has a set of *next states* (S_1 and S_2), and several *transition paths* that compose the state transition relation. Note that the Tree form of the AFFG is used in the Figure.

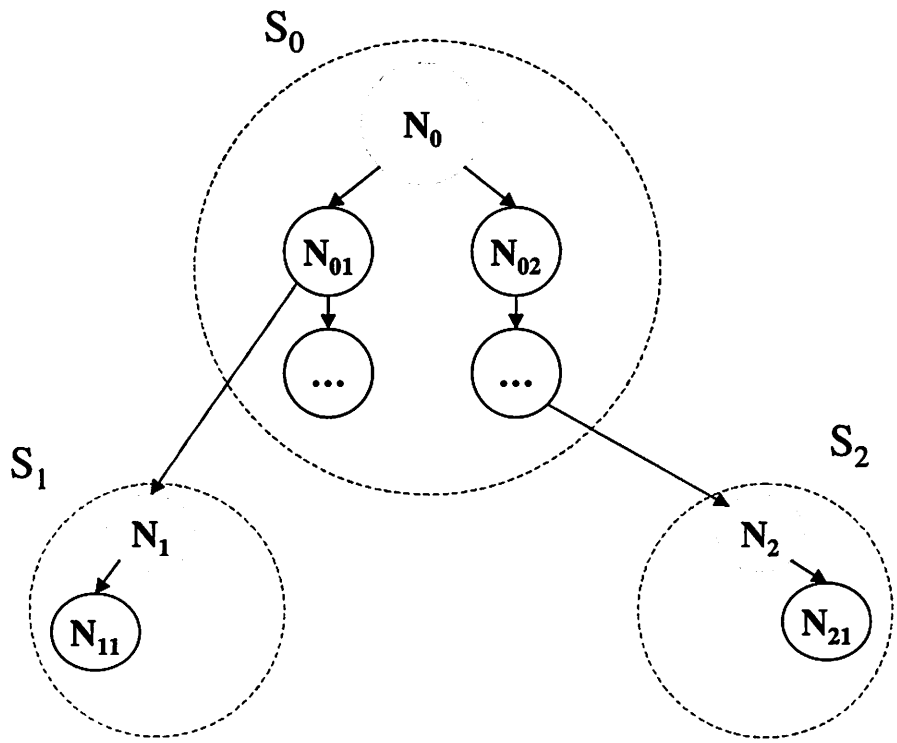


Figure 5.9: State in the AFFG

Operation *Macro*-Scheduling

Resource constraints limit the number of and binding of operations to resources in the final implementation [38]. However, I will not at this high level deal with *micro*-scheduling issues which make the problem quite intricate. I leave the resource allocation and scheduling to the lower level phases (performed by software or hardware compilers), and focus here on another useful process that I call *operation macro-scheduling*. The goal in this step is to make the definition of any computation *within a state* a *unique* indicator of the operation. If the FFG is in *Tree* form, this process will not change anything since its job would be subsumed by *Normalization* (see Chapter 4). In the case of the *Shared DAG* form of the FFG, normalization will handle most but not all such occurrences since an operation may depend on which path it is reached from (i.e. the execution context).

Algorithm 5.5.4 (Operation Macro-Scheduling)

Operation Macro-Scheduling(G)

begin

foreach state $s \in G$ **do**

Seen = $\emptyset$;

count = 0;

check_operations(G, s, Seen, count);

end foreach

end

Algorithm 5.5.5 (check_operations)

check_operations(G, n, Seen, count)

begin

```
    lseen = Seen; /* seen at this level so far */
    Changed = ∅;
    /* make operation identifier unique if seen, and append to seen, increment count, fix uses in
       node */
    Changed = make_unique(n, lseen, count);
    update_node_uses(n, Changed);
    foreach m ∈ DFS(Succ(n)) do /* DFS traversal */
        if m ∈ States(G) then
            return; /* Stop processing along this computation chain */
        else
            check_operations(G, m, lseen, count); /* recur */
        end if
    end foreach
end
```

In words, Algorithm 5.5.4 shown above does the following: Starting from a state s in the AFFG, we traverse the *successor nodes in the AFFG* (examining their associated operations as we go) in depth first order denoted by $DFS(s)$[16] until we hit the *next state boundary* if we encounter a *normalized* computation more that once *along a state transition path* then we make the definition identifier (destination of operation) of the second occurrence unique. The end result is that we have a *guarantee* that an operation definition is a exclusive indicator of an operation that must be **scheduled** *in the following micro-architecture stage.*

[16]DFS ordering of the nodes in the AFFG *starting* from node s

5.5.3 Allocation

I leave "operation" micro-scheduling and sharing to the later micro-architecture optimization stage (see Section 6.5.1), and focus here on determining whether computations need to have a register for *correct functionality* once a *state transition relation (TREL)* is built[17]. Algorithm 5.5.6 below shows the register allocation process. The sub-sections to follow will detail the steps it is composed of.

Algorithm 5.5.6 (Register Allocation Algorithm)

Register Allocation Algorithm(G)

begin

foreach node $n \in G$ **do**

foreach state $q \in G$ **do**

Seen = Identify Operations /* *Candidates* */

Build Frontier /* *Next State Boundary* */

Register Allocation and Creation

end foreach

end foreach

end

Identify Operations and Build Frontier

In this step we scan all the computations in a state q in the AFFG, and collect the destination *definitions* as *candidates* that need to be registered. During this traversal process as we are exploring the state we stop at the *next state boundary*. The states composing this

[17] We are not going to build the TREL yet, this step must be performed *before* building.

boundary are collected in a *Frontier*(q) as shown in Figure 5.10 (Figure used in Register Allocation Algorithm as well). This *frontier* is of crucial importance since it reflects the *schedule* we have associated with the AFFG, and is what will shape our register allocation requirements.

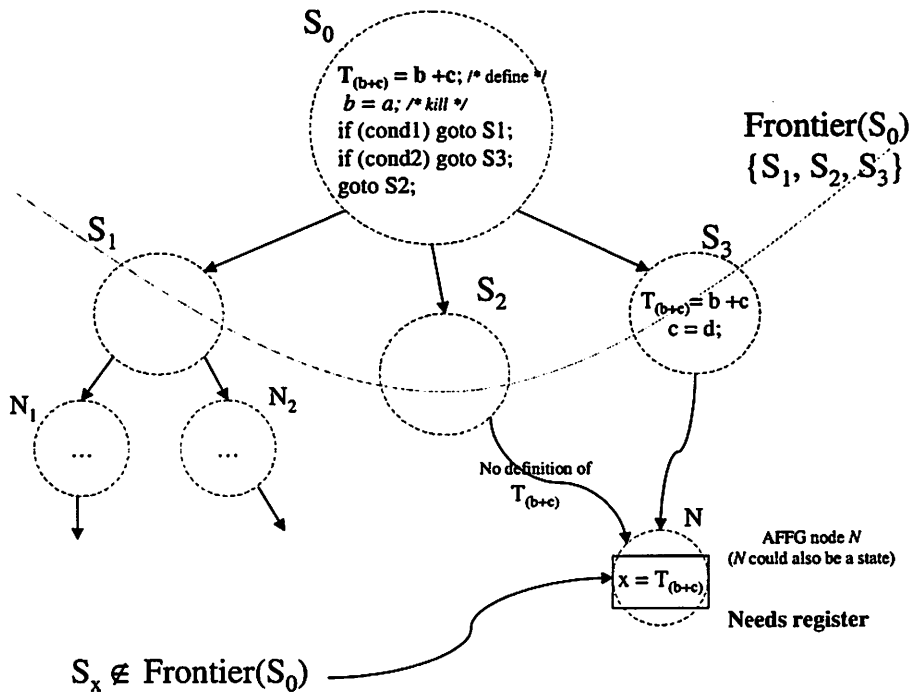


Figure 5.10: State Frontier and Register Allocation

Register Allocation

In order to perform register allocation, I have devised a new meet DFA problem *on the AFFG* called *Available Definitions.* This algorithm assumes that all the previous steps are performed (namely normalization on the FFG, and operation macro-scheduling on the AFFG), and uses the information that was gathered in those steps, and is tuned for maximum speedup improvement as will be described shortly. The input to the algorithm

is the AFFG G, and the $Frontier(q)$ for the state q we are processing; the frontier is of course the one built in the earlier step.

Algorithm 5.5.7 (Build Available Computations)

Build Available Computations(G, $Frontier(q)$)

begin

foreach node $n \in G$ **do**

$Reach(n)$ = *All definitions in AFFG; /* Initialize */*

end foreach

foreach state $s \in Frontier(q)$ **do**

$Reach(s) = \emptyset$; */* Initialize */*

while *there are changes* **do**

foreach node $n \in DFS(s)$ **do**

$Pass(n) = (Reach(n) - \emptyset) \bigcup Gen(n)$; */* Transfer Equation */*

$Reach(n) = \bigwedge_{P \in Pred(n)} Pass(P)$; */* Confluence Equation */*

end foreach

end while

foreach node $n \in DFS(s)$ **do**

foreach definition $d \in n$

If ($d \in$ Seen *and not*($d \in Reach(n)$) *and not*($d \in Gen_l(n)$[18])) **then**

/ Allocate register */*

create_register(d) unless d is a register;

end if

[18] Actually we check if it is defined *before* the use in question only, but I took the liberty of simplifying the pseudo-code; I insert a subscript l to distinguish from $Gen(n)$.

end foreach

end foreach

end foreach

end

The **Transfer Equation** $Pass(n) = (Reach(n) - Kill(n)) \bigcup Gen(n)$ reflects the fact that we are looking to see if there is a definition in this frontier state or not; note that I simplify the computation in the Algorithm by using $Kill(n) = \emptyset$ because in this case $Kill(n) = Gen(n)$ since I am looking for *definitions* (again operations have unique definitions in this stage), and the result of the two equations is the same. If there is no definition in this frontier state[19] then we need to *register* the operation from the previous state. If the definition is computed in the frontier state then we can just use the result immediately (i.e. the mapper (of the function onto the architecture) will generate a *combinational circuit in hardware or instruction in software* instead of creating a register and then a copy instruction). This idea is also shown in Figure 5.10. Furthermore, the reader may realize that in some cases there might *not be a* **real** *need* for adding a register (i.e. the function is not necessarily compromised if the register is not added); it could be that even if along some path in the frontier there is no re-definition (and this is what led us to decide on recommending that the earlier definition be registered), the earlier definition is in reality never "killed" and therefore always *available*, yet the Algorithm as shown does create a register (of course this is always *safe*). Algorithm 5.5.7 can be modified to include such information (with additional analysis in building the $Kill$ of the frontier to determine whether operands of the operation in question are modified thus forcing the need to register

[19] In other words, in *some* frontier state

or not), but the resulting algorithm will not be maximizing the speedup[20] but it would save needless registers. Again we see how optimization and constraints come into play; for our purposes here we will always use of Algorithm 5.5.7 for **maximum speedup**[21].

Register Creation

This step uses the result of the available computations, and then checks if a computation is defined in a node before it is used; this corresponds to routine `create_register` in Algorithm 5.5.7. If the test fails then a register is *recommended* by adding a *hint* (i.e. annotating the AFFG) to the architecture mapping phase as shown in this example. Assuming we initially had the following situation:

Example 5.5.3

output *outp;*

int *a, b, c;*

...

S1: /* state we are processing */

$T_{(b+c)}$ = b + c; /* operation definition: $T_{(b+c)}$ */

b = a; /* operation $T_{(b+c)}$ is *Killed* */

...

S2: /* state S2 $\in$ *Frontier*(*S*1) */

outp = $T_{(b+c)}$;

[20] We will always assume a register is faster than inlining (i.e. performing the computation again) here, this is architectural information we are assumed given, if that is not the case a differently tuned technique must be used.

[21] Current default in the toolset

For proper functionality we must now create a register, this is accomplished by associating a "hint" with this operation in the AFFG indicating that this should be registered, textually it can be expressed as follows (in textual form; in reality registers are attributes on the respective variables):

Example 5.5.4

...

S1:

$T_{(b+c)}$ = b + c;

$R_{T_{(b+c)}}$ = $T_{(b+c)}$;

b = a;

...

S2: /* state S2 $\in$ *Frontier*(*S*1) */

outp = $R_{T_{(b+c)}}$;

5.6 Function Architecture Co-design in the *Micro*-Architecture

System level design has a fractal nature; the problem has the same dimensions no matter at which abstraction level we examine it [100]. Function/architecture co-design can be performed at the *micro*-architecture level [89] in much the same manner as we described earlier at the *macro*-architectural level. In this Section, I introduce techniques geared towards this trade-off process between the intended functional behavior and the architectural primitives that implement it. I will revisit this issue in Section 6.5.2 where we try to find an *optimal mapping* of the function onto the components making up the

architecture by leveraging more on the application domain, and architectural specifics.

5.6.1 Operator Strength Reduction

An effective optimization in both runtime and size of hardware and software is *strength reduction* of operators in the AFFG quadruples. Given architectural estimate information about an operation's area, power, and performance cost we can *optimize the function* so that it performs better while preserving the functionality, by selecting a *cheap* operation or a *cheap set* of operations. if we consider the following example where we have a *sequence* of operations in the AFFG, *annotated* with a cost:

Example 5.6.1

...

output *x;*

t_1 = 3 * b; /* MUL: 10 units */

t_2 = t_1 + a;/* ADD: 2 units */

x = t_2; /* ASSIGN: 1 unit */

then *reducing* the multiplication operator in `t1 = 3*b`, to the equivalent `t1 = b + b + b` would be quite beneficial, the example would change to:

Example 5.6.2

...

$expr_1$ = b + b;

t_1 = $expr_1$ + b;

t_2 = t1 + a;

x = t_2;

Another example of strength reduction is changing operations like x = x - 1 to DECREMENT(x) where we give the architectural optimization a "hint" to optimize resources when performing said operation on x. The effect of such optimizations will be felt if there are conditions (and/or loops) in the execution such that we visit these operations many times over the lifetime of the task (see Section 5.3.5). So given some high level estimates of the architectural primitives supported by the architecture we can perform a number of *peephole optimizations* to try to get to the best *matching* of the operation or a *neighborhood* of operations to the given library in an optimal fashion. Of course we can only have some heuristics at this level where we apply a sequence (i.e. ordered set) of transformations to *improve* the result. While it may seem that such optimizations can have some limited improvement only, the reader should note that any educated algorithm that improves the AFFG in a local fashion, for example the algorithm might scan all the operations in the AFFG and replace constant multiplications by shifts or additions, is useful as a **hill-climbing** feature in the *AFFG Optimization Algorithm* shown earlier as Algorithm 5.5.1, where we can, once convergence is reached, apply this step to "shake things up" if you will, and perform another optimization loop to convergence. With this technique we can leverage simple techniques to potentially uncover larger reduction in redundant information[22]. To be able to do this step we have to rely on architectural estimation (or profiling and/or inference based on user knowledge as we did in the ROM algorithm).

Algorithm 5.6.1 (*Hill-Climbing* AFFG Optimization Algorithm)

[22] Or *inefficient* information representation

***Hill-Climbing* AFFG Optimization Algorithm(G)**

begin

shake_it_up = True;

AFFG_opt:

while changes to the AFFG **do**

Perform AFFG Optimization Algorithm *Steps*

end while

if (shake_it_up) **then**

shake_it_up = False;

Operator Strength Reduction

goto *AFFG_opt:*

end if

end

5.6.2 Instruction Selection

The discussion in the previous subsection was an example of how architectural constraints feed back information to the function so we can massage it to better *match* the architecture. This can also go the other way where at the functional level we can identify useful sets of operations to have in the architecture, and recommend that these instructions in the case of software, or resources in the case of hardware, be made available. This is called *Instruction Selection* where the function feeds information in the form of candidate primitives to the architecture. While this technique is not beneficial from a *hill climbing* perspective it will become useful in *optimal mapping* of the function onto a

suitable architecture as described in Section 6.5.2. Both directions are displayed visually in Figure 5.11 which shows how the AFFG is the *transfer medium* through which this information transfer and optimization happens.

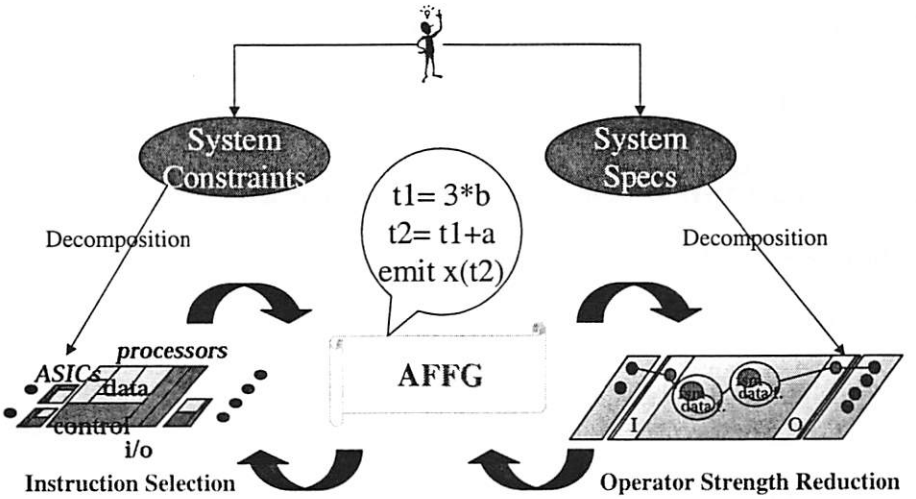


Figure 5.11: Operator Strength Reduction, and Instruction Selection

5.7 Future Directions

Many opportunities for future improvement and expansion of the function/architecture optimization and co-design engines remain, and my discussion here has only scratched the surface in *tuned optimizations.*

5.7.1 Optimization Opportunities

To give an example of further optimization and co-design opportunities, let me go back to the *relaxed I/O constraint* specialization for CFSMs. While I have presented in Section 5.5 (Algorithm 5.5.1) the benefit (reflected in variable reduction) of copy propagation,

that algorithm neglects to account for the potential of **Algebraic Identity Simplification**, the *Improved AFFG Optimization Algorithm* below tries to optimize such silly computations as `in + 0` into `in` by incorporating a new specialized normalization[23] step. Similarly *Available Expressions* can be relaxed to make expressions that depend on input valuations available if possible based on the CFSM sampling criteria.

Algorithm 5.7.1 (Improved AFFG Optimization Algorithm)

Improved AFFG Optimization Algorithm(G)

begin

while changes to the AFFG **do**

...

Relaxed Normalization

...

Relaxed Copy Propagation

...

end while

end

5.7.2 Wire Removal for Hardware Synthesis

In this subsection I would like to give a direction for future work where the DFA analysis techniques and their lattice-theoretic mathematical foundation framework can be applied to hardware logic synthesis; in particular I focus here on the *wire removal* problem identified by Khatri in [61]. The goal in wire removal is to attempt to reduce the number of

[23]Normalization presented earlier in Chapter 4 is *safe* and does not manipulate such constructs

wires between individual circuit components either by removing wires if they are *covered* by the remaining wires, or identifying a *better set of wires* in terms of a design metric (possibly number of wires, or total wire length) that can replace the original wires while keeping the functionality the same. The motivation of course is that global wires are costly in the context of Deep Sub-Micron (DSM) and we'd like to minimize them. This problem has been attacked using the notion of Sets of Pairs of Functions to be Distinguished (SPFDs) [106], a generalization of CODCs which stand for *Compatible Set of Observability Don't Cares*, in the context of multi-level and multi-valued Boolean networks. SPFDs capture the *flexibility* that can be used to implement a node in the Boolean network, and don't cares are used to reflect that the output node is not *controllable* or that the input change effect is not *observable* at this output.

Our goal in optimization is finding the best way to organize and use information, and this is an invariant whether we are trying to find the best set of operations, gates, or *wires* to implement a function in an optimal fashion. So, I contend here that, since *it's all about information representation and manipulation*, the SPFD notion can be *abstracted* and *generalized* to equivalent notions at the *operation* level instead of the current *wire* level[24], and subsequently generalized to the FFG node level, and then, if need to be, to the AFFG state level. In the SoC IP[25]-assembly domain, because of shear complexity logic synthesis cannot hope to have as large an impact as it would *inside* the IP modules. We need to carry these notions of *flexibility* to a higher abstraction and a *larger granularity*, we cannot talk about values of "signal wires" we must start to talk about *uses, and definitions* of

[24] "Bit" level if you will
[25] Intellectual Property

"avenues" of communication such as *collections of wires*, busses, registers and the like. This is where I believe a DFA analysis framework that supports the notions of information processing and is *safe* and based on an adequate design representation (such as the AFFG for heterogeneous control-dominated designs) can have a tremendous impact. To make this discussion concrete let us consider the following simple example.

Example 5.7.1

```
input in1, in2;
output out;
int x, y;
...
        x = in1;
        y = in2;
        t1 = x;
        out = t1 + t2;
...
```

In this example out is the output we are considering with immediate fanin t_1 and t_2. Therefore t_1, and x are two *alternate wires* for computing out; one can be replaced by the other because of some proximity considerations (say x is preferred because it is *available* in a "close" register). The notions of observability carry as well, for example changes in in2 are not observable at the output out. Also we can say that any valuation of y (or in2) is in the Satisfiability Don't Care (SDC) of computation output out. SPFDs carry as well in the notion of function/implementation co-design where we can identify *sets of operations* that

result in the same (*safe*) but *better*[26] computation at the output. The example is shown graphically in Figure 5.12.

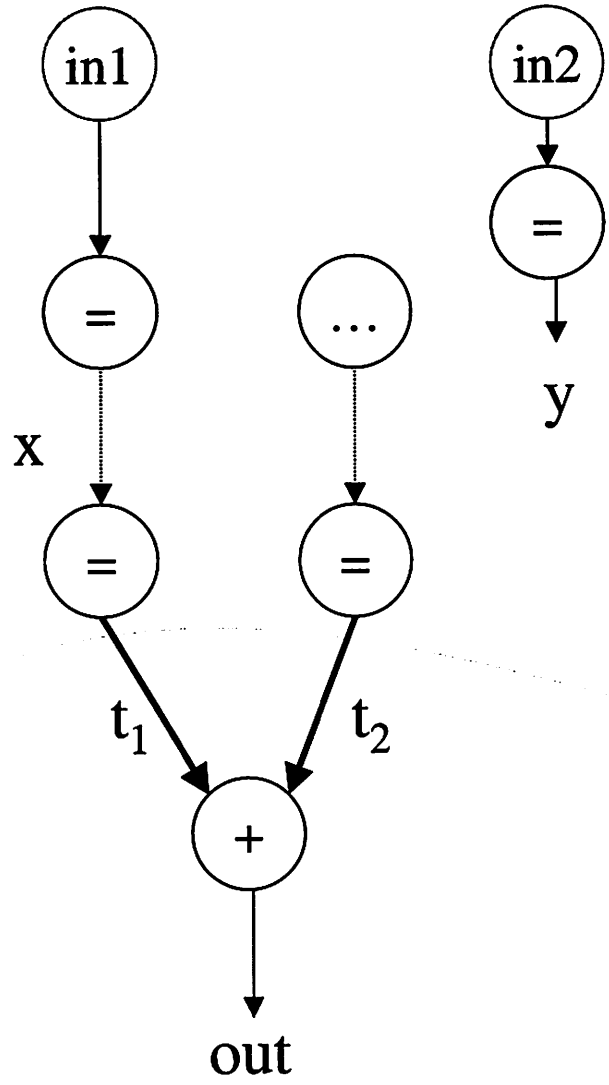


Figure 5.12: Flexibilities: Illustrative Example

I have made the case here that many of the problems we solve at lower abstract levels for, likely, a measly benefit can be turned into high powered function/architecture (or

[26]Less wiring to do in this context for example

function/implementation for purposes of this subsection) approaches if they are applied at the high level within a framework (the DFA framework is one instance) used to formulate the problems in a general enough fashion. I have used the wire removal problem and a simple non-rigorous but intuitive argument as a demonstration of this. Research into this problem is left to the future[27] ([111]).

[27]Courtesy: Abdallah Tabbara, NexSIS group

Chapter 6

Architectural Optimizations

6.1 Target Architectural Organization

6.1.1 Abstract Target Platform

I would like first to provide the reader with a concrete description of the *basic architectural organization* or *abstract platform* that my synthesis based flow targets. I will refer to this as the **macro**-architecture. I use the word *macro* in the sense that no specific implementation for any of the system design tasks (HW or SW) has been decided on, nor have any of the physical components (e.g. ASIC/FPGA for HW, micro-controller or micro-processor of specific data bit width and so on) of such an implementation been selected[1]. Stated more eloquently, this abstract platform has the following:

1. **Flexibilities:**

   - Any processor, micro-controller, or DSP can be used to run the software.

[1]This is a refinement of my earlier use of the word *macro*.

- An unlimited number of *hardware platforms* can be part of the architecture.
- Any *hardware platform* can be used as long as the proper interface glue is physically present, or a *reconfigurable* medium is available so that synthesis can generate adequate interfaces. The platform can be fully programmable as in the case of ASIC, fully reconfigurable as in the case of an FPGA, or fixed (pre-defined blocks, or an interconnection of LSI/SSI/MSI components for "glue logic").
- The *interface mechanism* can consist of a user defined **pool of resources** including: bus address and data bit widths, memory map for SW to HW interface, interrupts for HW to SW interface, and general I/O ports for direct communication, as well as dedicated ports for configuration purposes of the SW.
- The *interface between SW tasks*, as well as *scheduling* of the tasks can be done by an automatically generated RTOS supporting several policies, that uses the given *pool* to build the I/O drivers and services. Commercial kernels can also be interfaced to. See Section 7.6 for a complete discussion of this, as well as means of optimizing these services.

2. **Limitations:**[2]

- A single SW platform or partition is assumed. (This limitation can be lifted with multiple invocations of the synthesis engine, but is somewhat tricky, and not fully supported.)
- A single generic EISA[3]-like bus architecture with fixed-field decoding[4] is as-

[2]These are not limitations on the model of computation, or task-level (CFSM) semantics, just current limitations on what the synthesis engine can do, work is on-going to lift these in the future.

[3]Extended Industry Standard Architecture

[4]The memory map must be known ahead of time in order for synthesis to generate adequate decoders.

sumed, no other bus communication mechanisms are supported by default. The user can however lift this restriction by building what I call *arbiter CFSMs*; the work of Passerone in [92] is a good starting point, after adequate generalization, for automatically building such machines.

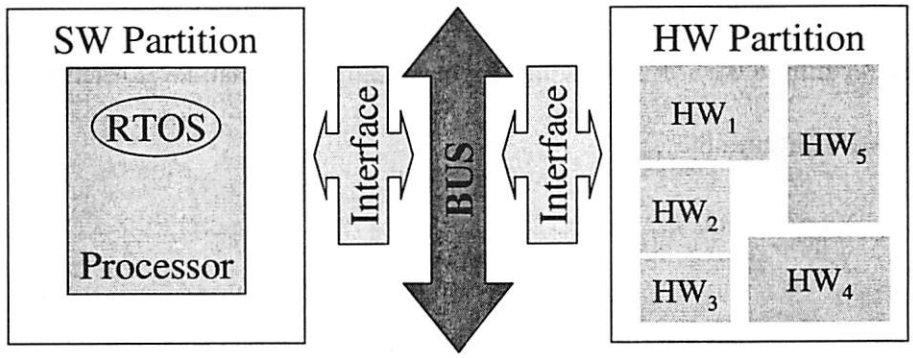


Figure 6.1: Abstract Target Platform

Conceptually, the *architecture in the large* is shown in Figure 6.1. Next, I refine this *architectural organization* to describe the *default task level macro-architecture* that the Polis synthesis engine builds, and then present in the next Section the final implementation structure.

6.1.2 Architectural Organization of a Single CFSM Task

The default task level *macro*-architecture that the synthesis flow builds for each task represents the CFSM transition function as a composition of:

- A *reactive block* and,
- a set of *combinational data-flow functions*[5].

[5]Functions should not have side effects, and their execution must complete in a bounded amount of time.

Data flow functions are implemented as arithmetic and logical expressions in the target language for SW, or library for HW. The reactive block specifies a multi-output function from a set of input variables (some of which can be outputs of data flow functions) to a set of intermediate *index* outputs. The function that is assigned to each CFSM output under any set of inputs is the one selected by those intermediate outputs. A single state from a simple CFSM is shown in Figure 6.2, and its task-level architectural organization is shown in Figure 6.3[6].

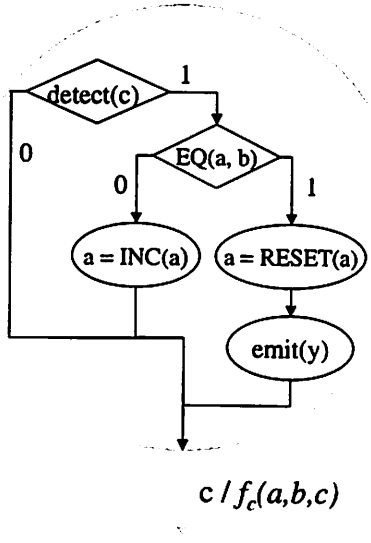


Figure 6.2: A Simple CFSM

The control flow decision making is contained within the reactive block shown in Figure 6.3 and implements the following function (adapted from [5]):

c	$a = b$	s_a	s_y
0	0	0	0
0	1	0	0
1	0	1	1
1	1	2	2

[6]For the *single* state, registers at outputs not shown

where the value 0 for s corresponds to no operation (or, equivalently, to feeding back the current value of a and y respectively), the value of 1 implies that a is incremented and y is not emitted (i.e., assigned value 0) and the value of 2 implies that a is reset to 0 and y is emitted (i.e., assigned value 1).

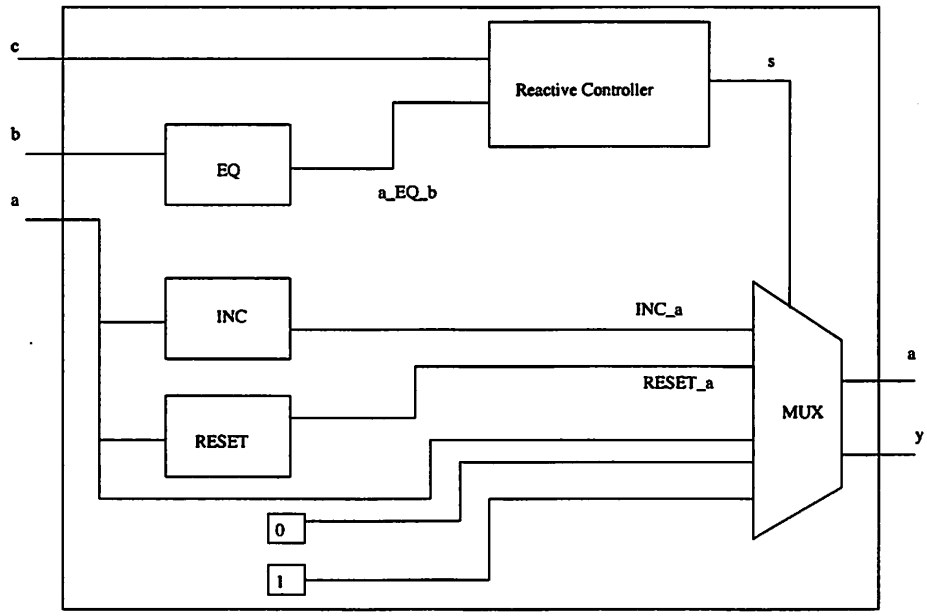


Figure 6.3: Task Level Control and Data Flow Organization

This is the *default* structure that synthesis builds for each task; it is an immediate natural reflection of the CFSM's

- reactive semantics in the reactive controller,
- FSM nature in the transitional structure of the machine, and
- extended FSM expressiveness with the associated data computations.

As we will see, this control and data flow structure does not necessarily have to be mapped **as is** to the *final* implementation, but is only a *starting* one that can be refined with

control decomposition optimization techniques at the task level described in this Chapter, as well as functional level decomposition and clustering at the system compositional level[7] as described in Chapter 8.

6.2 CFSM Network Architecture: SHIFT

The Software Hardware Intermediate FormaT (SHIFT) is a representation format for describing a network of EFSMs. It is a hierarchical netlist [5] of:

- Co-design Finite State Machines (CFSMs): finite state machines with reactive behavior
- Functions: state-less arithmetic, Boolean, or user-defined operations.

As I alluded to earlier in Chapter 3, a CFSM execution consists of four phases:

1. Idle, await trigger inputs
2. Sample inputs when invoked
3. Compute chain of operations
4. Emit outputs, return to Idle mode

A CFSM in SHIFT is therefore composed of input, output, state or feedback signals with initial values, as well as a transition relation (TREL) that describes the reactive behavior. Functions are used in the TREL to **ASSIGN** computation results to valued outputs. A function can be thought of as a combinational circuit in hardware or a function

[7] i.e. CFSM level

(with no side effects) in software. The CFSM network of SHIFT is shown in Figure 6.4 where the *implementation architecture* after the tasks are mapped onto the architectural organization is displayed.

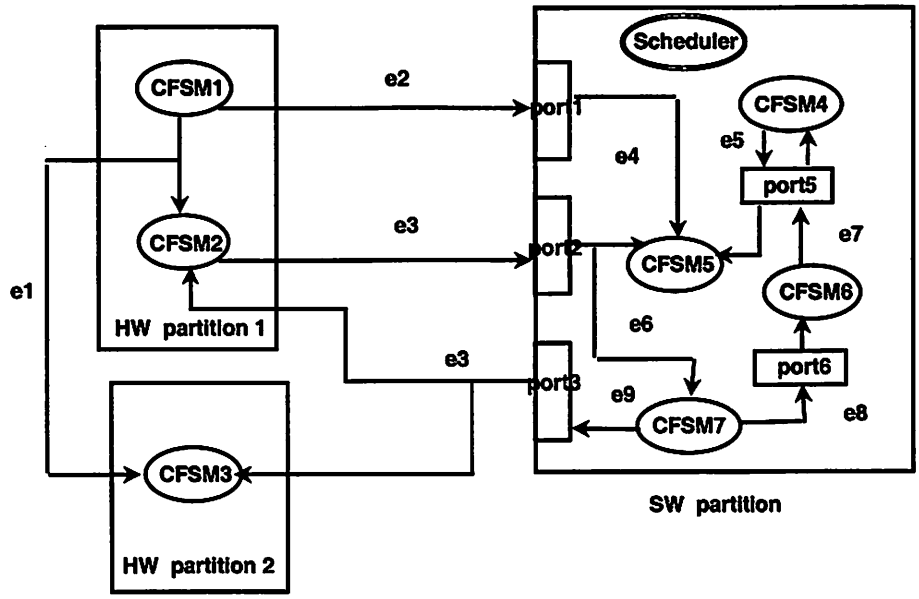


Figure 6.4: CFSM Network Architecture in SHIFT

6.3 Architectural Modeling

Architectural modeling is performed using an **auxiliary specification (AUX)** that can describe the following information:

- Signal and variable type-related information (definition of types used in the "front-end" (e.g. Esterel), re-definition of integer signals and variables, definition of unsynchronized valued signals),
- definition of the value of constants, and

- creation of a hierarchical netlist, by instantiating and interconnecting the CFSMs described in SHIFT.

6.3.1 Data Type Modeling

Data type modeling includes the following (adapted from Polis [94] manual):

- Definitions of user-defined types declared in the Esterel modules. The Esterel "front-end" allows the designer to declare such types, but their definition must be external. Suppose, for example, that declarations like:

```
type rgy;
type int8bit;
type uint8bit;
type float;
```

occur in one of the CFSM modules behavioral description. Then the following lines in the auxiliary file can define those types:

```
typedef enum { red, green, yellow } rgy
typedef mv 128 int8bit
typedef nb unsigned 8 uint8bit
typedef nb 31 float
```

The first type is enumerated, the second is signed and fits in 8 bits, the third is unsigned and also fits in 8 bits. The last declaration says that the "`float`" type is signed and fits in 32 bits (31 plus 1 for the sign).

- Re-declaration of Esterel integer variables as *integer subranges* or as *user-defined types*. Suppose, for example, that declarations like:

```
module m:
input a: integer;
...
var b, c, d: integer in
...
```

occur in the behavioral description. Then the following statements for the data types

```
mv a 128
mv b, c unsigned 256 in m
```

specify that those variables don't have the default number of bits for the `integer` data type in the target architecture, but 8 bits (`a` is signed, while `b` is unsigned). The `in m` clause specifies that the subrange declaration applies only to variables `b` and `c` of module `m`. The `nb` statement defines the number of bits that the variable can take (plus one if signed), while the `mv` statement defines the number of values (positive values if signed). Of course, `nb` is more informative for power-of-two ranges.

- Re-declaration of Esterel signals as *pure values*[8], without the control information, by using the following syntax:

  ```
  value a
  ```

  specifies that signal `a` is actually a pure value, that cannot be *awaited* or *watched*. This is especially useful for hardware CFSMs, because it saves one wire, otherwise used to indicate the presence of a signal.

- Definition of the value of constants using the `define` statement:

  ```
  define CONST_MEASURE 1000
  ```

[8]The Esterel *sensor* data type serves an equivalent purpose for input signals, but it cannot be used for output signals.

6.3.2 Component Interconnection

We can also instantiate and interconnect the CFSMs specified in the SHIFT file. Suppose that the following module was declared in Esterel:

```
module example:
input i(integer);
output o(integer), done;
constant norm;
...
```

Then the following netlist describes two cascaded instances of *example*, called *ex1* and *ex2* respectively. Note that internal interconnection signals like `o1_to_i2` need not be declared (but the types of the interconnected signals must match).

```
net two_examples:
input i1;
output o2, done1, done2;
module example ex1 [i/i1, o/o1_to_i2, done/done1, norm/125]; %impl=HW;
module example ex2 [i/o1_to_i2, o/o2, done/done2, norm/250]; %impl=SW;
.
```

SHIFT attributes, defining for example the implementation of each CFSM or hierarchical unit, can be specified as part of each `module` statement (a hardware or software implementation) as shown in the example above with the `%impl` indicator.

6.3.3 HW and SW Primitive Operation Library

Primitive, unary and binary operand, data flow functions, which are the *basic building blocks* for computations, consisting of arithmetic (such as `ADD`, `SUB`, `MUL`, ...), and relational operators (such as `EQ`, `GE`, `LT`, ...) are represented in SHIFT as mentioned earlier as sub-circuits whose computation is assigned to outputs by the reactive controller. The data flow functions implemented in SW as arithmetic and logical expressions in the target

language are *benchmarked* for the target processor, and estimates are built for a set of execution parameters as will be described in Section 7.9. In the case of HW they are implemented as BLIF networks, which are also *mapped* and subsequently characterized for the target hardware component library [102].

6.3.4 Limitations of Current Modeling and Future Improvements

The combined SHIFT/AUX architectural modeling has 2 major limitations:

- It provides for a *component* interconnection scheme instead of an *architectural view.*
- It models the system after a strict control/data path architectural decomposition in the SHIFT macro-architecture as described earlier.

The modeling is therefore limited to capturing the SHIFT macro-architecture, and a set of "micro-architectures" represented by the cost associated with operations, and the ability to specify data types and token bit widths in AUX. What is really needed in general is to provide the user with an architectural modeling capability. Most of the work in this area has focused either on special architectures (DSPs as in [107] and [50], Control/Data Flow as in [113], or Application Specific Integrated Processors as in [52]). In the future it might be possible to examine the basic elements of these methods and languages (LISA, ISDL, etc..) and define a foundation basis for a VHDL-based modeling language. VHDL is a standard language and many well supported tools for efficient synthesis and simulation are commercially available. This makes integration of automatically synthesized modules with external blocks (perhaps designed with different methodologies) quite straightforward. Using VHDL also permits us to leverage my system level co-simulation methodology de-

scribed in Section 2.3, thus expanding the capabilities of that method by providing more accurate and less restricted architectural estimates.

I would like to also state here that at high abstraction levels, VHDL may also be too detailed. For example we may need to express that a large set of implementations are valid as long as they abide by a set of *variations* such as partial-order reductions (i.e. all I/O traces equivalent under a given set of partial ordering constraints are equal), symmetry reductions, "ignoring" part of output traces during the response to an exception, and so on. The notion of *conformance* between various implementations would then be that every *finite sequence* of observations that may result from executing the detailed implementation may also result from executing the more abstract "golden" specification. VHDL cannot be solely used for such architectural (i.e. valid implementation set) specification. Propositional and temporal logics must be used as in the work of Tabbara in [117], that employs *Linear Temporal Logic*, for specifying the requirements that valid implementations of the design must fulfill. A VHDL modeling and validation framework can be used in conjunction with the latter for detailed modeling and validation (as in [117]).

In the rest of this Chapter I use the SHIFT *macro-architecture* as a user-provided target constraint, and perform some optimizations on this organization. I also describe *micro-architectural* optimizations guided by the micro-operation execution and size estimates, as well as the data type sizes provided by the user in the form of the AUX description.

6.4 Mapping the AFFG onto SHIFT

In order to perform synthesis, the AFFG is mapped into the SHIFT representation. The mapping is performed by *projecting* the AFFG representation onto SHIFT thus *decomposing* the AFFG behavior of each task into a single reactive control part, and a set of data path functions consistent with the default SHIFT macro-architecture. The *Mapping AFFG onto SHIFT* algorithm is shown next. `AUX` parameter in the algorithm stands for *architectural modeling information* as presented earlier in Section 6.3.1, while G is the task AFFG.

Algorithm 6.4.1 (Mapping AFFG onto SHIFT Algorithm)

Mapping AFFG onto SHIFT Algorithm(`G, AUX`)

begin

foreach state $s \in G$ **do**

build_trel(s.trel, s, s.start_node, G, AUX);

end foreach

end

The `build_trel` recursive routine builds the state *transition relation* including performing some book-keeping to determine whether a state needs to be *self-triggered.* A self-triggered transition is required in order to preserve the *task reactivity* when a state has no *triggering conditions*, which are conditional tests that depend on the input. The `self_trig` function call shown below inside `build_trel` checks for this and adds, if need be, the self-trigger transition to the TREL as well as a *trigger output* to all the states fanning into the state under consideration in order to activate this state transition. `build_trel` also

generates the list of data flow functions which includes both *HW/SW library primitives* like ADD, EQ, etc... as well as user-defined functions. This is shown in the algorithm below that demonstrates the key notions[9]. *Label*(n) for $n \in G$ is the set of ***labels* associated** with node n, in the context of the AFFG, these are the attributed operations linked to the node.

Algorithm 6.4.2 (build_trel)

build_trel(`Trel, s, n ∈ G, G, AUX`)

begin

ltrel = *Copy of All Elements in* Trel;

foreach operation *op* ∈ *Label*(n) **do**

/* *Mapping and Decomposition* */

if (*op* == (ARITHMETIC or RELATIONAL)) **then**

/* `dest = src1 binop src2,` *or* `dest = unop(src1)` */

Src1 = get_source(ltrel, *op*.src1);

Src2 = get_source(ltrel, *op*.src2);

create_subckt(`dest = Src1` *op* `Src2`);

else if (*op* == FUNCTION) **then**

foreach *src* in *op*.args **do**

Srcs.append(get_source(ltrel, *src*));

end foreach

create_subckt(`dest = FUNCTION(Srcs)`);

else if (*op* == ASSIGN) **then**

[9]Pseudo-code has been simplified to get the idea across; for the sake of precision I would like to mention here that the *Shared DAG* form of the AFFG introduces several additional dealings and book-keeping for instances of operations and the creation of multiple instances of sub-circuits from a single AFFG operation based on the path being traversed; these are not shown explicitly here.

```
        if (op.dest == OUTPUT|EVENT) then
            add_trel_entry(ltrel, op.trgt, OUTPUT|EVENT);
        else if (op.dest == INTERNAL) then
            add_register(op.trgt); /* create register */
            add_trel_entry(ltrel, op.trgt, INTERNAL);
        end if
    else if (op == CONDITIONAL) then
    /* if (src1) ... */
        self_trig(s, ltrel, op.src1); /* self-trigger */
        split_trel(ltrel, op.src1); /* split on false/true of src */
    else if (op == JUMP) then
    /* goto trgt; */
        if (op.trgt ∈ States(G)) then
            add_trel_entry(ltrel, op.trgt, NEXT_STATE);
        end if
    end if
end foreach
foreach m ∈ DFS(Succ(n)) do
    if m ∈ States(G) then
        store_trel_line(s, ltrel); /* Line in TREL table */
        return; /* Stop processing along this computation chain */
    else
        build_trel(ltrel, s, m, G, AUX); /* recur */
    end if
```

end foreach

end

The routines `get_source` and `add_register` reflect the SHIFT macro-architecture, where computations *within a state* have a *combinational immediate nature* and variables are *registered* so that their *state* can be sampled at the next task invocation (i.e. in next CFSM state). Consider the following example:

Example 6.4.1

```
input ...
output out;
int a, b, c, T(b+c);
...
S1:
        c = a; /* register here */
        goto D1; /* shared DAG */
...
D1:
        T(b+c) = c + b; /* subckt 1: a + b */
        out = T(b+c);
...
S2:
        T(b+c) = c + b; /* subckt 2: c + b */
        out = T(b+c);
```

For **subckt 1** we need to use **get_source(c)** to access the value of a and effectively compute **out = a + b** by creating the $T_{inst1_{(b+c)}}$ = **a + b** subckt *instance*, while in the case of **subckt 2** we only need create **subckt 2** immediately since there are no assignments *within the TREL* before this computation, and the **c** operand has the correct value since we created a *register* at the **register** label. This scenario is shown in the example for the *Shared DAG* form of the AFFG, since copy propagation is unable to reduce copies like the shown **c = a** by copy propagation because of the numerous control edges coming into the AFFG node containing the operation (i.e. varied contexts); this does not occur if the *Tree* form is used (see Chapter 3, and Chapter 4) where 2 separate computation instances get created[10]. It should be noted that the ROM algorithm presented in Chapter 5 can minimize the number of such instances by moving operations[11] (if possible) to places where copy propagation can do its work.

Several optimizations can be performed on the AFFG and the resulting SHIFT description **during this mapping process** given knowledge of the intended target. These optimizations *refine* the mapping steps in Algorithm 6.4.2 and are the topic of discussion in the next Section[12].

[10]Vahid in [123] refers to what I call instances here as clones (also generalized to procedures).
[11]That is "cloning" [123]
[12]Thus far, I have not said why we need the AUX parameter (information) in the routines

6.5 Architecture Dependent Optimizations

So far we have seen in Chapter 4 how the techniques used in software code optimization and high level synthesis can be applied to the task representation in the embedded domain, and "specialized" to the domain restrictions. If we are given additional architectural information, then we can do more restrictions and specializations leading to an increased level of optimization *before mapping* onto the final target as we have seen in Chapter 5. In this section, I present some specializations as they relate to the Network of CFSMs model of computation assumptions and restrictions that apply *after mapping* of the function onto the architecture, and how these reflect on more potential optimizations that can be performed *during the mapping process.*

6.5.1 *Macro*-Architectural Optimizations

Distributed Reactive Controller

In Section 6.1.2, I presented the *default* task level architectural organization which consists of a *single* reactive controller, a set of datapath circuits, and a multiplexer that selects the value assigned to outputs. Our experience, in the Polis group, with large industrial examples has been that the reactive controller can become quite large, and the BDD internally representing this control structure could consequently explode in size. If changing the CFSM granularity (i.e. defining a new functional decomposition, see Chapter 8 for a discussion) is not an option, then a more efficient organization in this case would be to distribute the control by decomposing the controller. We cannot, however, use the default structure as a starting point and then optimize it with BDD techniques such as variable re-

ordering since we would not be able to build the BDD to begin with. Decomposing the large reactive block and select multiplexer shown in Figure 6.3 for the CFSM of Figure 6.3 can be handled by partitioning techniques (BDD partitioning) introduced by McGeer in [83] in the area of cycle-based hardware simulation that deals with large FSM structures capturing synchronous circuits. An implementation of this partitioning technique in the HW/SW Co-design domain, and for the Polis CFSM model in particular, is being researched by Balarin and Chiodo in [6] using the concept of Control Transition Relations (CTRs). The reactive controller for each task is composed of many such CTRs, and data flow computations are represented by Data Transition Relations (DTRs). Both CTRs and DTRs compute a relation between the inputs and outputs; CTRs are *reactive* and operate on *trigger* inputs, while DTRs operate on *data* inputs and have no side effects. To evaluate a DTR or CTR for a given input means finding an output assignment that satisfies the Control or Data transition relation. DTRs capture single computations while CTRs represent a set of control conditions. CTRs can be merged to form larger control structures thus the similarity to *networks of BDDs* as opposed to a single BDD. While this former approach is a valid example of function/architecture optimization and co-design I would like to introduce here another technique of mine that can be performed on the **AFFG** to alleviate this *design explosion* problem. By using `if-then-else (ITE)` assignment operations where possible in the AFFG representation, we can relieve this problem by *moving some control into the datapath.* This serves to *break up* the reactive block, and the multiplexer. Consider the following example:

Example 6.5.1

```
output out;
int a, b, c, d, e;
if (d) then
        if (e) then
                out = a;
        else
                out = b;
        endif
else
        out = c;
endif
```

the out computation, and the `if ...` control statements can be replaced with:

Example 6.5.2

```
int a, b, c, d, e, T_out;
T_out = ITE(e, a, b);
out = ITE(d, T_out, c);
```

The process results in a smaller controller with a *tree of multiplexers* represented as ITE sub-circuits on the side instead of the one large multiplexer fed by the single large reactive block. In BDD terms, ITE operators translate some control into the data-path as shown in Figure 6.5 thus reducing the BDD size (even though the support increases). This technique serves as an excellent demonstration of the radical power of the function/architecture methodology. I am currently experimenting with the ITE **substitution heuristic** in con-

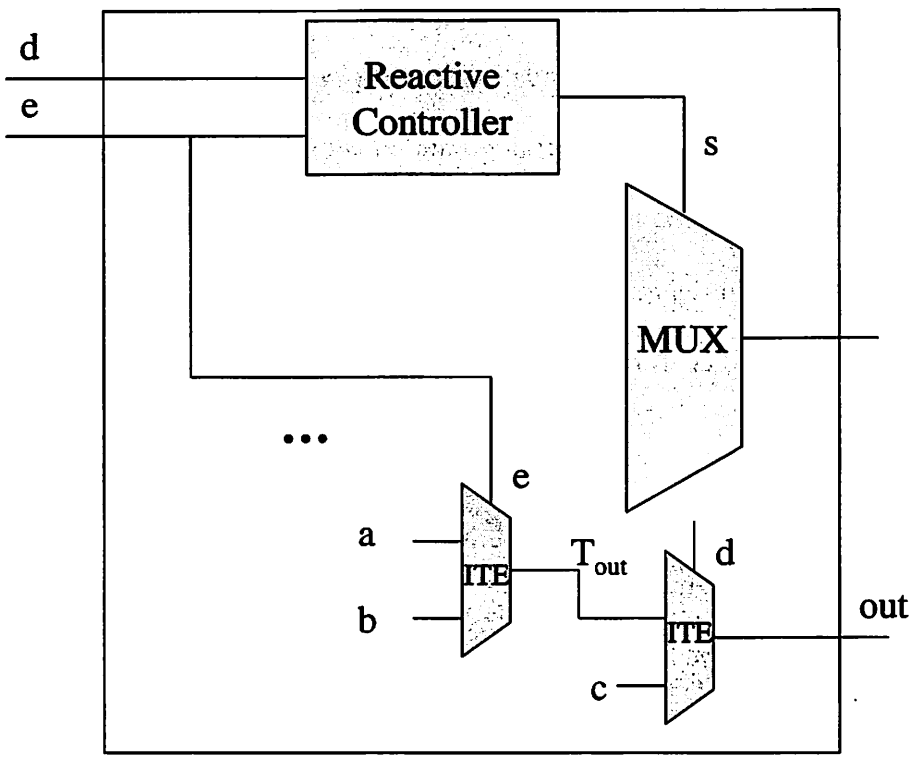


Figure 6.5: Moving Control into the Datapath

junction with *operation motion* in the ROM algorithm (see Chapter 5) since the ordering of the TESTs affects the number of ITE statements (BDD variable ordering problem), and computations can also *hide* candidate substitution opportunities.

SHIFT Sub-circuit Sharing

I have presented the *normalization* algorithm that helps identify similar computation operations, as well as *Available Expressions* analysis in Chapter 4, and then refined this in Algorithm 5.5.4 where *Operation Macro-Scheduling* is discussed. In this Section, I assume that the former steps have been performed on the FFG, and then the AFFG respectively, and discuss additional optimizations we can perform while mapping the AFFG onto the SHIFT target. Let us consider the example shown on the top of Figure 6.6.

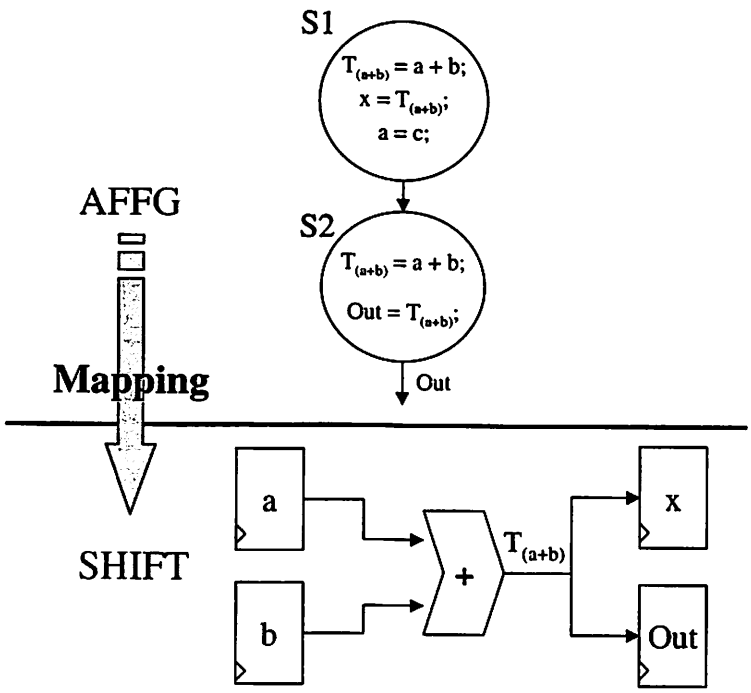


Figure 6.6: Sharing Computations During the AFFG to SHIFT Mapping

Available Expressions cannot eliminate the T_{a+b} **sub-circuit** from state S2 since the assignment a = c *Kills* that expression from $Pass(S1)$. However if we are mapping to the CFSM network target (SHIFT) where all the *variables are considered to have state* and therefore all the *variables are registered* then we can share the T_{a+b} **sub-circuit** in S1 and the similar computation T_{a+b} in S2 since a, and b are registered as shown in the bottom of Figure 6.6[13]. If this transformation is used in the final output generation, it **improves size** in both hardware and software but it reduces parallelism in the hardware, and the extra output capacitance might potentially be detrimental to the hardware performance in some cases. Using this optimization involves a *size for speed trade-off.*

Alternatively[14] we could have a *variant* of Algorithm 5.5.7 where we always **register** operations like T_{a+b} thus creating one sub-circuit instance in SHIFT and one register for its output for each such occurrence. However, this latter approach may not be as wise as we might initially think in terms of speedup in software since registers typically come at a premium, and *register spills* can happen and are typically more costly than "inlining" a computation since they involve memory access. So we cannot really make such a call on all such computations. I therefore keep the original variation of Algorithm 5.5.7 that creates registers to preserve functionality first, and perform some speedup in the cases where it is more involved computationally to check for preserving functionality, and use a *micro-architectural* analysis and synthesis technique to permit or undo the sub-circuit sharing optimization in the final output (see Section 6.5.2). Section 7.5 discusses how I handle this during synthesis within the Polis engine.

[13]For completeness I show the registers of x and out in SHIFT as well, but these registers have no bearing on this discussion.

[14]That is, yet another improvement technique

So, while the benefit of the *sub-circuit sharing* optimization cannot be determined at this high level, in the mapping step I take the approach[15] of *sharing* computations, thus **reducing the size** of SHIFT. A reduction in the SHIFT design representation helps the low level optimization and synthesis algorithms presented in Chapter 9; consequently this is a good goal for the mapping. Whether this actually makes it to the final synthesized output or not is irrelevant at this stage since the step is quite useful to do right now; the micro-architecture optimization can deal with the actual synthesis later once we have more information about the target.

Multiplexing Computation Inputs

If size of the hardware and software is of major concern, then multiplexing the *inputs of* the SHIFT *sub-circuits* is an architectural change that can help reduce the number of sub-circuits at the expense of more complex control. Let us consider Example 6.4.1 where I showed the need for two subckt *instances* of the $T_{(b+c)}$ operation: one for computing `a + b`, and another for computing the original form if you will of `c + b`. This was necessitated because after introducing the CFSM transition semantics the computation's operational semantics in the AFFG needed to be refined so as to depend on the execution *path*, and we ended up having to "replicate" the computation and tune it for each respective path. If we change the SHIFT macro-architecture and permit the use of multiplexers at the inputs of sub-circuits then we can remedy this situation where the essence of the normalized $T_{(b+c)}$ = `c + b` computation is preserved independent of the context; extra control handles checking which operand to pipe through to the sub-circuit. This is shown in Figure 6.7 where -c-

[15] Default in toolset

changes depending on the control (i.e. AFFG path) if `control` is `1` then we pick `subckt 1`, otherwise we pick `subckt 2`.

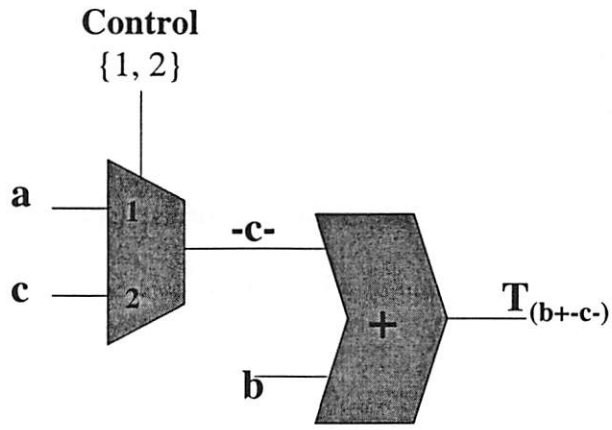


Figure 6.7: Multiplexing Computation Inputs

Function Sharing

In SHIFT we assume (and require the user to abide by this assumption) that functions are *stateless*, that is the function output depends solely on its inputs. If this is the case then we can *share* these function calls within a single state, and across states in the flavor discussed earlier for computation sharing (i.e. given that the argument variables are registered) thus generating an *optimized* SHIFT in the mapping step. Moreover, since function calls are typically quite costly, this step is bound to be useful in general for *both* size, and performance of the final output. I will always apply this technique while mapping to SHIFT in general. A micro-architectural approach can undo this optimization on a case by case basis for function instances, or inline the function call as discussed in the next Section.

Function Inlining

We currently rely on the target compiler to determine whether a function can be inlined to improve performance (at the expense of code size) using heuristics. The typical heuristic is as follows:

Proposition 6.5.1 Function Inlining Heuristic *If the* call overhead *of a function f is larger than the execution time of f's* body *then we can* improve *the performance by* in[illegible]ing *function f in its caller.*

Function inlining provides an opportunity for function/architecture co-design (see Chapter 5) in the AFFG representation if similar heuristics are used to *inline* the function *early* in the AFFG. The act of inlining opens the door for more task level optimizations by breaking the boundary between the caller and the callee. This is a direction for future exploration where *algorithms* for performing function inlining under code size constraints such as the one proposed by Leupers in [77] and improved *heuristics* can be used and generalized to encompass *instances* of function calls based on the context and the surrounding operations in the AFFG.

6.5.2 *Micro*-Architectural Optimizations

Data Type Optimization

During the final mapping of the function onto the architecture as we are traversing the AFFG, and building the CFSMs and the various interconnections and building bloc of SHIFT, we can perform *data type optimizations.* Since the Esterel front-end, SHIFT architectural modeling, and the AFFG data types are limited to the *integer* base type

currently, I will focus the discussion on describing how we can optimize the **number of bits**[16] needed to represent a variable in a correct fashion. The approach and technique can be easily extended in the future to perform data type optimization when intermixing several data type primitives e.g. integer and float, and possibly generalized (made more powerful) using interpolative (as in the work of Willems et. al. in [130]) or simulation-based (as in the work of Sung et. al. in [109]) word-length determination techniques from the DSP code generation domain. Consider Algorithm 6.4.2, and let $MAXINTBITS$ be the maximum number of bits needed to represent an integer type on the target platform, then we can optimize the number of bits as we traverse all the paths in the AFFG (i.e. covering the entire behavior) using the following procedure:

Procedure 6.5.1 Bit Representation Optimization

1. Inputs*: are assigned the user specified number of bits in AUX,* $MAXINTBITS$ *otherwise. Events are represented with* 1 *bit (and are unsigned).*
2. Constants*: of value cst are represented by* $ceil(log_2(cst))$ *bits if indivisible by 2, and an additional bit if divisible by 2, if unsigned*[17]*, with an additional bit for the sign if the constant is signed.*
3. Variables*' and* Outputs*' number of bits is initialized to* 0.
4. Sub-circuit outputs*: are represented by* 1 *bit if they are* relational *(and set to be unsigned),* $MAXINTBITS$ *otherwise.*
5. Function outputs*: are represented by the number of bits specified by the user,* $MAXINTBITS$ *otherwise.*

[16] Equivalently the number of values
[17] Constants 0 and 1 are 1 bit each

6. *ASSIGN operation: is the workhorse of this optimization. Whenever we have an operation of the form* `dest_var = src_var;` *then* `dest_var`*'s number of bits is decided as follows:*

```
if (src_var.bits > dest_var.bits) then
    dest_var.bits = src_var.bits;
    dest_var.is_unsigned = src_var.is_unsigned;
else
    /* do nothing */
end if
```

This method reduces the number of bits of internal variables (i.e. size of the registers), as well as the number of bits of the outputs thus leading to a reduction in the size of the interfaces between tasks in the case of HW tasks and HW to SW and SW to HW communication, as well as reducing the number of communication buffers in the RTOS for SW to SW communication (see Section 7.6).

Micro-Architecture Specialization Techniques

Once the *macro*-architecture is fixed there are several *micro*-architectural specialization approaches intended to optimize the performance[18] of the application function on the target platform. These specializations have either a control-oriented, or data-oriented focus. Within each camp we can identify several different emphases based on current research; these different techniques are still evolving, but I include here a broad classification based on the literature (such as [41])

[18]Taken to mean a specific design metric: size, power, speed

- **Data Processing Enrichment**
  - **Instruction Set (or Hardware Operation) Selection**: where the software instruction set (or hardware resources) is (are) tuned to the application, for example by the addition of arithmetic operations such as multiply-accumulate (MAC) instructions (as in [52]) or vector operations or components, or parallel execution units.
  - **Functional Unit Adaptation or Concentration**: where functional units are targeted to the application. These functional units could be adapted or parameterized for the application software (as in [79]), or have a concentration of functionality that is suitable for the application domain such as motion estimation, string manipulation, or pixel-operations [41].
- **Control Enhancement**
  - **Control Decomposition and Clustering**: where the functionality is broken down or grouped together in order to specialize or optimize component control [41].
  - **Communication Refinement**: where the architecture communication structure is optimized for the application, such as synthesizing adequate protocols between components (as in [92]), or building specialized bus structures for performance or power considerations (as in [36]).
  - **Storage Improvement**: where memory is restructured to optimize the access times or energy expenditure. These optimizations typically address the number,

size and distribution of memory banks, access ports and their access mechanisms, as well as the memory hierarchy (as in [78]).

6.6 Future Directions

While I have in the previous Section cited leading efforts in the *architectural specialization techniques*, I would like to point out that most of these approaches do *not* perform *optimization and co-design* of the function and architecture as I advocate in this work, but are mainly concerned with an optimal *mapping* of the function onto the architecture given some idea on what the intended application function is, and a set of suitable architectural choices. Function/architecture optimization and co-design in the *micro*-architecture is still in its infancy, and more research needs to be done in appropriate modeling of (and trade-off between) the function, and architecture at this level similar to the optimization and co-design at the AFFG level I have described in Chapter 5. The work of Choi ([31]) and more recently Benyamin ([12]) on ASIP synthesis is an example of research in this direction. Both works stay away from common *template matching* techniques, and start from the function and specialize it into a target architecture using a target compiler back-end (e.g. Trimaran compiler [119]). Both research endeavors have some promising initial investigation results.

Chapter 7

Hardware/Software Co-synthesis and Estimation

7.1 Hardware/Software Co-synthesis

My proposed overall co-synthesis flow is shown in Figure 7.1. The CDFG is built *after* performing the FFG task representation *architecture independent* optimizations, as well as the *architecture dependent* AFFG optimizations, followed by the *optimal mapping* step. After the AFFG is mapped onto an optimized CDFG we proceed with reactive

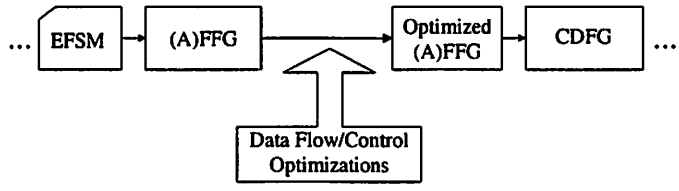


Figure 7.1: Our Optimization and Synthesis Flow

synthesis. In the next section I describe the CDFG representation, and the hardware and software co-synthesis techniques of the Polis co-design tool. A design environment based on hardware/software co-synthesis allows the designer to specify the system in a high level formal language (e.g. Esterel [16] front-end that our flow uses) by describing the functionality of each block and how blocks are connected together. Optimization and co-design on the *intermediate design representation* is then performed and subsequently a *synthesis engine* takes over by building an intermediate CDFG representation, and then generating the *best* design implementation as shown in Figure 7.2

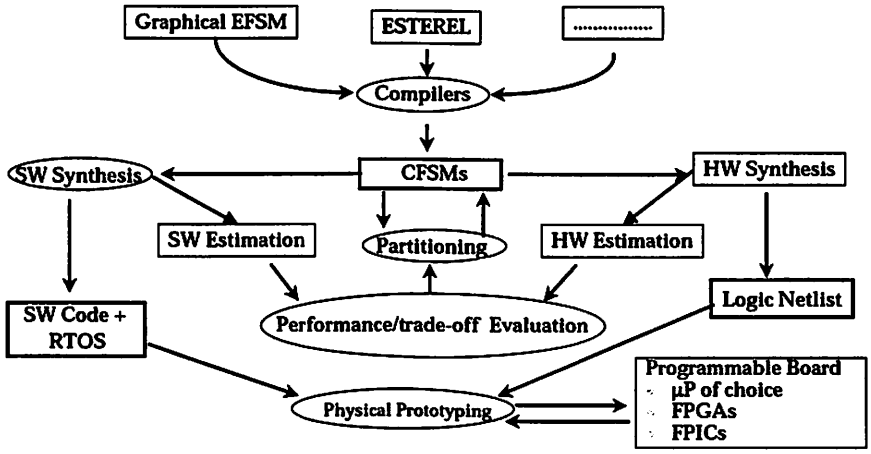


Figure 7.2: The Polis Design Flow

In this Chapter I describe the Polis co-design tool engines for:

- **Software Synthesis** A CFSM sub-network chosen for software implementation is mapped into a software structure that includes a procedure for each CFSM behavior, together with a simple Real-Time Operating System (RTOS). Synthesis of the CFSM behavior procedure is described in more detail in Section 7.2 to follow, while RTOS synthesis is described in Section 7.6.

- **Hardware Synthesis** A CFSM sub-network chosen for hardware implementation is directly mapped into an abstract hardware description format. This format can be BLIF ([102]), VHDL, or XNF for implementation on Field Programmable Gate Arrays (FPGAs).

7.2 Software CFSM Representation: The S-graph

Software synthesis in Polis is based on a Control-Data Flow Graph (CDFG) called S-graph [29]. The S-graph specifies the *transition function* of a *single* CFSM. Therefore it requires only conditional branch and assignment as primitives (augmented with arithmetic and relational expressions without side effects). An S-graph computes a function from a set of finite-valued variables to a set of finite-valued variables. The input variables correspond to input signals[1]. Each signal is a control signal, a data signal, or both, and can be associated with:

- a Boolean control variable, which is true when an event is present for the current transition, and
- an enumerated or integer subrange variable.

An S-graph is a Directed Acyclic Graph (DAG) consisting of the following types of nodes:

- **BEGIN, END** are the DAG source and sink nodes, and have one and zero children respectively,

[1]States, signals that are fed back in the CFSM network, are treated as a pair of input and output signals connected together.

- **TEST** nodes are labeled with a finite-valued function, defined over the set of input and output variables of the S-graph. They have as many children as the possible values of the associated function.

- **ASSIGN** nodes are labeled with an output variable and a function, whose value is assigned to the variable. They each have one child.

Traversing the S-graph from **BEGIN** to **END** computes the function represented by it. Output variables are initialized to an undefined value when beginning the traversal. Output values must have been assigned a defined value whenever a function depending on them is encountered during traversal of a *well-formed* S-graph. It should be clear that an S-graph has a straightforward, efficient implementation as sequential code on a processor. Moreover, the mapping to object code, whether directly or via an intermediate high-level language such as C, is almost 1-to-1.

7.2.1 S-graph Optimization

An S-graph is optimized for speed or size by re-ordering the nodes to minimize depth or size respectively. The actual re-ordering algorithm is implemented as dynamic variable re-ordering using the "sift" algorithm introduced in [98] on the BDD representing the CFSM characteristic function. There are three classes of variable orderings [5]:

- Ordering each output *after* its support yields an S-graph where all the decision computation is performed by **TEST**s. **ASSIGN** nodes are labeled with the data flow functions (see Chapter 6 for the SHIFT architectural organization).

- Ordering each output *before* its support yields an S-graph without **TEST** nodes. Each

ASSIGN node is labeled with the logical *and* of the enabling condition for the output multiplexer and the data flow functions.

- Other orderings give an S-graph with a mix of **TEST** and **ASSIGN**.

7.3 Polis Approach to Software Synthesis

The *S-graph*, introduced in Section 7.2, is used as an intermediate data structure. The S-graph, as mentioned earlier, has a direct representation in a simple C code[2] subset (consisting of assigns, tests, conditional and unconditional branches) referred to here as **CFSM-C**. The software synthesis procedure follows these main steps:

1. Translation of a given CFSM into an S-graph.
2. S-graph optimization (See Section 7.2.1) and code-size estimation (see Section 7.9).
3. Translation of the S-graph into a target language (CFSM-C).
4. Scheduling of the CFSMs and generation of the RTOS (see Section 7.6).
5. Compilation into machine code to be run on the target processor (left to the target compiler).

7.3.1 An Illustrative Example

I now introduce an example that will serve to illustrate the previously mentioned SW synthesis techniques (and also referred to in Section 7.6). Figure 7.3 describes a simple

[2]C is assumed to be the language the target compiler takes as input to generate processor specific machine code.

CFSM, implementing a seat belt controller that turns on the alarm if the driver does not fasten the seat belt 5 seconds after turning on the ignition key, and turns off the alarm after 10 seconds, or when the seat belt is fastened. The corresponding S-graph is shown in Figure 7.4. A fragment of the C code derived from an S-graph implementing it (i.e. CFSM-C) appears in Figure 7.5. The macro `detect_key_on_to_belt_control` returns 1 if the environment has sent the event key on to the CFSM. The macro `emit_alarm` emits the alarm event to the environment. The statement `v__st = 2` updates the CFSM state, and so on. The method for obtaining code cost estimates at the system level (size in bytes and time in clock cycles) is discussed in Section 7.10.

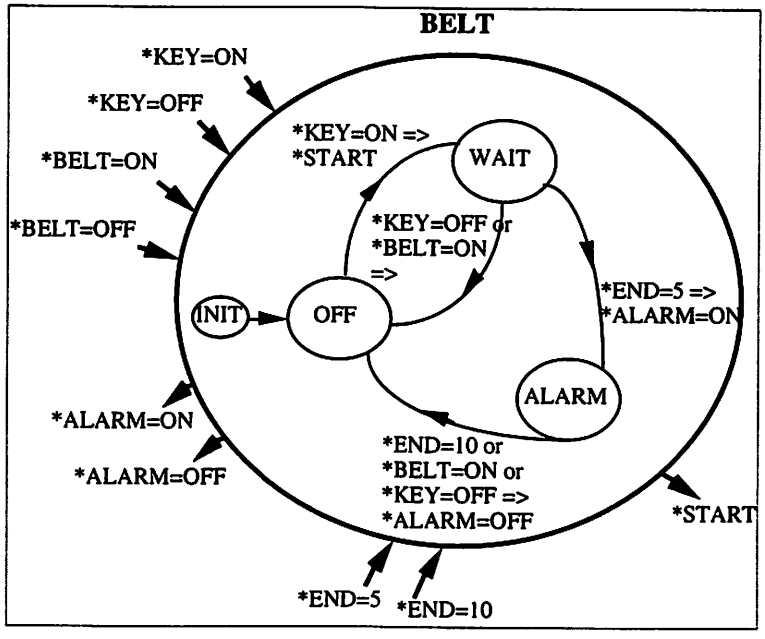


Figure 7.3: The CFSM of the Seat Belt Alarm Controller

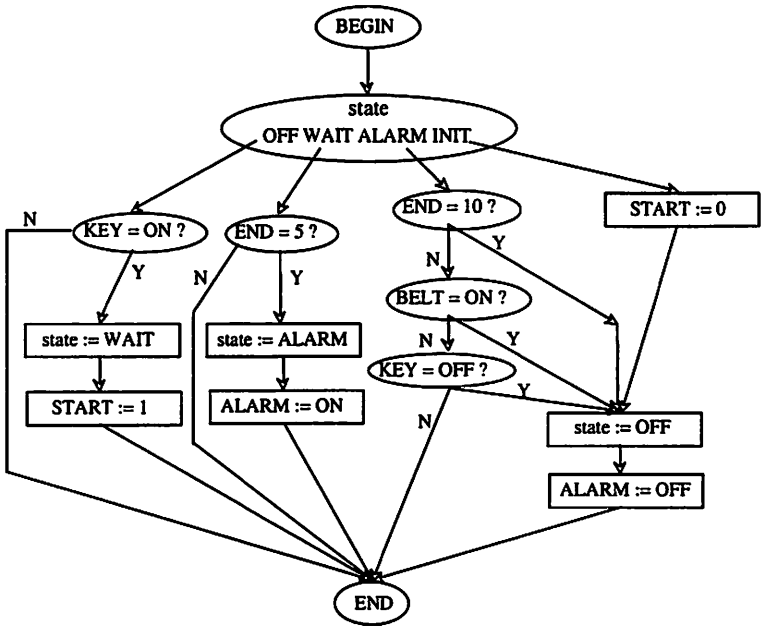


Figure 7.4: The S-graph of the Seat Belt Alarm Controller

7.4 Polis Approach to Hardware Synthesis

Hardware CFSM components of a design in POLIS are implemented as Finite State Machines (FSMs) that consist of a combinational part for the next state logic of the transition relation and the data path, and latches (all driven by the same clock) for the outputs and states. The result is a logic netlist (generated using classical logic synthesis techniques) that can be mapped to a specific technology.

7.5 Optimization and Co-design Guiding Co-Synthesis

In the earlier Chapters, I described high level optimization and co-design. Those steps give "hints" to the subsequent synthesis through the use of statements like `register`, `static ...` for registers, and `inline` for inlining function calls for example. Micro-

```
void belt_control()
{
...
/* check key_on event */
L1:       if(detect_key_on_to_belt_control)
                    goto L3;
          else goto L2;
...
          /* check CFSM state */
L3:       switch(v__st) {
          case 1: goto L6;
          case 0: goto L4;
          case 3: goto L14;
          default: goto L7;
          }
...
L7:       if (detect_e_timer_e_end_5_to_belt_control) {
                    ...
          }
          else {
                    goto L0;
          }
...
/* update CFSM state */
L10:      v__st = 2; goto L0;
...
/* emit alarm */
L12:      emit_alarm(); goto L13;
...
L0:
          return;
}
```

Figure 7.5: C Code for the Seat Belt Alarm Controller

architectural analysis can also guide synthesis. In particular when synthesizing statements in C for the SW, or BLIF for the HW, the synthesis engine must know whether to *inline* a computation as in the following code segment:

Example 7.5.1

```
x = ADD(a, b);
...
out = ADD(a, b);
```

or whether it should *register* this computation as in the following code segment:

Example 7.5.2

```
reg1 = ADD(a, b);
...
x = reg1;
...
out = reg1;
```

Such decisions can be guided by a *critical path* analysis in software and hardware using the estimation approach described in detail in Section 7.9. Research into proper micro-architectural guidance is still on-going within the Polis framework [73], and I will not get into it here; I will always inline all the computations[3] (therefore opting for *maximum speedup with no register pressure*) unless a register has been identified by the high level optimization and co-design and an architectural hint to this effect is provided to synthesis.

[3]Default in Polis

7.6 The Real Time Operating System

We have focused in our discussion so far on performing function/architecture optimization of the function and architecture. *Communication* is a very important aspect of the architecture; in particular for the execution of the SW tasks, the *Real-Time Operating System* (RTOS) plays a significant role in the performance and size of the final output. It is therefore critical to *optimize* the RTOS for the *specific* architecture. Embedded systems are typically implemented as a set of communicating hardware and software components. Typically several software tasks share a single processor. An RTOS is used to enable sharing and provide a communication mechanism between such tasks. While commercial RTOSs are available for most widely-used micro-controllers and they provide a significant reduction in design time and often lead to more maintainable systems, they are typically quite general and inefficient. Quite often, when performance is crucial, RTOSs are hand-coded by an expert for a particular application. This approach is obviously slow, expensive and error-prone. To that end, I present here an overview of some of the work I have performed in this regard with fellow Polis team members[4] ([8], and [5]) and emphasize the function/architecture co-design aspects of it, where the architectural constraints and the task function guide the RTOS synthesis process to generate an *efficient* RTOS. We endeavor to generate "low-level and simple C code" to implement the RTOS services but, as we will see, there is always an advantage to writing the services at the assembly level, however since our RTOS is built *around the application itself* we almost always generate a better OS in terms of size, and get quite close to, and sometimes surpass, hand-tweaked RTOS building.

[4]Courtesy: Felice Balarin, Attila Jurecska

As we will see, our technique can quite easily overcome other "high-level" approaches to RTOS design.

7.6.1 Overview of Scheduling Policies

Scheduling policies for real-time systems are generally classified as follows:

- *static* or *pre-runtime*, where tasks are executed in a fixed cyclic order. This order may or may not contain repetitions in order to cope with different expected task activation times and/or deadlines.

- *dynamic* or *runtime*, where the order of execution is decided at run time. Generally, the execution policy is priority-based, in that, at each instant, one among the set of ready tasks is dynamically chosen according to a *priority order*. Priority, intuitively, is a measure of "urgency" of each task, and can in turn be determined

  - *statically* at compile time, or

  - *dynamically* at run time.

  Moreover, runtime scheduling can be

  - *preemptive* if the currently executing task can be suspended when another task of higher priority becomes ready, or

  - *non-preemptive* otherwise.

The trade-off is in responsiveness and efficiency. However for control-dominated applications, typically the time of task readiness is unpredictable so it is more adequate to go with a runtime scheme that either polls the tasks periodically in a cyclic fashion or is

interrupt based. The underlying theory behind scheduling for real-time systems will not be reviewed in detail here, and the reader is referred to the discussion found in [5] and the references therein.

7.6.2 RTOS Synthesis in Polis

I present here a brief overview of RTOS synthesis in Polis, where the RTOS services (scheduler, event/value buffers, interrupts, etc..) are *tuned to the specific application* at hand. The RTOS generated by Polis for a given network of CFSMs has two major responsibilities:

1. Provide communication mechanisms (i.e. define `detect` and `emit` functions) among CFSMs implemented in SW and between the SW partition the OS is running on and HW partitions, and

2. schedule the execution of the SW tasks.

In Polis, the user can choose between two basic scheduling algorithms to coordinate SW CFSMS: cyclic scheduling, and static priority based scheduling. It is also possible to choose between preemptive and non-preemptive versions of static priority based scheduling. In either case, the RTOS keeps track of events a SW CFSM is sensitive to, and will not execute it unless at least one of those has occurred since the last execution of the SW CFSM. Therefore for each SW CFSM, and each event it is sensitive to, the RTOS maintains a flag which is set when the event occurs, and reset when that SW CFSM makes a transition Emitting an event thus requires setting these flags for all potential consumers. Detecting an event is implemented as a macro that checks if a flag is set. The value of an event is

communicated through a shared variable. While the RTOS distributes information about event occurrence to all the detecting CFSMs, it keeps only a single copy of the event value in order to save memory. The event emission and detection capability described above is sufficient for communication between SW CFSMs. For communication between the SW and HW CFSMs a "rendez-vous" style of communication is used, where the RTOS synthesis routines request the user to identify a *pool of resources* that the OS can use. These include: *ports*, *ISR tables*, as well as the *memory map* for memory-mapped I/O communication. This RTOS synthesis, and evaluation flow in Polis is shown conceptually in the flowchart of Figure 7.6.

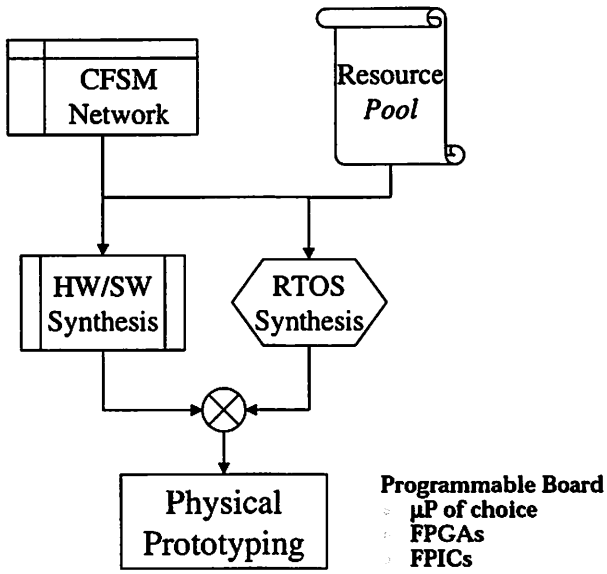


Figure 7.6: RTOS Synthesis and Evaluation in Polis

7.7 Interfacing Polis to Commercial RTOSs

The Polis generated RTOS can be substituted with an existing real-time kernel. The integration requires some modification by the addition of a *wrapper* layer around the automatically generated C code for the SW tasks, creation of new I/O tasks, and addition of code for initialization of the commercial RTOS services. Care must be taken to properly implement the Polis communication model, and preserve the semantics of the CFSM network Model of Computation (MOC).

In the next section, I present some data to demonstrate the effectiveness of tuning the RTOS to the application. The commercial RTOS I have used for the comparison for the Motorola 68HC11 platform is CMX which is a typical representative of commercially available kernels. It offers over 60 functions that enable the user to create multi-tasking applications for embedded controllers. Task, event, message, resource, timer, queue, and memory management take place through calling of these functions. The scheduler is based on pre-emption, and interrupts. Tasks can also cause immediate task switch if they become at a higher priority than the currently executing one. For the ARM7 target architecture I compare Polis against the open source version of VRTXoc [127]. The following changes were done to the normally Polis generated code to integrate it with the chosen kernel:

- The original Polis routines for emitting events have been changed to kernel function calls that signal events. To ensure that a SW CFSM is executed only if some relevant event happens, it is wrapped with code that awaits (by a system call) any event the CFSM task is sensitive to, sets proper event flags, and then executes the Polis generated behavior C code.

- The original polling task, has been re-implemented as an independent task triggered by a periodic external event (target or host timer for example).

- A new main program initializes the RTOS kernel by creating and triggering tasks and peripherals (if needed).

7.7.1 Experimental Setup and Results

I compared Polis generated and commercial RTOSs in several areas, namely[5]: task execution, event emission, distribution and detection, I/O operation execution, context switching, control and data code execution times, number of lost events, and code size. I report here some representative results based on emulating the very simple seat belt controller application example[6] presented earlier in Chapter 7 using a round-robin scheduling scheme for simplicity. For this small example I compared the RAM and ROM requirements as well as the following execution metrics (the same I/O primitives have been used and no events were lost):

- Context Switch: is the event sequence resulting from execution context transfer from the running task to a ready one. To be more concrete this includes overhead of *switch* from task to the RTOS and then to the other ready task.

- Schedule Latency: is the time taken for an event sensed to "activate" the task awaiting it (i.e. task is *ready*) and the time taken for that task to switch from ready to *running*. This includes the cost of *Event Delivery*.

[5]I present RTOS synthesis vs. commercial interface results here to explain the advantages and limitations of the synthesis approach, I introduce the general co-synthesis results in Chapter 9.

[6]I use a small sized application in order to make the RTOS effects more dominant; application memory requirements are negligible with respect to that of the RTOS.

- Event Emission: Time it takes the task to *emit* an output, meaning registering a new event with the RTOS as a result of the task's execution.

Note that I do not report *Event Detection* that is the time it takes the task to "check" event presence in its internal buffers since I opted to use the same internal buffers to be fair in the comparison and this number will always be the same for both, synthesized and interfaced to, kernels.

68HC11 Target Processor

In order to validate the Polis generated RTOS and compare it against the CMX RTOS for the Motorola 68HC11 micro-controller I used the prototyping environment at Cadence Berkeley Labs[7] [112] which consists of a complete Polis based design flow to generate the final software and hardware elements including boards with hardware components, and instrumentation (e.g. logic analyzer, emulator) as shown in Figure 7.7. This setup allows us to:

- compile, link, and download the software parts into the target,
- program the hardware parts into FPGAs, and
- debug in real-time the software code running on the target architecture.

The memory requirement comparison is shown in Table 7.1. While it is apparent that the Polis RTOS is much smaller than CMX the difference in RAM requirements is not clear at first glance. The details of the RAM requirements are shown in Table 7.2, which shows that since in Polis we use a "high-level" approach for RTOS synthesis using C code,

[7]Courtesy: Cadence Design Systems, Inc.

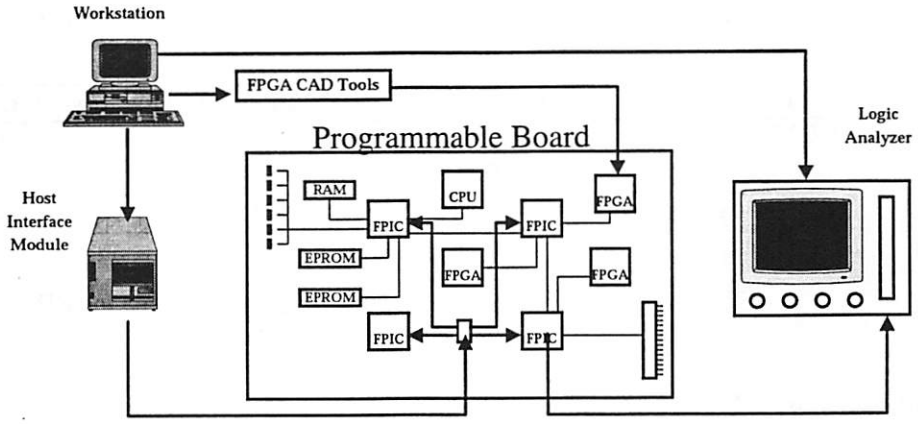


Figure 7.7: Prototyping Platform

RTOS	Code (ROM)	Data (RAM)
CMX	3191	4588
Polis	1391	4753
Improvement	**56.4 %**	**-3.5 %**

Table 7.1: Memory Requirement (in bytes) Comparison for 68HC11

RTOS	Heap	Stack	Data
CMX	512	2048	1796
Polis	4096	256	155
Improvement	**-87 %**	**88 %**	**91 %**

Table 7.2: RAM Requirement (in bytes) Comparison for 68HC11

RTOS	Context Switch	Schedule Latency	Event Emission
CMX	29	230	9
Polis	101	184	9
Improvement	**-71 %**	**20 %**	**0 %**

Table 7.3: Execution Time (in μs) Comparison for 68HC11

and CMX uses hand optimized assembly routines the *heap* requirements are much larger in Polis. This is made up for by the fact that we use *function calls* for scheduling whereas CMX keeps the function activation records on the stack and therefore has a large *stack* size expenditure. The *data* section [8] is about 12 times larger in CMX since the RTOS is generic and therefore has a larger number of functions, and global variables.

The execution metrics are reported in Table 7.3. It can be seen that while the Polis RTOS is clearly superior in terms of code size, the high level synthesis approach cannot beat the hand optimized assembly services. But it does come close in overall schedule latency (of course the reader should note that this benefit will dwindle with larger applications that have more tasks as context switch cost starts to dominate), in fact attention must be made in the interfacing and wrapping since the performance is fairly close an extra event buffer for example could throw the balance in favor of Polis.

[8] The other segments (interrupt vector table, register set) are the same and not shown

RTOS	Code (ROM)	Data (RAM)
VRTXoc	13676	556
Polis	5128	432
Improvement	**58 %**	**22 %**

Table 7.4: Memory Requirement (in bytes) Comparison for ARM7

RTOS	Context Switch	Schedule Latency	Event Emission
VRTXoc	4	52	19
Polis	2	14	4
Improvement	**50 %**	**72 %**	**79 %**

Table 7.5: Execution Time (in μs) Comparison for ARM7

ARMulator for ARM7

For the ARM7 architecture we used [44] the ARMulator from ARM's SDK 2.50 and compared Polis's RTOS against the open source version of VRTXoc [127]. The memory comparison results shown in Table 7.4 display that here Polis's RTOS is also quite superior to that of VRTXoc.

Table 7.5 shows the execution time comparison. It is clearly visible that here Polis's RTOS performs better than VRTXoc as well. The explanation is quite simple. VRTXoc is also written in C, and there is *no "assembly vs. C" advantage* compared to Polis RTOS. The benefit of using application specific RTOS optimizations is clearly visible here; while the method of task invocation in VRTXoc is inherently better (stack-based in VRTXoc vs. function calls in Polis), the smaller number of functions in Polis more than accounts for the potential performance drawback of its call strategy. Of course the advantage that VRTXoc presents is that it has several communication mechanisms not currently supported in the Polis automatically generated RTOS such as queues, and messages.

7.8 Optimizing the RTOS

I have so far discussed the effects of taking the application into account and tuning the RTOS synthesis, and I compared the output of this procedure to conventional generic kernels. I have not discussed the effects of *function optimization* (see Chapter 4) and *architectural optimization* (see Chapter 6) on the RTOS. Function optimization can lead to significant improvement in the *size* of the RTOS, as well as in *runtime* by targeting *task level optimizations*. The idea is that optimizations which *reduce local variables*[9] in the tasks reduce:

- Buffers in the RTOS: for the valued shared variables that are the value communication mechanism in the CFSM model. This is reflected in *code size* of the RTOS ROM.
- Stack Overhead: is reduced for the task since it has a small number of *local copies* of these variables performing the "sampling" at each task invocation.
- Call Overhead: is reduced since the stack itself is reduced. This is a performance improvement that favors function calls i.e. the performance of a function call based approach is improved relative to an RTOS approach that keeps everything on the stack. Of course it should be noted that this optimization improves the memory requirement of the latter approach since the size of the stack is reduced.

Architectural optimization helps in the RTOS *size* and *speed* by performing *data type optimizations* as discussed in Section 6.5.2 thus reducing the data type size of the communicated information, saving in the call stack overhead and the RTOS buffer sizes.

[9] As I discussed earlier, FFG level optimizations do not change the I/O behavior so no event buffers are affected, but value buffers that store global shared variables are, and the number of these buffers can be significant.

The previous comments indicate therefore that optimizations at the task level can improve significantly the RTOS performance whether it is an externally interfaced RTOS service or an application specific RTOS. Results on RTOS optimization due to task level optimizations are reported in Chapter 9. The optimization analysis can also potentially lead to other improvements in the application specific approach as discussed next.

7.8.1 Future Directions

Our aim in the RTOS generation is to use ***application specific knowledge*** to improve the RTOS quality over commercial kernels. Results show that our approach is competitive to generic kernels, and our aim here has been to emphasize the need for taking the ***application*** demands and the ***function*** specifics into account, and not just take advantage of the architectural features, which is where most kernels spend their time optimizing. I expect that a synergy between function/architecture and application is key to optimizing the RTOS, and my investigation supports this conclusion. From the experimentation, I have identified several potential areas for further optimization, these include:

- Function Calls: Using function calls to call the tasks from the scheduler is a straightforward approach for automatic synthesis, but it needs to be compared against other approaches in particular ***task inlining*** where tasks are inlined in the scheduler. This promises to be quite useful in a cyclic scheduling scheme to reduce the context switch overhead.

- Macros and Pointer Comparison: These are currently used in the Polis RTOS for event detection within a task, and also for event emission. This approach seems to be

quite successful in advanced architectures (e.g. ARM7 results) but it is less favorable (especially since it demands memory) on simple architecture platforms (e.g. 68HC11 results).

- Optimization Analysis: This technique can be used to guide the RTOS synthesis (encompasses the first two bullets). For example heuristics can decide whether to inline a task or not based on the task body, parameters, as well as the current scheduling policy.

While my presentation here is applicable to any SW partition with a specific RTOS, the current Polis synthesis restriction requires a single SW partition. We hope to lift this restriction in the future and explore scheduling and communication schemes across these SW partition boundaries.

7.9 Measuring the Final Implementation Cost

There are several ways to measure the effect of the optimizations and subsequent synthesis of software on the final implementation:

- Rapid Prototyping, and Emulation: A rapid prototyping and emulation environment was presented in Section 7.6 [112] for performing some experiments. While this is the most accurate implementation cost measurement method it is the most time (and resource) consuming, and is not suitable for rapid high level guidance, or quick results feedback, however, in some cases there is no substitute for this either from the accuracy perspective or from the benchmarking view. The alternatives listed below require

an initial benchmarking and validating overhead that must be done on the target architecture before relying on the simpler (less accurate) approaches.

- Target Architecture Profiling: This approach is quite accurate but is also time consuming, and needs to be performed for every application (in fact every optimization must be followed by a synthesis run) so it is quite laborious. Instrumentation tools help alleviate the laborious aspects of this process. For example I will use in some of my reporting the `pixie` instrumentation of the ATOM analysis tool [108].

- Software Estimation: is the least accurate between the previously mentioned techniques, but if it is conservative and fast, it can be quite useful for synthesis result comparisons where *relative* improvement measurement is of utmost importance, and the *absolute* value is of secondary importance. For getting the absolute measurement there is no substitute to profiling or emulation.

Software estimation is the main technique I use to evaluate the final implementation, and the result of the function/architecture optimization and co-design approach. In this Section I give an overview of this approach.

7.10 Software Estimation

In our optimization and co-design framework, software cost estimation can be used for rapid evaluation of synthesis quality and hardware/software co-simulation without the overhead of full prototyping of the system.

7.10.1 Overview of Software Estimation in Polis

In the Polis system, code cost (size in bytes and time in clock cycles) is computed by analyzing the structure of each S-graph node, for example:

- the number of children of a **TEST** node (a different timing cost is associated with each child), and
- the type of tested expression. For example, a test for event presence must include the RTOS overhead for event handling, and reading an external value must include the execution time of the driver routine.

A set of cost parameters is associated with every such aspect, and is used to estimate the total cost of each node. For example the costs in clock cycles on a 68HC11 target processor using the Introl compiler for the simple CFSM shown earlier in Figure 6.2 are displayed on the corresponding S-graph nodes in Figure 7.8.

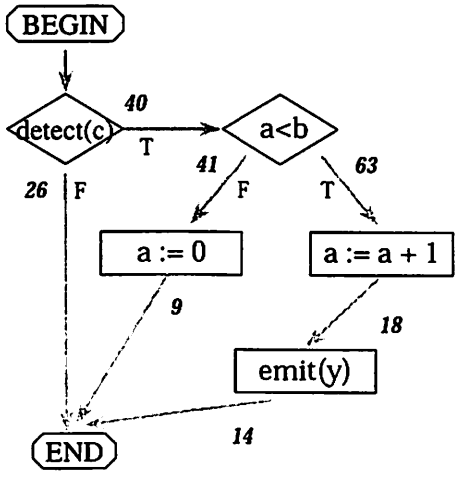


Figure 7.8: A Simple S-graph Annotated with Execution Cost

The parameters can be derived either automatically, or by hand (e.g., by inspecting the assembly code after synthesis and compilation for the target processor) as shown in Figure 7.9. In the former case, the processor library maintainer needs to compile a set of benchmarks and analyze their size and timing by using a profiler for the target system.

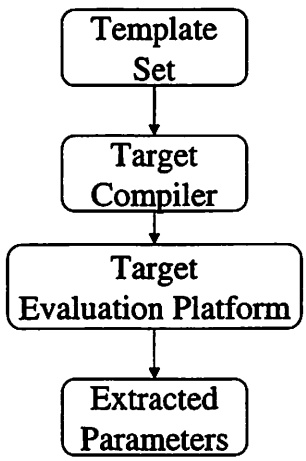


Figure 7.9: Parameter Extraction Flow

7.10.2 Modeling the Synthesized Software

The key aspect of the software performance estimation method implemented in Polis, referred to in this dissertation as the **macro-modeling method**, is the fact that the software synthesis flow is aware of the structure of the program being synthesized. The S-graph structure (see Section 7.2) is very similar to the final code structure, and therefore helps in cost and performance estimation. The procedures generated by Polis which implement the task behavior can be represented by a simple model as follows [110]:

```
(1)   function ( )
      {
```

(2) Initialization of local variables (assignment statements);

(3) Directed Acyclic Graph of *if*, *switch*, and assignment statements;

(4) Event buffer cleanup and *return*;

}

The execution time $T(p_i)$ and code size S of a procedure that follows the above structure can therefore be modeled as (adapted from [5]):

$$T(p_i) = T_{pp} + kT_{init} + T_{struct}(p_i)$$

$$S = S_{pp} + kS_{init} + S_{struct}$$

Where:

- p_i is an execution path of the procedure,
- T_{pp} is the execution time for entering and exiting the function ((1) + (4) in the above procedure model),
- T_{init} is the average execution time for initializing a local variable ((2)),
- k is the number of local variables, and
- T_{struct} is the execution time for the structure of mixed conditional statements generated from **TEST** nodes in the S-graph and assignment statements generated from **ASSIGN** nodes ((3) in the above procedure model). T_{struct} is therefore the execution time along a path determined by the structure of (3) and the CFSM inputs as sampled by the RTOS at the beginning of the transition.

Similarly, S_{pp} is the code size for entering and exiting the function, S_{init} is the average code size for initializing the local variables, and S_{struct} is the total code size for (3). As mentioned

previously, an S-graph and the C code generated from the S-graph have the same structure, and the kind of C statement (i.e. *if*, *switch*, or assignment) depends on the type of the corresponding node (**TEST** or **ASSIGN**) and the associated variable type in the S-graph. The execution time and code size of each C statement that appears in the S-graph depend on the kind of C statement, the code generated by the target compiler, and the performance of the target CPU. Therefore, the T_{struct} and S_{struct} can be modeled as

$$T_{struct}(p_i) = \sum_{i \in p_i} C_t(node\_type\_of(i), variable\_type\_of(i)),$$

$C_t(n, v)$ above is the execution time for node type n and variable type v , and

$$S_{struct} = \sum C_s(node\_type\_of(i), variable\_type\_of(i)),$$

where $C_s(n, v)$ is the code size for node type n and variable type v. The target software domain can be modeled by a set of $C_t(n, v)$ and $C_s(n, v)$. The assumptions and simplifications made in the previous equations include:

- the target compiler does not perform global optimizations [10], and
- the effects of cache and pipelining are neglected which allows both $C_t(n, v)$ and $C_s(n, v)$ to be represented by a set of fixed cost parameters for each S-graph construct[11].

A set of cost parameters can be obtained if this approach is followed by using simple benchmark programs containing a mix of the C statements that appear in the task and analyzing the execution time and code size of these programs on the target compiler

[10] This is true of most embedded processors today

[11] This is a more serious limitation, and will be addressed shortly.

and the target CPU as was shown earlier in Figure 7.9. Also, T_{pp}, T_{init}, S_{pp}, and S_{init} can be determined beforehand, because they are constant and independent of the structure of (3) since in Polis *global shared variables* are used for value communication, and therefore no parameter passing is done[12], and the number of these variables is known apriori.

7.10.3 Cost Parameters

Estimation in Polis uses a parametric model of the target processor and its compiler in order to estimate the execution time and code size. The method consists of two major parts. First the cost parameters for the target system are determined. Second, those parameters are *applied* to the S-graph[13], to compute the minimum and maximum number of execution cycles and the code size. The execution cost for any S-graph path can also be computed of course but is not automated currently. Each node is assigned two specific cost parameters (one for timing and one for code size), depending on the type of the node and the type of the input and/or output variables of the node. Edges also have an associated cost, as the `then` and `else` branches of an `if` statement generally take different times to execute. The parameters for execution time or code size correspond to the kind of C statements generated from a node in the S-graph:

- a **TEST** node `detecting` the presence of a signal (which yields an RTOS function call),
- a **TEST** node branching on a multi-valued expression (which yields an *if* or *switch* statement),

[12]Otherwise these costs would vary depending on the number and sizes of these parameters
[13]That is the S-graph is annotated with these costs

- an **ASSIGN** node emitting a signal (which yields an RTOS function call),
- an **ASSIGN** node which assigns an expression to a data signal (which yields an assignment).

In the case of a **TEST** node with two outgoing edges, the cost parameters for each edge (i.e. the true case and the false case) are stored explicitly. For a **TEST** node which has more than two edges, the execution time for the k-th edge is represented as $T_{switch} = C_{base} + kC_{case}$, by using two parameters C_{base} and C_{case}. The other parameters for the execution time and code size are defined for

- pre-processing and post-processing for a C function (together these correspond to T_{pp} and S_{pp}),
- a branch operation (generated from a *goto* statement),
- initialization of a local variable (corresponds to T_{init} and S_{init}),
- average execution time and size for pre-defined software library functions, and
- the size of integer variables.

Synthesized programs may also contain primitive data flow and user-defined functions. Estimation is done for the primitive data flow functions such as ADD, and EQ by considering an average execution time and code size for these pre-defined functions[14]. To improve the accuracy of the estimation for a program which contains these functions, or to model user-defined functions the estimator takes additional parameters for each such

[14]This is quite reasonable for a RISC type of architecture

function. These additional parameters are defined with a quintuple: (name, maximum execution delay, minimum execution delay, code size, and result bit width). With this mechanism, hand-written functions or externally synthesized code called by a CFSM can be characterized for software cost estimation.

7.10.4 Modeling a new processor: ARM

In order to model a new processor, it is necessary to extract the cost parameters for that processor. The cost parameters are determined for each target system (CPU, compiler) by using a set of sample benchmark programs. Each `if` or `assignment` statement which is contained in these functions has the same style as one of the statements generated from a **TEST** or an **ASSIGN** node in the S-graph. The analysis can be done either with a profiler or an assembly level code analysis tool for the target CPU. If neither a profiler nor an analysis tool are available, it is also possible to specify each cost parameter according to the architecture manual of the target CPU, by predicting the code generation strategy of the target compiler.

We have recently used the *macro-modeling* approach described earlier to estimate the execution speed and code size on the ARM7TDMI and the ARM920T processors [23]. The latter processor has a 4-stage pipeline, as well as a 16Kb data cache and a 16Kb instruction cache. We added these benchmarked processors to the Polis library that consists of the MIPS R3000, and the Motorola 68332, and 68HC11 processors[15]. Our estimated results were validated with the ARMulator, and for the ARM7 the estimates were off by a conservative (pessimistic) 10%-20%. For the ARM9 architecture, with caching and pipelining

[15] Has been made available for release

effects, the estimates were off by 20%-30%. This experience led us to several interesting on-going research directions to address the shortcomings of the estimation method as discussed next.

7.10.5 Limitations of the Software Estimation Technique and Future Improvements

Clearly a more accurate analysis technique (e.g. for handling pipeline and cache effects), for example based on a cycle-accurate model of the processor [97][66], is needed to validate the *final implementation.* But both early synthesis result evaluation as well as the architecture exploration phase can be carried out much faster, as long as the accuracy of estimation is acceptable for the task at hand. I am exploring opportunities for *statically* modeling caching and pipeline effects by trying to capture the essence of some of the optimization techniques used to improve performance/code size in pipelined and cached machines. Our aim is to understand *why* and *how* these approaches affect performance, size, and power. If this can be accomplished then we can not only optimize for a target architecture but also have a better handle on cost with rapid heuristics based on the optimization techniques thus developing a *synthesis for predictability* approach [89] where final output can be estimated quickly *statically* and *accurately.* These CDFG[16] level optimization techniques include:

- **TEST** *node rescheduling*: this is currently limited to variable sifting as described in Section 7.2.1, and does not exploit re-ordering based on the common path. If the latter is done, not only is the code *better*; this restructuring has been shown by Castelluccia

[16] S-graph

and Dabbous in [27] as well as the references in that work, to improve code *cache layout*. The idea is to order the most frequently executed branches to be consistent with compiler predictions and the final code output that determines the locality of this code in the cache. In fact this optimization, by the same argument, improves pipelining as the pipeline does not have to be flushed after every failed prediction.

- *Function call inlining*: where we estimate using simple heuristics whether a function call will be inlined or not and this can save considerably by which measure the estimation is off (possibly an order of magnitude in estimation error). A simple heuristic that compilers use was introduced in Proposition 6.5.1.

- *Sharing of computations*: where we can perform some early analysis (possibly using the optimization engines) to determine the likelihood that an operation will be "inlined" (i.e. computed again) or "shared" (i.e. used from a register). We can simply for example see what the high level optimizations discussed in Chapter 5 had *recommended* or *hinted* must be done, and tune our estimate to that likely scenario.

That said, let me re-iterate the basic idea. The macro-modeling technique by Suzuki ([110]) can be improved on slightly with statistical techniques but it is close to the best that you can do at the high level *statically* as opposed to compilation-based techniques as in [71]. The crucial improvement would then be to optimize program behavior so that it is close to the "ideal" predicted by this method so as the prediction will not be too off, and remain useful in the early design phases. Another simple improvement to the macro-modeling technique would be expanding the parameters to consist of *set of values* that capture different aspects for example with inlining, with/without a cache miss, etc...

as in the work of [72]. We can also solicit user input (profiling is not an option for rapid estimation, while early "guidance" from the user is) on probabilities of test conditions in order to identify the common path in the CDFG, in the same spirit of what I did for *operation motion* in Chapter 5 but where we compute frequency of path execution in the CDFG that is identify the *common case* as opposed to the visit probability of the states (collection of paths) in the AFFG.

7.11 Hardware Estimation

As described earlier CFSMs implemented in hardware are currently synthesized assuming that each transition requires exactly one clock cycle, by using classical RTL and logic synthesis techniques. Estimation is performed on the BLIF (mapped and un-mapped) network representation using structural estimation techniques on the multi-level logic network (see [102] for details).

Part III

Overall Co-design Flow, Results, Conclusions, and the Future

Chapter 8

Function/Architecture Optimization and Co-design Flow

In the previous Chapters, I layed the foundation for the formal function/architecture optimization and co-design methodology. I presented a suitable abstract representation on which function/architecture trade-off is performed through refinement of the function, and abstraction of the architecture in Chapter 3. I then focused on discussing optimization of both control and data and how I introduced it into the co-design process as a crucial player in the analysis and redundancy removal of the information embodied in the function as it is constrained by the application and architecture demands. I overviewed architecture-independent optimizations in Chapter 4, architecture/function trade-offs in Chapter 5, and then mapping of the function onto the architecture in Chapter 6. Integration with synthesis was presented in Chapter 7. In this Chapter, I would like to give the

overall picture[1], and the complete *Vertical Integration* of the overall co-design process using the Function/Architecture Organization. The concept is demonstrated in Figure 8.1; the co-design methodology I develop in this work aims to assist designers struggling with application demands, and the increasing design complexity starting from the highest specification levels down to the final implementation.

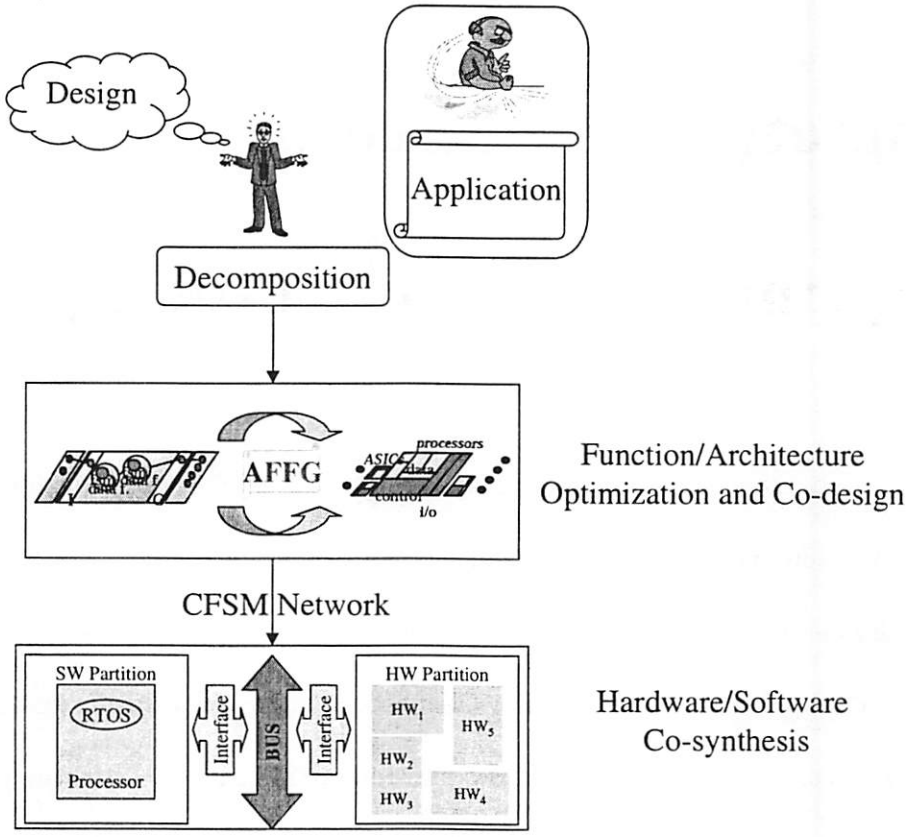


Figure 8.1: Overall Co-design Process: Concept Flow

[1] As promised in the *Dissertation Contribution* Section of Chapter 1

8.1 *Inter*-CFSM Optimizations

I have focused so far on describing *task level* optimizations where a CFSM task is represented by the *Architecture Function Flow Graph (AFFG)*, or *intra*-CFSM optimizations that are independent of the task's implementation in hardware or software. In this Section I motivate and then build a method for performing a flavor of *inter*-CFSM optimizations. Ideally, it would be most appealing if the lattice-theoretic information analysis framework could be generalized and extended to capture the inter-CFSM processing network, followed by redundancy removal on the network as a whole. Realistically, however, this is a hard problem at the abstract CFSM interconnection level because performing such an analysis involves studying *all the possible behaviors of the network* (see Abstract Interpretation of [34]). The analysis would become manageable (in the sense that suitable heuristics can be devised) only if the implementation attributes of the network tasks are fixed. In the case of an *all HW* concurrent implementation the problem is equivalent to *sequential optimization* known in the synchronous hardware domain. A DFA framework would be better suited in the latter case of hardware inter-EFSM optimization, in my opinion, because of the DFA framework's expressiveness and compactness (Tabbara in [111] is evaluating such an approach). If an *all SW* implementation is chosen, and a task execution schedule is known or can be derived statically then the analysis and optimization can also be performed (as in the work of Murthy [86] in SDF).

Our goal here has always been to perform *safe* optimizations *efficiently* in a *co-design* environment. *Efficiency* means that the designer can perform trade-off analysis and optimization quickly given a function and a set of constraints, so carrying out a time

consuming optimization cannot be part of the *co-design* steps in the framework intended for architectural exploration and evaluation, but only at the final implementation steps (preceded by Hardware/Software partitioning approaches such as [125] and scheduling). *Safety* has meant throughout this work the preservation of:

1. The CFSM[2] task I/O semantics, and

2. the CFSM network MOC semantics.

In order to preserve the CFSM semantics, and be able to perform an adequate *optimization of the network as a whole*, I therefore propose the following *inter-CFSM optimization approach.*

Proposition 8.1.1 CFSM Network Optimization Approach *In order to improve the CFSM network performance (size and/or speed), we iterate on* functional decomposition. *By exploring different* task boundary configurations *we can uncover a* better *network organization where intra-CFSM optimizations can be leveraged more effectively, thus optimizing the network performance as a whole.*

This is the general idea; I make this statement more concrete in the next Section where the functional decomposition strategy is presented.

8.2 Functional Decomposition

Because of the limitations on the scalability of synthesis engines (i.e. size of the low level CDFG representation, for example the S-graph in Polis may explode for large

[2]EFSM in general, CFSM in Polis

tasks), users typically decompose a system into an initial *modular* description. This is the *initial* functional decomposition we have been assuming so far where each such module is represented as a CFSM task. However, the price of communication between CFSMs may increase significantly (RTOS scheduling and communication overhead [6], as well as interfaces among HW components) so this need not be the final functional decomposition. My goal here is to improve this initial functional decomposition. The main idea behind my procedure is that the larger the size of the task the more we can benefit from 2 aspects:

- The communication overhead between tasks is reduced (scheduling/interfaces), and
- the task optimizations will perform "better" as there are more uncovered opportunities for optimizations that exploit the now broken boundary i.e. some of the *inter*-task are now *intra*-task chances for enhancing the performance.

So what I propose to do is to ***break the CFSM boundaries*** by ***clustering*** the various tasks. Since the CFSM Model of Computation (see [5]) guarantees that the composition of CFSMs has the same exact behavior as the individual components together in the system, then this clustering is *safe* i.e. it preserves the system semantics. Furthermore, the process of clustering two *composed* CFSMs *currently* involves building the *product machine* for the two EFSMs[3]. This is, of course, now a *synchronous composition*, and user input and guidance may be required to make sure that the process does not over restrict the sets of behaviors that the user had in mind since the asynchronous CFSM composition includes a larger set of *valid behaviors* than the now combined single CFSM. The functional

[3]In the future, it might be possible to move computations between these 2 asynchronously composed CFSMs without building the synchronous product machine

decomposition procedure[4] is shown in the procedure 8.2.1 flow chart below. This is the *conceptual* approach where: starting from an initial functional decomposition at a *modular* granularity, we perform **greedy pairwise clustering based on a closeness metric** [5] where closeness [122] is measured based on *architectural communication cost estimates.* We then run the FFG/AFFG optimization flow and compute an estimate of runtime and code size. The process is iterated, thus building larger and larger tasks, until a previously set (code, area) size threshold is reached which could for example be proportional to the size of the instruction cache in the architecture or a limit on the HW component area, or we reach *diminishing returns* in terms of runtime; this will eventually happen since typically optimizations reach their usefulness limit when the control becomes too complex.

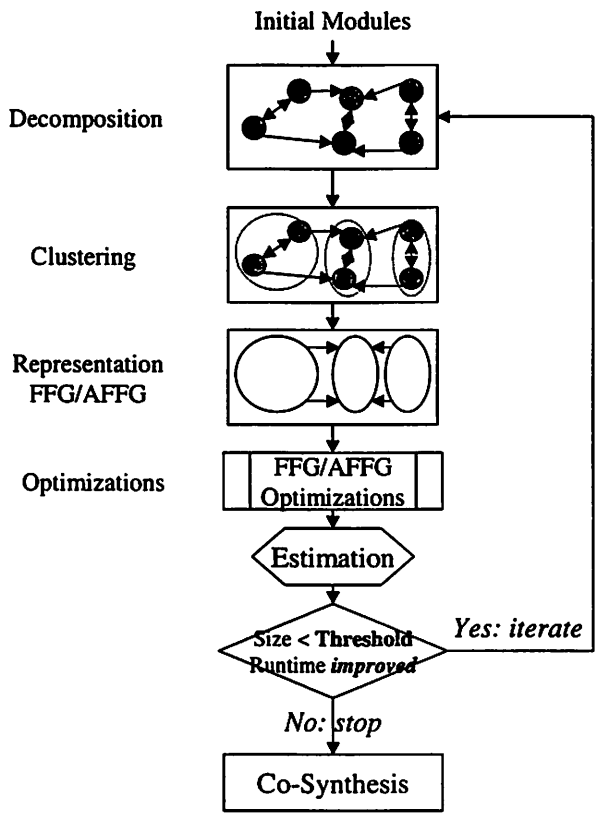
As I mentioned this can only be a "hypothetical" approach since the complexity of clustering is $O(m^2)$ and the outer iteration is $O(m)$ where m is the number of modules or tasks at the initial given granularity. While m is not typically large, the overhead spent in optimizing each task is not trivial and will get costly as the tasks in the partition get larger. A more practical approach can be conceived where both size and *expected* runtime are modeled as functions that *constrain* the closeness metric to begin with, so as *not* to incur the overhead of actually calling the FFG and AFFG optimization algorithms. These functions can be obtained *heuristically* based on a *structural analysis* of the FFG[6], for example to name some of the parameters: node count, average node size (i.e. operation count within node), average in-degree, out-degree, and graph depth (see Chapter 4); the only complexity

[4] To clarify, this is a *clustering* approach so as to get us a better overall functional decomposition.

[5] Alternatively (with some extra complexity) we can find the strongly connected components, or use Kernighan/Lin [60] k-way partitioning.

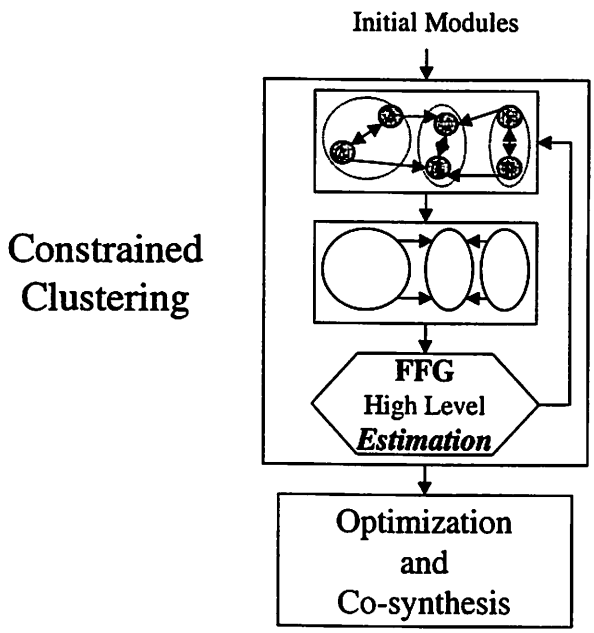
[6] A simple function for size could be the total of all operations in the FFG, and for runtime the number of operations in the longest control path.

Procedure 8.2.1 Conceptual Functional Decomposition Improvement Using Clustering



involved is that of building the FFG at the front-end, data is collected while building the graph. We therefore perform a ***constrained* greedy pairwise clustering based on a closeness metric** which is also polynomial in the number of modules m (i.e. $O(m^2)$, and $O(m^3)$ for overall flow) but there is no intermediate optimization step cost (again m is typically small). The practical procedure, shown in the flow chart below, serves to *improve* the functional decomposition starting point for our optimizations, and to benefit from the *inter*-CFSM optimization potential.

Procedure 8.2.2 Practical Functional Decomposition Improvement Using Clustering



8.3 A Comprehensive Function Architecture Co-design and Optimization Flow

Pulling together all the methodology and practical algorithms and heuristics, the concrete implementation of the concepts outlined in Figure 8.1 is shown in Figure 8.2. After the user inputs the initial functional decomposition, constrained clustering using high level estimation on the FFG identifies a better functional decomposition for the system tasks that is more suited for optimization. This is then followed by the architecture-independent, and architecture-dependent optimization phases, and then HW/SW, RTOS, and interface co-synthesis.

8.4 Software Implementation

This brief Section is included for *Research Transfer* purposes and can be safely skipped by the un-interested reader. Further documentation can be found in the updated Polis manual [94]. The function/architecture optimization and co-design methodology has been integrated into the Polis co-design framework [94]. The FFG and AFFG optimization algorithms have been implemented in four components, as Figure 8.2 shows, that operate on the FFG internal representation *successively refining* it in the top-down flow with *attributes*. I have chosen to expand on Polis's framework of *point tools* which permits ease of experimentation, and the building of user-defined *flows*, so CLIF (see Appendix A) is used for data interchange[7] among the different optimization and co-design engines. Internally

[7]Default in current toolset; alternatively, an API is available for the building of executable flows instead of the "shell"-based approach

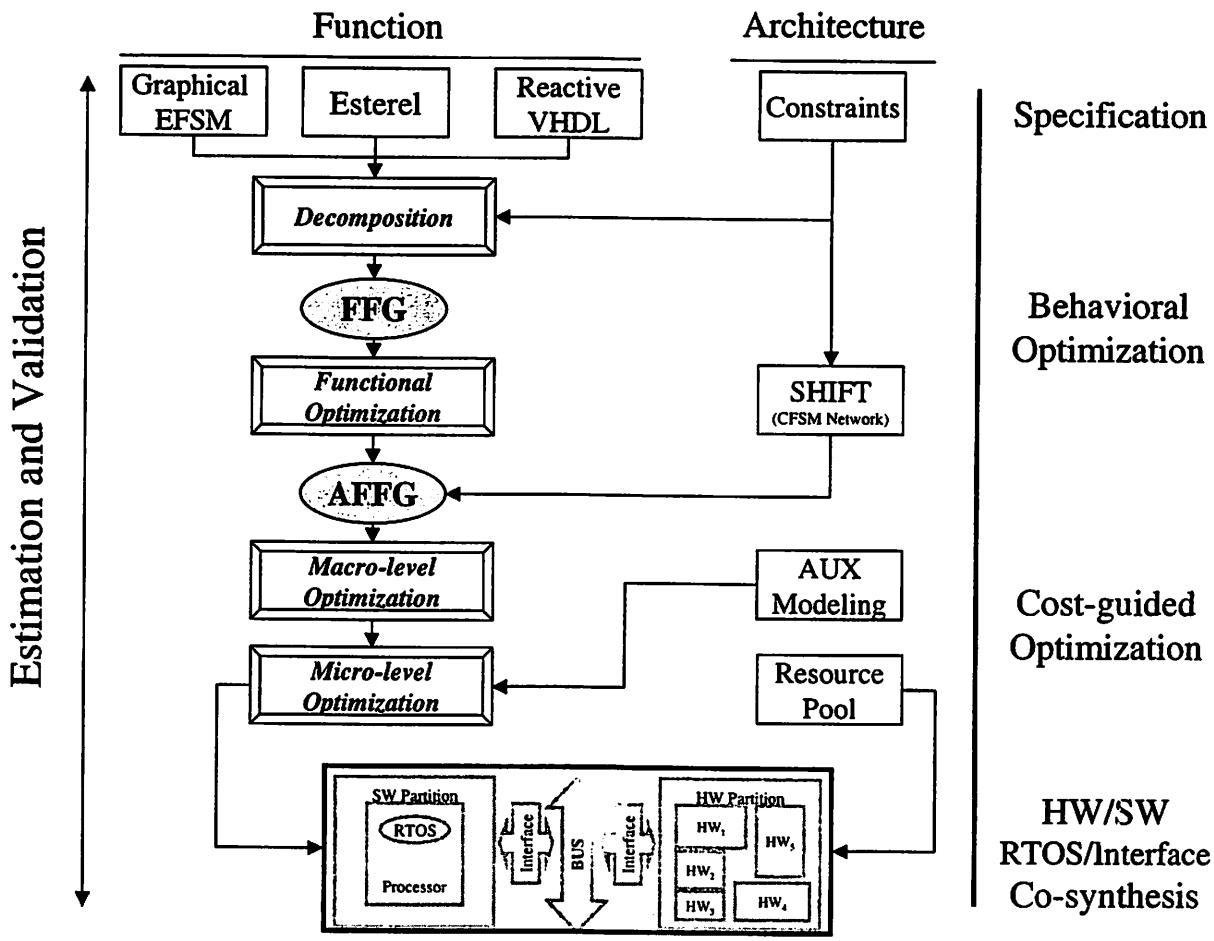


Figure 8.2: Overall Co-design Process: Concrete Flow

these components have a *default optimization flow* which can be modified *interactively* or through code modification of the default API call set. These components are:

1. Front-end: The Esterel compiler is used to build the EFSM from the Esterel input description. `oc2clif` augments the front-end analysis and builds the internal FFG representation.

2. Function Optimization: `clifopt` performs this step on the FFG representation.

3. Macro-level Optimizations: `clifmap` implements this refinement and optimization step on the AFFG representation; it assumes the CFSM network semantics, and the SHIFT macro-architecture, unless otherwise specified[8].

4. Micro-level Optimizations: `clif2shift` performs this step starting from the AFFG representation using the AUX/SHIFT micro-architectural constraints.

5. HW/SW, RTOS, and Interface Co-synthesis: using the Polis engines guided by the SHIFT/AUX, and user-specified *resource pool*[9] constraints as discussed earlier in Chapter 7.

The toolset is shown in Figure 8.3. As displayed in the Figure (and Figure 8.2), the tools can be integrated into any co-design environment, the architectural constraints that guide the toolset need only be changed. The FFG/AFFG design task representation on which the toolset operates has been implemented in C++ using an infrastructure of data types and access methods from the **Library for Efficient Data types and Algorithms**

[8]In which case user must modify the API, and point to the target architecture's own (user-defined) optimization methods.

[9]Polis architecture configuration file

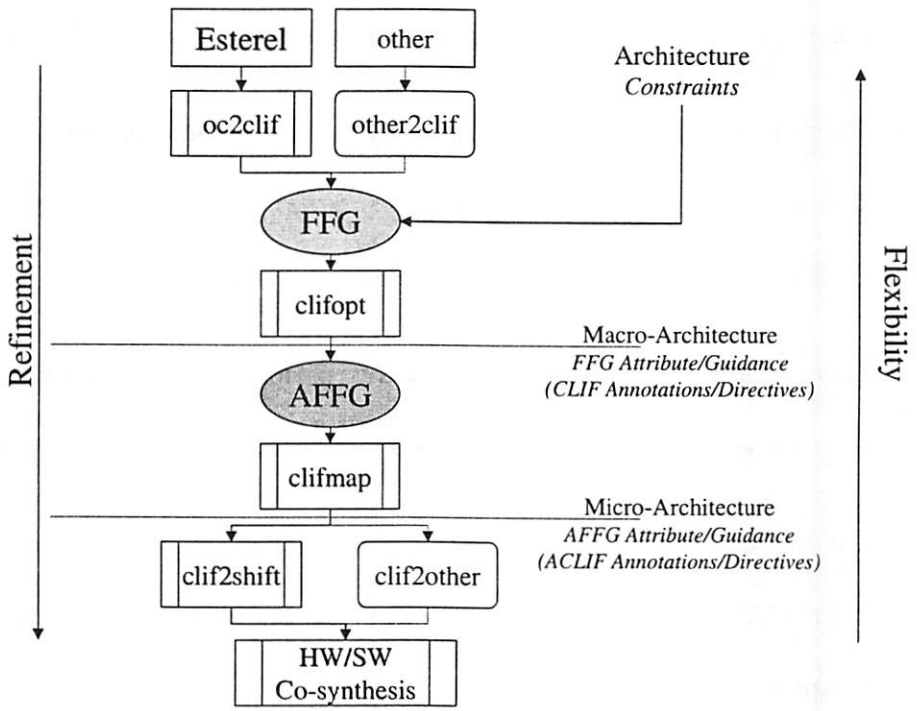


Figure 8.3: Overall Co-design Process: Toolset Flow

(LEDA)[10] in particular the [75]:

- *Parameterized Graph* data type: derived from directed graph data type realized by doubly linked lists of nodes and edges. Most operations take constant time, while iterators typically take $O(n+m)$, n being the current number of nodes and m the current number of edges.

- *Linear Lists* data type: realized by doubly linked linear lists; all operations used in the toolset on this data type take constant time except for *search* and *rank*[11] which take linear time ($O(n)$) in the number of elements n.

- *BFS*, and *DFS*: breadth first and depth first search algorithms that operate on the parameterized graph.

The *BitSet* data type is somewhat clumsy in LEDA[12] and therefore the `Gnu libg++`[13] Bit manipulation package has been used instead for convenience[14]. Other C++ (and C) data structures ([129]) were built by the author to augment the aforementioned ones, for example to store *attributes* in the AFFG. My software architecture design follows Object Oriented Programming (OOP) techniques (see the notable work of Booch in [24]) so my algorithmic contribution is preserved in the (well documented) C++/C code implementation in addition to this dissertation and the manual write-ups.

[10] LEDA Version 3.3 with `g++-2.7.2` on Alpha OSF and Sparc Solaris, also tested on the most recent (at time of this manuscript preparation) LEDA Version 4.0 with MSVC++-6.0 on the Windows NT platform

[11] Used to implement "hashing"

[12] Version 3.3 does not have one, while the dynamic integer set found in version 4.0 does not work correctly on the aforementioned Sparc and Alpha platforms

[13] Version 2.7.2

[14] BitSets can contain logically infinite sets of non-negative numbers in libg++ and have an optimal representation, and a wealth of operations.

Unfortunately, sound methodologies and processes that enable design re-use and component-based design, and permit developers to concentrate on creating algorithms and techniques that deal with the real issues (instead of the mundane) have still not taken hold in the software and hardware design communities yet. For best portability of the toolset code, and to relax the current dependence on LEDA[15], I am currently attempting to rebuild the aforementioned data type foundations in Java within the NexSIS effort [111] in conjunction with developing techniques[16] for SoC IP assembly (see Chapter 10).

[15]Research License
[16]Courtesy: Abdallah Tabbara, NexSIS Architect

Chapter 9

Synthesis Results

In this Chapter I report various results to support my claim that function/architecture optimization and co-design is indeed an improvement over other current day co-design approaches. I compare the improved and original co-design process using the Polis co-design tool; the results that compare the Polis methodology to other approaches reported by Balarin et. al. in [5] (such as the synchronous composition done in Esterel [16]) carries by transitivity since I use Polis here as the baseline[1]. I will, unless otherwise stated, present results for the *(A)FFG Tree Form* which, as I described in earlier Chapters, provides for the *best quality* output. The reader should also assume that *all* the aforementioned optimization techniques have been applied *except* Relaxed Operation Motion (ROM), unless otherwise stated. I will dedicate later in the Chapter several result tables and analyses for the latter, and also compare the *Tree* vs. the *Shared DAG* form in terms of output quality and running times, but report in the main result tables what the toolset default[2] co-design flow obtains.

[1] Of course, the reader should keep in mind that my approach can also be applied in other co-design tools thus improving their output quality as well.

[2] Tree Form with no ROM

The results have been collected by exercising several *representative applications* with typical input, and validating by trace comparison[3] with the *golden trace*; both obtained from the application designer.

9.1 A Communications Domain Application Example: An ATM Server

9.1.1 The Design

Let us consider a case study from the communication networks domain: an ATM server [26]. The target system is a server that performs support *control functions* for ATM Virtual Private Networks (VPN), like the control of the bandwidth of the outgoing flow, and a message discarding technique to avoid node congestion. Virtual private networks are used to interconnect LANs of multi-site users. An ATM backbone is used in order to provide efficient use of resources in a changing traffic scenario. The input of the system is a stream of ATM cells belonging to the set of active Virtual Channel Connections (VCC). Cells are buffered inside the switch. The buffer is divided into FIFO queues, one for each output Virtual Path Connection (VPC). Cells are forwarded to the proper FIFO according to the entries in the internal routing table as shown in Figure 9.1.

The ATM server performs the following functionalities:

- Statistical multiplexing of the incoming flows.
- Buffer management: the Message Selective Discarding (MSD) technique avoids node

[3]In addition to the *FFG Interpreter* (the FFG simulation engine), and the VHDL validation method described in Chapter 2

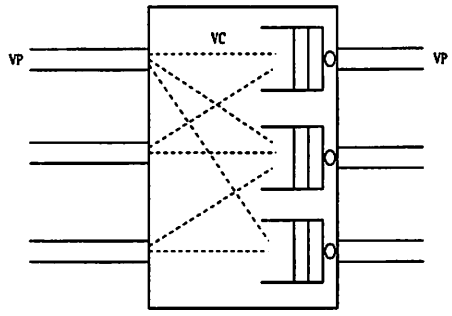


Figure 9.1: Operational View of the Buffer Inside the ATM Server

congestion by preserving the integrity of messages.

- Egress policing: the bandwidth of the outgoing flows is controlled by a Virtual Clock scheduling technique that provides fair bandwidth allocation among the queues.

In Figure 9.2 a high-level description of the functional blocks is given. The timing constraints of the system are tight. The processing of every incoming cell has to be done before the next cell arrives, i.e. within $2.72\mu s$, for a link rate of 155 Mbit/s.

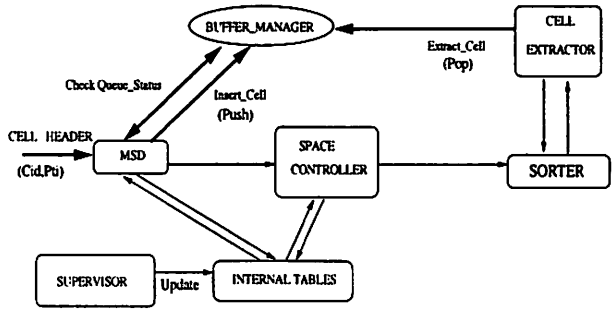


Figure 9.2: A High Level Description of the ATM Server

9.1.2 Modeling the Design

The system has been modeled as a network of 15 CFSMs in Polis. This is considered to be a medium to large design example; a very large design is typically composed of twice as many CFSMs. Esterel is used to describe the blocks, then the design is manually partitioned into hardware and software modules. VHDL co-simulation has been used to validate the system (using the method described briefly in Section 2.3), and evaluate the partitioning and the implementation alternatives [43]. After this partitioning step I have endeavored to apply the function/architecture optimization approach to see how to improve the size and performance of design tasks; you should recall that in the architecture-dependent stages I opted to *bias* optimizations towards *better runtime* (See *registering* at the macro-architectural level described in Chapter 5, and *inlining* computations at the micro-architectural level described in Chapter 6).

9.1.3 Synthesizing the Controller

I report here results for the controller synthesis and compare the *original* synthesis process in Polis, and my *improved* synthesis process that incorporates optimization and co-design performed on the (A)FFG design representation. Table 9.1 shows the results for SW synthesis of the various design tasks; The `memory` and `sub_sort` tasks have 2 instances each in the design (one instance only shown in Table). The results are *estimates* for *average runtime* and *code size* on the target platform using the *macro-modeling* method described earlier in Section 7.9; for the Alpha I have used the `pixie` instrumentation of the ATOM analysis Tool for *profiling* [108]. In detail, the SW platforms are:

- Alpha 21164 64-bit (400 MHz) processor: representing the high end 32-bit and 64-bit RISC processors such as those cited in the EEMBC [40] suite using GNU's gcc compiler **with the highest level of optimization**[4] since gcc has been ported to most of these embedded processors in the form of the Cygnus GnuPro compiler [35].

- ARM7 and ARM9: family of Harvard architecture RISC processors where the estimation is derived from benchmarks run on the `ARMulator` using the ARM Software Development Kit[5] (see Section 7.10.4).

- Motorola 68HC11: representing the 8-bit/16-bit data (4 MHz) widely used CISC micro-controllers where the estimation is derived from benchmarks run using the `Introl` compiler and debugger.

The results in the Table are ordered based on the CDFG node count improvement (from high to low). The *number* of nodes is roughly proportional to code size, in reality node *type* must also be taken into account, so estimation and profiling techniques must be applied as shown in the adjacent columns. The results show that for *most* cases we have a sizable improvement in *both* runtime, and code size. The improvement varies with the specifics of the target architecture but it can be clearly seen that the improvement can be **up to 25 %**. Tasks that have a considerable data computation portion reflect the most improvement as exemplified by `first_cell` the 4th task listed in Table 9.1 which displays an improvement on the order of *20 %* in both runtime and code size on the various architectures. The rest of the *top 5* tasks show an improvement of about 9 % in runtime, and twice that in code size except for `space_controller` where code size increases. Let

[4] `gcc -03`
[5] SDK v. 2.50

Task under Co-design	CDFG nodes	Alpha 21164 cycles	Alpha 21164 bytes	ARM7TDMI cycles	ARM7TDMI bytes	ARM920T cycles	ARM920T bytes	68HC11 cycles	68HC11 bytes
memory	17	134	1112	4114	520	2921	544	253	345
memory	12	122	856	4114	376	2921	396	244	257
% Improvement	**29**	**9**	**23**	**0**	**28**	**0**	**27**	**4**	**26**
space_controller	118	160	1928	586	1260	471	1320	817	951
space_controller	90	152	2156	524	1256	439	1312	800	923
% Improvement	**24**	**5**	**-11**	**11**	**0**	**7**	**1**	**2**	**3**
lqm_arbiter	128	142	1948	425	2288	402	2376	443	1578
lqm_arbiter	99	141	1564	405	1712	383	1800	404	1298
% Improvement	**23**	**1**	**20**	**5**	**25**	**5**	**24**	**9**	**18**
first_cell	137	260	3004	784	1608	682	1640	734	1112
first_cell	112	198	2368	688	1320	580	1352	588	898
% Improvement	**18**	**24**	**21**	**12**	**18**	**15**	**18**	**20**	**19**
extract_cell	100	109	1216	396	1112	304	1180	314	918
extract_cell	89	104	1144	343	992	283	1048	298	780
% Improvement	**11**	**5**	**6**	**6**	**5**	**7**	**11**	**5**	**15**
sub_sort	175	369	5008	770	2324	700	2352	800	1728
sub_sort	161	336	5360	734	2388	662	2420	734.5	1902
% Improvement	**8**	**9**	**-7**	**5**	**-3**	**5**	**-3**	**8**	**-9**
msd_technique	260	178	3796	639	3244	496	3372	572	2525
msd_technique	241	177	3448	587	2984	485	3100	572	2281
% Improvement	**7.3**	**1**	**9**	**8.1**	**8.0**	**2.2**	**8.1**	**0**	**9.7**
counter	27	92	740	371	500	317	516	274	317
counter	27	81	588	371	500	317	516	274	317
% Improvement	**0**	**12**	**21**	**0**	**0**	**0**	**0**	**0**	**0**
supervisor	90	201	2136	971	1236	677	1264	535	713
supervisor	90	191	2124	965	1228	671	1256	524	703
% Improvement	**0**	**5**	**1**	**1**	**1**	**1**	**1**	**2**	**1**
sorter	17	70	516	356	300	264	316	206	189
sorter	17	69	492	356	300	264	316	206	189
% Improvement	**0**	**1**	**5**	**0**	**0**	**0**	**0**	**0**	**0**
arbiter_sc	22	49	412	222	396	188	424	198	318
arbiter_sc	22	49	412	222	396	188	424	198	318
% Improvement	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
arbiter_sorter	22	60	464	233	412	200	440	208	324
arbiter_sorter	22	61	464	233	412	200	440	208	324
% Improvement	**0**	**-2**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
collision_detector	20	130	904	392	692	364	748	395	613
collision_detector	20	132	904	392	692	364	748	395	613
% Improvement	**0**	**-2**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Table 9.1: Software Synthesis Results for ATM Server Co-Design

us now examine these tasks that do not have a uniform improvement in both runtime and size. Task `space_controller` and to a lesser degree the `sub_sort` task *trade-off* code size for speed. In fact, examining the synthesized C code confirms my conjecture and shows that indeed the runtime improvement is the result of using a pre-computed value in the optimized output as opposed to performing the computation again[6]. The *bottom 4* tasks do not show any improvement at all, and the synthesized code is identical; the *negative* improvement in the profiled output[7] can be explained by the fact that I am exercising the *application as a whole* so it is quite conceivable that *resource pressure* (such as register pressure and subsequent "spill" into memory) would account for such minor *jitter*.

We can also estimate the **overall system improvement** in the following fashion:

- Application size: consists of two components, the RTOS size and the *cumulative* task size, so we can use the sum of the RTOS size (see Section 7.8 for an explanation of why and how the RTOS is improved in my co-design approach) and the total task size as a measure of the overall code size improvement.

- Application speed: can be estimated for a specific scheduling policy. For example I will present shortly results for a *round robin* scheduling scheme. If we neglect the RTOS improvement in terms of context switch and task calls (measured for a simple application in Section 7.7.1), the overall speed improvement can be obtained for a cyclic schedule with no repetition by adding up all the speed improvements in the tasks themselves.

[6] I can tell the interested reader that *copy propagation* was the key that enabled this computation re-use.
[7] Assuming *perfect* profiling even though a 1-2 % error is reasonable

Co-design Process	RTOS Size (Alpha bytes)	Overall Code Size (Alpha bytes)	Overall Speed (Alpha cycles)
Original	39652	72319	1954
Improved	36072	58828	1813
% Improvement	**9**	**19**	**7**

Table 9.2: Estimated Overall Improvement for the ATM Server Co-design

Table 9.2 shows the RTOS size improvement to demonstrate how the RTOS size is indeed improved, and then gives an estimate of the overall size and speed improvement of the ATM server application assuming a round-robin scheme[8]. The Table shows an RTOS size improvement of 9 %, an overall improvement in code size of 19 % and a speed improvement of 7 %. Of course this application is control dominated and we would expect more improvement the more abundant data computations are; these suspicions will be confirmed shortly.

I think it is worthwhile for me here to re-emphasize a point I had made in the "proof-of-concept" example of Chapter 3; a point that is shown clearly in this Chapter in the results of Tables 9.1 and 9.2. An optimizing target compiler *does not* uncover all the potential optimizations when applied on the low-level CDFG representation, otherwise there would've been *no* improvement results in the aforementioned Tables. Since the optimization approach introduced in this work does indeed result in a significant[9] then there is a tremendous value in applying "compiler-like" optimization techniques at the *high representation level* (i.e. FFG/AFFG), and even more so if such techniques are *guided* by constraints that are solicited from the user, and others derived from the target architecture. I present results on guided optimizations in Section 9.3.2, let us now examine the effect of optimizations on the synthesized hardware.

[8]This is the scheme used in the profiling of Table 9.1 in order to be *fair* to the tasks.

[9]Again, even more significant in data-rich examples as we will see shortly

For *hardware synthesis* I report results on 3 representative tasks from the ATM server design: `first_cell`, `lqm_arbiter`, and `sub_sort` ordered based on improvement in BLIF network node count from least to best. The results are measured from an *unmapped implementation-independent optimized* (using script.rugged from SIS [102]) BLIF network and shown in Table 9.3. Results clearly show expected speed improvements when the network is mapped to a target library as reflected by the literal count in the Sum of Products (SOP), and size (area) improvement as reflected by the node and latch counts shown. Since the HW optimizations at the low level examine the circuit as a whole, they in fact manage to look at much more global optimizations than their counterparts in SW which get bogged down by the task call (invocation) boundaries as I discussed early in this dissertation in Chapter 3. The downside, of course, is that these optimizations are heuristics, not algorithms, and also are performed on a very low granularity so they run out of steam soon as the tasks become larger. The largest task that can be handled in acceptable "compilation" time consists of about 500 nodes, I had to stop the unoptimized version of *sub_sort* after over 10 minutes of CPU time, and report the numbers *before* the low level HW optimization[10]. The reader should also note that the *outputs* of these tasks, and *inputs* from other tasks to these tasks will also be optimized by the *bit-width optimization* discussed earlier as part of the micro-architecture optimizations in Chapter 6 (not shown in Table).

[10] script.rugged of SIS

HW Task	BLIF nodes	BLIF latches	Literals (SOP)
first_cell	278	160	1402
first_cell	275	153	1317
% Improvement	**1**	**4**	**6**
lqm_arbiter	188	152	789
lqm_arbiter	134	107	593
% Improvement	**29**	**30**	**25**
sub_sort	885	226	21408
sub_sort	200	137	850
% Improvement	**77**	**39**	**96**

Table 9.3: Hardware Synthesis Results for ATM Server Co-design

9.2 An Automotive Dashboard Controller

Before turning our attention to representative data rich examples, let us consider another control dominated design. The application is a simplified car dashboard controller from the automotive domain adapted[11] from the one described in [5] and distributed with the Polis co-design tool. The system depicted in Figure 9.3 is modeled in a hierarchical fashion. There are five computation chains: the speedometer, the odometer, the tachometer, the fuel and water, and the belt controller. Each computation chain is a network of CFSMs.

This is a relatively large design example composed of 30 CFSMs; and a larger number of computations than the ATM sever presented in the previous section. Table 9.4 displays the original and improved co-design process[12] estimated SW synthesis results on the Polis benchmarked architectures for a few design tasks. We can see the same order of improvement, about 10-30 % in runtime and code size, as that found in ATM (slightly better

[11] I simplified the AUX by removing the bit sizing done manually (within the AUX module instantiation) in the distributed example with Polis [94], and moved constant declarations to inside the modules in order to emphasize the ability of the improved process to automatically uncover constants, and size the bit fields of internal/output variables as I described in Chapter 6.

[12] Ordered by improvement in CDFG node count

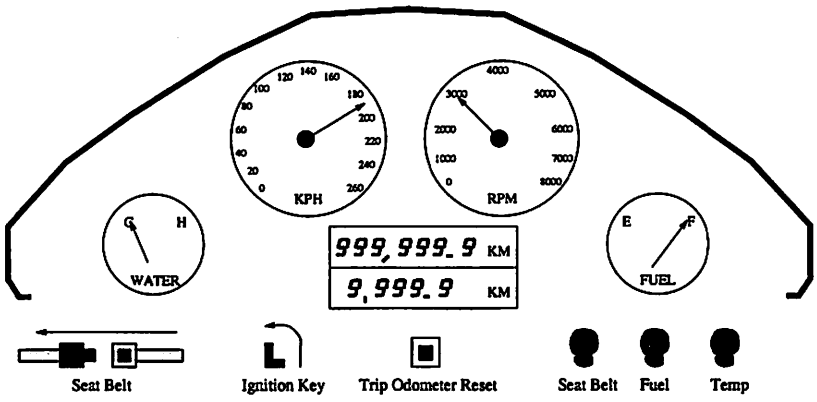


Figure 9.3: Automotive Dashboard Controller (from [5])

results because of the increased number of computations and constant manipulations); this is typical for control designs that have some computations. My observations i.e. RTOS size, and overall size and performance, done in the previous Section carry over to this example[13] where the **size of the RTOS is reduced by over 15 %.** These observations indeed apply to *typical control-dominated applications.*

9.3 Results on *Data-rich* Control Designs

In this section I use some benchmarks from the literature to demonstrate the effectiveness of the optimization and synthesis approach in the quality of the output. I intend to demonstrate on these examples the benefits of incorporating optimization into the co-design process of control-dominated designs that have "intelligence" embodied in a significant data computation portion.

[13]Size of tasks cannot be neglected in this rather large example, also improvement in context switching must be taken into account.

Task under Co-design	CDFG nodes	ARM7TDMI cycles	ARM7TDMI bytes	ARM920T cycles	ARM920T bytes	68HC11 cycles	68HC11 bytes
alarm_compare	30	309	484	284	500	278	314
alarm_compare	12	262	296	234	312	196	190
% Improvement	**60**	**15**	**39**	**18**	**25**	**30**	**39**
speed_cross_display	200	1097	4488	1055	4504	7558	5137
speed_cross_display	124	1016	3536	972	3552	6784	4037
% Improvement	**38**	**7**	**21**	**8**	**21**	**10**	**21**
belt	73	422	1236	363	1276	482	850
belt	49	361	948	298	988	328	676
% Improvement	**33**	**15**	**23**	**18**	**23**	**32**	**21**
engine_cross_display	201	828	3400	784	3416	4183	3361
engine_cross_display	137	782	2584	738	2600	3734	2387
% Improvement	**32**	**6**	**24**	**6**	**24**	**11**	**29**
debounce	33	368	628	349	644	309	350
debounce	25	327	512	304	528	276	307
% Improvement	**24**	**11**	**19**	**13**	**18**	**11**	**12**
speed_count_pulses	26	347	448	289	468	237	276
speed_count_pulses	20	336	380	277	400	235	251
% Improvement	**23**	**3**	**15**	**4**	**15**	**1**	**9**
measure_period	44	383	736	324	768	319	506
measure_period	39	346	672	285	704	288	483
% Improvement	**11**	**10**	**9**	**12**	**8**	**10**	**5**
odo_count_pulses	44	487	828	438	848	408	496
odo_count_pulses	41	465	764	414	784	395	481
% Improvement	**7**	**5**	**8**	**6**	**8**	**3**	**3**

Table 9.4: Software Synthesis Results for Car Dashboard Controller Co-design

9.3.1 Benchmarks from the Software Domain

I present here results on two benchmarks from the software domain: Quick Sort[14] *fragment* example used by Aho in [2], and Insertion Sort[15] *fragment* from [129]. Both fragments are shown in Figure 9.4 where l and r are the left and right boundaries of the array to be sorted. I have adapted the examples to be reactive by periodically reading from, and writing to, the external environment. These examples have been used quite extensively in the software optimization domain, and I hope they will serve here to put my optimization and co-design approach in perspective.

```
i = l; j = r; v = a[r];
while(1)
{
        do i = i + 1; while (a[i] < v);
        do j = j - 1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
}
x = a[i]; a[i] = a[r]; a[r] = x;
```

Quick Sort Fragment

```
for (i = l+1; i <= r; i++)
{
        tmp = a[i];
        for (j = i; ((tmp < a[j-1]) && (j>l)); j--)
                a[j] = a[j-1];
        a[j] = tmp;
}
```

Insertion Sort Fragment

Figure 9.4: Software Benchmarks

Synthesis results for a SW implementation are displayed in Table 9.5. It can be seen from the data that the representation and consequent optimizations are able to improve considerably final code quality in all cases. We can also see a case (`Insert` task) where the high level optimizations prevented design explosion. The latter, as we have seen from

[14] $O(n^2)$ algorithm
[15] $O(nlog(n))$ algorithm

Task under Co-design	CDFG nodes	Alpha 21164 cycles	Alpha 21164 bytes	ARM7TDMI cycles	ARM7TDMI bytes	ARM920T cycles	ARM920T bytes	68HC11 cycles	68HC11 bytes
quick_sort	327	439	7392	820	4108	833	4176	1165	2797
quick_sort	219	153	3188	434	2304	414	2364	592	1540
% Improvement	**33**	**65**	**57**	**47**	**44**	**50**	**43**	**49**	**45**
insertion_sort	SHIFT explodes								
insertion_sort	246	142	3444	529	2732	488	2780	516	1761
% Improvement	**unlimited**								
With ROM *earliest possible placement* Further Optimization									
quick_sort	219	153	3188	434	2304	414	2364	592	1540
quick_sort	194	195	3000	484	1876	415	1916	554	1087
% Improvement	**11**	**-22**	**6**	**-10**	**19**	**0**	**19**	**6**	**29**
With ROM *lifetime minimization* Further Optimization									
quick_sort	219	153	3188	434	2304	414	2364	592	1540
quick_sort	191	195	2976	432	1856	414	1896	549	1086
% Improvement	**13**	**-22**	**7**	**1**	**19**	**0**	**20**	**7**	**29**

Table 9.5: Software Synthesis Results for Benchmarks Co-design

the results of Table 9.3, is a quite common occurrence in hardware synthesis because of the smaller granularity of the low level representation. The size of the RTOS (on Alpha platform) for the Quick Sort design is reduced by **45 %**. The results are pretty impressive, and serve to show that the conventional co-design flow is unable to handle the optimization opportunities available in the benchmarks because of the presence of a considerable data computation portion, while the improved co-design flow comprised of both data flow and control optimizations is able to manage quite well.

The second part of Table 9.5 shows the result of running ROM with *earliest possible placement* additional optimization onto the earlier result of co-design. The data shows that clearly register pressure can be felt on architectures that are not register rich particularly the Alpha and the ARM7; this is less of an issue for the ARM9, and almost a non-issue for the

CISC 68HC11. The task code size is improved significantly[16] by about 10-20 % however the additional register have a "side-effect" cost in the RTOS buffers; **the RTOS (not shown) grows by 7 %** and this tempers the overall benefit. Since the task size dominates over the RTOS size in this example, overall the earliest possible placement does have its benefits, the additional registers (beyond what the macro-architectural analysis recommended initially) seem to help slightly in terms of code size no matter what the target architecture[17]. The third part of the Table displays the data for attempting to use ROM but with *lifetime minimization* to alleviate the register pressure problem. Clearly, the situation is better when the lifetime of the registers is minimized; but we see it does not radically change the problem and rather improves the result for register rich RISC architectures, and for the code compact CISC. The careful reader should suspect that the profiled result is telling us something, and the picture for the other architectures (ARM, and 68HC11) is not as rosy as the estimation[18] paints. The reader is correct; *profiling* gives a measure of output cost *based on frequency of execution* of the various branches in the reactive task. Creating additional registers is bound to reach the point of limited return and in fact, as we see from the above results, start to be more expensive if ROM is not *guided.* I address this issue in the next Section by revisiting the Reactive Knoop example I introduced in Chapter 5, and evaluating the *cost-guided* version of ROM.

Before concluding this section I would like topresent some data to compare the average execution times of the original and improved co-design process where the latter uses

[16]In fact, computations (i.e. sub-circuits in SHIFT) are reduced by 40 % (not shown explicitly in Table)

[17]Speaking generally, of course, if a certain task in on the "critical path" (for example is repeated quite often in the schedule) then ROM is ideal for optimizing such a task, and improve overall performance on candidate architectures, and code size significantly.

[18]which averages `min` and `max` of runtime

the (A)FFG Tree or the Shared DAG form. The first two rows in Table 9.6 compare the time it takes to build the SHIFT representation in the improved process starting from the designer input including:

- The front-end analysis and building of the FFG,
- the FFG functional level optimization,
- the AFFG macro-architectural level optimizations,
- the AFFG micro-architectural level optimizations, and finally
- the mapping of the AFFG onto SHIFT

and the time it takes the original Polis flow from the front-end analysis to the immediate building of SHIFT. The Table uses the CDFG node count as a rough estimate of the output quality, and the Alpha cycles and bytes as a more accurate measure; the lower the numbers the better the expected output. The percentage columns compares the data of the improved (Tree, or DAG form) flow to the original. The data is for the Quick Sort benchmark because it is quite large and rich with optimization opportunities. Without a doubt, the results show that the larger the size of the input the slower the low level CDFG build and optimization algorithms are. Building the CDFG is proportional to n^2 in the number of the FFG nodes n, while the optimizations take *constant time on average* (see Chapter 4) and only take $O(n^2)$ in the worst case. Since most of the time is spent (as the data shows) at the low level, the optimizations make a lot of sense at the high level; they speed up the process tremendously and result in a substantial benefit in output quality[19]. It is interesting to

[19]This is truly a win-win situation!

Co-design Steps	Original Flow	Improved (Tree)	% Improv.	Improved (Shared DAG)	% Improv.	DAG vs. Tree
SHIFT build (sec)	0.24	3.56	**-93.3**	2.66	**-91.0**	*25.3*
CDFG Synthesis (sec)	8.12	1.00	**87.7**	5.98	**26.4**	*-83.3*
Overall Time (sec)	8.36	4.56	**45.5**	8.64	**-3.2**	*-47.2*
Output Quality Comparison						
CDFG (node count)	327	219	**33**	240	**27**	*-9*
Alpha (bytes)	24944	13256	**47**	13488	**46**	*-1.7*
Alpha (cycles)	439	153	**65**	158	**64**	*-3*

Table 9.6: Time and Quality Comparison for the Quick Sort Benchmark Co-design

examine the comparison in the Table of the running time and quality of output between the FFG Tree and DAG representation forms. For this large data-rich example the Tree elaborate form is a clear victor in the quality and the running time because of the dominance of the low level CDFG synthesis on the running time. It should be evident to the reader that these comparison results apply only to such examples and not small and/or mostly control examples[20]. In this latter case, the situation is typically reversed where optimization analysis cost is more dominant and using the DAG form starts to make sense.

So, to sum up this discussion, we again have a "continuum[21]" of trade-offs, and knowledge of the target application can *adapt the co-design flow* to be as best as possible: the DAG form should be used if there is a very limited optimization potential to maximize use of the designer's time; if on the other hand quality is of utmost importance, and the

[20] Of course, the large and rich in data examples are what I am targeting, hence the use of the Quick Sort example

[21] I use the word rather loosely here, of course, we end up discretizing this *continuum* into several refinement of function (abstraction of architecture) levels

goal can indeed be attained by a more elaborate analysis then the Tree form should be used. The reader may be wondering why I care about runtime to begin with, since it seems my goal is to improve the output quality; and if that's the case an analysis and optimization overhead is acceptable. To that reader I say that her/his statement is not entirely correct; I have from the outset of this work stated that my goal is to improve *both* productivity and output quality. My aim is not to develop a *compilation* framework where optimization typically means extra tool runtime; my goal has always been to improve the *co-design process* as a whole. If the proposed function/architecture *co-design* framework is to succeed, it must permit **rapid and focused optimization** where *several architectural and functional alternatives* can be generated up to the CDFG for synthesis on which estimation can be performed. I stress here that my work addresses a co-design framework that is based on an exploration and evaluation of the co-design alternatives, albeit an *educated* exploration equipped with a formal and sound optimization and evaluation methodology.

9.3.2 Synthesis Result of the Reactive Knoop Example

In the previous Section, I did not perform *cost-guided* Relaxed Operation Motion (ROM), so here I pick up the *reactive version* of the Knoop example ([67]) I used in Chapter 5 and demonstrate through synthesis that there is indeed a potential benefit even on this small design example if architectural costs are known and guidance is used. I collect performance estimation results for the 68HC11 target architecture, and also the ARM7TDMI and ARM920T architectures (a CISC and a RISC target respectively). I report the results for a very valuable cost metric that can be used in guided ROM: the task **Worst-Case Execution time (WCET)**. The WCET corresponds to the longest computation path (i.e.

Co-design Method	CDFG nodes	68HC11 cycles	68HC11 bytes	ARM7TDMI cycles	ARM7TDMI bytes	ARM920T cycles	ARM920T bytes
w/out ROM	125	454	934	497	1276	377	1332
w/ ROM	126	432	904	474	1252	353	1308
% Improvement	**-1**	**5**	**3**	**5**	**2**	**6**	**2**

Table 9.7: Worst-Case Response Time Results of the Reactive Knoop Example

maximum) if there is no task pre-emption. WCET for a task is very useful for schedule validation to check if the system timing constraints are met, and also for system resource utilization analysis, and schedule optimization. I report the results for the Reactive Knoop example, then for the same example with guided ROM applied, using the static estimation method as shown in Table 9.7. The WCET path belongs to the **targeted** state S2 of the Reactive Knoop example, and the next to worst path corresponds to targeted state S8 where the improvement is about 4 % (not shown). Table 9.7 shows a benefit of about 6 % in response time. The number of nodes in the CDFG for synthesis increases because of the addition of registers once operation motion is performed. The improvement provided by ROM in *both* code size and runtime is more apparent in the register-rich ARM9 architecture.

Our goal in guided ROM in the previous paragraph was to improve the worst-case, let us for argument's sake try to see if the optimized result turns out to be useful on *average.* We can try to use a *typical* application profile that is *consistent with the initial given and/or derived frequency of inputs and tests measure* to collect data but we cannot conceivably get adequate profile coverage to the degree done in the static frequency execution analysis. A more productive method for collecting data, is to *statically* collect information from the synthesis DAG. This method involves building a low-level[22] Bayesian

[22]S-graph level

net, or Markov process, that uses the state visit frequencies computed earlier (which gets applied to the state condition of the DAG) and the input and test frequencies to determine the probabilities of execution of the *DAG nodes*[23]. A *weighted average cost* C_{ave} is then used in the estimation (see Section 7.10) [5]:

$$C_{ave} = \Sigma p_{ij}(C_t(node\_type\_of(i), variable\_type\_of(i)) + C_e(i,j)),$$

where p_{ij} is the conditional probability of going from node i to node j, given that node i is being executed, and $C_e(i,j)$ is the edge cost for edge e_{ij}.

In order to measure the *average task response time*, I use the frequency estimates of the state visits (see Chapter 5), map those onto the corresponding paths in the CDFG, and then measure the improvement using the weighted average approach. Each state corresponds to one or more paths in the CDFG[24] so I *average* the costs of the different paths[25] of each state assuming a *uniform* probability distribution for the conditionals in order to get a single *state execution path cost*. The computed data for the ARM920T is shown in Table 9.8, where the state visit probabilities are repeated for convenience. The reader should be able to see that the cost *with ROM* is lower for the later states and higher for the initial states after ROM is applied. I then add up the *visit probability weighted cost* of these representative paths for all the states (i.e. S1, S2, ..., S10), and report these numbers in Table 9.9.

So, in this example where I assumed equal probability of conditionals, ROM results in better expected worst-case performance (WCET) because of the improvements in *targeted* states S2, and S8, but because of the high probability of visit for state S1, on *average* the

[23] S-graph nodes, or better yet can be done on *basic blocks*
[24] i.e. the S-graph
[25] See Section 7.10

State	Probability	ARM920T Cost	Weighted Cost
w/out ROM			
S1	0.15	289	43.35
S2	0.07	303	21.21
S3	0.15	254	38.10
S4	0.07	264	18.48
S5	0.15	282	42.30
S6	0.046	280	12.88
S7	0.024	295	7.08
S8	0.09	302	27.18
S9	0.1	299	29.9
S10	0.15	247	37.05
w/ ROM			
S1	0.15	326	48.90
S2	0.07	291	20.37
S3	0.15	254	38.10
S4	0.07	263	18.41
S5	0.15	282	42.30
S6	0.046	280	12.88
S7	0.024	289	6.94
S8	0.09	297	26.73
S9	0.1	293	29.30
S10	0.15	247	37.05

Table 9.8: Runtime Cost of States and their Visit Frequencies

Co-design Method	ARM920T cycles
w/out ROM	278
w/ ROM	281
% Improvement	**-1**

Table 9.9: Average Response Time Results of Reactive Knoop Example

response time is almost the same and not improved[26] because the operations end up residing in costly state S1. In fact, for the given example and input distribution, the best solution found by guided ROM if we consider visit frequencies is the *original* example itself! Of course if there had been a *larger difference* in these visit frequencies then we would have also been able to find an expected improvement on average as well by using appropriately guided ROM.

[26]This is not surprising, since ROM is not properly guided; it's using worst-case information as opposed to state visit frequencies.

Chapter 10

Conclusions and Future Research Opportunities

I have presented my work on function/architecture optimization and co-design of embedded systems; a methodology that applies to both hardware and software synthesis for ASIC and ASIP targets, as well as for *programmable platforms*. With increasing market pressures including shrinking time to market and rising cost of layout masks, the importance of the latter architectural target cannot be over-emphasized. Figure 10.1 displays the envisioned required paradigm for programmable platforms[1] and how function/architecture co-design comes into the picture. The Figure is intended to show that we typically have an incompletely specified, possibly "vague" (non-deterministic if you wish) functional specification captured by the trapezoid. Similarly for the architecture I use an inverted trapezoid

[1] Figure created from key concepts by all of Edward Lee ("shadows"), Richard Newton ("components", "refinement and abstraction"), Alberto Sangiovanni-Vincentelli and Mark Pinto ("platforms"), Kurt Keutzer ("architectural exportation"), and Bassam Tabbara ("guided and constrained optimization and co-design").

to emphasize that several possible alternative architectures (parameterizations of a platform if you wish) may be suitable for realizing our intent. The specification casts a *shadow* on the architectural space in the refinement levels on *how* it can be realized. In turn the application architectural specification space sheds a *light* on *what* can be realized with the architecture as required by the application. The architecture can be more **powerful** than what the typically *restricted* functional specification can describe; in fact this is most often the case in architectures with large memories and ways to access these memories (essentially Turing Complete), where the architecture is definitely more powerful than what we would like to describe (or even can in a restricted language) at the functional level.

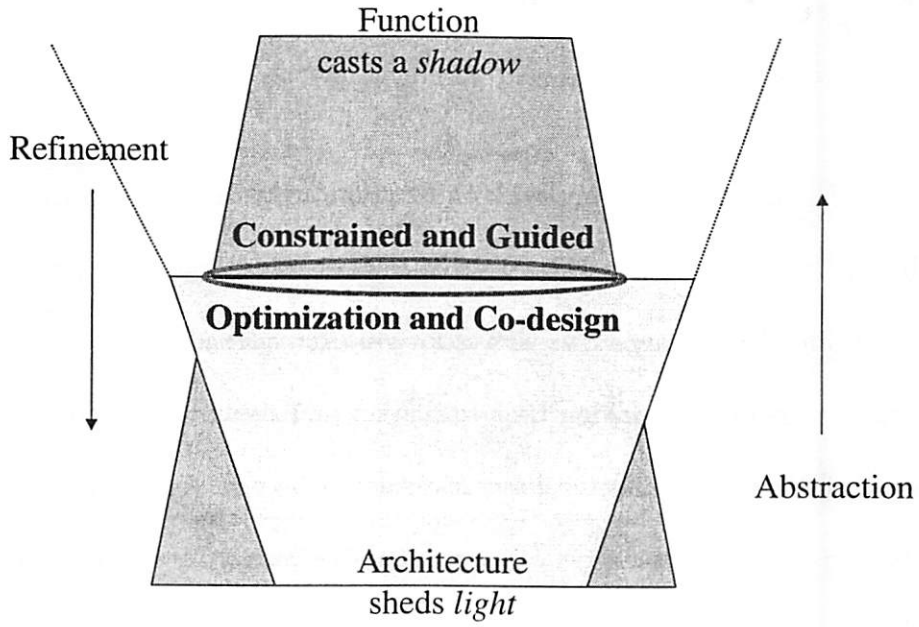


Figure 10.1: Function/Architecture Optimization and Co-design for Platforms

This light shows us what we can do in *guided and constrained function/architecture optimization and co-design.* This is what this work is all about. The "higher" we can get

the light to dispel the shadows the *better* our chances of finding a suitable and beneficial trade-off between function and architecture. This effort is limited only by our ability to "abstract" the architectural constraints adequately. This process is not simple, and it is the limiting factor that decides at which abstraction we can work. For example, if we limit our abstraction to gate delay timing we can only do gate sizing and driver insertion to fix our timing closure problems, if on the other hand we are able to specify "visit probabilities" on the EFSM states we can do co-design at the high level to improve performance much like we did in Chapter 5.

To perform this function/architecture optimization and co-design starting from the high abstraction level, and at every level of refinement until implementation where HW/SW co-design is performed, I have introduced a novel intermediate design representation, and a formal methodology and framework for actively trading-off between the function and the architecture using constrained (from the top level) and guided (using bottom-up estimates) optimization. My work builds on the research in global data flow and control optimizations from the software domain which has been typically applied in a limited fashion to assembly code generation for micro-processors starting from hand-written code. The notions I use are also in the "spirit" of high level synthesis, an area which has been active for the past 2 decades but which may have set itself up by biting more than any methodology can reasonably chew.

Through the use of *separation of concerns*, *abstraction*, *decomposition*, and *successive refinement* in a synthesis (top-down) constraint-driven (bottom-up) methodology I have been able to circumvent some of the pitfalls of typical "push-button" approaches. I

have argued that **boosting design productivity** cannot be the major focus of system level co-design tools; attention should be paid as well to enhancing the **quality** of the final synthesized output that will run on the target architecture. I have shown that *design representation level data flow and control optimizations* have a considerable positive improvement effect on synthesized software and hardware. The optimizations at the high level assist the lower (abstraction) level optimization algorithms and heuristics in the co-design flow as well. This is because typically the lower level algorithms deal with smaller granularity optimizations and therefore perform better on smaller inputs. It should be noted, as well, that the computation cost of the data flow and control analysis and optimization to improve synthesis quality should not be viewed as an overhead since it is most often recovered, as we have seen, by the significant speed up of the low level synthesis techniques.

I believe that my theoretical and practical work sets the groundwork for sound and more involved function/architecture optimizations. In particular, in the future, aside from the various expansions and improvements to function/architecture optimization and co-design both for estimation of guiding metrics as well as in optimization techniques themselves as I outlined in the respective Chapter's future directions sections, I'd like to explore what I neglected to dwell on in this dissertation. The key areas for further investigation I believe are: *Functional Decomposition*, *Cross-"Block" Optimization*, and *Task and System Level Algorithmic Manipulations*.

In this work I have assumed an initial functional decomposition possibly given by the user, and proposed an iterative flow (see Chapter 8) for improving this functional partitioning through the use of "collapsing" given an initial "small"-grain decomposition.

Decomposition remains an open problem. It would be interesting to think of metrics that can generalize hardware/software partitioning techniques to more of *constrained and guided functional decomposition* where possibly the user constraints and architectural performance estimates can guide the early exploration at this stage.

Cross-"Block" Optimization: is also a very interesting avenue of investigation. We have started to look at this under the auspices of the NexSIS effort ([111]) within the Gigascale Silicon Research Center (GSRC [45]) where we are focusing on an interconnection of *hardware* **Intellectual Property (IP)** blocks. Intellectual property (IP) is the single biggest issue facing the electronic industry today [48]. Grenier in [48] states that while we continue to go lower and lower on line width and put increasing numbers of transistors on a chip, there is no corresponding increase in capability to 'wire up' these components. This gap can be bridged with re-usable blocks that are "hooked up". This is the essence of IP and design re-use so that the designer does not have to start from scratch at every design start and miss the market window. Developing an *IP modeling and assembly* framework that is based on sound theoretical foundations (such as those of the DFA framework) where the interconnection of IPs: soft (RTL), firm (gate), and hard can be *optimized*, and *composed* in an *optimal* manner that satisfies the design budget constraints (for example, addresses the *timing closure* problem) is the promising solution, that we propose and are working towards, to the design woes in the IP mix-and-match domain. Generalizing the framework to represent and subsequently optimize a mix of HW and SW IPs is left for the future.

Another area for research that is closely tied with the previous two directions is that of performing user-guided *algorithmic manipulations*. In this work I have assumed

that I/O traces cannot be "touched" at the functional optimization level for lack of any knowledge, and then I relaxed this at the macro-architectural level to permit manipulations ***within state boundaries*** if doing so is permitted by the implementation architecture. An interesting avenue for exploration is to allow the user to specify a set of ***valid algorithmic manipulations*** at the ***task level*** and also at the ***system level*** such as the ones I alluded to in Section 6.3.4: partial reduction, symmetry reduction, "don't care" response of I/O traces. Task level optimizations that I have discussed can leverage the additional task I/O manipulation knowledge to optimize the targeted metrics such as: WCET, average runtime, and code size. At the system level, task scheduling and resource allocation can make use of the given knowledge as well; the RTOS schedule may be tuned to optimize system response time, assuming the analysis and optimization is not computationally prohibitive of course.

Finally, I'd like to ***thank you***, the reader, for your patience in reading the material presented in this dissertation. I have tried to make the organization as simple as possible, introducing and building on concepts as the need arose in hopes of getting my message across. Documenting my research has certainly been a worthwhile endeavor; I can only hope that your expectations have been met, and that reading this dissertation was as much a pleasurable educational experience for you, as it has been for me writing it.

Bibliography

[1] Abernathy, W., "The Productivity Dilemma: Roadblock to Innovation in the Automobile Industry", *The John Hopkins University Press*, 1978.

[2] Aho, A.V.; Sethi, R.; Ullman, J.D., "Compilers: Principles, Techniques, and Tools" *Addison-Wesley*, 1988.

[3] Baker, W., "Application of the Synchronous/Reactive model to the VHDL Language", Technical Report UCB/ERL M93/10, *UC Berkeley*, 1993.

[4] Balarin, F., "Worst-case Analysis of Discrete Systems", *ICCAD*, 1999.

[5] Balarin, F.; Chiodo, M.; Giusto, P.; Hsieh, H.; Jurecska, A.; Lavagno, L.; Passerone, C.; Sangiovanni-Vincentelli, A.L.; Sentovich, E.; Suzuki, K.; and Tabbara, B., "Hardware-Software Co-Design of Embedded Systems: The POLIS Approach", *Kluwer Academic Publishers*, MA, USA, May 1997.

[6] Balarin, F.; Chiodo, M., "Software Synthesis for Complex Reactive Embedded Systems", *ICCD*, 1999.

[7] Balarin, F.; Chiodo, M.; Giusto, P.; Hsieh, H.; Jurecska, A.; Lavagno, L.; Sangiovanni-

Vincentelli, A.; Sentovich, E.M.; Suzuki, K., "Synthesis of Software Programs for Embedded Control Applications", *IEEE Transactions on CAD*, 1999.

[8] Balarin, F.; Chiodo, M.; Jurecska, A.; Lavagno, L.; Tabbara, B.; Sangiovanni-Vincentelli, A., "Automatic Generation of a Real-Time Operating System for Embedded Systems" *CODES* March 1997.

[9] Banning, J., "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables", *ACM Symposium on Principles of Programming Languages*, pp. 29-41, 1979.

[10] Barton, D., "The System Level Design Language Initiative", at: `http://www.inmet.com/SLDL`, 1999.

[11] Batchelor, R., "Henry Ford, Mass Production, Modernism, and Design *Manchester University Press*, 1994.

[12] Benyamin, D.; Mangione-Smith, W., "Function Unit Specialization through Code Analysis", *ICCAD*, 1999.

[13] Bergamaschi, R.A., "Behavioral Network Graph: Unifying the Domains of High-Level and Logic Synthesis", *DAC*, 1999.

[14] The Berkeley Wireless Research Center (BWRC), see web pages at: `http://bwrc.eecs.berkeley.edu`

[15] Bern, J.; Meinel, C.; Slobodova, A., "Efficient OBDD-based Boolean manipulation in CAD beyond current limits" *DAC*, 1995.

[16] Berry, G., 1996. See http://www.inria.fr/meije/esterel/

[17] Berry, G.; Gonthier, G., "The Esterel Synchronous Programming Language: Design, Semantics, and Implementation", *Science of Computer Programming*, vol.19, no 2, pp. 87-152, 1992.

[18] Birkhoff, G., "Lattice Theory", *American Mathematical Society Colloquium Publications*, 1967.

[19] Bouissou, M.; Martin, F.; Ourghanlian, A., "Assessment of a Safety-Critical System Including Software: A Bayesian Belief Network for Evidence Sources", *Reliability and Maintainability Symposium*, 1999.

[20] Boussinot, F.; Doumene, G.; Stefani, J., "Reactive Objects", *Annales des Telecommunications*, vol. 51, no 9-10, pp. 459-473, 1996.

[21] Caspi, P.; Halbwachs, N.; Pilaud, D.; Plaice, J., "A Declarative Language for Programming Synchronous Systems", *ACM Symposium on Principles of Programming Languages*, pp. 178-188, 1987.

[22] Chang, H.; Cooke, L.; Hunt, M.; Martin, G.; McNelly, A.; Todd, L., "Surviving the SOC Revolution: A Guide to Platform-Based Design", *Kluwer Academic Publishers*, 1999.

[23] Bjuréus, P.; Tabbara, B., "High Level Software Cost Estimation", EECS 249 Project, *UC Berkeley*, Fall 1999.

[24] Booch, G., "Object-Oriented Design", *Benjamin-Cummings*, 1990.

[25] Bryant, R., "Graph-Based Algorithms for Boolean Function Manipulation" *IEEE Transactions on Computers*, 1986.

[26] Cappellari, A.; Licciardi, L.; Montanaro, A., Paolini, M.; Lavagno, L., "Specification of an ATM Design for The Evaluation of a HW/SW Co-design Methodology." *CSELT Technical Report*, 1996.

[27] Castelluccia, C.; Dabbous, W.; O'Malley, S., "Generating Efficient Protocol Code from an Abstract Specification", *IEEE/ACM Transactions on Networking*, August 1997.

[28] Chiodo, M.; Giusto, P.; Hsieh, H.; Jurecska, A.; Lavagno, L.; and Sangiovanni-Vincentelli, A., "Hardware/software Co-design of Embedded Systems" *IEEE Micro*, Vol. 14, Number 4, pp. 26-36, 1994.

[29] Chiodo, M.; Giusto, P.; Jurecska, A.; Lavagno, L.; Hsieh, H.; Suzuki, K.; Sangiovanni-Vincentelli, A.; Sentovich E., "Synthesis of Software Programs for Embedded Control Applications", *DAC*, June 1995.

[30] Choi, C.; Ha S., "Software Synthesis for Dynamic Data Flow Graph", *IEEE International Workshop on Rapid System Prototyping*, June 1997.

[31] Choi, H.; Park, I.; Hwang, S.; Kyung, C., "Synthesis of Application Specific Instructions for Embedded DSP Software", *ICCAD*, 1998.

[32] Chow, F.C., "A Portable Machine-Independent Global Optimizer–Design and Measurement", *Ph.D. Thesis*, Stanford University, 1983.

[33] Cortadella, J.; Lavagno, L.; Sangiovanni-Vincentelli, A.L., "Embedded Code Optimization via Common Control Structure Detection", *CODES*, 1997.

[34] Cousot, P.; Cousot, R., "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints", *ACM Symposium on Principles of Programming Languages*, 1977.

[35] Cygnus GNU, http://www.cygnus.com, 2000.

[36] Dally, W.; Poulton, J., "Digital Systems Engineering", *Cambridge University Press*, 1998.

[37] Dawkins, R., "The Selfish Gene", *Oxford University Press*, 1976.

[38] De Micheli, G.; Ku, D.; Mailhot, F.; Truong, T., "The Olympus Synthesis System for Digital Design", *Center for Integrated Systems Stanford University*, 1992.

[39] Devadas, S.; Newton, A., "Decomposition and Factorization of Sequential FSMs", *ICCAD*, 1998.

[40] The EDN Embedded Microprocessor Benchmark Consortium (EEMBC), http://www.eembc.org, 1999.

[41] Ernst, R., "Embedded System Architectures" in "Hardware/Software Co-Design: Principles and Practice", *Kluwer Academic Publishers*, 1997.

[42] Ferrari, A.; Sangiovanni-Vincentelli, A., "System Design: Traditional Concepts and New Paradigms", *IEEE ICCD*, 1999.

[43] Filippi, E.; Lavagno, L.; Licciardi, L.; Montanaro, A.; Paolini, M.; Passerone, R.; Sgroi, M.; Sangiovanni-Vincentelli, A., "Intellectual Property Re-use in Embedded System Co-design: an Industrial Case Study", *ISSS*, Dec. '98.

[44] Gao, M.; Xu S.; Tabbara B., "Interfacing POLIS to Mentor Graphics VRTX RTOS", EECS 249 Project, *UC Berkeley*, Fall 1999.

[45] Gigascale Silicon Research Center, Richard Newton, Director; Kurt Keutzer Co-director at `http://www.gigascale.org`

[46] Goossens, G.; Lanneer, D.; Vahoof, J.; Rabaey, J.; Van Meerbergen, L.; De Man, H. "Optimization-based Synthesis of Multiprocessor Chips for Digital Signal Processing, with CATHEDRAL II", *International Workshop on Logic and Architecture Synthesis for Silicon Compilers*, 1988.

[47] Green, C., "Eli Whitney and the Birth of American Technology", *Boston, Little, Brown*, 1956.

[48] Grenier, J., "Dataquest Forecast – Semiconductor Industry", excerpts from talk at *SEMI/Dataquest Outlook Dinners*, Austin and Dallas, November 1999.

[49] Grotker, T.; Schoenen, R.; Meyr, J., "Unified Specification of Control and Data Flow", *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1997.

[50] Guerra, L.; Fitzner, J.; Talukdar, D.; Schlager, C.; Tabbara, B.; Zivojnovic, V., "Cycle and Phase Accurate DSP Modeling and Integration for HW/SW Co-Verification", *DAC*, June 1999.

[51] Hailperin, M. "Cost-Optimal Code Motion", ACM Transactions on Programming Languages and Systems, Vol. 20, No. 6, pp. 1297-1322, November 1998.

[52] Hanono, S.; Devadas, S. "Instruction Selection, Resource Allocation, and Scheduling in the Avis Retargetable Code Generator", *DAC*, 1998, pp. 510-515.

[53] Harel, D., "StateCharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, vol. 8, 1987.

[54] Hecht, M.; Ullman, J., "Analysis of a Simple Algorithm for Global Flow Problems", *ACM Principles of Programming Languages*, October 1973.

[55] Hsieh, H.; Sangiovanni-Vincentelli, A.; Balarin, F.; Lavagno, L., "Synchronous Equivalence for Embedded Systems: a Tool for Design Exploration", *ICCAD*, 1999.

[56] Hsu, Y.; Tsai, F.; Liu, T.; Lin, S., "VHDL Modeling for Digital Design Synthesis", *Kluwer Academic*, 1995.

[57] Kam, J.B., Ullman, J.D. "Global Data Flow Analysis and Iterative Algorithms", *J. ACM*, Vol. 23, No. 1 , January 1976, pp. 158-171.

[58] Kam, J.B., Ullman, J.D. "Monotone Data Flow Analysis Frameworks", *Acta Informatica*, Vol. 7, 1977 pp. 305-317.

[59] Kennedy, K., "A Comparison of Two Algorithms for Global Data Flow Analysis", *SIAM J. COMPUT.*, Vol 5, No. 1, 1976.

[60] Kernighan, B.; Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell System Technical Journal*, Vol. 49, No. 2, 1970, pp. 291-308.

[61] Khatri, S. "Cross-talk Noise Immune VLSI Design Using Regular Layout Fabrics", Ph.D. Dissertation, *UC Berkeley*, December 1999.

[62] Kin, K., Vice-president, Worldwide Sales, Consumer and Internet Products Segment, IBM Technology Group, IBM Corporation, "Technologies in the Post PC age", Excerpt from Dr. Kin's address at the SEMI/TSIA Industry Strategy Symposium in Taiwan on November 29, 1999.

[63] Köbler, J.; Schöning, H.; Torán, J.,"The Graph Isomorphism Problem: Its Structural Complexity", *Birkhauser Boston*, 1993.

[64] Kubiatowicz, J., "Introspective Computing", *UC Berkeley*, 1999.

[65] Kildall, G., "A Unified Approach to Global Program Optimization", *ACM Symposium on Principle of Programming Languages*, 1973, pp. 194-206.

[66] Klein, R.; Leef, S., "New Technology Links Hardware and Software Simulators" *In Electronic Engineering Times*, June 1996.

[67] Knoop, J.; Rüthing, O., "Optimal Code Motion: Theory and Practice", ACM Transactions on Programming Languages and Systems, Vol 16. No. 4, July 1994, pp. 1117-1155.

[68] Knoop, J.; Rüthing, O.; Steffen, B., "Lazy Code Motion", *ACM SIGPLAN*, Vo. 27, No. 7, pp. 224-234, 1992.

[69] Knuth, D., "An Empirical Study of FORTRAN Programs", *Software Pract. and Exper.*, Vol. 1, No. 2, 1971.

[70] Knuth, D., "The Art of Computer Programming", Addison-Wesley, 1973.

[71] Lajolo, M.; Lazarescu, M.; Lavagno, L., "A Compilation-based Software Estimation Scheme for Hardware/Software Co-Simulation", *CODES*, 1999.

[72] Lajolo, M.; Lavagno, L.; Sangiovanni-Vincentelli, A., "Fast Instruction Cache Simulation Strategies in a Hardware/Software Co-Design Environment", *ASP-DAC*, 1999.

[73] Lajolo, M.; Tabbara, B., "Software Optimization in Polis", *Personal Communication*, 2000.

[74] Lavagno, L.; Sentovich, E., "ECL: A Specification Environment for System-Level Design", *DAC*, 1999.

[75] LEDA, see web pages at `http://www.mpi-sb.mpg.de/LEDA`

[76] Lemone, K.A., "Design of Compilers: Techniques of Programming Language Translation", *CRC Press*, 1992.

[77] Leupers, R.; Marwedel, P., "Function Inlining under Code Size Constraints for Embedded Processors", *ICCAD*, 1999.

[78] Li, Y.; Wolf, W., "Hardware/Software Co-synthesis with Memory Hierarchies", *IEEE Transactions on CAD*, 1999.

[79] Liem, C.; May, T.; Paulin, P., "Instruction-set Matching and Selection for DSP and ASIP Code Generation", *EDAC*, 1994.

[80] Lindstrom, G., "Static Evaluation of Functional Programs", *SIGPLAN Notices*, Vol. 21, No. 7, 1986.

[81] The LUSTRE-ESTEREL Portable Format Version OC5, *Manual*, 1997.

[82] Marlowe, T.J.; Ryder, B.G., "Properties of Data Flow Frameworks", *Acta Informatica*, Vol. 28, pp. 121-163, 1990.

[83] McGeer, P.; McMillan, K.; Saldanha, A.; Sangiovanni-Vincentelli, A.; Scaglia, P., "Fast Discrete Function Evaluation Using Decision Diagrams", *ICCAD*, 1995.

[84] Microsoft Research: Decision Theory and Adaptive Systems Group at: `http://research.microsoft.com/msbn/default.htm`

[85] Morel, E.; Renvoise, C., "Global Optimization by Suppression of Partial Redundancies", Commun. ACM, Vol. 22, No. 2, pp. 96-103, 1979.

[86] Murthy, P.K.; Bhattacharya, S.S.; Lee, E.A., "Joint Minimization of Code and Data for Synchronous Dataflow Programs", *Journal of Formal Methods in System Design*, July 1997.

[87] Petri, C.A., Kommunikation mit Automaten, *Dissertation*, Bonn, 1962.

[88] Narayan, S.; Vahid, F.; Gajski, D., "System Specification with the SpecCharts Language", *IEEE Design and Test of Computers*, Dec, 1992.

[89] Newton, A.R., Personal Communication, Berkeley, 1999.

[90] Nash, S.G.; Sofer, A., "Linear and Non-Linear Programming", *McGraw Hill*, 1996.

[91] Niemann, R., "Hardware/Software Co-design for Data Flow Dominated Embedded Systems", *Kluwer Academic Publishers*, 1998.

[92] Passerone, R.; Rowson, J.A.; Sangiovanni-Vincentelli, A., "Automatic Synthesis of Interfaces between Incompatible Protocols", *DAC*, 1998.

[93] Pearl, J., "Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference", *Morgan-Kaufmann*, 1988.

[94] The Polis group, Polis Home Page, http://www-cad.eecs.berkeley.edu/~polis

[95] The Ptolemy Group, Ptolemy Home Page, http://www.eecs.berkeley.edu/~ptolemy

[96] Rabaey, J., "Component Based Design and The PicoRadio Design Driver", GSRC Workshop, December 1999.

[97] Rowson, J.; Sangiovanni-Vincentelli, A., "Interface-Based Design", *DAC*, 1997.

[98] Rudell, R., "Dynamic Variable Ordering for Ordered Binary Decision Diagrams", *ICCAD*, November 1993.

[99] Sabel, S., Zeitlin, J., "World of Possibilities : Flexibility and Mass Production in Western Industrialization, *Cambridge University Press*, c1997.

[100] Sangiovanni-Vincentelli, A., Personal Communication, Berkeley, 1999.

[101] Semiconductor Industry Association, "The National (International) Technology Roadmap for Semiconductors", 1997 (1999).

[102] Sentovich, E., "Sequential Circuit Design Using Synthesis and Optimizations" *ICCAD*, 1992.

[103] Sgroi, M., "Model and Design of an ATM Server for Virtual Private Networks" *Personal Communication*, 1997.

[104] Shultz, S., "New ITRS Roadmap Portends Massive Design Challenges Ahead", *ISDN Magazine*, December 1999.

[105] Simonyi, C., "Intentional Programming - Innovation in the Legacy Age", Presented at *IFIP WG 2.1*, June 4, 1996, see web page at:
`http://research.microsoft.com/ip/ifipwg/ifipwg.htm`

[106] Sinha, S.; Brayton, R., "Implementation and Use of SPFDs in Optimizing Boolean Networks", *ICCAD*, 1998

[107] Siska, C., "A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools", *ISSS*, 1998.

[108] Srivastava, A.; Eustace, A., "ATOM: A System for Building Customized Program Analysis Tools", *SIGPLAN*, 1994.

[109] Sung, W.; Kum, K., "Word-Length Determination and Scaling Software for a Signal Flow Block Diagram", *ICASSP*, 1994.

[110] Suzuki, K.; Sangiovanni-Vincentelli, A., "Efficient Software Performance Estimation Methods for Hardware/Software Co-design" *In Proceedings of the Design Automation Conference*, pp. 605-610, June 1996.

[111] Tabbara, A., Tabbara, B., Brayton, B., Newton, A.R., Sangiovanni-Vincentelli, A.L., "NexSIS: An IP Modeling and Integration Framework", GSRC Workshop, December 1999.

[112] Tabbara, B.; Balarin, F., "Hardware/Software Co-design Experiments", Co-design Laboratory *Cadence Berkeley Labs*, Summer 97, and Summer 98.

[113] Tabbara, B.; Filippi, E.; Lavagno, L.; Sgroi, M.; Sangiovanni-Vincentelli, A., "Fast Hardware-Software Co-simulation Using VHDL Models", *DATE*, March 1999.

[114] Tabbara, B.; Tabbara, A.; Sangiovanni-Vincentelli, A.,"Hardware and Software Representation, Optimization, and Co-synthesis for Embedded Systems", Technical Report UCB/ERL M00/7, *UC Berkeley*, 2000.

[115] Tabbara, B.; Sangiovanni-Vincentelli, A., "Data Flow and Control Optimizations for Hardware and Software Synthesis", *SRC Graduate Fellowship Conference*, Sept. 1999.

[116] Tabbara, B.; Tabbara, A.; Sangiovanni-Vincentelli, A.,"Task Response Time Optimization Using Cost-Based Operation Motion", *CODES*, May 2000.

[117] Tabbara, B.; Tabbara, A., "Simulation-Oriented Behavioral Verification", *Mentor Graphics Users' Group Conference*, Sept. 1999.

[118] Tjiang, S.W., "Automatic Generation of Data-Flow Analyzers: A Tool for Building Optimizers", *Ph.D. Thesis*, Stanford University, 1993.

[119] Trimaran, "An Infrastructure for Research in Instruction-Level Parallelism" at: `http://www.trimaran.org/` , 1999.

[120] Trivedi, K., "Probability and Statistics with Reliability, Queuing and Computer Science Applications", *Prentice Hall*, 1982.

[121] University of California Office of Technology Transfer, see web page at: `http://www.ucop.edu/ott/permissn.html`

[122] Vahid, F., "A Three-step Approach to the Functional Partitioning of Large Behavioral Processes, *ISSS*, 1998.

[123] Vahid, F., "Procedure Cloning: A Transformation for Improved System-Level Functional Partitioning", *TODAES*, 1999.

[124] Vahid, F., "Procedure Exlining: A Transformation for Improved System and Behavior Synthesis", *ISSS*, 1995.

[125] Vahid, F.; Gajski, D., "Incremental Hardware Estimation During Hardware/Software Functional Partitioning", *IEEE Transactions on VLSI Systems*, Sept. 1995.

[126] Van Rompaey, K.; Verkest, D.; Bolsens, I.; De Man, H., "CoWare-a Design Environment for Heterogeneous Hardware/Software Systems", *Euro-DAC*, 1996.

[127] VRTXoc Open Source Licensing, *Mentor Graphics* see web pages at: `http://www.mentor.com/embedded/downloads/index.html`

[128] Wagner, T.; Maverick, V.; Graham, S.; Harrison, M. "Accurate Static Estimators for Program Optimization", *ACM SIGPLAN*, 1994.

[129] Weiss, M.A., "Data Structures and Algorithm Analysis in C++", *Addison-Wesley*, 1994.

[130] Willems, M; Burgens, V.; Grotker, T.; Meyr, H., "FRIDGE: An Interactive Code Generation Environment for HW/SW Co-design", *ICASSP*, 1997.

Appendix A

C-Like Intermediate Format (CLIF) for Design Representation

I list below the LALR(1) [2] grammar of the *C-Like Intermediate Format (CLIF)*, the *concrete syntax* or textual interchange format for the **Function Flow Graph (FFG)** *abstract* syntax. Attributes for the **Attributed Function Flow Graph (AFFG)** are represented as *directives* in CLIF typically as comments using the `';'` or `/* ... */` delimiters. I should point out that fellow researchers and I in the NexSIS endeavor are working on an *XML* version of this concrete syntax to simplify parsing, and improve extensibility and portability of CLIF [111]. The listing below shows the reader all the currently supported constructs (including data types, as well as some primitive pointer support not used currently in the design flow (it has no corresponding structure in the front-end)).

```
/* A CLIF representation is made up of a declaration list then an
 * operations list.
 */
```

```
program:      decl_list inst_list

decl_list:    decl_list decl
              |decl

inst_list:    inst_list inst
              |inst

/* A declaration is that of a variable or signal.
 */

decl:          type  ID '=' expr ';'
              |type  ID ';'
              |type ID '(' args ')' ';'
              |type ID '(' ')' ';'

args:         type
              |args ',' type

/* Expressions are
 */

expr:          '+' terms
              |'-' terms
              |terms

terms:         terms '+' term
              |terms '-' term
              |terms '|' term
              |term

term:          term '*' factor
              |term '/' factor
              |term '%' factor
              |term '&' factor
              |factor

factor:        '(' expr ')'
              |'~' factor
              |integer
              |character

/* An operation has several formats.
 */
```

```
inst:          label ':' inst
              |variable '=' simple_expr ';'
              |'*' variable '=' variable ';'
              |if '(' variable ')' goto label ';'
              |goto label ';'
              |exit '(' integer ')' ';'
              |cout '<<' string ';'
              |';'

label:        STATE
              |DAG
              |ID

simple_expr:   variable binop variable
              |unop variable
              |variable '[' variable ']'
              |variable

variable:      ID
              |ID '(' argms ')'
              |ID '(' ')'

argms:        variable
              |argms ',' variable

type:          int
              |int '*'
              |input
              |Input
              |output
              |Output
              |Local
              |local
              |sensor

unop:          '!'
              |'-'
              |'*'
              |'&'

binop:         '+'
              |'-'
              |'*'
```

```
|'/'
|'%'
|and
|or
|eq
|ne
|'<'
|le
|'>'
|ge
```