Design Methodology for Run-time Management of Reconfigurable Digital Signal Processor Systems

Roy Allen Sutton

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-63
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-63.html

May 17, 2006

Design Methodology for Run-time Management of Reconfigurable Digital Signal Processor Systems

by

Roy Allen Sutton

B.S. (Prairie View A&M University of Texas) 1993
M.S. (University of California, 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, BERKELEY

Committee in charge:
Professor Jan M. Rabaey, Chair
Professor Kristofer S.J. Pister
Professor Paul K. Wright

Spring 2006

Design Methodology for Run-time Management of Reconfigurable Digital Signal Processor Systems

by

Roy Allen Sutton

Abstract

Design Methodology for Run-time Management of Reconfigurable Digital Signal Processor Systems

by

Roy Allen Sutton

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Jan M. Rabaey, Chair

In recent years, significant attention has been given to emerging reconfigurable digital signal processing (DSP) systems. They can undergo in-system hardware changes in order to adapt to post-design external variations in system constraints, stimulus, algorithms, and optimization objectives. This adds degrees of freedom and levels of complexity to the design process that are not well considered by contemporary computer aided design (CAD) methods. This research presents a methodology and CAD design environment, based in a quantitative approach, that support the implementation of domain-specific DSP systems which cope with post-design changes by leveraging reconfigurable hardware architectures.

Run-time methods and dynamic reconfiguration comes at the costs of overheads. The proposed methodology limits these overheads by imposing a framework that creates multiple pre-characterized templated-mapping design implementations that are organized as candidates under the discrimination of one or more system schedulers. At run-time, the managing scheduler selects implementations according to its assigned optimization objective given the resource availability. To deal with the tight constraints on cost-performance inherent within DSP systems, the framework introduces mapping modes that further guide and/or limit run-time scheduler choice in template selection (dynamic resource "type" allocation), resource binding, and configuration. These modes allow for design-time trade-offs between predictability and flexibility. A formal model that defines the components and their interactions is presented to provide some rigor in the methodology application.

A design environment and tools development framework is presented that adheres to the methodology and formal model. It includes a simulator for use in quantitative design exploration. The framework accounts for use-costs in computation, communication, and reconfiguration in terms of area, energy, as well as time. Others metrics of interest (such as position, etc.) can be easily incorporated. Detailed quantitative statistics for system behavior are accounted for and visually presented. This aids the designer during iterative design refinement. Moreover,

performance metrics can be analyzed at run-time by system schedulers as a guide for statistical-based adaptation.

The effectiveness of the proposed methodology is demonstrated across a database of generated designs and for an actual multi-user DSP system design. The results for each are presented in separate chapters. The appendix includes a tutorial that discusses the environment, framework, an existing CAD tools.

Professor Jan M. Rabaey
Dissertation Committee Chair

To my loving family,
the sacrifice of Joshua,
and the memory of
Donald Steven Smith...

Contents

List of Figures **v**

List of Tables **viii**

Listings **ix**

1 Introduction **1**
- 1.1 Digital Communications Revolution . . . 3
- 1.2 Reconfigurables Need New Methodologies . . . 4
- 1.3 Research Goals and Organization . . . 10

2 Reconfigurable Digital Signal Processing Systems **12**
- 2.1 System Examples and Features . . . 13
- 2.2 Dynamic Change in Algorithms . . . 15
  - 2.2.1 Forms of Change . . . 16
- 2.3 Reconfigurable Architectures . . . 20
  - 2.3.1 Cost Distribution . . . 23
- 2.4 Mapping and Design-flow . . . 25
- 2.5 Chapter Summary . . . 27

3 Templated-Mapping Design Methodology **29**
- 3.1 Algorithm Mapping . . . 31
- 3.2 Mapping with Templates . . . 34
  - 3.2.1 Templated-Mapping . . . 35
  - 3.2.2 Candidates . . . 36

3.2.3 Mapping Modes . . . 39
3.2.4 Mapping Paradigm . . . 43
3.3 Offline Template Construction . . . 46
3.3.1 Algorithm Analysis . . . 47
3.3.2 Implementation Alternatives . . . 52
3.3.3 Run-Time Management Properties . . . 55
3.3.4 Implementation Cost Estimations . . . 58
3.4 Run-Time Template Use . . . 60
3.4.1 Template Selection . . . 60
3.4.2 Template Instantiation . . . 64
3.5 Method Formalization . . . 68
3.5.1 Component Model . . . 70
3.5.2 Execution Process Model . . . 80
3.6 Chapter Summary . . . 89

4 Templated-Mapping Design Environment 91
4.1 Core Infrastructure . . . 93
4.1.1 Internal Data Model . . . 93
4.1.2 Event Recording Mechanisms . . . 97
4.1.3 Report Generation Interfaces . . . 100
4.1.4 Data Model Extension . . . 102
4.2 Simulator Framework . . . 103
4.2.1 Algorithm Modeling . . . 105
4.2.2 Architecture Modeling . . . 108
4.2.3 Candidate Implementation Modeling . . . 109
4.2.4 Scheduler Modeling . . . 110
4.2.5 Static Design Analysis and Optimization . . . 116
4.3 Simulation Execution Flow . . . 118
4.3.1 Simulation Condition Monitors . . . 120
4.4 Environment Design-Flow . . . 121
4.4.1 Tool Scripting . . . 124
4.4.2 Graphical User Interfaces . . . 126
4.5 Chapter Summary . . . 127

5 Design-flow Evaluation **129**
5.1 System Component Composition 130
5.2 Example System Simulation 137
5.3 Chapter Summary 142

6 A Design Exploration Example **146**
6.1 System Description 147
6.2 Templated-Mappings 150
6.3 Design Exploration 153
6.3.1 Active User Sensitivity 154
6.3.2 Run-time Optimization Objectives 162
6.4 Chapter Summary 164

7 Conclusions **167**
7.1 Directions for Future Work 169

Bibliography **173**

A Terminology **183**

B Design Environment Tutorial **185**
B.1 Templated-Mapping 187
B.2 Tool Types 188
B.3 Data Formats 189
B.4 File Structure 191
B.5 Tools 200
B.6 Reports 209
B.7 Framework Expansion 211
B.7.1 Stimulus Modeling 214
B.7.2 Task Modeling 215
B.7.3 Scheduler Modeling 216

Index **221**

List of Figures

1.1 Growing body of wireless communication standards 4
1.2 Reconfigurables enable flexible yet efficient design 7

2.1 Forms of dynamic change in algorithms 17
2.2 Reconfigurables emerge to balance efficiency and flexibility 20
2.3 Architecture template as a building block 22

3.1 System mapping time-line 33
3.2 Templated-mapping introduction 35
3.3 Mapping algorithms to reconfigurable architectures 38
3.4 System mapping modes 39
3.5 Mapping with sets of templated-mappings 45
3.6 ASIC cost variation for DCT algorithms 49
3.7 JPEG encoder algorithm example 51
3.8 Algorithm graph instantiation and node implementation alternatives 54
3.9 Properties for run-time management 56
3.10 Resource configuration topology schemes 66
3.11 System-level design modeling language primitives 81
3.12 System model: Component structure and relations 83
3.13 System model: Algorithm, architecture, and scheduler execution . 86

4.1 Skeleton of the design environment core libraries and data model . 94
4.2 State trace example: resource utilization and concurrency 98
4.3 Simulator framework key objects and select functions 104
4.4 One implemented algorithm modeling MoC 106

4.5 Methodology: System architecture base model 108
4.6 Methodology: Candidate implementation conceptual model 109
4.7 An example scheduler: the stdbe and its life-cycle 112
4.8 Static design analysis dependency enforcement 118
4.9 Simulation base execution control flow 119
4.10 Condition monitors and scripting framework for design exploration 121
4.11 Methodology design-flow responsibility 123
4.12 Methodology design-flow sequence 125

5.1 Generated algorithm families examples 131
5.2 Modeling families of heterogeneous reconfigurable architectures . . 133
5.3 Generated candidate and system of candidates example 134
5.4 Run summary simulation example 139
5.5 Candidate selection performance metric 140
5.6 Candidate selection quality simulation example 141
5.7 Concurrency and utilization simulation example 143
5.8 Time-use distribution simulation example 144

6.1 MPEG-4 simple profile decoder block diagram 148
6.2 Multi-user decoder system resource 150
6.3 IDCT kernel templated-mapping 151
6.4 IDCT kernel decorated templated-mapping 152
6.5 Run-time scheduler execution time vs. concurrent active users . . 155
6.6 Select dynamic statistics vs. concurrent active users 157
6.7 Quality and overhead vs. concurrent active users 158
6.8 Timing performance vs. active users 160
6.9 Resource utilization vs. time 161
6.10 Select dynamic statistics vs. scheduler task ordering function . . . 163
6.11 Mapping selection quality vs. scheduler optimization function . . 165

7.1 Domain-specific systems: Multi-level networks and bus interconnect 170
7.2 Dynamic algorithm schedule management 172

B.1 A templated-mapping example 186
B.2 Design environment directory structure 192

B.3 Design database directory structure 195
B.4 Simulation results directory structure 197
B.5 Design-flow graphical user interfaces 205
B.6 Graphical tools from other sources 208
B.7 Chart report examples 212
B.8 Graph report examples 213

List of Tables

1.1 Basic comparison of some wireless communication standards . . . 6

2.1 System operation cost distribution for reconfigurables 24

3.1 Candidate mapping template selection properties 61
3.2 System model components 69
3.3 System property functions 79

4.1 Task life-cycle: flow of execution 107
4.2 System scheduler base responsibilities 111

5.1 Simulation example run key 137

6.1 Static and dynamic MPEG-4 decoder profile 149

B.1 Design environment tool types 189
B.2 Design environment file formats 190
B.3 Design environment command line tools 200
B.4 Implemented report flows 210

Listings

4.1 Simulation event log example 101
B.1 A parsed simulation run metafile example 193
B.2 Simulator command line synopsis 202
B.3 Stimulus function modeling example 215
B.4 Task modeling example 217
B.5 Scheduler example: Pre-static analysis initialization 218
B.6 Scheduler example: Post-static analysis initialization 219
B.7 Scheduler example: Done task processing code fragment 220

Acknowledgments

This research was funded by various sources over the years. A special thanks goes to AT&T Bell Laboratories (then, now Lucent technologies) for their generous Cooperative Research Fellowship Program (CRFP) which supported the first five years of my study and to Sandia National Laboratories for their participation in the CRFP and for sponsoring a stimulating internship full of relevant and practical exposure.

I am particularly indebted to my advisor, Professor Jan M. Rabaey for his consistent support and guidance. His unwavering high standard and example has been admirable. I am truly appreciative to have had multiple opportunities to participate in his research efforts. I would like also to thank my committee members, Professor Kristofer S. J. Pister and Professor Paul K. Wright. Additionally, I am appreciative of Professor K. Robert Brayton for taking time to chair my qualifying examination. There have been many U.C. Berkeley EECS faculty that have inspired, given time and encouraged my work. Among the many are Robert W. Brodersen, Edward A. Lee, Andrew R. Neureuther, Paul R. Gray, Randy H. Katz, and John Wawrzynek.

A special thanks goes to Sheila Humphreys, Professor Brodersen, and Professor Andrew R. Neureuther for their life-course changing introduction to U.C. Berkeley through the SUPERB program. A special acknowledgement goes to Anantha

Chandrakasan and Kevin Kornegay for their fittingly superb mentorship. This was one of the most exciting times of technical growth that I can remember. I am amazed at what can be accomplished in such a short period of time when experience, motivation, resource, dedication, encouragement, and access align.

I would like to express my gratitude to Tom Boot, Fred Burghardt, Susan Mellers, Brian Richards, and Kevin Zimmerman for supporting the work environment, computers, laboratory equipment, and software infrastructure. Each always gave beyond that which was required and established a stabilizing presence.

There have been so many fellow students who I have had the great fortune to encounter during my participation over the years in the Infopad, Spartan, VGI, Pleiades, and Pico Radio projects. It is clear that I would not be able to acknowledge everyone. Please know that I am grateful you to you all. Yikes!, I have been here too long.

I would like to thank Vason P. Srini, Adrian J. Isles, and Johnathan M. Reason, for countless hours of stimulating conversation that was often challenging, often inspiring, and at times a bit random. I can only imagine how different my experience would have been in their absence.

And finally I would like to express my gratitude to my parents, immediate family, and extended family for their incredible love throughout the years. This accomplishment would not have been possible without them. Your continued

sacrifice and belief made all the difference. And to Donald Steven Smith, please forgive me for not finishing in time for you to share in this moment. For without you it is quite clear that I would not have reached this milestone.

—Ilah Grande (near Angra do Reis), Brazil, June 17, 2005.

Chapter 1

Introduction

> Engineers participate in the activities which make the resources of nature available in a form beneficial to man and provide systems which will perform optimally and economically. L. M. K. Boelter, 1957.

Digital Signal Processing (DSP) enables the design of systems that continue to find favor in growing proportions across multiple market sectors. From low-cost consumer electronics to high-end super-computing, interest expands largely due to the increasing capacity, diminishing size, favorable economy, and the reliability of very large scale integrated circuit (VLSI) technologies. Applications from wireless communications, multimedia processing, personal digital assistants, manufacturing and control systems, to radio astronomy, naming just a few, all benefit from, and in-turn drive, the growth of the electronic design industry. Today' embedded systems are complex and heterogeneous. In addition to DSP, they

often contain numerous other components of various types. For example, they may additionally contain digital components such as micro processors, memory, application-specific integrated circuits (ASIC), and field programmable gate arrays (FPGA); as well as analog components, such as analog to digital (A/D) and digital to analog (D/A) converters; and increasingly environmental transducer such as sensors and actuators.

The algorithms in contemporary DSP systems are subjected to regular revision as a result of continual optimization and innovation. Competition-driven pressure to deploy on-schedule make it common for these complex systems to be released with flaws, known and unknown, in order to avoid losing valuable market share. The pace of change has been continually eroding product life cycles. In view of piles of discarded obsolete electronic devices, one could argue that a new metric has emerged that distinguishes a solution from another based on it's ability to remain relevant across the greatest window of change. Inasmuch, except for a few high-volume or dedicated or safety-critical systems, ASIC-based solutions are loosing favor to more flexible hardware ones that can be reconfigured in-system to adapt to the changing requirements in order to extend product life[TB01].

For many years DSP has continually benefited from the ever shrinking VLSI dimensions. Improving DSP architecture and algorithms enable new capabilities that allow engineers and computer scientists to design new – faster, smaller, and

cheaper embedded systems. These new systems, due to their advantages, remain in demand and the expanding markets continue to rewards these design efforts with capital. This, in-turn, drives further research and development in the underlying technologies. Over the past decade or so wired and wireless communications has become a key driving factor in the fervent increase in the rate of this expansion.

1.1 Digital Communications Revolution

When electronic systems interconnect to one another the resulting capability is greater than the sum of the parts. As a result, new classes of applications and services emerge[KBP+04]. As these systems become increasingly integrated into the fabric of our environment[KKP99][GHI+04], key successful components—algorithms, protocols, and even entire sub-systems—become standards for future system development[ZSGR01]. Unquestionably, there is continual pressure to innovate while simultaneously maintaining standard compatibility with existing and legacy systems; in order to retain the advantage of connectivity and information exchange. Standards are proliferating.

For some insight, see figure 1.1. It shows several competing and complementary wireless communication standards that have found broad acceptance in the

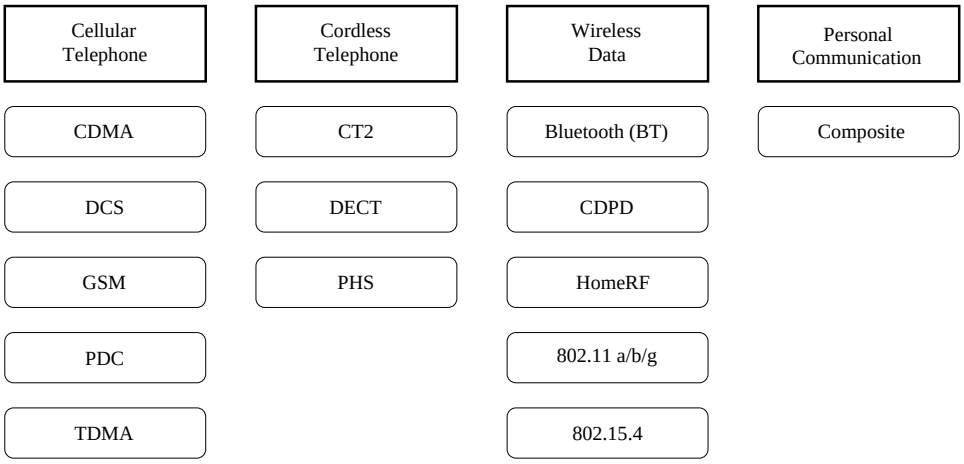


Figure 1.1: Growing body of wireless communication standards

marketplace. Not only are standards proliferating, but are also new embedded systems and devices that materially rely upon such wireless communication standards. Moreover, this is occurring at an increasingly dizzying pace. It appears evident that future devices will increasingly communicate with one another and will need built-in flexibility to adapt to new standards[LB04]. They will require system-level strategies[LSR03] that enables families of continuously changing algorithms to efficiently share flexible reconfigurable hardware resources[SS05].

1.2 Reconfigurables Need New Methodologies

In response to the need to balance efficiency and in-system flexibility, reconfigurable processor based designs[Sut98] are rapidly moving from niche in-

terest and are appearing in mainstream designs[Har04]. This is having a major impact on traditional electronic design automation (EDA) tools and design flows[RVN02] since there are fundamental differences in the approach of system built on reconfigurables from those utilizing general-purpose and application specific processors[Gri04]. Most notably is the additional degrees of freedom accompanying resource configuration management.

For general-purpose[HR97] and application specific processors[Bro92][CPRB03], there has been voluminous research in the areas of compilation and synthesis which yield acceptable results[PKK97]. In both cases the functional operations of processor resources are fixed at fabrication for some target application; either broad general-purpose coverage or limited specific coverage. By contrast, reconfigurable platforms retain post-fabrication programmable structures that must be configured prior to use by a mapped algorithmic computation.

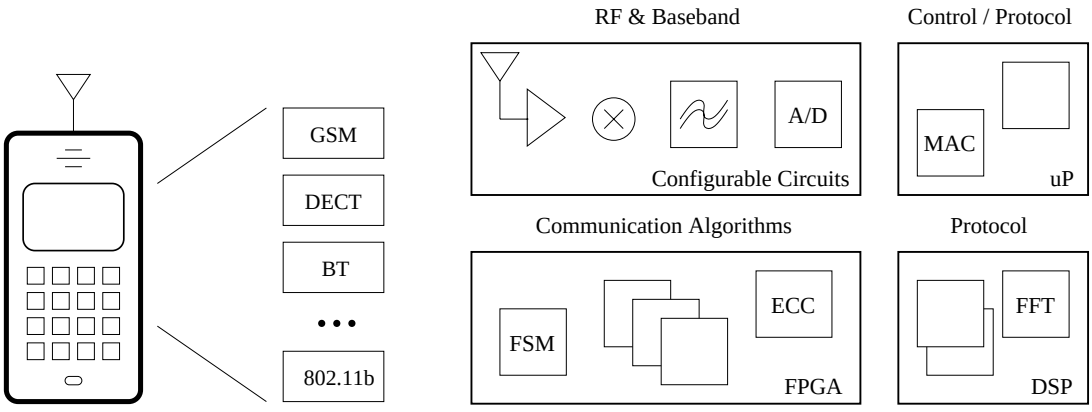
Consider, for example, the complimentary wireless communication standards identified in table 1.1. Shown are one cellular telephone (GSM), one cordless telephone (DECT), and two wireless data (802.11b and Bluetooth) standards. Each operates within different network coverage range, 10x variations in data rates, and differing algorithm computation requirements. A typical user is less concerned about the underlying communication standard details as they are about the ability to use an application within some minimum quality of service (QoS).

Table 1.1: Basic comparison of some wireless communication standards

	Coverage Range	Frequency Range (Mhz)	Multiple Access	Channel Bit Rate
GSM	10Km	Rx:1805-1880 Tx:1710-1785	TDMA/FDMA	270.833 kb/s
DECT	100m	1880-1900	TDMA/FDMA	1.152 Mb/s
802.11b	100m	2401-2462	CSMA/CA	11 Mb/s
Bluetooth	10m	2402-2480	Frequency Hopping	1 Mb/s

Let us assume the application goal of the user is to place a voice telephony call within an environment covered by multiple overlapping communication networks and that each network has sufficient unused capacity and signal reception to meet minimum QoS expectations. Furthermore, the user wishes to first make a connection from a Bluetooth headset to the device, which in-turn, will place the call using either a cellular, cordless, or 802.11/voice-over-IP (VoIP) connection. With a multi-standard communication device, see figure 1.2(a), the user can consider other factors to differentiate and select amongst the three connectivity options. Each standard provides significantly different regions of coverage, network use-cost, and device energy expenditure behavior. Under these conditions, the system can allow the user to choose the method of connectivity based on tertiary objectives (following connectivity and QoS).

Figure 1.2(b) shows a conceptual block-diagram of a reconfigurable radio



(a) Multi-standard device

(b) Reconfigurable radio sub-system

Figure 1.2: Reconfigurables enable flexible yet efficient design

sub-system that might implement the multi-standard wireless communication functionality proposed above. It shows radio frequency and analog baseband processing implemented with configurable circuits, communication operations implemented in an FPGA, lower-level protocol processing in a DSP, and higher-level protocol processing in a general purpose microprocessor. This resulting radio sub-system is configurable and able to implement multiple standards[CG00]... in theory and even in prototype. The challenge comes from the real-world market constraints of cost, reliability, size, energy-efficiency, etc. The design solutions space is large, composed of multiple interdependent NP-complete sub-problems. And, it is not entirely static since even the so-called "standards" occasionally undergo revision.

The designer faces all of the well study issues of hardware/software co-design as

well as a host of new challenging design dimensions[FBK98][BH01] that are derived from reconfigurable hardware, changing algorithms, and dynamic environments [FBK98][NB02]. Consider the following list of a new design issues:

- When there are multiple hardware resources available that could implement a required algorithm function, which one should be chosen? Each choice affects future opportunity.
- When should resources be allocated and/or configured? Just before they are required? Just in time? On demand? After an algorithm function has completed, should its resources remain in reservation or be released for use by other algorithm functions?
- Who and/or what should be responsible for resource configuration? A central authority? A distributed scheme? The algorithm function itself?
- Who should be responsible for observing and managing the response to changing algorithms, such as the switch to a new quality level, and changing environments, such as limited resource and/or reduced connectivity conditions?
- What can be done at design time to limit the overhead cost that accompanies run-time management?

- For a given system resource — reconfigurable architecture, programmable platform, etc. — what are the limits of system adaptability? Will the system degrade gracefully or fail abruptly under changing conditions?

- What hardware architecture features are especially beneficial or detrimental for a given system application domain?

- What is the best method to capture the dynamic nature of the algorithms, stimulus, and heterogeneous reconfigurable architectures during the design process?

- Can there be future-planning or only statistical observation of past performance in the run-time management scheme?

- Given the complexity of the highly-concurrent, multi-application, multi-resource, run-time dynamic systems, how are designs verified for correctness or robustness? What statements about reliability can be made?

- What is a good strategy for accepting new algorithms into and destroying old algorithms from an active system at run-time without affecting disjoint and/or concurrent behavior?

Existing design environments[Wan01] do not go far enough to address these essential questions. New methodologies, design environments, and tools are needed.

1.3 Research Goals and Organization

The objective of this research is to explore the questions raised in the previous section with the intent to propose a new methodology, and base framework that provides computer assisted system-level design for run-time managed reconfigurable digital signal processors in a dynamic, multi-standard, multi-use application space.

Up to this point, this chapter has introduced the area of focus and has motivated the need for this research. In Chapter 2 a few additional system examples are described along with the unifying features that make them relevant to this research. One feature, dynamic change in system algorithms — a key force behind the need for run-time methods — is classified into five forms, each with unique opportunities. The assumed reconfigurables architecture building blocks and the use-costs are detailed. Finally, typical design-flows are described. Chapter 3 lays the foundation for the material contribution of this research; a proposed methodology and design framework that supports system-level design of run-time managed dynamically reconfigurable digital signal processing systems. Along with the methodology and framework, a mathematical model is presented to formalize the design semantics. Chapter 4 advances the methodology from theory to practice by presenting the CAD design environment developed in compliance to

the proposed methodology. Care was taken to implement a modular and extensible design framework rather than a one-off proof-of-concept application (at a significant cost of time). The details of the environment, its based framework, software architecture, and extensibility are discussed. And, a simulator — a key design environment tool that provides quantitative performance evaluation — is detailed. Chapter 5 demonstrates the effectiveness of the methodology exercising the design environment against a database of generated system examples. Additional design tools, developed to generate families of algorithms, architectures, and mappings, are discussed. They are then used to automate the evaluation of the design-flow over numerous structured scenarios. A set of 13 parallel-independent, 15 randomly-interconnected, and 17 serial-parallel connected algorithms are each mapped to 36 generated architectures and simulated with various run-time optimization objectives. In Chapter 6 a multi-user DSP system design is described, explored, and refined to validate and clarify the effectiveness of the methods on a real system design. Chapter 7 concludes this dissertation and highlights some areas of future work. A design environment tutorial is presented in Appendix B that covers some implementation details and additional tools developed during this research effort.

Chapter 2

Reconfigurable Digital Signal Processing Systems

> That which is static and repetitive is boring. That which is dynamic and random is confusing. In between lies art. John A. Locke.

More and more digital signal processing (DSP) systems are making use of reconfigurables [CEL+03] [KBP+04] [ZPG+00] to deal with design-time constraints and post-design uncertainties. In this chapter we will start with a discussion of a few system examples and their common features, including the increasing prevalence of post-design dynamic change in system algorithm computation. We will then consider a classification of this computation change. Next, we will have a look at some reconfigurable architectures. Finally, we will end

this chapter with a discussion of typical design-flows that are used to map system algorithms to these architectures.

2.1 System Examples and Features

In Chapter 1 we described a multi-standard communication device that is able to work in different network environments without the need for dedicated hardware in each; one flexible device is configured as needed. This is by no means an isolated example. There are many system scenarios emerging that similarly benefit from the use of run-time reconfiguration techniques[YJ05][GNVV04].

Consider a network of wireless sensor nodes[PKB99][ECKM04]. These millimeter-sized devices are resource constrained and collaborate to yield aggregate system functions. They may include both sensors, such as temperature, humidity, and movement, and actuators, such as alarms, rheostats, and light controls. Their applications are varied, but a typical use is a smart building, that conserves energy, for example. These battery-powered devices must perform under tight energy constraints to conserve energy and scavenging techniques have been proposed to replenish energy stores from vibration, light, radio frequency, and other environmental sources. The computation performed might include node locationing[SRB01], data compression, radio-link processing[ZSGR01], closed-loop control system, etc.

A system includes thousands of identical nodes that self-organize. Clearly each node will have responsibilities that vary with time.

As a third example, consider an autonomous reconnaissance vehicle that receives command by radio link. Sequences of directives are sent as part of an overall mission objective that are carried out by the vehicle each in turn. A directive might involve vehicle navigation and sampling of environmental elements. Given the autonomous nature of the vehicle, it will need to continuously identify and avoid obstructions. As a final brief example, consider an all-media gateway that is part of a home environment. A broad-band network connection provides an aggregation of various media data; high definition television, digital satellite services, internet data, etc. The gateway provides a distribution and collection point into and out of the home and it performs data transformations and routing for all media encoding standards.

In each of these examples there are vastly different applications, data-rates, and use budgets. There are, however, several commonalities. For starters, each uses digital signal processing as a basis to carry-out their system functions. They also use digital communications and are part of network of systems where information is exchanged in digital form. The communication takes the form of streams of encoded data with well-defined standards for the exchange. Each system is expected to be responsive and efficient, producing results give stimuli within per-

formance constraints in time and energy. Fortunately, the system computations have both spatial and temporal concurrency that can be exploited during design to help meet the performance constraints and efficiency requirements. The systems also are implemented with diverse components such as analog-to-digital converter, application specific integrated circuits, and microprocessors, and use multiple models of computation. And finally, the systems are increasingly being confronted with computation that change over time. In the next section we will spend some time discussing this last point as it is a major driver in the trend towards reconfigurable systems solutions.

2.2 Dynamic Change in Algorithms

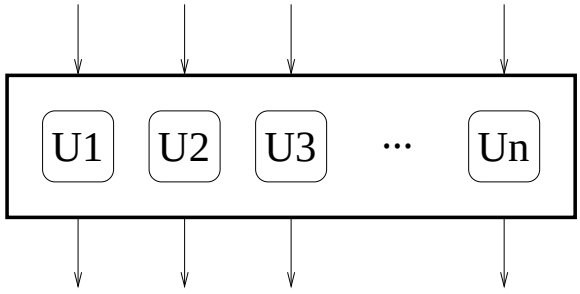
Given the nature of the data they process, DSP computations are inherently multidimensional. Comparisons and transformations in time and space translate into algorithms that are composed of concurrent loops that iterate over groups of operations. This results in computational structures where a relatively large number of operations are performed on each datum. These looping structures are often called "kernels" of computation. As described in the previous section, these computation kernels are beginning to change with increasing frequency. For example, a new directive might restructure the efforts of the autonomous vehicle

that results in an entirely different computational focus. Or the sensor network node might transition from a locationing phase into a data forwarding phase. After studying numerous systems that deal with changing computation, five forms have been identified and are shown in figure 2.1(a)-(d).

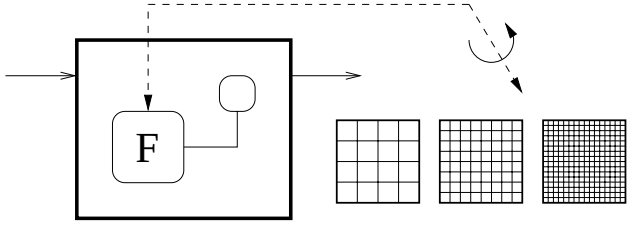
2.2.1 Forms of Change

Shared Use(r): Due to design physical limitations and cost concerns, systems implementations will often combine users and uses on a common set of resources as shown in figure 2.1(a). In doing so, the computations must share resources in time, one after another, and in space, several at a time. A multi-user application is one that has more than one independent computation that are largely identical in structure; each performing the same function for one of the users in the system. A multi-use application has more than one independent computation that are not necessarily similar in structure. In a multi-user scenario, opportunity exists to simply timeshare the configured resources as needed by the data requirements of each user. And as for the multi-use case, more effort is required to prepare the resources prior to each new use.

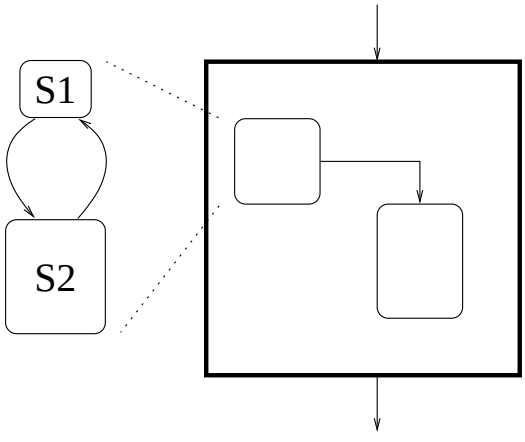
Variable Quality: Often, multimedia systems transition into new environments where the quality of service requirements varies. For example, when the band-



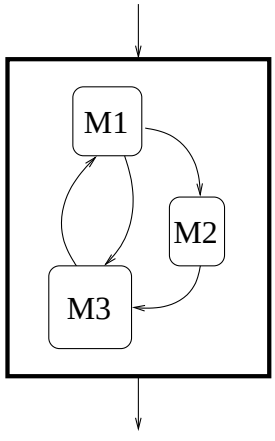
(a) Shared use, shared users



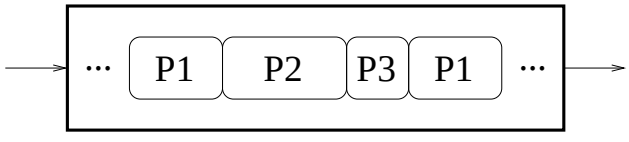
(b) Variable fidelity, quality of service



(c) Multiple standards



(d) Triggered regions



(e) Sequenced regions

Figure 2.1: Forms of dynamic change in algorithms

width of a network becomes restricted, the data-rates will too be limited. Under such a scenario, opportunity exists to adapt the computation effort according to the limited data rate. This adaptation may be beneficial to conserve computational resources for other users (or uses) and/or to conserve other use-constraints, such as energy. Figure 2.1(b) depicts this form of computation change.

Multiple Standards: When information is exchanged between two or more systems, each participant must agree upon a common representation in order to encode and decode the information according to some well-defined standard. Standards are commonly used for data and multi-media representation, policy for collaboration, and digital communications. Standards continually improve as new research produces improved efficiency and higher quality representations. The appropriateness of a particular standard varies with application and environment. For a given application, there usually are many standards that provide similar function, such as a image encoding methods. Increasingly, a system needs to be flexible in order to support such an array of complementary and competing standards that are used in different scenarios. Figure 2.1(c), shows the concept of multi-standard computation change. Standard $s1$ and $s2$ both operate on the same data type (say for example audio data), but use different techniques and may be used interchangeably as governed by the agreements assumed between the collaborative systems.

Triggered Regions: When the computation effort and system control-flow remains within a kernel until the occurrence of some externally triggered event, as depicted in figure 2.1(d), we call this form computation change triggered regions. In some ways, it is similar to the operation of a finite state machine (FSM), except for that in an FSM, the system enters into a state where it remains waiting for the next transition. The triggered region form implies that a kernel continues to utilize some computational behavior while the region is active. Upon some external event, the active kernel becomes inactive and is replaced with a new region of computation (a new kernel); where it remains until the next transition event occurs.

Sequenced Regions: When the computation transitions through a predetermined cycle of kernels in time as depicted in figure 2.1(e), we call this form of computation change sequenced regions. Change of this form arises out of the data and control dependencies in an algorithm structure. A system implementation can use a pipelined approach (using resources for each kernel that operate concurrently) when time performance is important, or can time-share fewer resources, when resource constraints dictate that area conservation is more important. Sequenced regions are essentially a special case of triggered regions where the transitions between kernels are known a priori.

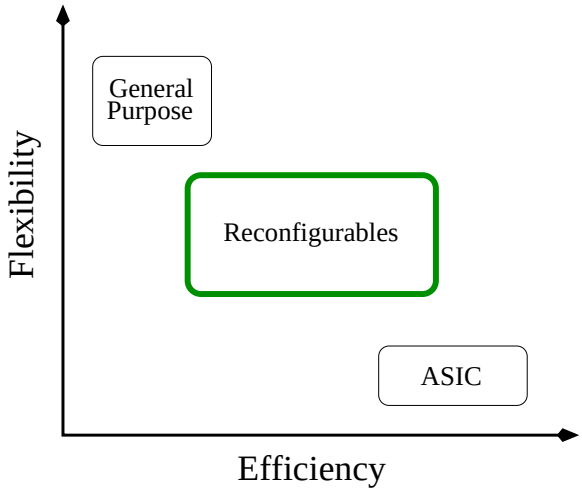


Figure 2.2: Reconfigurables emerge to balance efficiency and flexibility

2.3 Reconfigurable Architectures

Application-specific integrated circuits (ASIC) provide efficient design solutions since custom resources can be assigned as required by the algorithm to exactly meet performance constraints. However, these solutions are inflexible since they are essentially dedicated to specific applications. At the other extreme are general purpose microprocessors. They can perform any computation over time, but must expend both more time and more energy (for a given algorithm) in comparison to its custom counterpart. The trade-off[AZW+02] between efficiency and flexibility for both the ASIC and microprocessor is shown in figure 2.2.

Interest in using reconfigurable platforms for DSP continues to grow due to their ability to strike a balance[ASI+98] between system performance (efficiency)

and flexibility. Commonplace design constraints such as time-to-market pressure, an expanding base of standards for multi-media and digital communications, increasingly adaptive and multi-use systems, shrinking system dimensions, and limited-power source systems, to name a few, all seem to point towards the use of programmable techniques on reconfigurable hardware.

Current integrated circuit technology presents opportunities to conceive and implement elaborate programmable hardware architectures[Har01]; and over the past fifteen years, or so, there have been many[CH02]. Except for when there are clear structural relationships between an algorithm and architecture, it is difficult to know when one programmable architecture is better for a given application than another. Architectures such as programmable DSP, very long instruction word (VLIW)[MD04], multiple instruction multiple data (MIMD)[SSR98], single instruction multiple data (SIMD), field programmable gate arrays (FPGA)[CKR+03], each have been demonstrated to perform well in specific applications. Recent research indicates that heterogeneous architectures composed of reconfigurable modules are well suited at providing both efficiency and flexibility over a domain of algorithms. An architectural template is advocated that can be used to compose hardware architectures given the collection of algorithms in the domain[AZW+02]. Figure 2.3 shows the template[1]. It consists of heterogeneous programmable mod-

[1]The configuration cache memories and configuration controller have been added, proposed by the results of this research, and will be discussed more in section 7.1.

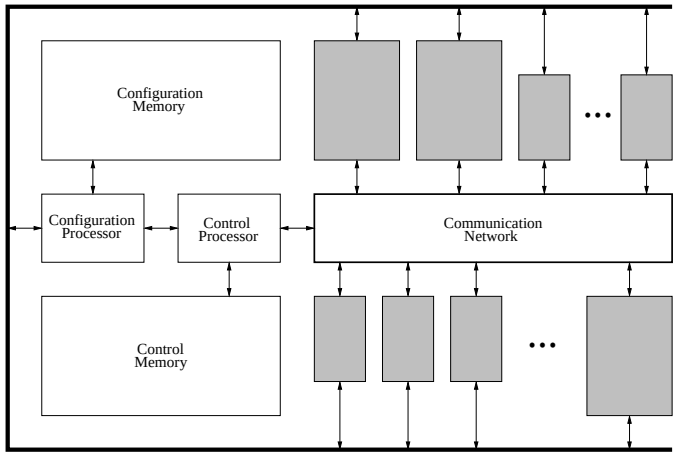


Figure 2.3: Architecture template as a building block

ules, a communication network, a control processor, and memories.

This architecture template is assumed as the basic model for reconfigurable processor composition in this research. We feel that it is appropriate given that the template is general and can be used to create many specific architectures. We further assume that at least a portion of the hardware can be configured dynamically while other portions of the hardware continue with computation.

From the perspective of this research, we are concerned with the run-time reuse of these resources (modules, interconnect, memory) and operate under the following assumptions. (1) A schedulable unit of behavior in an algorithm requires some known collection of these system resources. (2) Some portion of these system resources can be reconfigured dynamically at run-time. (3) All communications between these units of behavior are assignable to interconnect structures such as

system networks and/or buses.

Based on changes in the environment or system-level objectives the assignments of resources are redistributed to efficiently adapt to this change. In essence the trade-off of flexibility and efficiency is done while the system is in operation. To achieve this goal, a method needs to account for the core expenditure costs and overhead costs of a current operating point and any new proposed operating point in order to know which dynamic adaptation is appropriate.

2.3.1 Cost Distribution

Currently, reconfigurable architectures use one of three strategies to configure the programmable hardware resources; either serial scan chains, random addressed, or concurrent scan chains. In each case, time and energy are required to configure the resources. And area must be reserved in the architecture to implement the configuration topology. These costs are overheads that must be considered when adaptation is desired.

Table 2.1 shows a breakdown of the operational cost distribution for run-time reconfigurable systems. In table 2.1(a) is shown the distribution categories. In table 2.1(b) is shown the dimensions for each category element. Algorithms use resources to carry out the computation and communication. The resources must

Table 2.1: System operation cost distribution for reconfigurables

	computation	communication	scheduling
resource execution	core expenditure	core expenditure	overhead
resource configuration	overhead	overhead	overhead

(a) Cost components

time use	energy use	area use

(b) Cost component dimensions

be configured prior to execution for the desired computation. Schedulers are required to control the execution and configuration according to some predefined system policy. Each component minimally has the dimensions of time, energy, and area. Therefore there are eighteen types of system use-costs that can be managed at run-time for each system object. Only the execution of algorithm computation and communication directly contribute to the core expenditures that go towards system behavior. The other components are necessary overheads that must be carefully tracked in order to keep overall system operation costs minimized.

Configuration delays impose a big performance penalty on run-time reconfigurable systems. In response, it has become common for reconfigurables to support partial reconfiguration schemes that allow for a select subset of the architecture to be reconfigured while the others continue in computation (largely) unaffected.

To further help mitigate the cost of configuration, some architectures support difference-based configuration schemes, where only the configuration vector differences are used to change reconfigurable resource behavior. This can significantly reduce the time and energy required to reconfigure when the differences are small. However, it obviously comes at a cost of additional computation required to derive the difference. To reduce the time and energy it takes to move configurations across communication structures, from remote memories for example, configuration caching[LCH00] has also been proposed. The cache can significantly reduce the costs of configuration[DST99] when they exist locally within a cache memory. It comes, however, at the cost of both management and memory area overhead.

A designer may need to consider a combination of module-based, difference-based, and configuration caching to limit overheads when frequent reconfiguration cannot be avoided. And, in any case, the run-time management decisions will need to be aware of these outlined costs in order to optimize system performance.

2.4 Mapping and Design-flow

Given an algorithm and architecture, a mapping is required from the algorithm components to some collection of the architecture resources. Often, the algorithm coding, the architecture specification, and the mapping between them

are collectively optimized using structured exploration that considers incremental changes[CPRB03]. With each change, the performance is evaluated to identify a ranking and to determine if the change is beneficial. Mapping requires both an allocation of resources types and an assignment to specific architecture resource instances. The process is constrained by system performance requirements, available resource types, and available resource instances. Overlapping demand for resources, stemming from behavioral concurrency within the algorithm, further constrains the efforts.

Whenever performance (use-cost limitations) is a factor, a design flow always begins with a characterization of the algorithm to identify the parts with the most stringent requirements[Wan01]. This is defined by the amount of computation that must be completed within some specified performance constraints associated with the computation. Typically, performance constraints will be associated with collections of behavior and therefore the notion of a use-budget emerges. The budget identifies aggregate expenditure for the collection. When this budget is exceeded, new allocation and assignments must be considered to reduce the aggregate expenditures. And when there are savings (the budget has excess), allocations and assignments can be reconsidered to relax the performance so that other constrained mappings can benefit by making use of the excess. Once a mapping is chosen that meets all of the constraints, the configuration bit-streams can

be generated for the architecture modules and interconnect.

This is the typical design-flow that yields optimized mappings of applications for reconfigurable architectures. It works well to produce efficient designed-time mappings. However, it is not well suited to deal with changing computation that is increasingly found in DSP systems[SS05]. For this, new mapping schemes and new design methodologies are required. Moreover, they need to support efficient run-time mappings with limited overheads. This is the prime focus of this research effort.

2.5 Chapter Summary

In this chapter we have taken a look at DSP system examples and the common features, that merit the use of run-time managed design methods. Next, we propose a classification, along with the corresponding opportunities, for the forms of computation change observed in systems. We then discuss reconfigurable architectures from the perspective of a run-time manager. We note that dynamic reconfigurables enable run-time trade-offs between flexibility and efficiency. We then present a breakdown of the use-costs associated with these architectures. Subsequently, the architecture template assumed throughout this research is described. And finally we conclude by discussing the typical design-flows used to

map algorithms to architectures and discuss why new methodologies are required.

Chapter 3

Templated-Mapping Design Methodology

> ...templates exploit the expertise of two distinct groups. The expert numerical analyst creates a template reflecting in-depth knowledge of a specific numerical technique. The computational scientist then provides "value-added" capability to the general template description, customizing it for specific contexts or applications needs. Jack Dongarra,"Templates for the Solution of Linear Systems" 1995.

System designs that utilize digital processors all follow a similar overall flow that includes specification, algorithm selection, algorithm coding, hardware selection and/or design, coding-to-hardware mapping, and scheduling. Each of these design stages are interdependent and are often iteratively optimized in-concert as well as in part. During iterative design refinement, implementations are

analyzed against their specification according to key metrics to establish ranking. During mapping, the codes of the algorithm are translated into forms that are meaningful in context of the hardware architecture.

In this chapter we will introduce algorithm-to-architecture mapping and will defined some terms that are subsequently used to delineate when mapping is performed. We then discuss mapping with templates,[1] a method proposed by this research to limit the overhead associated with run-time mapping schemes. Next we describe the concept of candidates, equivalent implementations for some system behavior. We then define modes that are used to guide run-time decisions based on designer expectations of system behavior. We will then discuss a new mapping paradigm, proposed by this methodology, and define when, mappings are constructed, and how they are later used. We will discuss how properties are used to distinguish between alternative candidate implementations and enable systems schedulers to make optimized decisions. Finally a formal set-based model is defined along with the requirements for execution control.

[1]Referred to throughout this text as "templated-mapping"

3.1 Algorithm Mapping

The goal of an algorithm coding is to unambiguously capture the intended system behavior. Many algorithm coding languages [BL02] [PMC03] [GZD+00] [EL03] and schemes have been proposed and developed, each having a particular area of strength[LZ05]. Most practical models for digital processor design are based on communicating sequential processes [Hoa78] where the algorithm is divided into units of sequential computation that operate concurrently and communicate with one another. This representation lends itself to graph-based analysis[Kri98] [HR97] with the computation represented as nodes and the communication represented as edges. The target hardware architecture is the primary factor that establishes which coding scheme and model of computation forms the best fit. For example, languages that code algorithms as procedures of sequential operations are a good fit for general purpose processor that executes in a similar load-execute-store sequential fashion. However, for architectures that support multiple concurrent operations, languages and schemes that expose algorithm concurrency are desired since there is a more natural fit with the multiple resources of the architecture.

Although designers always have the choice to implement algorithms directly using the native machine codes of the hardware architecture, this is a very time

consuming and error prone method. And, due to time-to-market pressures and rising algorithm complexity, this is feasible only under very limited circumstances. Contemporary systems employ increasingly sophisticated algorithms and depend heavily on high-level language constructs. These descriptions therefore must be converted into a form suitable for execution on hardware. This conversion process is at the heart of what mapping is all about.

Something as seemingly simple as the addition of two scalar operands takes on many different forms based on a selected architecture and mapping style. For example, the numbers must be converted into a representation that is appropriate for the architecture. If the operands are larger than that supported by the architectures maximum word size then the addition must be distributed over multiple operations. Storage must be reserved for the operands and result. The appropriate hardware resource that can implement the addition must be assigned. And, if this resource is a shared by other portions of the algorithm, then it must be scheduled whenever there is concurrent demand.

The computational effort required to map dynamic algorithms to reconfigurable architectures is significant and finding optimal solutions are nontrivial. Therefore designers resort to computer aided design, heuristics, and iterative methods to explore the design space for acceptable solutions. The desired result of this design effort is an implementation that provides correct function, meets all the

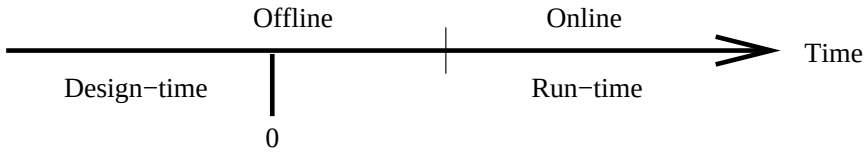


Figure 3.1: System mapping time-line

required system constraints, and performs satisfactorily according to some metrics of interest (which typically relate to the constraints). When there is a change in the intended system behavior, more specifically, it's algorithm, the entire process must be repeated starting with refining the algorithm down to once again iterating over the design space. Therefore, for systems with changing algorithms, this design process is highly undesirable.

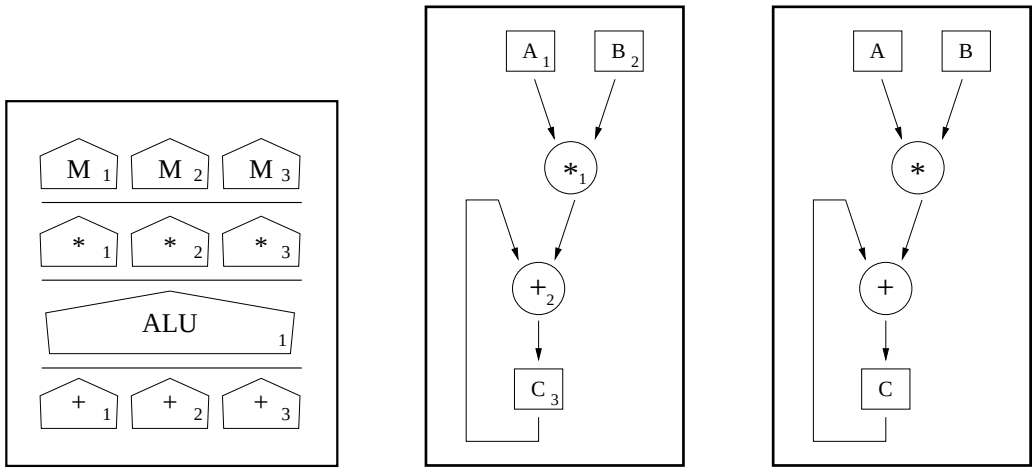
Clearly a more flexible approach is required that will allow systems to adapt to the emerging frequent changes that has been described in section 2.2. To discus the desired changes in design-flow, and proposed the template-based mapping, some distinction is required regarding definitions of time. From the perspective of system design completion there are three distinct time phases. See figure 3.1. So far in the previous paragraphs as we have talked exclusively about design-time. When a design has been completed and placed in-service we call this time "time zero." Afterwards, we further make the distinction between offline and online, where online is the time at which the system begins to perform its intended function. We also called this time run-time interchangeably. The time between

post-design and run-time we call off-line. For example, a system that has been placed in-service but is not performing computations on the system algorithm is said to be operating off-line. This time can be used to perform system optimization and analysis.

3.2 Mapping with Templates

To manage the post-design change observed in contemporary algorithms new approaches are required[TB01]. Since it is prohibitive and inefficient to reserve dedicated hardware to handle post-design changes in resource demands, new approaches must manage post-design resource remapping and reconfiguration. This remapping can be done online, offline, or some combination of the two.

To limit interruptions in system operation, it would be preferred to perform the remapping online. However, this come with significant overheads. The computational cost of translating algorithmic coding descriptions into architecture mappings is high. Moreover, contemporary design flows are exploratory/interactive and therefore difficult to embedded within a system design. Additionally, for resource limited devices, this design scheme would be impractical. Therefore it is desired to move as much of the design process as possible into that region prior to placing the system in service (time zero) with only minimal online computation



(a) Resources (b) Bound (c) Template

Figure 3.2: Templated-mapping introduction

required for system adaptation. Towards that goal, a templated-mapping scheme is proposed.

3.2.1 Templated-Mapping

In a templated-mapping, resource "types" allocations are identified but particular instances are not. In this manner a mapping is specified using its form, or by "template." It is a partial mapping in that resource instance binding is postponed. By specifying a mapping in form, resource allocation, configuration, and binding can be optimized in post-design time. Moreover, the template can be reused to reassign resources under changing load conditions.

To illustrate the concept of templated-mapping see figure 3.2 which shows re-

sources and two ways of mapping the algorithm code fragment $C+ = A * B$ to the resources. In 3.2(a) is the set of system resources. They consist of three dedicated adders, a single configurable arithmetic logic unit (ALU), three dedicated multipliers, three memory blocks, and an interconnect network.

In figure 3.2(b) is a mapping that assigns each operand and operation to specific system resources. Each time this algorithm fragment is ready for execution, these specified resources will be required. If, at that time, any of these specified resources were unavailable, then this algorithm fragment would be blocked from execution and would have to wait until they were available.

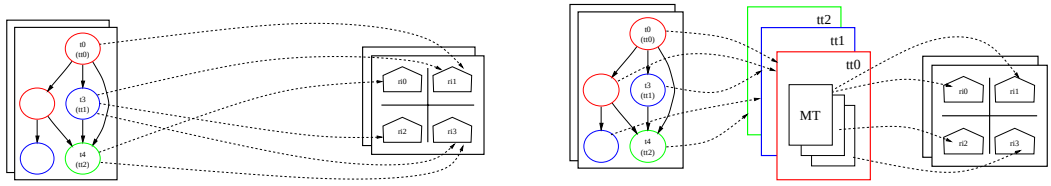
Alternatively, figure 3.2(c) shows a templated-mapping. The mapping is identical except for the fact that no resource instances are directly specified. The advantage of a mapping template is that resource binding and configuration occurs when needed and therefore if a particular resource instance is busy, at some point in time, then an alternative resource instance can be considered. The disadvantage is the overhead required to perform the run-time template instantiation.

3.2.2 Candidates

Mapping templates allow run-time flexibility by postponing resource instance allocation, configuration, and binding. When a mapping is required, a scheduler

can identify appropriate resources and perform these operations according to a management policy. The flexibility offered by mapping templates is significantly increased when multiple templates are available for a given portion of an algorithm. The template shown in figure 3.2(c) and discussed in the previous section gives a mapping form. Other forms are often possible given the system resource set. Depending on how the computation is scheduled and the selected mapping-architecture, it could take one cycle, in a full parallel direct-mapping implementation, or multiple cycles, using an iterative approach and fewer resources[ZPTB99]. If the ALU implements both multiplication and addition, then it alone, in conjunction with the memory references, could implement the computation with an appropriate control sequence.

We call the set of one of more templated-mapping implementation alternatives "candidates." By definition, each mapping templates has identical input-to-output behavior. In other words they are behaviorally equivalent transformations of the algorithm coding similar to that described in [RCHP91]. The term candidate has been chosen since any of the templates from the set may be considered at run-time, and one is selected as needed. By combining candidates and mapping templates, both resource instances and groups of resource types can be reassigned as needed with limited run-time overhead. Resources can be reassigned as needed given dynamic demands and mapping form alternatives can be considered under



(a) Fixed (b) Flexible

Figure 3.3: Mapping algorithms to reconfigurable architectures

dynamic system-level constraints and objectives.

Figure 3.3 illustrates the concept of multiple mapping templates. In 3.3(a) the nodes of the algorithm are directly mapped to resources and therefore are fixed and inflexible. By contrast, figure 3.3(b) represents the same algorithm computation with candidate mapping templates. This strategy allows for flexible post-design reassignment of resources while keeping the overhead of remapping within reasonable quantifiable limits. The strategy comes with two types of overhead. Computational overhead, since the alternative partially-mapped implementations require management and storage overhead, since the template form (implementation) must be available. It should be obvious that a tradeoff in flexibility and overhead is present. By increasing the number of candidates mapping templates, greater implementation alternatives are available for in-system consideration — increased flexibility. However this comes with cost increases in both computational management[2] and templates storage.

[2]Having a greater number of mapping templates increases the computation required to search

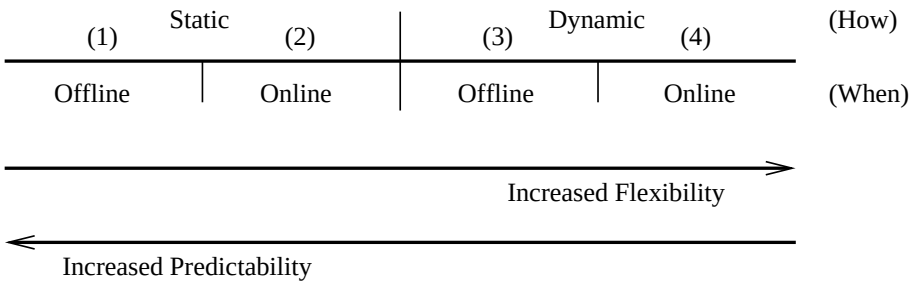


Figure 3.4: System mapping modes

3.2.3 Mapping Modes

Many system designs have absolute fixed limits on cost expenditures for certain behavioral components. These costs limits can be in the use of time, energy, and/or area. For example a deadline establishes a time expenditure limit for some identified section of the algorithm. Limits often are further distinguished using the terms "hard" and "soft." A hard limit, such as a hard real-time deadline, indicates that if the expenditure is exceeded, then the system has failed. Whereas, soft limits are expenditure guidelines that indicate how a "good" system is to perform under normal circumstances. Exceeding soft limits does not necessarily indicate system failure, but it does indicate degraded performance. With the increasing pervasiveness of fixed-budget reconfigurable systems, the specification and control of cost expenditure limits in energy and area becomes increasingly desirable (in addition to time).

and select amongst the candidates.

To address the increased uncertainty that arises from run-time dynamic mapping, a mapping mode scheme is proposed. The aim of the mode is to allow for trade-offs in flexibility and predictability. Therefore, in circumstances where there are no limits on cost expenditure, more flexibility can be allowed. And in circumstances where there are limits, more predictability can be controlled. Figure 3.4 shows the four types of mapping modes. A mode establishes both when a mapping is to occur and how the mapping-binding is retained afterwards. A mapping can occur either online or offline and can be retained after its use — statically, or disposed after its use — dynamically.

Modes provide for increased control over critical algorithm mappings with cost budgets. There are two types of mappings that can be controlled with modes, templated allocation selection and resource instance binding. These two types are discussed in the following sections.

Template Selection

Each algorithm node and edge that is mapped to the architecture using a template is assigned one of four template selection mapping modes. (1) Templated selection mapping modes that are offline and static are pre-selected prior to run-time and this selection is maintained throughout the life of node or edge. (2) Those that are offline and dynamic are also pre-selected. However, after the first

invocation, the selection is released and therefore must be re-established prior to subsequent use. (3) Templated selection mapping modes that are online and static are selected just once, prior to the first invocation, and are subsequently kept for future invocations. (4) And finally, those that are online and dynamic offer the greatest flexibility (and greatest overhead) since with each invocation a selection must be made from amongst the candidate mapping templates. In this manner continual adaptation—to changes in system resource load—is possible with each invocation.

Resource Binding, Reservation, and Configuration

Each mapping template has one or more references to system resources. Each resource reference is assigned a mode that indicates when the resource is reserved and how long the reservation remains valid. We call this mode the resource reservation mode. As with template selection, there are four possible values to differentiate between increased flexibility and increased predictability. (1) Resource reservations that are assigned the mode offline–static provide the greatest predictability since their resources are guaranteed to be ready each time they are required by the mapping template. The reservation is static, and therefore cannot be used by other mappings. (2) The reservation mode online and static indicates that the reservations occurs just prior to the first template use and are statically

retained thereafter. This scenario works well when there is a need to accommodate dynamic algorithm change while simultaneously limiting cost expenditures. (3) Those that are offline and dynamic provide for predictability in the initial reservation, however subsequent to use, the reservation is freed and therefore may be used by other mappings. This configuration is a good match for applications that desire low overhead startup since initially resources are guaranteed to be available during initialization and can be subsequently freed as the algorithm transitions out of this system phase. (4) And finally, the greatest flexibility is offered by the online and dynamic reservation of mapping template resources.

Mode Use Scenarios

For the case of a single algorithm element with multiple mapping templates, each having a single resource reference, there are sixteen[3] possible combined mode configurations that results from the candidate selection mode together with the resource reservation mode. Based on algorithm behavior, system stimulus (data patterns), and resource characteristics — such as event duty cycle, frequency, cost constraints, configuration time, concurrent demands, etc. — an appropriate mode that balances the required predictability, for a single mapping, and flexibility for system-level resource re-mapping, can be explored.

[3]Most realistic mappings will have significantly more since with each additional resource reference the total possible modes increases by a factor of four.

For the case of a critical component that has a tight budget, high duty cycle, and mappings to heavily loaded resources, more rigid mapping modes would be prescribed. And those components with lower duties cycles and fewer constraints are prescribed more adaptive and flexible modes.

3.2.4 Mapping Paradigm

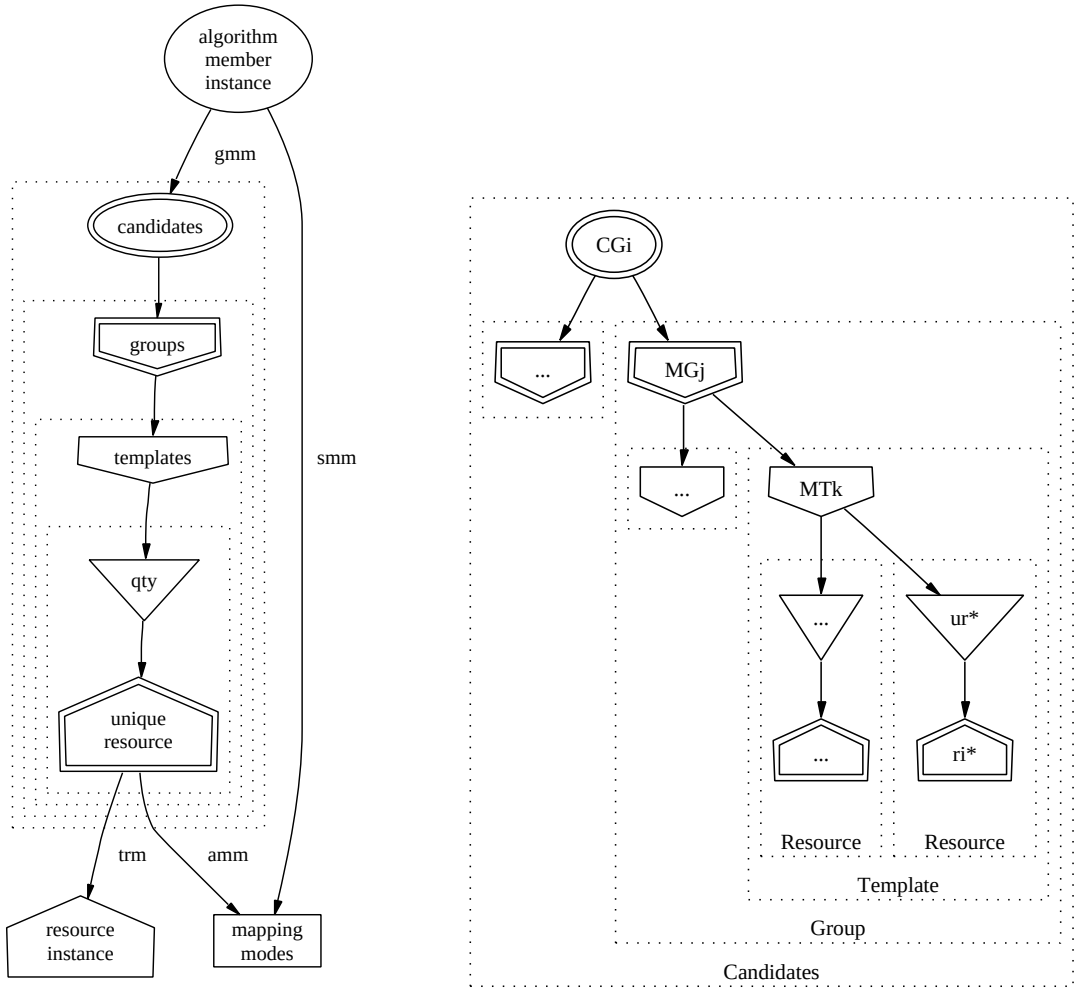
For a given system, the algorithm describes the operations to perform on the system data. This data is transformed by the system as governed by the algorithmically codes behavior. An algorithm however, is merely a description — or coding — of the desired data manipulation and control-flow behavior. The actual computations are carried out by integrated circuit functions and interconnect. The hardware circuits are designed with the application in mind such that appropriate functions and interconnect is available in the hardware architecture. These components, various functions and interconnects, make up what is called the resources of the architecture — named such since they are available for algorithm computation. Mapping is the merging of algorithm coding to architecture resources. As previously described, this mapping problem is usually non-trivial and requires significant effort. Therefore, designers resort to CAD tools that transform system models. Moreover, these CAD algorithms themselves require significant computation.

A driving goal of this research is the need for dynamic resource management that enables system adaptations and run-time re-optimization under changing algorithm and system stimulus conditions. To feasibly support this, most of (or as much as possible) the mapping design problem needs to be completed prior to system deployment. In the previous sections, a templated mapping scheme has been presented that establishes a framework toward this objective. In this and the subsequent section we move toward formalization of this scheme.

Mapping Kernels and Communication

When behaviors are reused regularly throughout the algorithm, the behavior can be identified and used as a building block for algorithm composition. In this manner, these unique algorithm components can be mapped once and reused as required. Mapping occurs for two types of algorithm constructs; algorithm nodes and algorithm edges. Nodes ultimately are mapped to integrated circuit functions that provide the required operations to implement the coded behavior represented by the node. Edges are mapped to interconnect that provides communication between functions. For an algorithm graph, each node and edge instance requires a mapping.

With template-based mapping, each component of the algorithm graph is mapped either directly or indirectly to a group of potential hardware implemen-



(a) Levels

(b) Structural view with model identifiers

Figure 3.5: Mapping with sets of templated-mappings

tations. Recurring algorithm behaviors are mapped to unique behavior representations, called computation kernels. The levels of indirection and the structure of candidate mapping templates are shown in figure 3.5. Figure 3.5(a) shows the levels between an algorithm instance and an architecture resource instance and the relations in between. Figure 3.5(b) gives a view of the structural organization of mapping templates. Essentially, templates are organized in a three-level tree with level one being the tree roots, or the base reference for a candidate group. At the second level is one or more mapping groups. And finally, one or more mapping templates are tree-descendants of each group.

3.3 Offline Template Construction

Templated-mapping design requires a set of mapping templates, or candidates, to be constructed for each schedulable algorithm component. During this design phase, algorithm descriptions, mappings, and architectures are iteratively refined together. The features of the system computation and data patterns are carefully analyzed to extract characterizations that can be used to improve upon the performance of the design implementation. After the hardware architecture is selected, template construction can proceed. For a given architecture and algorithm, the set of mapping templates is fixed. And, although alternative algorithms (to-

tally new/different system behavior) can be map to the architecture, this requires additional analysis, new mapping templates, and a relevant scheduling scheme.

In short, template construction involves analysis of algorithm features, the design and organization of multiple implementations, and the deferral of implementation resource type allocation and resource instance binding.

3.3.1 Algorithm Analysis

Often there are numerous algorithm codings to implement a desired DSP algorithm behavior. The algorithms can have significant difference in cost performance as shown in figure 3.6. Research CAD tools, one that performs behavioral transformations [RCHP91] and another that implements a give system description [Bro92], were used to synthesize ASICs for nine DCT algorithms under constant throughput and constraints[PKK97]. Normalized design area and normalized design power is presented in the figure. The upper chart shows the designs resulting for high-throughput constraint designs and the lower chart shows those resulting from low-throughput constraints. As can be seen in the high-throughput cases, different codings for the DCT algorithm can have a difference of over 24x in area requirement between the best and worst case. Ignoring the direct mapped implementation, the difference is still 7x. Similar observations can be made for energy use-costs. A more pertinent observation from the perspective of dynamic

reconfigurable computing is the difference between the high throughput and low throughput scenarios. Although the normalized charts do not depict the absolute magnitude differences, there clearly is a dramatic decrease in cost requirements between the two. This gap represents an opportunity for run-time resource redistribution under changing requirements if implemented on a reconfigurable architecture.

Algorithm analysis has been heavily studied over the past four decades; especially so for sequential [LA93][HR97] and parallel [Kri98] "programs". The general goal is to characterize algorithm coding behavior against a specific architecture. Recently, approaches that use sub-parts of the reconfigurable architecture resource to monitor other parts where system behavior has been mapped have been proposed[SC04]. In this manner direct measurement of algorithm performance is possible.

For architectures designed for a target algorithm, optimization can be applied to both the algorithm coding and architecture specification concurrently during design exploration. Understanding the algorithm basic blocks, block invocations frequencies, and block-to-block communication patterns is critical in crafting efficient architecture mappings. Moreover, understanding the architecture — such as its instruction set, word width, and memory architecture — is important when coding or refining the algorithm. To understand these behaviors, designers rely on

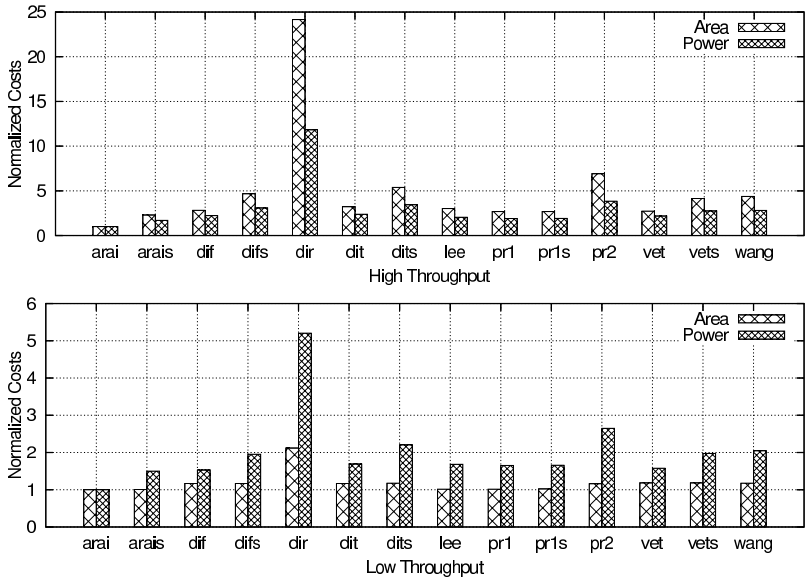


Figure 3.6: ASIC cost variation for DCT algorithms

static and dynamic algorithm analysis techniques like those presented in [Wan01] or [JHK+05], for example. In general, these processes are referred to as algorithm profiling. Static techniques are typically used to characterize the basic blocks in terms of their operations, control, and data sets, to access relative complexities and computation effort distribution across basic blocks. Dynamic analysis techniques provide for characterizations of block-to-block interactions, data dependent control flow, invocation frequencies, etc. One of the more useful results from dynamic algorithm analysis is a view of the basic block duty cycles. When these duty cycles are combined with basic block cost requirements (time, energy, etc.), an architecture-independent estimate of algorithm performance can be obtained.

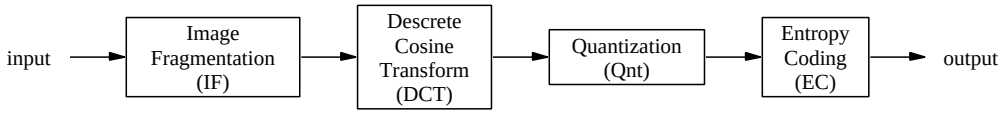
This analysis can be used as a road-map during the refinement of the algorithm, mappings, and architecture.

For a general purpose processor (GPP) or general purpose digital signal processor (GP-DSP), where and instruction set simulator (ISS) is likely to exists, cycle-accurate results of algorithm performance can be obtained by direct measurement of execution cycles from within the ISS, like in the example [CPC+99]. Using this method, the algorithm is iteratively refined, compiled and analyzed, until acceptable results are obtained. Often this "software" estimation technique is used as a base metric to characterize algorithms even when the ultimate target is a custom or mixed hw/sw architecture. This is the case for the design example presented in Chapter 6.

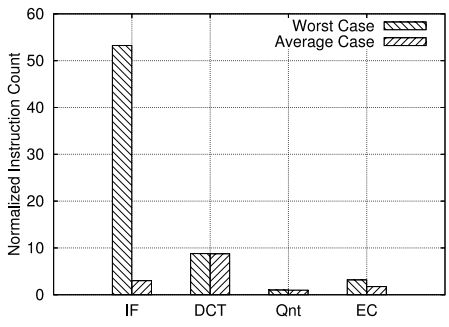
As an example, consider the JPEG encoder algorithm depicted in figure 3.7. The algorithm's block diagram, static analysis chart, and dynamic analysis chart is shown. The algorithm is a public domain implementation optimized for a GP-DSP. The algorithm is divided into four blocks; image fragmentation, discrete cosine transform, quantization, and entropy coding. The estimates are from a DSP56600 instruction set simulator with an image composed of 180 blocks of 8x8 pixels [CPC+99].

The static analysis[4] shows chart, figure 3.7(b), both the worst case instruc-

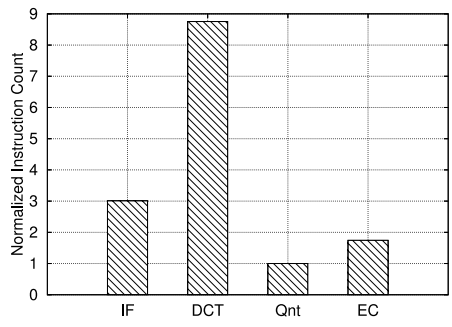
[4]This analysis is obtained from a cycle accurate instruction set simulator. The average case



(a) Root-level block diagram



(b) Static analysis



(c) Dynamic analysis

Figure 3.7: JPEG encoder algorithm example

tion count and average case instruction count for each iteration of the algorithm blocks. From this we see over a 50x difference in instruction counts across the blocks. Notice that for the image fragmentation block that there is a 17x difference in worst case and average case counts. This illustrates that for certain data dependent behavior there can be significant difference in computation costs based on system stimulus and that performance estimation based solely on worst case performance can lead to overly pessimistic results.

The dynamic analysis chart, figure 3.7(c), gives a clear picture of the domi-

is actual performance measured from real stimulus. The worst case, although also measured, can be estimated by summing — over each instruction of the algorithm coding — the cycles required to execute each instruction (CPI) on the target architecture.

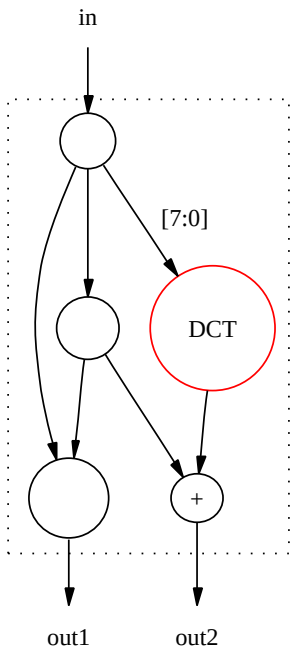
nant runtime behavior. Although the DCT has 5x less instructions than IF, it contributes 3x more to the total accumulated system instruction count. Therefore it should be considered a top priority for optimizations and algorithm coding, mappings, and architecture constructs.

3.3.2 Implementation Alternatives

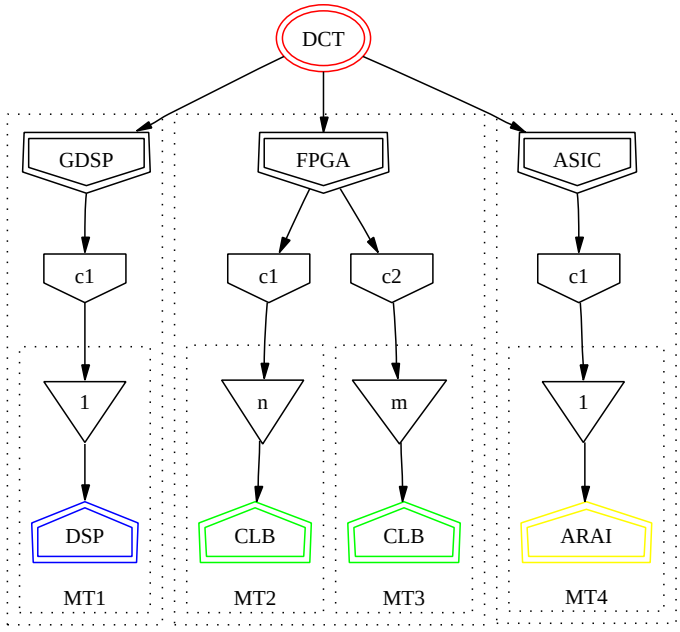
To achieve low overhead run-time algorithm mapping, the set of mapping templates are constructed during design-time. For algorithm behaviors that have loose constraints on cost-performance, such as background or auxiliary system behaviors, opportunity exits in the offline region for construction. This set of candidates subsequently can be flexibly interchanged during run-time optimization. Traditional design flows optimize cost contributors on critical design paths. For example, for a timing critical constraint, the computation and communication path between the input and output — and the resources to which they are mapped — are refined to minimize these critical cost components. For run-time reconfigurable systems these critical paths are no longer fixed and therefore fluctuate with changes in the algorithm, stimulus, and architecture mappings. Therefore it is advantageous to have numerous mapping implementations that span the area–energy–delay designed space that can be selected appropriately at run-time to cope with the temporal system changes.

Each implementation alternative is constructed according to an appropriate design methodology for the model of computation, algorithm coding, and target architecture resource. They may be generated with computer assistance — such as compiled or synthesized, or may be manually constructed. In either case, the mapping is templated and not bound to the particular architecture resources. Any unused instance of the resources type required by the template may be used. What results is a deferred resource binding scheme.

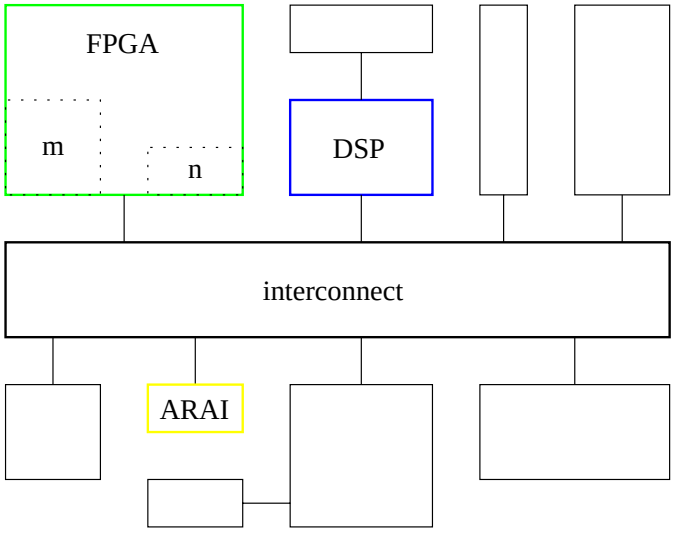
As an example, consider figure 3.8. In 3.8(a) an instantiated graph from the system algorithm is shown. It has a single input and two outputs with a composition that is divided into five parts — represented as graph nodes — each having varying complexity. The most costly kernel has been identified and label DCT. Numerous algorithm coding optimizations and architectural transformations are considered for the DCT, like that discussed in section 3.3.1. Due to system constraints and architecture resources, shown in figure 3.8(c), four implementations are selected. (1) One that uses a "software" compiled implementation for a general purpose DSP. (2) Another that uses an performance-optimized FPGA design flow. (3) A third using the same FPGA flow to yield an area-optimized implementation. (4) And finally, a "silicon compiler" design flow is used to synthesize an ASIC implementation. Each implementation is templated, by deferring the resource instance binding, and becomes a candidate mapping for the DCT algorithm



(a) Graph instance



(b) Candidate mapping templates for DCT kernel



(c) System hardware architecture

Figure 3.8: Algorithm graph instantiation and node implementation alternatives

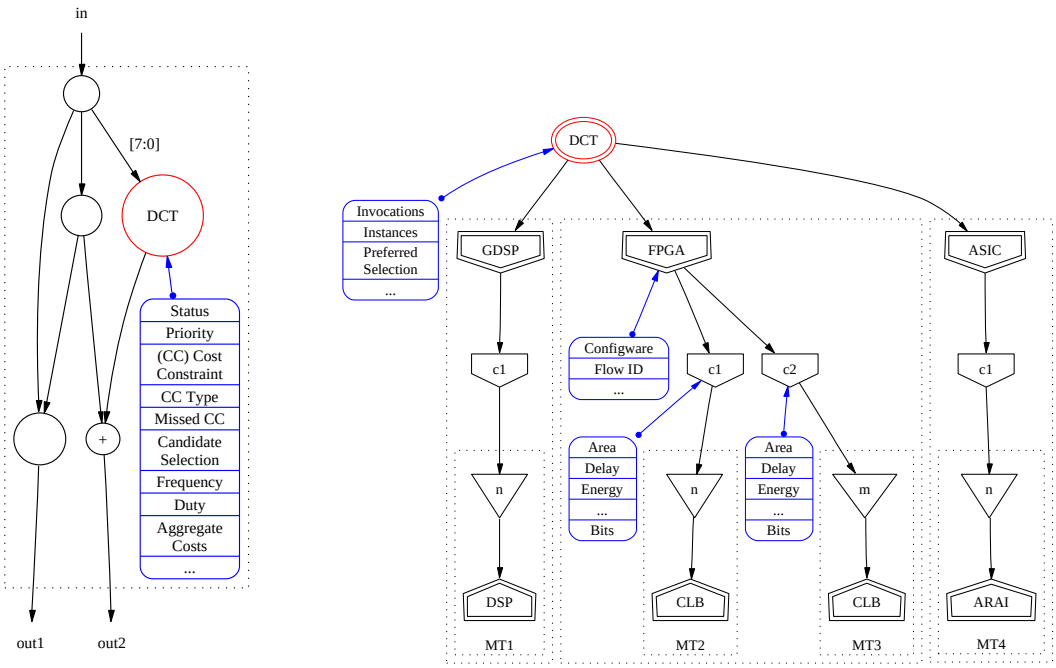
kernel. The Candidate mapping templates are show in figure 3.8(b)

3.3.3 Run-Time Management Properties

Dynamically adaptive systems require flexible run-time management strategies. Mapping templates provide a flexible framework for organizing implementation alternatives that can be appropriately chosen at run-time. The management of these alternatives adds to the existing responsibility of run-time scheduling. One solution for scheduling would be to make arbitrary choices to resolve execution partial orderings[5]. This, for obvious reasons, is not desirable under almost all circumstances. It is preferred that the scheduler make informed choices at each step. This can only be achieved if the scheduler is able to discern relevant information about the objects that are part of the scheduling action. This information can be organized using system object attributes or property assertions.

Any run-time scheduling scheme that aims to make optimized scheduling choices requires property assertion for each participating object. For example to discern priority amongst algorithm behavioral nodes, a priority value can be asserted for each. Similarly, if two new algorithm graphs instances dynamically request admittance concurrently, then property about each could be asserted to distinguish between them. Others system objects that benefit from property as-

[5]We in fact explore this strategy of random ordering in Chapter 5 and Chapter 6.



(a) Node: DCT kernel

(b) Candidates for DCT

Figure 3.9: Properties for run-time management

sertions are graphs edges, candidate mapping templates, computational resources, interconnect resources, etc.

Figure 3.9 revisits the instance graph and candidates shown in figure 3.8 with an example set of properties that could be used by a scheduler in a run-time management scheme. Some common property assertions classes for system objects are: (1) object state, (2) object parameters, (3) object use cost estimates, and (4) object use measurements or metrics.

The properties attached to the graph node 'DCT" in figure 3.9(a) represents the list of properties most often used in this research. The properties fall into three general categories: (1) those that establish relative importance, such as priority and age; (2) those that specify use-constraints associated with the algorithm node, such as cost-constraint and cost-constraints type; and (3) those that annotate statistical performance history, such as aggregate costs, missed cost-constraints, and computation duty-cycle. The candidate mapping templates shown in figure 3.9(b) has example properties that: establish a relative importance, such as number of referring instances and preferred mapping template selection; annotate design information, such as vendor ID and configuration bit-stream memory location; and mapping template cost-estimations, such as area, delay, and energy. For cost-constrained systems, cost estimations are of particular significance since they are used to guide run-time decision-making. Section 3.3.4 examines this subject a bit

more.

3.3.4 Implementation Cost Estimations

Strategies for estimating implementation costs are a subject of ongoing research. As the complexity of systems increase so does the importance of fast and accurate cost estimation. The templated mapping framework presented in this chapter makes no assumption about model accuracies. During the beginning stages of design, "conceptual-level" modeling can be used. In this case, high-level estimations of templates implementation costs can be obtained using techniques such as those presented in [BRL97]. General trends of system behavior can be observed and key cost components identified. As the system description is refined toward actual implementation, models of increasing detail level are chosen for particular algorithms, target architectures, and implementation mappings.

For heterogeneous systems, with mixed models of computation and model refinement levels, no single unified estimation strategy currently exists that works equally well across the diverse computation domains. More accurate results are obtained when domain-specific[6] estimation strategies are employed. For example, efficient estimation with reasonable accuracies (8%-20%) has been reported

[6]In this context, domain-specific refers to specific combinations of algorithms, models of computation, architectures, and system descriptions.

for ASIC adder implementations [OZD+03]. For more complex algorithm constructs with behavior described in System-C, [GLMS02] proposes a rapid estimation scheme. For FPGA targets, [BFS04] proposes an efficient scheme for rapid area estimation that does not require the timely traversal of vendor-specific technology mapping tool-flow. And as a final example, consider the GPP or GP-DSP. These architectures have large-grain complex structures with multi-stage pipelines, multi-level caches, complex interrupt strategies, that makes accurate estimation particularly challenging. However, [NR95] discusses these issues and presents effective methods for efficient estimation. When multiple GPPs are required, estimation strategies such as [HAA+96] exists.

This is by no means meant to provide a comprehensive survey of the work on cost estimation techniques. The goal here is to simply underscore the point that diverse techniques exist and, more importantly, modern system design flows need be a manifold for diverse schemes. Design implementation cost "estimation abstracts" recorded as mapping template property assertions, allow diverse estimations schemes to be unified in an internal representation which may in turn provide context for run-time optimization strategies.

3.4 Run-Time Template Use

Candidates provide multiple implementation alternatives. The alternatives enable flexibility by allowing run-time selection of appropriate implementations. This flexibility strategy creates an obligatory run-time selection requirement. It becomes an additional responsibility of system scheduling. Here we discuss the orthogonal issue of mapping templates selection and resource binding.

Algorithm components for computation and communication have one or more architecture mapping implementation alternatives. At some point prior to execution a specific implementation must be selected, its resources allocated, configured, and bound. Dependent on the design strategy, selection mode (section 3.2.3), and scheduling scheme, this can occur at design time, offline, or some time online prior to execution.

3.4.1 Template Selection

Each candidate uses some distribution of system resources and has associated implementation costs in area, delay, and energy. Selections for cost-critical portions of the algorithm are generally performed in advance — either at design-time or offline — to facilitate predictable cost-performance. The run-time selection is guided by system-level objective functions that aim to optimize some system cost

Table 3.1: Candidate mapping template selection properties

area	time	energy
	configuration time, t_c	configuration energy, e_c
	computation time, t_e	computation energy, e_e
total area, a	total time, t_t	total energy, e_t

(a) Simple relation examples

Ordering		Strictness	
ascending	descending	absolute	relative

(b) Selection search directives

distributions. These functions use system-level observations in their attempts to avoid local performance minima. They monitor static and/or dynamic system properties to maintain a view of current system behavior. The monitored system properties typically fall into one of three categories: (1) simple property relations, (2) composite property relations, and (3) arbitrary/independent relations.

Simple Relations

Simple relations are the easiest to describe and therefore are presented first. With each selection, consideration is given to a single static or dynamic candidate mapping template property and its impact on overall system cost performance. Table 3.1(a) shows an example of some simple relations.

For each relation there is the notion of selection search ordering and strictness

as summarized in table 3.1(b). Ordering indicates whether the simple property is maximized or minimized — ascending order or descending order. Strictness indicates what is to occur when the identified selection, for some reason, cannot be instantiated. For absolute strictness, no other selection consideration is to be made. Therefore the associated algorithm graph member, to which the mapping template is paired, must strictly wait until the select mapping template implementation may be used. Relative strictness indicates that additional selections, in decreasing optimization order, are to be considered, in succession, until one that is capable of instantiation is identified.

Composite Relations

Composite relations are used when optimization objectives consists of several system properties in combination. The objective functions consist of multiple input terms from the system properties of interest. With function evaluation, selection proceeds as with the simple relation case. Many such functions can be constructed given a particular system scenario, and for the sake of discussion, considered optimization objective function given by the polynomial equation 3.1.

$$obj = c_1 \times a^{k_1} + c_2 \times t_c^{k_2} + c_3 \times t_e^{k_3} + c_4 \times e_c^{k_4} + c_5 \times e_e^{k_5} \tag{3.1}$$

Here in this example, the designer can choose appropriate coefficients c_1, c_2, c_3, c_4, c_5 and power terms k_1, k_2, k_3, k_4, k_5 to construct a suitable run-time optimization function for dynamic template selection given system constraints and objectives, where a, t_c, t_e, e_c, e_e are defined in the table 3.1(a). In practice, a function such as this would not be evaluated directly as shown. This would impose significant overhead unless implemented in dedicated hardware. To reduce the implementation penalty, an encoding — exact or approximate — would be used; such as with a look-up table based approach.

Arbitrary Selections

Arbitrary selections are, in essence, selections at random. Given the desire to construct specific system performance, it would seem counterintuitive to consider strategies based on arbitrary selection schemes. However, when such selection is combined with weighted feedback, self-adjusting adaptive schemes are possible. Moreover, such selections have the minimum possible cost overheads. For example, let for each selection an observation be made about the prior selections performance — good or bad. Under some circumstances, this statistical good/bad behavioral observation might allow the system to statistically refined its selection criteria and converge to inherent optimization operating points. The system designer can incorporate their knowledge of "normal" behavior by utilizing various

weighted random distributions — such as uniform, triangular, geometric, exponential, etc. For highly dynamic and unpredictable systems, statistical approaches, such as caching [LCH00], are often beneficial.

3.4.2 Template Instantiation

After a suitable templated-mapping has been selected, the template is instantiated by (1) reserving the specified resources, (2) scheduling the configuration of the implementation bit-streams, and (3) binding of the constructed implementation to the parent algorithm member (node or edge).

Reservation

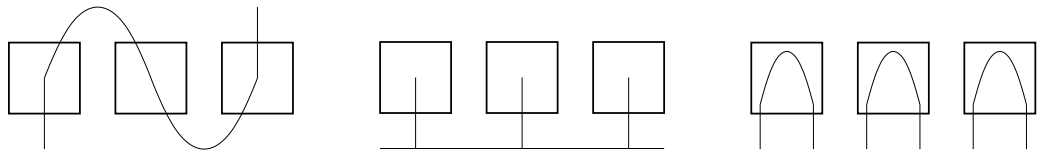
Resource reservation is a rather straightforward operation. Since the architecture resources are fully known, an efficient low-overhead data structure can be used to manage the status for each system resource. Minimally, storage for $(1 + n)$ bits are required for each resource instance. One bit to indicate resource reservation status and n bits to identify the reserving algorithm member; where $2^n \geq (n_e + n_n)$ for n_e, n_n equal to the number of edge instances and number of node instances in the algorithm, respectively.

Configuration Scheduling

To simplify the task of configuration management and reduce the associated overhead, it is best to use a single scheme across the resources managed by a single configuration controller. For partitioned systems with multiple configuration managers, different schemes can be used as appropriate in each particular resource group. Each will be based on either a random or serial access scheme. Which ever the case, the configuration bit-stream cw_{jk} for a resource ri_{jk} specified by the mapping template is, by definition, available at run-time. Depending on the access scheme, this configuration will consist of one or more sequences in the following generalized form.

Pre-amble	Configuration data word(s)	Post-amble

Prior to resource configuration, an ordering for each cw_{jk} must be imposed. This ordering will be directly derived from the configuration architecture of the resources. For example, consider the three architecture topologies shown in figure 3.10. In the case of serial access, figure 3.10(a), the ordering is fixed and no further consideration is required. In the random access case with a single configuration bus, figure 3.10(b), configuration ordering is not specified and therefore is arbitrary. However, in the case of multiple/parallel access configuration architectures, figure 3.10(c), the configuration scheduling order is non-arbitrary when



(a) Serial (b) Random (c) Concurrent

Figure 3.10: Resource configuration topology schemes

any of the configuration costs for a cw_{jk} differ from another. The problem is similar to a computation scheduling problems where there are concurrent tasks and parallel resources with the goal of minimizing the configuration schedule cost-performance. Based on system-level requirements, this configuration scheduling subproblem would have as its objective to minimize either configuration energy or configuration time.

Bit-stream Downloading

With an established configuration schedule, the work of downloading the configuration bit-streams to the required resources can precede. It is well understood that this configuration effort has significant costs and time and energy and in many cases can be the limiting factor of performance for systems based on reconfigurable architectures[DST99].

If the run-time configuration schedule is managed by a single processor, then the work of resource configuration will be fully serialized. Moreover, when multi-

ple algorithm members require configurations concurrently then further ordering considerations are required. Either (1) the complete schedule can be computed for some time-step follow by complete configuration or (2) scheduling can be intermixed with configuration with some increased implementation complexity/overhead. In this latter case, priority can be given to algorithm components of greater importance. For example, if multiple tasks are concurrently ready for resource configuration, each having varying priority, then a single-processor scheduler might choose to configure and dispatch in succession each task in the order established by the task priorities. In contrast, if this single-processor-scheduler schedules each in succession and then subsequently dispatches each, then there will be greater latency for the high priority tasks.

Dependant on the system characteristics, namely the frequency of resource configuration and concurrent configuration demand, a multi-processor scheduling and configuration approach could be advantageous. With such an approach, one or more processors would handle configuration scheduling and one or more would handle bit-stream downloading in a distributed manor. Clearly in this case, a mixed approach would be appropriate where scheduling processors immediately handoff the task of bit-stream loading to a configuration processors.

Binding Mode

As discussed in section 3.2.3, mapping template selection and architecture resource reservation both have corresponding modes which establish when (and for how long) an allocation selection is to occur and when (and for how long) a resource reservation is to be maintained. The multi-level mapping from algorithm to resource instance identifies the binding. Irrespective of when a selection or reservation is performed, those which are of the mode type dynamic (as opposed to static) are released after use. Dynamic mapping templates selections are released after use and dynamic resource reservations are freed after use.

In the following section a mathematical system model formulation is presented for algorithms, architecture resources, and templated-mapping. This model provides a precise language for describing and designing systems using mapping templates.

3.5 Method Formalization

The behavior of a system establishes its meaning in the context of the system environment. The crafting of this behavior is the overall goal of the design exploration process. The formal definitions of system behavior therefore establishes its execution semantics. Many different language, models, and schemes have been

Table 3.2: System model components

System Closed Sets	$UN, NS, ES, UG, UR, R, MM, RR, MT, MG, CG$
Dynamic Operations	$I(S), I(g), D(S), D(g), T(S), T(g)$
System Open Sets	S, A
Set Relations	$nsm, asm, trm, amm, gmm, smm, stm$
Property Functions	$setP, getP$

proposed and used to define functional, denotational, and operational semantics. But all have the common goal of unambiguously specifying meaning of system run-time behavior. The properties of such a formalism are directly tided to the ability to understand, execute, and analyze the system model[LZ05].

The execution semantics defined within this research consists of (1) a mathematical set-based model of the system components and (2) a state diagram formalization of how these components interact to define system behavior. For an example of a formal execution semantic definition see [DGM02]. The system component model can be found in section 3.5.1. The component interactions and scheduling are formalized in section 3.5.2. Together they form a road map for design using the proposed templated-mapping methodology and establishes a standard for cad-tool development. Chapter 4 documents a simulator implementation that conforms to the specification defined here.

3.5.1 Component Model

The system model consists of eleven closed (static) sets, six operations for managing dynamic algorithms, one dynamic system scheduler set (an open set), one dynamic system algorithm set, seven set mapping relations, and two generic property management functions. These components are summarized in table 3.2 and a structural view is shown in figure 3.12. The model can be divided into eight parts. (1) That which models algorithm and scheduler components, (2) that which models resource components, (3) that which models dynamic graph constructs, (4) that which models the system scheduler, (5) that which models the system algorithm, (6) that which models the system resource, (7) that which models mapping templates, and (8) that which models system property management. The key model relations are shown in equations (3.2)-(3.42).

Algorithm and Scheduler Behavior Components

UN is the set of n_{un} unique nodes un_i.

$$UN = \bigcup_{i=1}^{n_{un}} un_i \tag{3.2}$$

$NS = \bigcup_{i=1}^{n_{ns}} NS_i$ is the set of n_{ns} graph node sets. Where NS_i is the set of n_{gn_i} graph nodes in NS_i and nsm is a mapping for each node gn_{ij} onto a unique node

in UN.

$$NS_i = \bigcup_{j=1}^{n_{gn_i}} gn_{ij} \tag{3.3}$$

$$NS = \bigcup_{i=1}^{n_{ns}} \bigcup_{j=1}^{n_{gn_i}} gn_{ij} \tag{3.4}$$

$$nsm : NS \rightarrow UN \tag{3.5}$$

$ES = \bigcup_{i=1}^{n_{ns}} ES_i$ is the set of edge sets, where ES_i is the set of directed communications between the nodes of NS_i. Edge $ge_{ijk} \in ES_i \iff$ a connection exists from node $gn_{ij} \in NS_i$ to node $gn_{ik} \in NS_i$ for $j \neq k$, (we call gn_{ij} the source node and gn_{ik} the sink node).

$$ES_i = \bigcup_{j=1}^{n_{gn_i}} \bigcup_{k=1}^{n_{gn_i}} ge_{ijk} \tag{3.6}$$

$$ES = \bigcup_{i=1}^{n_{ns}} \bigcup_{j=1}^{n_{gn_i}} \bigcup_{k=1}^{n_{gn_i}} ge_{ijk} \tag{3.7}$$

$UG = \bigcup_{i=1}^{n_{ug}} ug_i$ is the set of n_{ug} unique algorithm graphs. Where the relation ug_i defines a directed acyclic graph where $NS_i \in NS$ is the node set and $ES_i \in ES$ is the edge set.

$$ug_i = (NS_i, ES_i) \tag{3.8}$$

$$UG = \bigcup_{i=1}^{n_{ug}} (NS_i, ES_i) \tag{3.9}$$

Resource Components

UR is the set of n_{ur} unique system resources ur_j.

$$UR = \bigcup_{j=1}^{n_{ur}} ur_j \tag{3.10}$$

Dynamic Graph Operations and Functions

The following operations and functions have been defined to manage open sets. There is no ambiguity in the reuse of the operation and function identifiers since the identifiers together with the arguments uniquely identifies each.

$S_c = I(S_o)$ defines the operation called instantiate, that constructs a copy of the set S_o. We write $S_c = I(S_o)$ which means reserve the set identifier S_c and copy each member element of S_o into S_c such that $S_c = S_o$ by construction (we call S_o the original and S_c the copy). Specifically, let S_o be a countable set with $n = |S_o|$ members and a one-to-one sequence $< e_i >_{i=1}^{n}$.

$$I(S_c) : reserve(S_c); S_c = I(S_o) \equiv S_c = \bigcup_{i=1}^{n} e_i \tag{3.11}$$

$g_c = I(g_o)$ defines the operation called called instantiate, that constructs a copy of the graph g_o. We write $g_c = I(g_o)$ which means reserve the graph relation identifier g_c and copy the set N_o and set E_o of $g_o = (N_o, E_o)$ into g_c with the

relation order preserved such that g_c and g_o are isomorphic by construction.

$$I(g_c) : reserve(g_c); g_c = I(g_o) \equiv g_c = (N_c = I(N_o), E_c = I(N_o)) \tag{3.12}$$

$D(S_c)$ defines the operation called destroy, that removes each element from the set S_c, and releases the set identifier S_c.

$$D(S_c) : D(S_c) \equiv S_c = \{\emptyset\}; release(S_c) \tag{3.13}$$

$D(g_c)$ defines the operation called destroy, that removes each element from the sets N_c and E_c where $g_c = (N_c, E_c)$, and releases the graph identifier g_c.

$$D(g_c) : D(g_c) \equiv D(N_c); D(E_c); release(g_c) \tag{3.14}$$

T is the inverse mapping function for each set S_c instantiated with $I(S_o)$. For example, given $S_{c1} = I(S_o)$ and $S_{c2} = I(S_o)$, then $T[S_{c1}] \equiv T[S_{c2}] \equiv S_o$. Therefore $T[S_c = I(S_o)]$ provides indirect access to S_o and we may write $S_{c3} = I(T[S_c])$ with $T[S_{c3}] = S_o$.

$$T : S_c \rightarrow S_o \tag{3.15}$$

T is the inverse mapping function for each graph g_c instantiated with $I(g_o)$. For example, given $g_{c1} = I(g_o)$ and $g_{c2} = I(g_o)$, then $T[g_{c1}] \equiv T[g_{c2}] \equiv g_o$.

$$T : g_c \rightarrow g_o \tag{3.16}$$

Partitioned System Scheduler

S is the open set of one or more graphs sg_i that compose the scheduler S where each sg_i is an instance of some unique graph from UG. Specifically, $sg_i = I(ug_\star)$, where $ug_\star = (NS_\star, ES_\star)$ and $ug_\star \in UG$. Therefore it follows that $SN_i = I(NS_\star)$, where $NS_\star$ has $n_{gn_\star}$ nodes, and $SE_i = I(ES_\star)$, where $ES_\star$ has up to $n^2_{gn_\star}$ edges[7]. We call this open set S the partitioned system Scheduler. $|S|$ indicates the number of scheduler graphs at a moment. We call the graph sg_1 the root scheduler.

$$S = \bigcup_{i=1}^{(open)} sg_i \tag{3.17}$$

$$sg_i = (SN_i, SE_i) \tag{3.18}$$

$$SN_i = \bigcup_{j=1}^{n_{gn_\star}} sn_{ij} \tag{3.19}$$

$$SE_i = \bigcup_{j=1}^{n_{gn_\star}} \bigcup_{k=1}^{n_{gn_\star}} se_{ijk} \qquad se_{ijk} \in SE_i \iff ge_{\star jk} \in ES_\star \tag{3.20}$$

Dynamic System Algorithm

A is the open set of graphs ag_i that compose the algorithm A where each ag_i is an instance of some unique graph from UG. Specifically, $ag_i = I(ug_\star)$, where $ug_\star = (NS_\star, ES_\star)$ and $ug_\star \in UG$. Therefore it follows that $AN_i = I(NS_\star)$, where $NS_\star$ has $n_{gn_\star}$ nodes, and $AE_i = I(ES_\star)$, where $ES_\star$ has up to $n^2_{gn_\star}$ edges. We

[7]This bounds the fully connected undirected graph, but is pessimistic in the case of a DAG.

call this open set A the dynamic system algorithm. $|A|$ indicates the number of algorithm graphs at a moment and for $A = \{\emptyset\}, |A| \equiv 0$. The mapping asm assigns each algorithm graph instance $ag_i \in A$ to a scheduler graph instance $sg_j \in S$. For $sg_j = asm(ag_i)$, we say scheduler partition instance sg_j manages the algorithm graph instance ag_i.

$$A = \bigcup_{i=1}^{(open)} ag_i \tag{3.21}$$

$$ag_i = (AN_i, AE_i) \tag{3.22}$$

$$AN_i = \bigcup_{j=1}^{n_{gn_\star}} an_{ij} \tag{3.23}$$

$$AE_i = \bigcup_{j=1}^{n_{gn_\star}} \bigcup_{k=1}^{n_{gn_\star}} ae_{ijk} \qquad ae_{ijk} \in AE_i \iff ge_{\star jk} \in ES_\star \tag{3.24}$$

$$asm : A \to S \tag{3.25}$$

Static System Resource

$R = \bigcup_{j=1}^{n_{ur}} RI_j$ is the closed set of resources. Where RI_j is the set of rq_j instances of the unique resource ur_j and $rq_j \in \aleph$ for $\aleph$ the set of all natural numbers. We call ri_{jk} the k^{th} instance of unique resource ur_j and closed set R the static system

resource.

$$RI_j = \bigcup_{k=1}^{rq_j} ri_{jk} \tag{3.26}$$

$$R = \bigcup_{j=1}^{n_{ur}} \bigcup_{k=1}^{rq_j} ri_{jk} \tag{3.27}$$

Candidate Templated-Mappings and Modes

MM is a set of whose members are abbreviations for Offline-Static, Offline-Dynamic, Online-Static, Online-Dynamic, respectively. They indicate when a mapping is to occur—either online or offline—and how the mapping is to be bound—either statically or dynamically. We call this the set of mapping modes.

$$MM = \{OffS, OffD, OnS, OnD\} \tag{3.28}$$

$RR = \bigcup_{j=1}^{n_{ur}} RR_j$ is the set of resource reference sets. Where RR_j is the set of all possible references rr_{jk} for resource instance set $RI_j \in R$. Therefore rr_{jk} is a reference of k instances of resource ur_j and $1 \leq k \leq rq_j$.

$$rr_{jk} = (ur_j, k) \tag{3.29}$$

$$RR_j = \bigcup_{k=1}^{rq_j} rr_{jk} \tag{3.30}$$

$$RR = \bigcup_{j=1}^{n_{ur}} \bigcup_{k=1}^{rq_j} (ur_j, k) \tag{3.31}$$

$MT = \bigcup_{i=1}^{n_{mt}} MT_i$ is the n_{mt} member set where MT_i is the n_{ur} member set. The mapping trm assigns each mt_{ij} of MT_i to one of (1) the empty set or (2) a

member of RR_j. For each $mt_{ij} \in MT_i \iff trm[mt_{ij}] \neq \emptyset$. Therefore $MT_i \subseteq RR$ is a mapping subset of resource references and we call MT_i a mapping template. Moreover, each $MT_i \in MT$ is unique or $MT_j \neq MT_k$ for $j \neq k$. The mapping amm assigns one member of MM for each $mt_{ij} \in MT_i$. We call amm the resource allocation mode mapping and trm the template reference mapping.

$$MT_i = \bigcup_{j=1}^{n_{ur}} mt_{ij} \tag{3.32}$$

$$trm : MT_i \rightarrow \{\emptyset \cup RR_j\} \tag{3.33}$$

$$amm : MT_i \rightarrow MM \tag{3.34}$$

$$MT = \bigcup_{i=1}^{n_{mt}} \bigcup_{j=1}^{n_{ur}} mt_{ij} \tag{3.35}$$

MG is the set of n_{mg} disjoint subsets of MT. Specifically, $MG_i \subseteq MT$ and $MG_j \cap MG_k \equiv \emptyset$ for $j \neq k$.

$$MG = \bigcup_{i=1}^{n_{mg}} MG_i \tag{3.36}$$

CG is the set of n_{cg} disjoint subsets of MG. Specifically, $CG_i \subseteq MG$ and $CG_j \cap CG_k \equiv \emptyset$ for $j \neq k$. The mapping gmm assigns one-to-one each member of $\{UN \cup ES\}$ to a member CG_i of CG. The mapping stm assigns a member MT_j, from the mapped CG_i, for each member of $\{UN \cup ES\}$. The mapping smm assigns each member of $\{UN \cup ES\}$ to a member of MM. We call smm the selection mode mapping, gmm the algorithm graph member mapping, and stm

the selected mapping-template mapping.

$$CG = \bigcup_{i=1}^{n_{cg}} CG_i \tag{3.37}$$

$$gmm : \{UN \cup ES\} \rightarrow CG \tag{3.38}$$

$$stm : \{UN \cup ES\} \rightarrow MT_j \tag{3.39}$$

$$smm : \{UN \cup ES\} \rightarrow MM \tag{3.40}$$

Given MT is the set of unique mapping templates, MG is the set of disjoint subsets of MT, and CG is the set of disjoint subsets of MG, each member CG_i is a subset of subsets of mapping templates that are mapped to the algorithm members by gmm, each assigned a mapping mode in MM by smm.

System Properties

The model presented in the preceding paragraphs defines a framework for template-based mapping of dynamic algorithms on static resources. No assumptions are made about specific system properties that might be relevant to a particular run-time scheduling scheme beyond the constructs of the base framework. Inasmuch, in practice, one will likely need to associate properties germane to the run-time scheduling scheme of focus.

To facilitate the association of arbitrary system properties, two functions are

Table 3.3: System property functions

Set Property	$setP(identifier, keyword) = value$
Get Property	$value = getP(identifier, keyword\ )$

defined that allow identifiers to be associated with keyword-value pairs. They are shown in table 3.3. One function sets the association and the other gets the current associated value for the specified identifier and keyword[8].

Example: To set the physical location of each resource one could use the keyword "position" and value relation "(x, y)" and iterate over the resource using the resource identifier for each association:

$$[SetP(ri_{jk}, possition) = (x_{jk}, y_{jk})]_{j=1}^{n_{ur}}\Big|_{k=1}^{rq_j} \tag{3.41}$$

Or, to assign a priority pri_{ij} to each node an_{ij} in each of the current graphs ag_i of algorithm A, using the keyword "priority," one would write:

$$[SetP(an_{ij}, priority) = pri_{ij}]_{i=1}^{|A|}\Big|_{j=1}^{n_{gn_i}} \tag{3.42}$$

[8]A pair of functions like those in table 3.3 are required for each unique type-tuple $(identifier, keyword, value)$ used in a specific model.

3.5.2 Execution Process Model

In this section the description of how system components, specified in the previous section, interact during execution to yield system behavior is presented. First, a brief introduction to the assumed system-level design language primitive constructs is presented. Then, a presentation of the component structure and their relations is discussed. Finally, the run-time interaction between the system components is specified under the heading scheduling.

Primitive Constructs

In order to remain focused on the contribution goals of this research the methodology has been built on top of SystemC [GLMS02], a system-level design language. Many other choices exist such as SpecC [GZD+00], Verilog, and VHDL, to name just a few. SystemC is an extension of C++, has an open software architecture and free reference implementation, and significant research and industry involvement. However, any of the aforementioned could have been chosen as a foundation. As depicted in figure 3.11, a system that is composed of three basic categories; algorithms, schedulers, and architectures. Each of these are specified using the constructs of the design language. Rather than attempt to cover these well studied and documented primitive constructs we simply refer to the literature and assume the existence of a few basic capabilities as outlined in

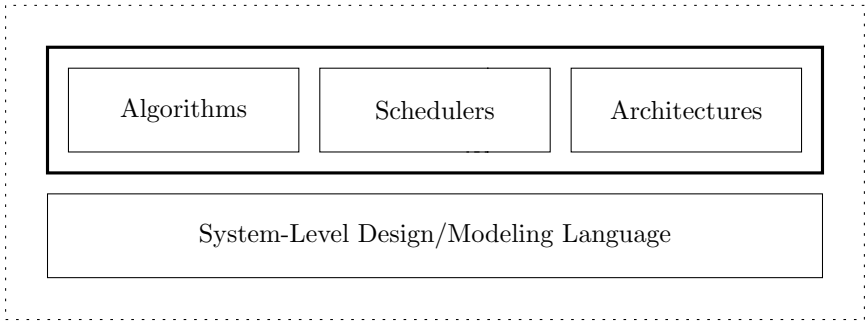


Figure 3.11: System-level design modeling language primitives

the following paragraph.

We assume descriptive language for (1) hierarchical structure composition, (2) behavioral concurrency specification for system components, (3) synchronization and communication primitives, and (4) a model and mechanism for time control and consumption. On top of the base primitive constructs we build our methodology and methodology-specific primitives. It is helpful but not essential when more advanced constructs exist such as signals, timers, mutexes, queues, rich data types, etc., since they can be built on top of the primitives as required.

Structure

A graphical display of the set model is shown in figure 3.12. It shows each of the sets that make up the system model and their member elements[9]. The set

[9]Sets MG and CG have been omitted for clarity of the diagram. These sets are disjoint subsets of the mapping template set MT

mappings are shown with solid-curved lines that indicate how members of one set are related to members of another. The undirected-dashed lines show structural composition. For example, the resources R are composed of unique resources from UR. The directed-dashed lines indicate how the dynamic operations are used to expand and reduce members in the open sets A and S at run-time.

System run-time behavior adapts to external change requests that are directed to the scheduler S. System execution is distributively managed by a member of S. If a change request can be accommodated, based on current performance and constraints, then the scheduler implements the required system changes. Algorithms are composed of unique graphs UG that are built from node sets NS and edge sets ES. Each graph node is one of a unique node UN. The unique nodes and edges have one or more unique candidate implementations from the set of mapping templates MT. Each mapping template specifies a set of resource references RR that identifies the algorithm-to-architecture mapping requirements. Each resource reference specifies the quantity of and resource identifier from the system resource R. The system resource is a fixed collection of instances for each of the available unique system resources UR. Each mapping templates selection and resource allocation has a mapping mode MM that indicates when and for how long the selection or allocation is to be implemented.

A system design specifies each of the components and the component relations.

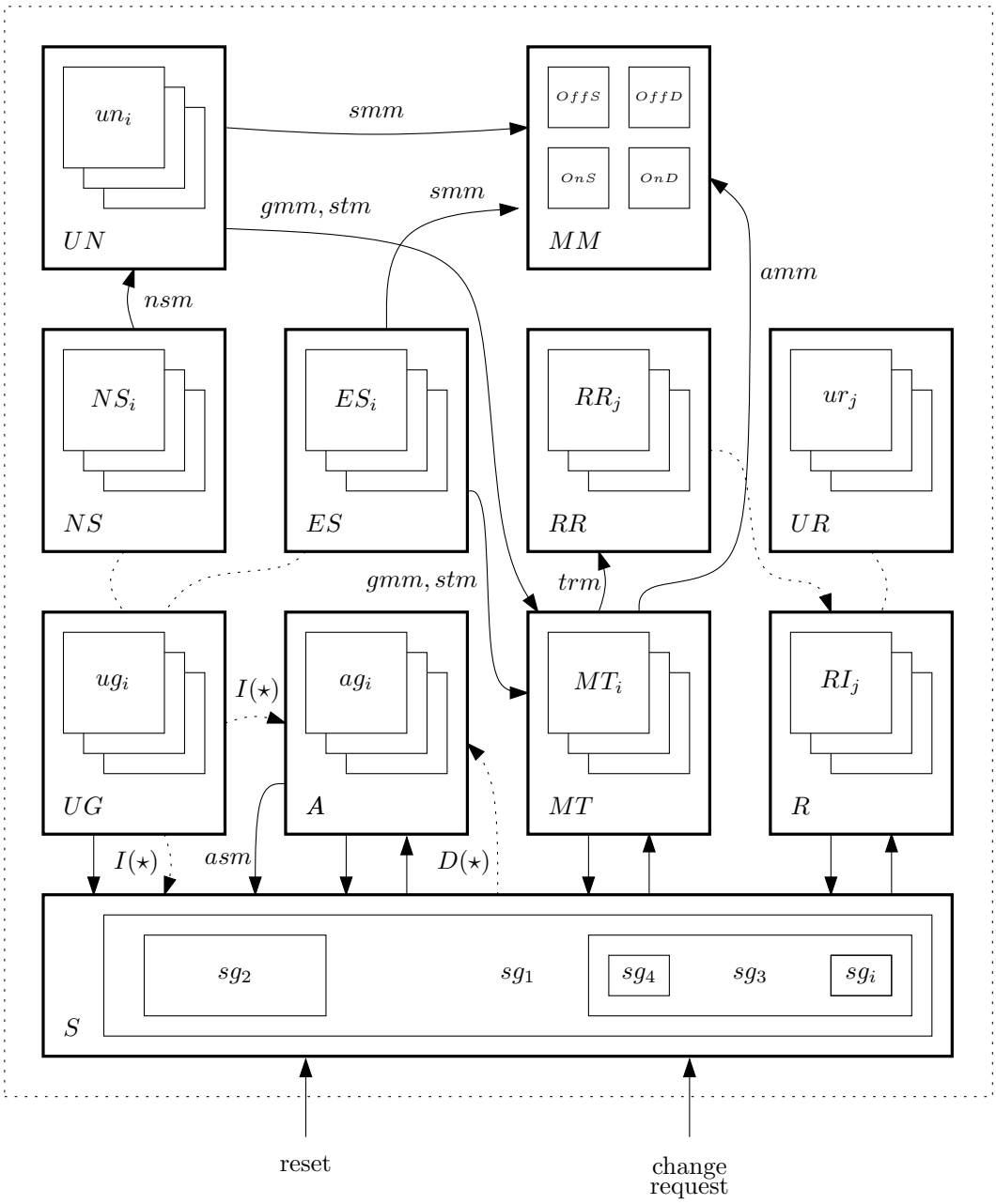


Figure 3.12: System model: Component structure and relations

The scheduler S acts as the authority to (1) resolve concurrent resource demands, and (2) dynamically choose implementations from MT based on run-time context, (3) expand, contract, or change the algorithm A and scheduler S at the request of external events. These choices are governed by system objectives and policy to carry out run-time optimization within limits of current constraints.

Given the need to maintain both local and global control over the resource pool, the scheduler S is hierarchically organized. Descendant schedulers are delegated responsibility from ancestors, but remained under the authority of the delegator. A scheduler can create and destroy descendants, but the reverse is deemed illegal.

To provide the relevant internal context during runtime decision-making the design methodology and scheduler make use of component property annotations via $setP$ and $getP$. These annotations are concise descriptions or abstracts that provide distinctions during computer aided design and run-time management.

Scheduling

From the perspective of system executions specification we need only focus on the ordering of node execution since we can make the assumption that the passive communication execution is in-lined with the source (or sink) node. Figure 3.13(a) shows the system algorithm A containing a single graph ag_i that has been instantiated from UG with three nodes ($an_{ij}, an_{ik}, an_{im}$), two internal edges

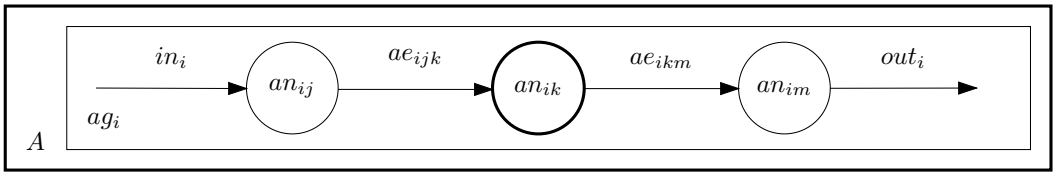
(ae_{ijk}, ae_{ikm}), and two external edges (in_i, out_i). Each node operates concurrently with one another and transitions through the life-cycle shown in figure 3.13(b).

As shown, each node is in one of eight states; $status(an_\star) \in \{$`dormant, waiting, ready, blocked, configure, running, interrupted, done`$\}$. The node status is governed by (1) the node input event condition and (2) the assigned system scheduler. For the purpose of discussion we let node an_{ik} be assigned the scheduler sg_n, $asm(an_{ik}) = sg_n$, and have the single input ae_{ijk} and single output ae_{ikm}.

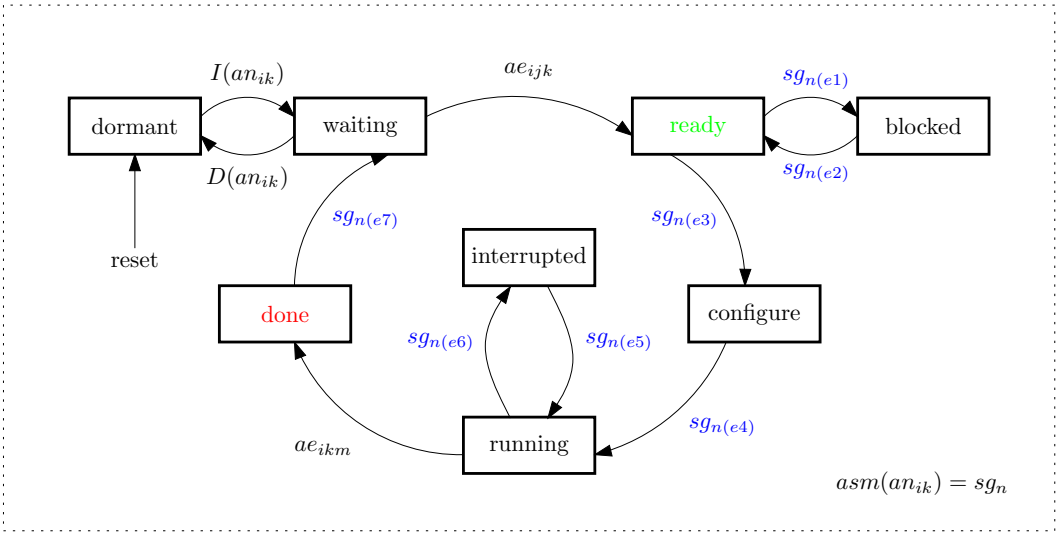
`dormant` — *Definition:* nodes that are inactive. *Transition:* after instantiation, $I(an_{ik})$, each node enters into `waiting`.

`waiting` — *Definition:* nodes waiting on input event conditions. *Transition:* upon the occurrence of input event ae_{ijk}, node an_{ik} transitions to `ready` where it awaits service from assigned scheduler sg_n. *Transition:* upon the occurrence of $D(an_{ik})$, the node returns to `dormant`.

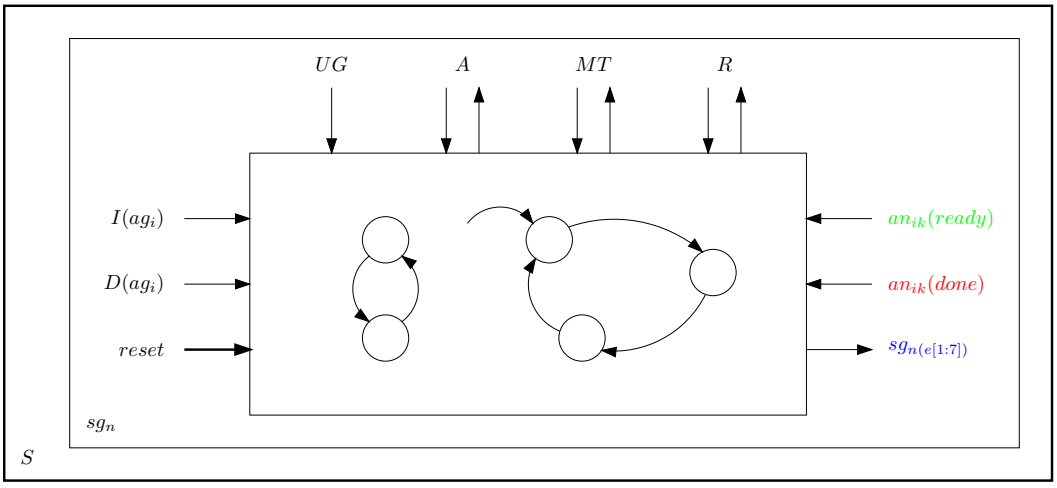
`ready` — *Definition:* in this state, the scheduler sg_n searches the available implementation alternatives based on (1) the choice of assigned templated-mappings $gmm(an_{ik})$, (2) the available resource R, (3) the system optimization objectives, and (4) the cost-constraints. *Transition:* if the scheduler identifies an acceptable implementation from MT_i, an_{ik} is advanced to `configure`. *Transition:* if no



(a) Algorithm node $an_{ik} \in ag_i \in A$



(b) Algorithm node an_{ik} execution life-cycle



(c) Scheduler $sg_n \in S$

Figure 3.13: System model: Algorithm, architecture, and scheduler execution

selection can be made, then an_{ik} is advanced to `blocked`.

`blocked` — *Definition:* nodes with satisfied input condition for which there are no "suitable" architecture resources for any of its templated-mappings. *Transition:* a node an_{ik} transitions back to `ready` at the sole discretion of the assigned scheduler sg_n. Typically a scheduler implementation will revisit `blocked` nodes when there is a change in resource reservations and/or run-time optimization objectives.

`configure` — *Definition:* in this state, the scheduler reserves the resources specified by the selected templated-mapping and downloads the configurations. A scheduler will need to implement "configuration scheduling" to order the concurrent nodes that are ready for configuration. *Transition:* after the complete set of resources specified by a template are configured, the node is advanced to `running`.

`running` — *Definition:* nodes that are performing computation and may generate output events such as ae_{ikm}, often called execution. *Transition:* a scheduler may temporarily stop the execution of a node by saving its state and advancing it into the `interrupted` state. *Transition:* when a node completes its execution it notifies the scheduler which in-turn advances the node to `done` where it awaits "post-execution" processing.

`interrupted` — *Definition:* a node with in-progress execution that has had

its execution state saved and has been temporality halted. *Transition:* at the discretion of the scheduler, the node's state may be restored and returned to `running`.

`done` — *Definition:* a node that has voluntarily completed execution and is ready for "post-execution" processing. Each node in `done` is reviewed for "template de-construction." That is to say, its reserved resources may be releases and its mapping template may be de-selected. These are both governed by the associated reservation and selection modes. As with `ready` and `configure`, a scheduler will need to implement "done scheduling" to order the processing of concurrent nodes that are ready for 'post-execution" processing. *Transition:* afterwards, the scheduler sg_n advances the node to `waiting` where the behavior life-cycle repeats.

The design methodology makes no further assumptions about a particular scheduler implementation other than to define its responsibilities and to specify the framework for managing system components. Figure 3.13(c) shows a block-diagram of a scheduler with its inputs and outputs from and to other system components. The scheduler can be implemented in software — on a general purpose processor — or in dedicated hardware. It can be designed as a single monolithic process node, distributed across multiple coordinating nodes, or can incorporate concurrent independent processes. In the latter case, and additional mapping relation beyond $asm(\star)$ is required to specify which scheduler process

manages node-ready events $an_{ik}(ready)$ and node-done events $an_{ik}(done)$.

3.6 Chapter Summary

A framework and methodology for post-design management of dynamic change in digital signal processing system algorithms, architectures, and stimulus, is proposed. The architecture resource is assumed to consists of a closed set (fixed) of reconfigurable elements. The algorithm is composed of an open set (variable) of pre-analyzed static behavioral graphs that can be dynamically instantiated and destroyed in response to system behavior change requirements. The framework uses a structured approach to organize pre-designed mapping implementation alternatives for the behavioral graph elements that can be chosen and implemented with minimal run-time overhead by a dynamic, distributed, and hierarchical system scheduler. The re-targetable mapping alternatives are named templated-mappings. The system scheduler is assigned the responsibility of candidate selection and run-time template instantiation.

A classification of mapping time is proposed to clarify the discussions in dynamically reconfigurables system design that includes design-time, run-time, offline, and online. Mapping modes are defined that allow design trade-offs in flexibility and predictability for each templated-mapping. The design exploration

steps for offline template construction and the responsibility for run-time template use are detailed. Finally, a formal system model is defined that enumerates the components of the design methodology and provides an execution model of how they concert to compose system behavior.

Chapter 4

Templated-Mapping Design Environment

> The path of precept is long, that of example short and effectual. Lucius Annaeus Seneca (4 BC - 65 AD).

In this chapter we will describe the implementation of a design environment that uses templated-mapping and implementation candidates. The design environment consists of a core infrastructure and collection of design tools that facilitates the composition, static analysis, and dynamic analysis of run-time reconfigurable systems using a templated-mapping design methodology. The infrastructure, built on top of SystemC[1], establishes a common data-model, framework,

[1]SystemC was chosen simply to increase the rate of progress in the development of the design environment. Numerous other starting points were possible—and considered. However,

and collection of libraries. The tools are built on top of this infrastructure and its libraries. A description of this core infrastructure is presented in Section 4.1. An essential tool for this research, the system simulator, has been structured with a modular framework in order to encourage and facilitate its extension. This simulator software architecture is described in Section 4.2 and its execution semantics are described in Section 4.3. Some of the other design environment support tools are introduced in a tutorial in Appendix B.

The overall design environment is structured to provide a platform for the implementation and iterative refinement of system algorithms, architecture mappings, and schedulers. Some care has been taken to limit unnecessary constraints imposed on the system component model composition and their interactions beyond that defined by the methodology proposed in Chapter 3. In Section 4.4, a discussion of the existing design-flow that emerges out of the methodology, design environment, and the current tools is presented. The chapter is concluded with a summary.

---

the open-source status combined with compiled execution made this an acceptable choice.

4.1 Core Infrastructure

A skeleton diagram of the design environment data model and development libraries is shown in figure 4.1. To enhance the readability, focusing on the essential elements relevant to this research, most of the detailed parameters, states, and operations have been omitted from the diagram. This model is used throughout the environment to capture the description of algorithms, architectures, and "reconfigurable" templated-mappings.

4.1.1 Internal Data Model

Shown in figure 4.1, the model has objects to support simulation, event recording, results reporting, static-design analysis, and model data input/output. Also identified are objects for iterating over the model data-structures and specifying algorithmic performance constraints. Finally, there are objects to facilitate the recordation of static cost-performance, dynamic cost-performance, and run-time state statistics. In the center of the figure, three groups of objects are identified: (1) system algorithm, (2) candidate implementations, and (3) system architecture. These are the core structures that implement the methodology proposed by this research.

Starting with (1), a system algorithm, it is shown that an algorithm is com-

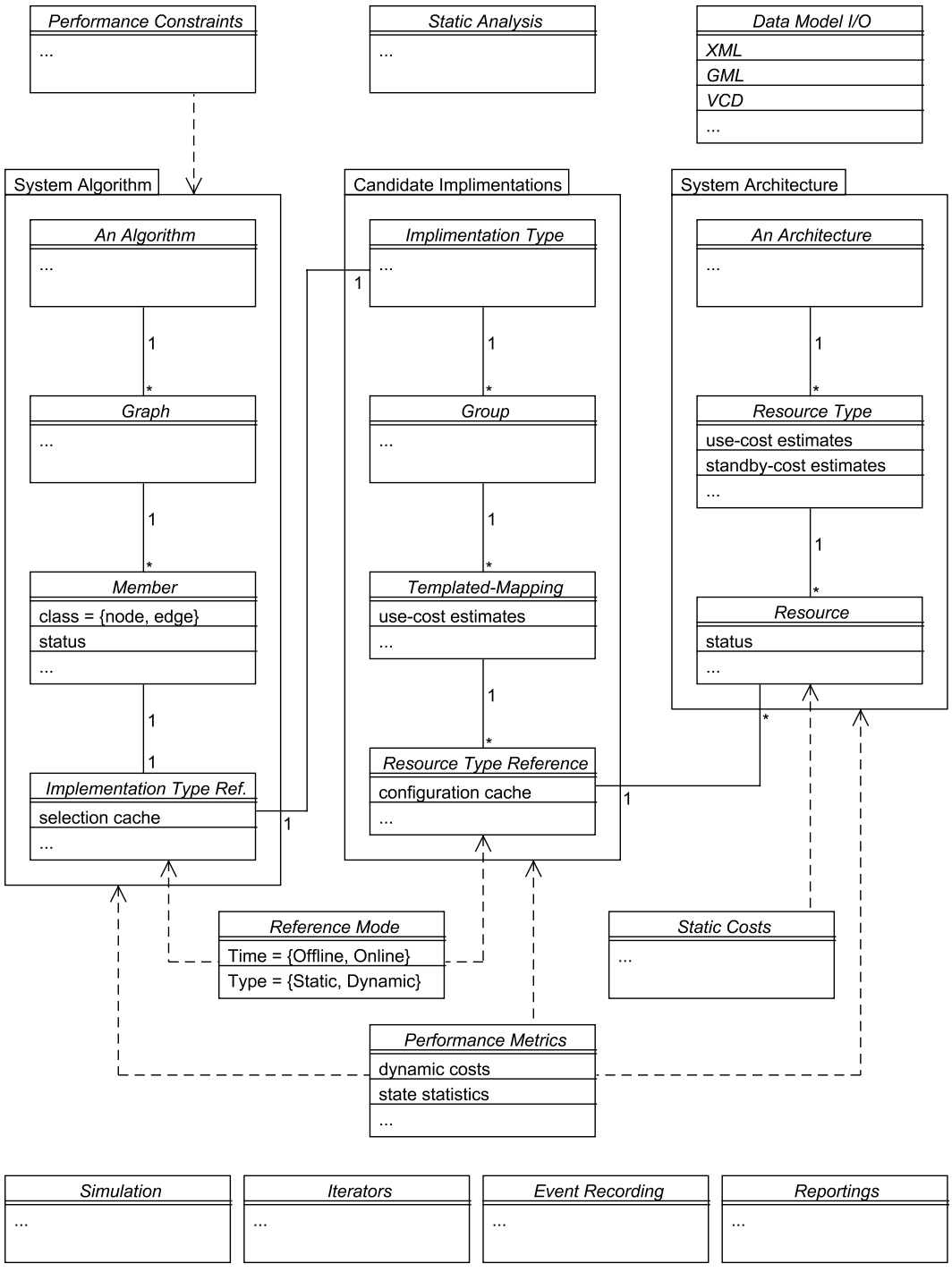


Figure 4.1: Skeleton of the design environment core libraries and data model

posed from one or more graph objects. Each graph has one or more members where a member can be either a node or an edge and has a public status. Each member has an implementation type reference that refers to a candidate implementation type and has a selection cache.

Next (2), each implementation type has one or more groups and each group has one or more templated-mappings[2]. Each templated-mapping has a collection of estimates for the cost associated with its use and has one or more resource type references that specify the resource requirements of the mapping. A resource type reference has a configuration cache associated with the state of the reference and specifies some quantity of resource type instances in the system architecture. The costs of interest to a particular system design varies dependant upon numerous factors. Neither the design methodology nor the design environment limits the dimensions of the cost components. As of this writing, the design environment considers the consumption of time, energy, and area for reconfiguration and computation. However, it could easily be extended to incorporate other use-costs of interest.

As for (3), system architecture, the model specifies an architecture as having one or more resource types that each have used-cost estimates and standby cost-estimates. Each resource type has one or more resource instances and associated

[2]A templated-mapping specifies an implementation structure but does not identify specific resource instances; hence the use of the adjective templated.

status.

The implementation type reference and resource type reference both have an associated reference mode, as shown below the objects in the figure. As introduced in Section 3.2.3, the mode is used to define when the object is de-referenced, either offline or online, and how it is to be managed afterwards, either with a static reference replacement or continuous dynamic de-referencing with each invocation. The selection cache and configuration cache are used to manage the state associated with implementation and resource-type references, respectively.

To facilitate the inter-operability of design environment tools with other tools, open (or public) standard data formats have been selected for persistent store where appropriate. The extensible markup language (XML) has been chosen as the primaries data model file I/O format. It has been gaining in popularity in recent years due to a number of factors: its extensibility, its textual encodings, and its meta-format focus. It allows both the logical structure and field data to be incorporated within a single text file and therefore can be used to store a wide variety of data structures. Put another way, given a properly formatted XML data file, a parser can determine both the data structure and data content by reading the file contents. It integrates well within an Internet-based designed environment and, with the help of style sheets, it can be transformed into human-readable representations.

It is worth mentioning here, that the graphs markup language (GML) and vector-change dump (VCD) format have been incorporated to leverage existing tools for (1) system graph construction and (2) the evaluation of dynamic system state transition. For more on the data formats of the design environment see Appendix B.3 and for a complete list of file formats see table B.2.

4.1.2 Event Recording Mechanisms

Timing in concurrent systems is of high importance. In systems that have a single computation resource or a single thread of control, the issue of time often converges to that of performing each operation as quickly and efficiently as possible, as ordered by the control and data dependencies of the algorithm. In systems where there are multiple control treads and/or computational concurrency, relative timing of events becomes a primary focus of concern in the design process. At each step in time, when there is concurrent behavior and control or data interdependency, the choice of first progress can affect the correctness of overall system behavior. Or, in a worst-case scenario, it can even lead to deadlock. To help manage these issues and the complex solution space that results from the design of threaded and concurrent systems, event recording mechanisms and report generation libraries have been structured into the core infrastructure.

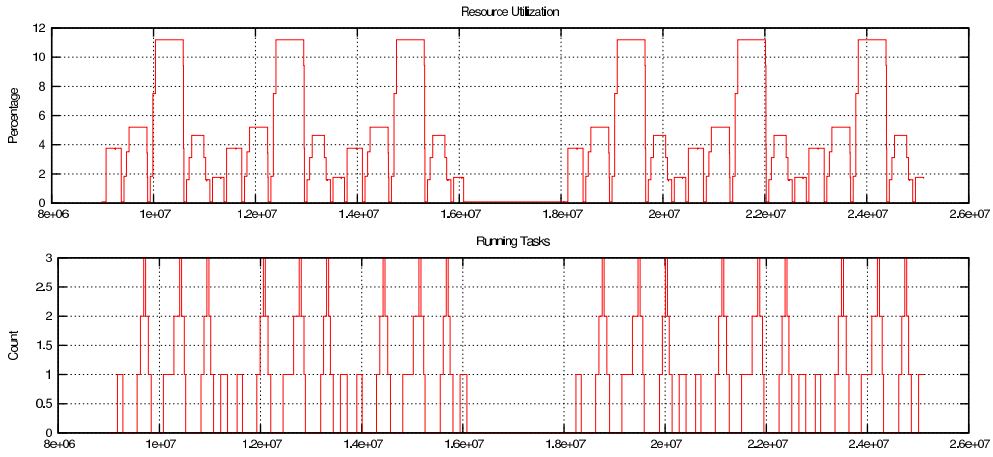


Figure 4.2: State trace example: resource utilization and concurrency

Dynamic State Recording

To effectively optimize system behavior, a designer needs accurate views of the dynamic internal interactions of system components and/or their state over time. Inasmuch, state recording mechanisms have been incorporated into the design environment infrastructure. As an example, see the trace charts shown in figure 4.2.

The upper chart shows overall system resource utilization over time. This trace is generated by a specifically registered state-recording function that computes and records this value with each system event. This mechanism allows virtually any systems statistic to be observed over the simulation run. The lower chart shows a count of the current running system tasks over time. This value is directly

maintained by the simulation kernel and therefore it only need be recorded at each system event.

Simulation Output Logging

Reconfigurables DSP systems can at any time have multiple active threads ready for computation. As discussed in Chapter 2, the applications of interests are typically characterized with ample concurrent operations. Consider once again the trace profile of running tasks shown in the lower chart of figure 4.2. We see that there are, at times, three (3) concurrent tasks[3] running on the architecture over the window of interest. This system design methodology supports distributed scheduler development and therefore if the system represented in this dynamic trace had a separate scheduler for each thread, there could be more active computation threads at any point in time[4]. Each thread operates independently and is capable of modifying system state. Even in this very simplistic example, the need should be clear that a system designer must have a precise time-stamped view of every system event in order to understand the details of system behavior. The design environment defines a set of routines to deal with this matter. Listing 4.1 shows an excerpt of one structured event log generated during the simulation corresponding to the trace above. When using the environment library primitives

[3]A task is equivalent to a thread in the context of this design environment.

[4]If the scheduler were performing other background or system management functions.

for system composition, this process of event logging is structured automatically.

4.1.3 Report Generation Interfaces

The structure, state, parameters, accumulated static/dynamic performance cost metrics, and use-cost estimates — both defined by the base methodology and extended by a particular system design — can be reported to system graphs, human-readable text files, and raw data text files. A collection of routines has been written to selectively search and/or iterate over a design database to compile such system reports. This reporting mechanism utilizes a configuration parameter structure to define custom views for each report assembly pass. It can be used to generate comprehensive comparisons of multiple designs in an automated fashion.

This approach has been implemented in the design environment simulator. A design solution space can therefore be explored in a systematic and structured way. The resulting design database-of-reports can be further processed to synthesize higher views of the design space. The post-simulation report object of the simulator framework, discussed in the next section, makes use of this recording core infrastructure facility.

Listing 4.1: Simulation event log example

```
SYSROOT: simulation start
SYSROOT: running until all clusters have completed 3 time(s)
8988290 ps:     TASK: t[pas.ct0.ci0.t0] ready, signaling t[rtos.schedule]
8988290 ps: SCHED-MQ: signaled
8988290 ps:   select: c[*.message] selected for t[rtos.schedule]
8988290 ps:    alloc: rt[sch]x1 allocated for tc[rtos.schedule.*.message],
                      0/1 remain available
8989570 ps:  [DEBUG]:  qsize = 0, events = 0, current task = t[pas.ct0.ci0.t0]
9016770 ps: TASKSSUM: task status summary (i=23, s=1, b=0, c=0, r=0, d=0)
9016770 ps:  dealloc: for tc[rtos.schedule.*.message]: freeing ( rt[sch]x1 )
9016770 ps: deselect: tc[rtos.schedule.*.message] selection released
9016770 ps:   timing: t[rtos.schedule] done at 9016770 ps,
                      signaled at 8988290 ps, started at 8988290 ps,
                      response = 0 s, active = 28480 ps, delay = 28480 ps
[SECTION DELETED]

9068450 ps: SCHED-CL: new phase
9068450 ps:   select: c[*.dispatch] selected for t[rtos.schedule]
9068450 ps:    alloc: rt[sch]x1 allocated for tc[rtos.schedule.*.dispatch],
                      0/1 remain available
9069730 ps: SCHED-CL: list ( t[pas.ct0.ci0.t0] )
9069730 ps: SCHED-CL: t[pas.ct0.ci0.t0]: configuring ( rt[rt3]x23 rt[rt4]x23 )
9170930 ps: SCHED-CL: signaling t[pas.ct0.ci0.t0]
9181330 ps: SCHED-CL: existing configuration = 0, new configurations = 1,
                      for 1 selected task(s)
9181330 ps: TASKSSUM: task status summary (i=23, s=0, b=0, c=0, r=1, d=0)
9181330 ps:  dealloc: for tc[rtos.schedule.*.dispatch]: freeing ( rt[sch]x1 )
9181330 ps: deselect: tc[rtos.schedule.*.dispatch] selection released
9181330 ps:   timing: t[rtos.schedule] done at 9181330 ps,
                      signaled at 9068450 ps, started at 9068450 ps,
                      response = 0 s, active = 112880 ps, delay = 112880 ps
9249686 ps:     TASK: t[pas.ct0.ci0.t0] done, signaling t[rtos.schedule]
9249686 ps:   timing: t[pas.ct0.ci0.t0] done at 9249686 ps,
                      signaled at 8988290 ps, started at 9069730 ps,
                      response = 81440 ps, active = 179956 ps, delay = 261396 ps
9249686 ps:     TASK: t[pas.ct0.ci0.t2] ready, signaling t[rtos.schedule]
9249686 ps:     TASK: t[pas.ct0.ci0.t5] ready, signaling t[rtos.schedule]
9249686 ps:     TASK: t[pas.ct0.ci0.t8] ready, signaling t[rtos.schedule]
9249686 ps: SCHED-MQ: signaled
9249686 ps:   select: c[*.message] selected for t[rtos.schedule]
9249686 ps:    alloc: rt[sch]x1 allocated for tc[rtos.schedule.*.message],
                      0/1 remain available
9250966 ps:  [DEBUG]:  qsize = 3, events = 1, current task = t[pas.ct0.ci0.t0]
9278166 ps:  [DEBUG]:  qsize = 2, events = 1, current task = t[pas.ct0.ci0.t2]
9305366 ps:  [DEBUG]:  qsize = 1, events = 1, current task = t[pas.ct0.ci0.t5]
9332566 ps:  [DEBUG]:  qsize = 0, events = 1, current task = t[pas.ct0.ci0.t8]
9359766 ps: TASKSSUM: task status summary (i=20, s=3, b=0, c=0, r=0, d=1)
9359766 ps:  dealloc: for tc[rtos.schedule.*.message]: freeing ( rt[sch]x1 )
9359766 ps: deselect: tc[rtos.schedule.*.message] selection released
```

4.1.4 Data Model Extension

The design environment internal data model can be extended in one or more of the following three ways. (1) Additional system properties can be associated with any model object of figure 4.1. By adding new properties an implementation can incorporate additional design-time information that can later be used to guide run-time optimization. As an example, a designer might annotate each resource with its physical coordinate information, which could subsequently be considered by run-time optimization functions. In like fashion, (2) a designer can incorporate additional state variables for each model object. For example, consider that under some circumstances it is beneficial to know accumulated resource use (in terms of iterations or total run-time). In such a scenario, the designer can associate a state variable for each resource object of interest and provide run-time behavior to manage the additional variable state; in this example, its accumulated use. Finally, (3) additional data structures can be constructed and associated with any model object. Simulation kernel operations exist to register a new object property associations by name and to return its current associations by name.

4.2 Simulator Framework

On top of the data structures and library functions, presented thus far in this chapter, are built the design environment tools. This, and the following, section describes a tool that has been developed to assist in the design of reconfigurable DSP systems by performing event driven cycle-level simulation of the same. A command line synopsis of the shell interface can be found in listing B.2. One objective during simulator development was for it to have a modular framework wherever possible so that algorithms, dynamic mappings, architectures, and schedulers could be constructed with interchangeable model components. This helps support/encourage design space exploration since it reduces the iteration delay between alternative system evaluations. Moreover, successive runs can be evaluated using design dithering where slight variation of one model component is changed at time to evaluate system sensitivities and robustness. As an example, a designer might opt to compare various combinations of run-time optimization functions to see the impact of overall system performance.

Figure 4.3 shows some of the key objects of the simulator framework and corresponding select member functions. The framework roughly divides into four parts: (1) simulation kernel, (2) simulation control, (3) systems modeling, and (4) systems analysis. An algorithm model, architecture model, candidate imple-

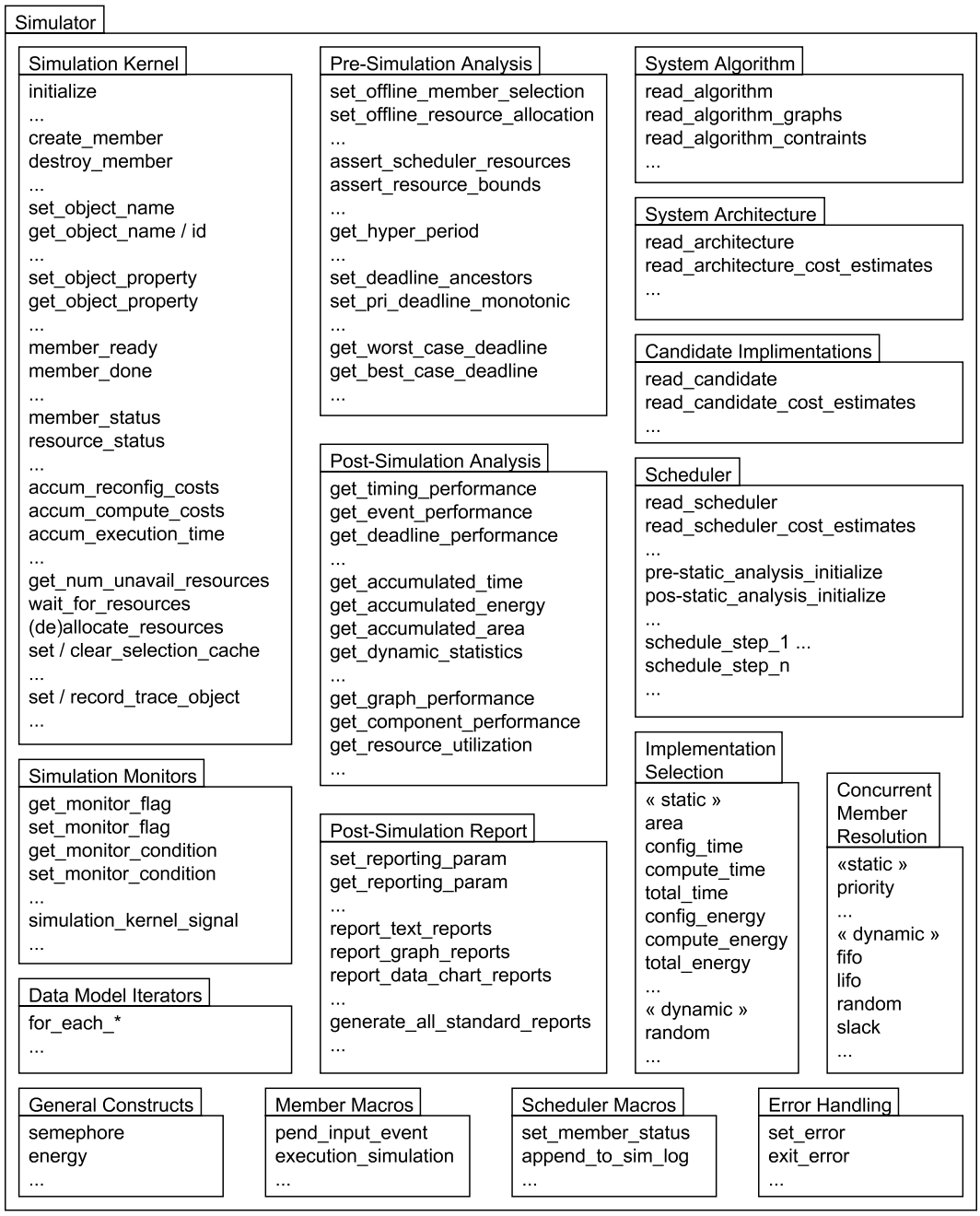


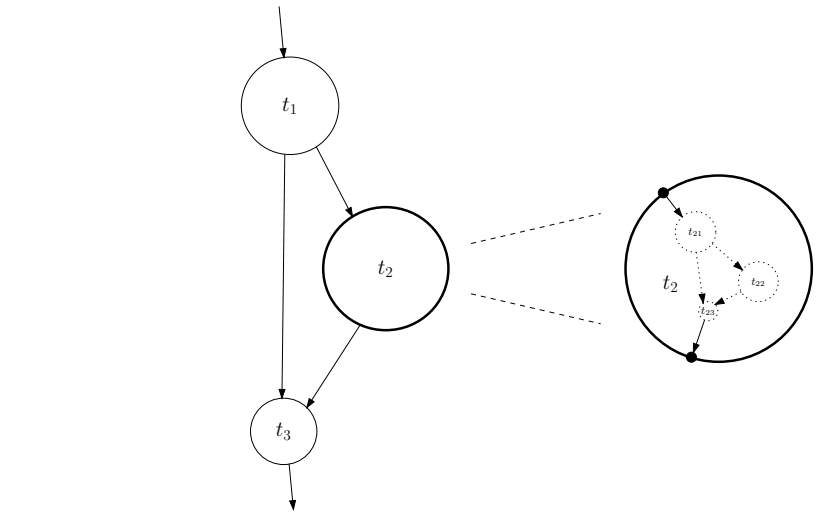
Figure 4.3: Simulator framework key objects and select functions

mentation model, and scheduler model all have multiple views. For example, an algorithm model has a graph view, a behavioral view, a constraints view, and an interface view. The simulator framework has functions for reading the required views for each system model component. The remainder of this section describes each part of the simulator framework in more detail.

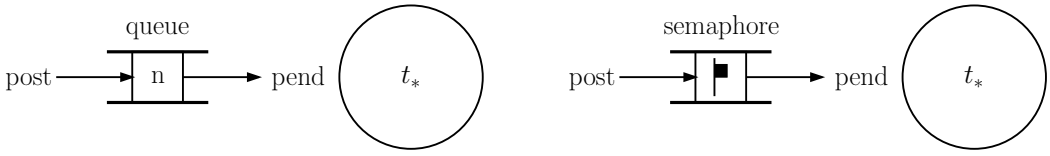
4.2.1 Algorithm Modeling

An algorithm is described using graphs, where nodes represent tasks[5] and edges represent communication between tasks. An exact model of computation is not specified by the methodology. This is left to the discretion of the designer as appropriate for particular system design. Once a model of computation is chosen, it must remain constant within the context of the local graph (called a cluster in the simulator). Additionally, the scheduler that manages the graph computation must be aware of model assumptions. This does not prohibit other algorithm models from coexisting within a single system design. However, each models of computation does require a runtime scheduler and separate subgraph that is aware of the representation. In summary, both the design methodology and design environment supports multiple and mixed models of computation (MoC).

[5]it is assumed that a task has a single thread of control and therefore a task and a thread are one in the same.



(a) HCDFG



(b) Event Queue

(c) Event Semaphore

Figure 4.4: One implemented algorithm modeling MoC

The designs completed during this research utilize, as shown in figure 4.4(a), hierarchial control data flow graphs (HCDFG) with, as shown in figure 4.4(b), event message queues and, as shown in figure 4.4(c), semaphores. These concepts are well formed elsewhere and will not be discussed here. In appendix B, listing B.4, an example task code fragment that makes use of this computation model is shown which makes this description effectual.

The system scheduler, described a few sections below, is responsible for re-

Table 4.1: Task life-cycle: flow of execution

Phase	Task Status
1	input events satisfied (per encoded input activation logic)
2	signal parent scheduler - ready for computation
3	wait for implimentation/configuration assignment from scheduler
4	perform data and control behavior ("the task")
5	request simulation kernel to record performance estimations
6	signal parent scheduler - done with computation
7	wait for next input event condition

source management and ordering of execution. The simulation framework makes use of an explicit request and granting scheme that is encoded into an algorithm to coordinate between the algorithm and its parent scheduler. Table 4.1 summarizes the flow of execution (life cycle) of a task from the perspective of the task.

An algorithm task informs its parent scheduler when it needs service and subsequently waits for a response. When appropriate, the scheduler informs the task to resume. At some point prior to completion, the task informs the simulation kernel to account for the performance estimates based on the implementation and resources assigned by the scheduler. After the task completes, it again informs the assigned scheduler so that post-processing, if any, can be performed.

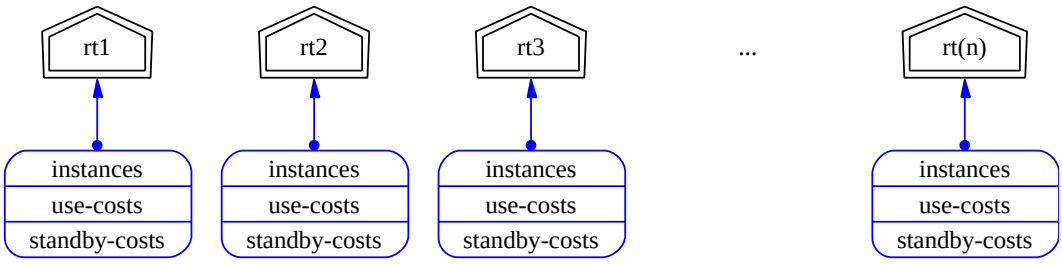


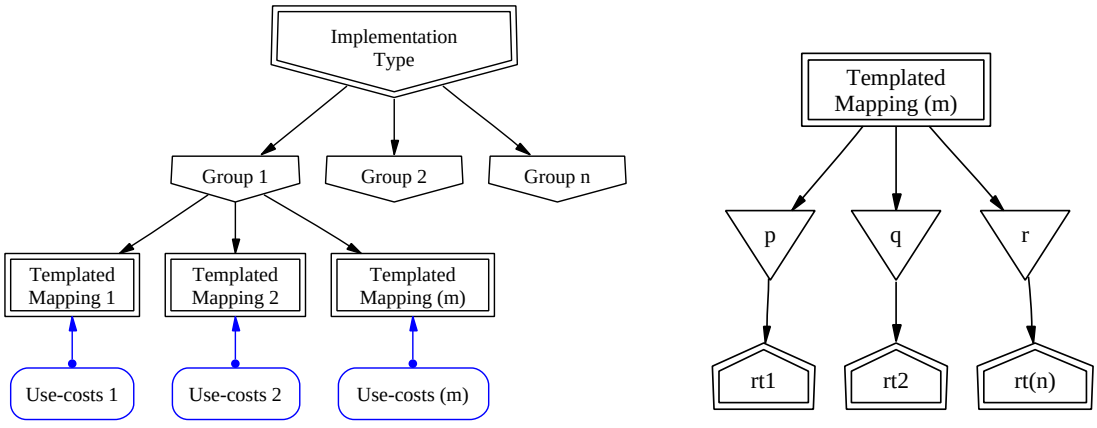
Figure 4.5: Methodology: System architecture base model

4.2.2 Architecture Modeling

From the perspective of run-time management strategies, the architecture can be abstracted into resource types and associated resource instance counts. Both computation and communication hardware architectures can fit within this abstraction[6]. A templated-mapping implementation specifies resource types which allow for more flexible run-time resource assignments under varying conditions.

An example system architecture specification for resources types rt_1 to rt_n is presented in figure 4.5. As shown, the methodology incorporates base properties for resource use-cost and resource standby-costs. Additional properties may be associated and considered by specific run-time schedulers as desired.

[6]This would not be the case for run-time schemes that employ sub-task re-optimization, just-in-time compilation, or synthesis techniques; all of which would require awareness of specific algorithm operations, hardware resource operation capabilities, and hardware interconnect structures.



(a) Candidate Implementations

(b) Templated-Mapping

Figure 4.6: Methodology: Candidate implementation conceptual model

4.2.3 Candidate Implementation Modeling

For a given algorithm, each node and edge (computation and communication) is assigned to one, and only one, mapping template type during system design exploration. This type is the base object of a structure that contains one or more templated-mapping implementations which maps to the architecture resource. The entire structure is collectively called the candidate implementations. The methodology is strict about the representation of the candidate structure as represented by the data model of figure 4.1.

Conceptually, as shown in figure 4.6(a), the collection of templated-mappings establishes the set of implementation alternatives. As shown in 4.6(b), each alternative specifies the set of resource types and instances required for the map-

ping[7]. Each alternative has an equivalent black-box input-to-output behavior, but different internal structure, resource requirements, and use-costs; the greater the number of alternatives, the greater the run-time flexibility, due to the wider choice of implementation options. However, this increasing flexibility is accompanied by increasing overhead for runtime search and implementation storage.

The candidate implementations are read from a design database into the internal data model during simulation elaboration. This will be discussed in more detail in the following section.

4.2.4 Scheduler Modeling

A system must have at least one registered scheduler. However, there is no methodology or simulator framework-imposed upper limit on the number of schedulers that may exist in a system design. For a given algorithm graph there must exist at most one MoC and at least one parent scheduler. Hierarchical sub-graphs may use some other MoC as long as the sub-graph component interface behaves according to the communication model of the incorporated parent graph. When a graph is instantiated, its member elements are assigned to a scheduler. More specifically, each member receives two scheduler assignments: (1) a reference for

[7]Please refer to the Section 3.5 for a rigorous mathematical model.

Table 4.2: System scheduler base responsibilities

	Responsibilities Description
1	Resolution of overlapping resource demands
2	Run-time implementation selection and reconfiguration
3	Post-execution member reference cache policy enforcement

which scheduler to signal when the member is ready for computation[8] and (2) a second reference on which to signal when the member has completed the computation[9]. These references may be assigned to the same scheduler. This relational mapping is typically established at design time. However, it is conceivable that a scheduling scheme, that supports run-time parent-schedule re-assignment, could be devised.

The basic responsibility of a scheduler within the context of this methodology is: (1) to make a run-time choices in the ordering of execution when there are concurrent demand on a limited resource; (2) to select, and configure if necessary, appropriate implementations amongst the available candidate templated-mappings; and (3) to decide what to do with the templates and resource allocations after the execution has completed. Each choice is guided by run-time optimization functions. Table 4.2 summarizes the responsibilities.

As an example application of scheduler modeling, consider the single-threaded

[8]See phase 2 of table 4.1

[9]See phase 6 of table 4.1

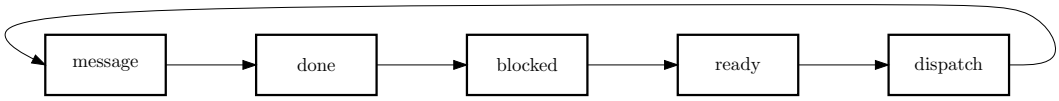


Figure 4.7: An example scheduler: the stdbe and its life-cycle

dynamic best effort (stdbe) list scheduler shown in figure 4.7. As the name suggests, this scheduler consists of a single task (thread) and has five processing sub-phases. (1) One phase processes signal messages from assigned graph members; the message phase. (2) Another phase processes the members that have completed execution; the done phase. (3) Another processes all members that are blocked a waiting on resource availability; the blocked phase. (4) Another processes those members that are ready for execution; the ready phase. And finally, (5) one dispatches the computation; the dispatch phase. The scheduler begins with the message processing phase whenever a managed graph member signals for service. The done phase scheduling code can be found in listing B.7.

Optimization Objectives Functions

Of the three scheduling responsibilities listed in table 4.2, two are optimization problems that can make use of run-time functions to guide each choice. At any moment, a sample of the system state variables can be evaluated according to some heuristic encoding in order to optimize the respond toward some performance

objective.

When there are multiple demands on a resource, a resolution of conflict need be determined. Likewise, when a computation or communication has multiple candidates available, then a selection needs to be made amongst them. These two optimization problems are not independent. If performed separately, for example the ordering of concurrent demands is committed in a first step without consideration of the available implementation selection and how they relate to the overall optimization objectives, localized minimize can be reached that miss the optimum objective. Although the heuristic complexity will increase, in many instances, it should consider the ordering and selection optimization together in order to circumvent localized minima. A combined optimization provides a broader view of the solution space and increases the opportunity to reach global optimal points at each heuristic evaluation. For the sake of discussion, these two optimizations will be introduced separately below.

Overlapping Service Requests: A required scheduler responsibility, whenever there are multiple demands on system resources at overlapping moment in time, is to enforce a resolution policy. At the most basic level, the scheduler itself is a resource and therefore[10] whenever there is concurrent behavior ready for execution, there is overlapping resource demand. There also may or may not be

[10]This assumes that the concurrent demanding members are assigned to a common scheduler.

overlapping demand for other non-scheduler system resources; which in-practice is more relevant. In this situation the scheduler must establish an ordering based on some optimization objective. There has been significant research in classic scheduling on this subject and many schemes exist. The structured design of the simulator framework, allows many ordering functions to be easily constructed and considered. Currently both static-parameter ordering function and dynamic-parameter ordering functions have been developed. For a sample, see the "Current Member Resolution" object of figure 4.3.

Templated-mapping Selection: Before a computation or communication can be dispatched an acceptable mapping must be selected and configured from amongst the available candidates. The selection optimization evaluations of relative performance are performed against the template-mapping properties defined by the base-methodology and/or any additional arbitrary user-defined properties relevant to a run-time optimization strategy. See the "Implementation Selection" object of figure 4.3. The search has a three-part configuration that establishes the objective function behavior: (1) the search objective function itself, (2) the ordering directive of the search, and (3) the matter of whether the search should accept sub-optimal selections or deliver only the optimum — its' strictness.

The first step under any search configuration is to order-rank the selections

according to the objective function and the ordering directive. Next, if the search specifies that only the optimal be returned, then it is returned and the scheduler can dispatch the execution once all resources are available, allocated, and configured. If the optimization objective specifies that sub-optimal selection are permissible, a heuristic can apply secondary objectives to select amongst the orderings; for example, return the first that would allow execution to proceed as soon as possible in order to minimize response delays.

Post-Execution

To guide the decision process for post-execution reference de-construction, the scheduler considers the run-time choices as defined by the reference modes of both (1) the member template reference and (2) each resource type reference. In essence, a graph member maps to a collection of specific resource instances by way of a multi-level reference set. These references are stored in a multi-level cache, as depicted in figure 4.1, that can be independently configured to, upon subsequent de-referencing, bypass the run-time implementation search processing, and/or bypass the run-time resource allocation processing, and/or bypass the run-time resource configuration processing. This can have significant benefit in the reduction of overhead under certain system conditions.

4.2.5 Static Design Analysis and Optimization

The evaluation of system run-time performance can be, and sometimes is, called dynamic design analysis; called such since the evaluations are with regards to how the system behaves over time while in operation. It should be little mystery that this time-dependent, or run-time, behavior strongly correlates with the systems static structural compositions. Understanding this static composition is key to understanding and refining the dynamic run-time system performance. This is the goal of static design analysis and optimization.

Given the importance of static design analysis[11] in relation to design exploration and iterative refinement, the simulator incorporates a framework for the structured management of this design activity. As with the other simulator "frameworks," a system designer may code additional routines of interest and incorporate them within the framework to support their design effort. To clarify what is meant by static design analysis, consider the process, well documented in the literature, of assigning priorities to system tasks using a deadline monotonic scheme. This and several other analysis have been incorporated within the framework.

In general, the objective is to either: (1) check that some condition does or does not exist, property assertion; (2) analyze the system and report the observations;

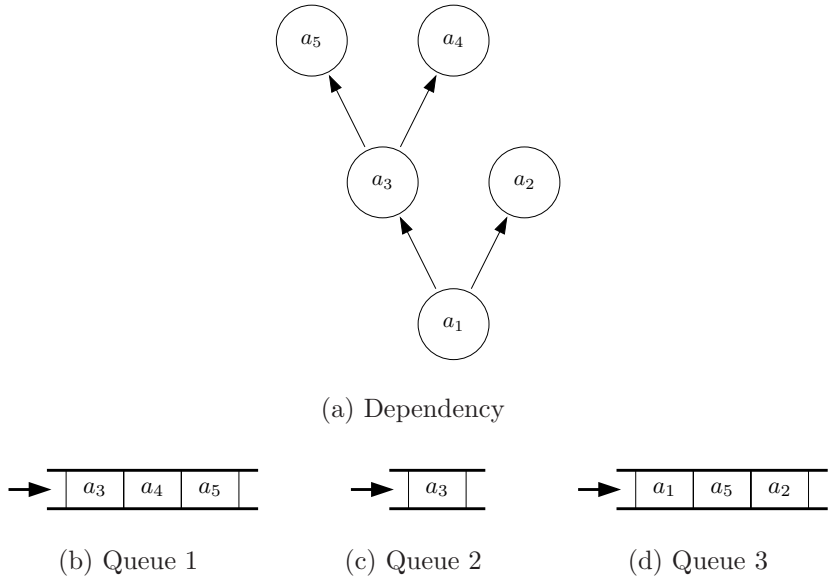
[11]Within static design analysis we include static designed optimization.

get-analysis; or (3) optimize the system and update the relevant design elements, set-optimization. A naming practice of assert_name, get_name, and set_name, has been established to clarify the analysis objective by its name. An example of some implemented static design analysis routines are identified in figure 4.3. See "Pre-Simulation Analysis" and "Post-Simulation Analysis."

Analysis Order Dependency

In a modular and orthogonal manor, a design can be statically analyzed and optimized by the set of developed routines in a specified order. The sequence of analysis steps can quickly grow and become a challenge to manage. Moreover, there are often dependencies in the order of analysis application. To assist in this organizational effort, the simulation framework incorporates an analysis registration capability that allows each of routine to publish its name, its type, and its analysis dependency. Subsequently, during static design analysis, checks can be made to ensure that the analysis is conducted within the required context. For example, all dependent analysis and optimizations can be evoked as required and ordering errors introduced by the designer can be checked and enforced. See listing B.5 for a code example of the analysis request interface.

For clarification, consider figure 4.8(a) that shows an example dependency tree



(a) Dependency

(b) Queue 1 (c) Queue 2 (d) Queue 3

Figure 4.8: Static design analysis dependency enforcement

for a set of five static design analysis routines[12]. In this example, the requests in both queues of figures 4.8(b) and 4.8(c), with the analysis being performed in the order indicated by the arrow, would yield equivalent results. The request indicated in the queue of figure 4.8(d) would yield an analysis dependency error.

4.3 Simulation Execution Flow

The simulator has been implemented within a framework perspective in order to facilitate lowered overhead interchange of system models. This does, however, come at the cost of some simulation overhead. The division of orthogonal simula-

[12]Analysis 3 depends on both analysis 4 and 5.

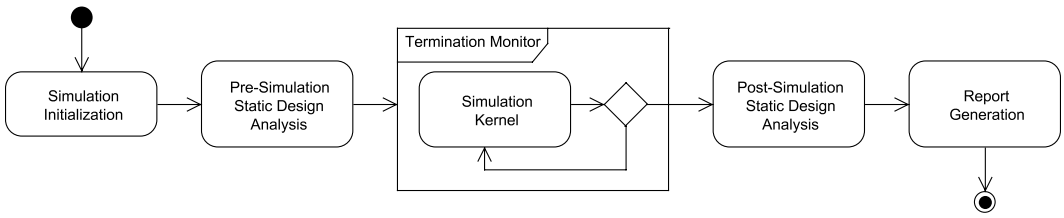


Figure 4.9: Simulation base execution control flow

tion function into separate objects modules according to a standardized framework is not only good for software development; it also enables the reuse of these objects at higher levels of abstractions. In this section the basic simulation execution flow is presented. Subsequently, the simulation condition monitor is introduced. The condition monitor is used to create scripted simulation execution-flows.

The basic flow of execution involves a linear flow starting with simulation initialization, to pre-simulation static designed analysis, then the simulation itself, next a post-simulation static design analysis, and finally the reports generation. This base-case execution flow is shown in figure 4.9.

A simulation begins with initialization. System algorithm, system architecture, candidate implementations, and system schedulers are read from permanent store into internal data structures. Each object model sequences through the necessary steps to initialize — in preparation for simulation. For example, views for algorithm behavior, algorithm graph structure, and constraints are read into the internal data model. Subsequently, the queue of pre-simulation static design

analysis functions are executed one-by-one. Next, the simulation execution runs to termination. Termination is defined by: (1) sufficient simulation cycles, (2) sufficient simulation time, (3) algorithm completion, or (4) until signal by the simulation monitor. After simulation, the queue of post-simulation static design analysis functions are executed one-by-one. And finally, the report generation facility is run and the simulation exits.

The next section introduces the simulation condition monitor and how it is used to create scripted simulation execution flows that can be used to automate fine-grained design exploration and optimization.

4.3.1 Simulation Condition Monitors

A condition monitor framework enables the observation of the overall simulation environment and facilitates the structured management in the simulation execution control flow. The basic intent of this capability is to allow structured early termination under specific known conditions such as run-away execution, non-convergent behavior, or deadlocked simulation, for example. More importantly however it is useful to created higher-levels scripted "wrappers" in the simulation execution flow.

When condition monitors are combined with simulation execution, static de-

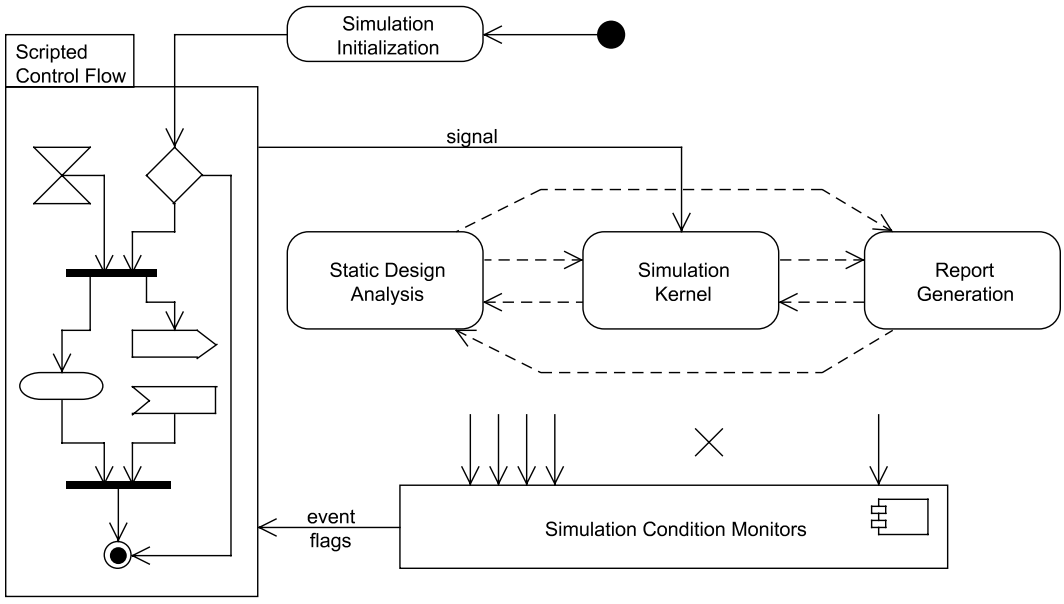


Figure 4.10: Condition monitors and scripting framework for design exploration

sign analysis, and the reporting framework, new design exploration capabilities emerge. This concept is depicted in figure 4.10.

4.4 Environment Design-Flow

The simulator developed in conjunction with this research is one part of an overall design process proposed for reconfigurable DSP. It supports the design effort by providing quantitative evaluation of the performance of a given system design. At this point it is useful to have a look at the broader perspective within which the design environment, and design environment tools, fit. Two diagrams

are discussed below which identify the activities that must be completed during the design process. One figure shows a responsibility assignment for the activities and the other shows the design-flow and their dependencies.

As depicted in figure 4.11, there will typically be at least two classes of designers involved; system designers and domain specialist. A system designer manages the highest-level overall design issues as they relate to system-level behavioral objectives. This responsibility will often be supported by CAD tools, such as a system-level simulator. A domain specialist applies specific and detailed domain knowledge to yield design components that offer relevant system building blocks. A domain specialist will be assisted by CAD tools such as compilers and synthesizers. The figure shows one breakdown of design-activity responsibility assignment. It is by no means definitive. Rather it should be viewed as a road-map to identify the general relationships between designers, system simulators, compilers, and synthesizers in the proposed methodology. One observation is that even in the presence of high automation — such as compilation, synthesis, and system-level simulation — the design process requires significant designer intervention.

Figure 4.12 repeats the system design activities in a form that identifies the dependent ordering that exists between them. This figure also exposes the iterative nature of the design refinement process. Starting at the top of the figure, at the solid circle, there are both concurrent activities and synchronized design

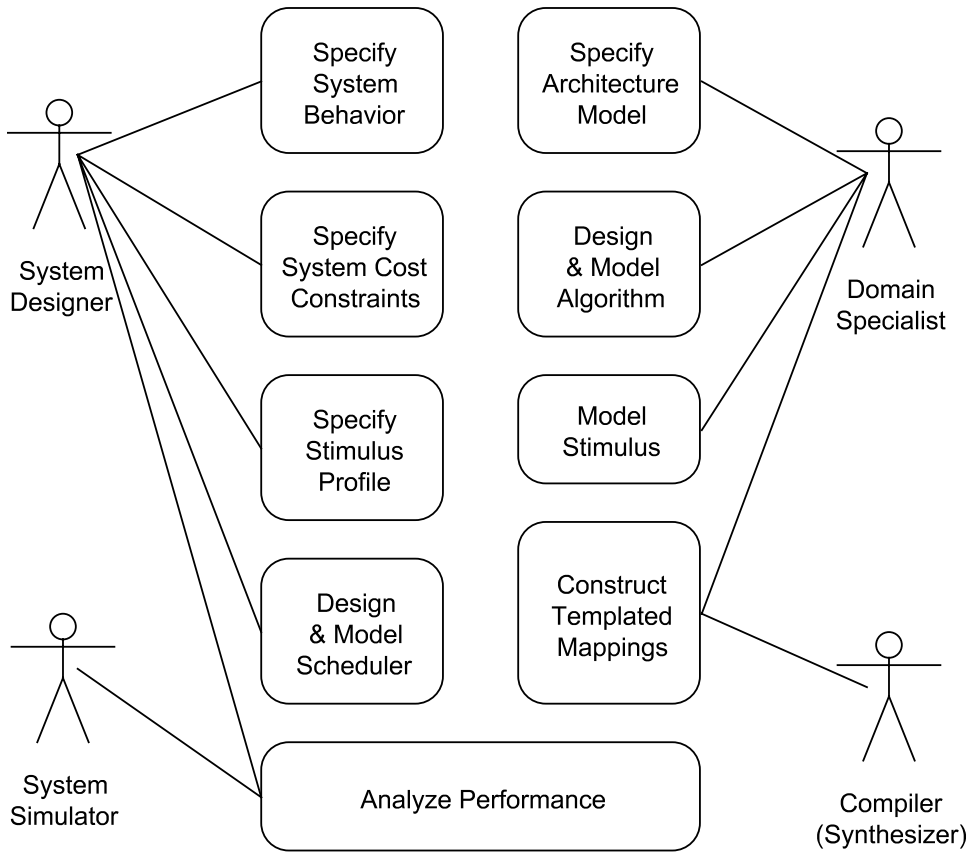


Figure 4.11: Methodology design-flow responsibility

phases that are required to create the system. After a design pass, it is analyzed to determine its ranked performance. If the ranking result is satisfactory, the design process is complete. Otherwise, subsequent iterative passes, and design refinements, are repeated.

Given the immense size of the solution space, a designer need devise a strategic "divide and cover" approach to systematically evaluate and rank designs within the solution space[13]. First, a coarse-grained division of design alternatives are ranked to identify the subsequent region, or regions, of interest. Next, these sub-regions are subdivided, in like manor, and the process continues until the ranking results converge.

4.4.1 Tool Scripting

Design exploration involves both course-grain and fine-grained system design refinement. The notion of fine-grained refinement has been discussed in section 4.3; using scripted simulator control flow. For coarse-grain refinement, design environment tools scripting can be useful. In addition to the system simulator, there were four other design environment tools developed. These tools are summarized in table B.3. By adhering to the design environment framework and additionally exposing the tool run-time configuration options at the shell command line

[13]One approach to design exploration.

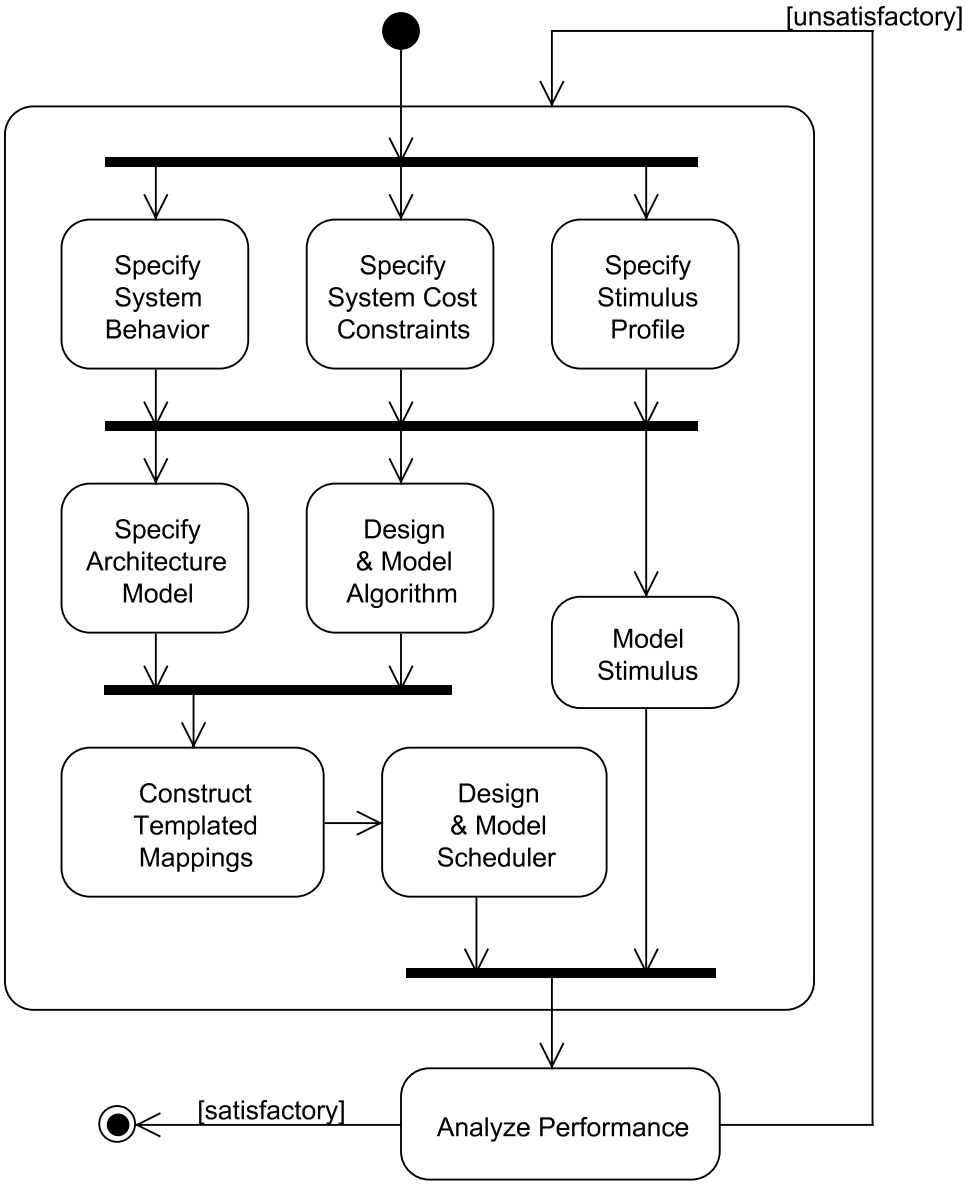


Figure 4.12: Methodology design-flow sequence

level[14], tool scripting can significantly accelerate the course-grain design refinement process.

4.4.2 Graphical User Interfaces

Design environment design tools with shell command line interfaces were found to be most useful for encapsulating sub-parts of the overall design flow. This expedited the design exploration by allowing incremental evaluations via scripts in an automated and structured approach as has been discussed earlier in this chapter. However, the resultant datasets were very difficult to manage without the aid of graphical user interfaces (GUI). It may be of some interest to note that although there were four separate GUI tools developed, none were "shell tool front ends." For example, none simply provide a convenient way of managing the command line options (one-to-one) of some equivalent shell tool. Rather, they were most useful in design-flow scripting, encapsulating complex sequences with a parameterized execution.

So as to not get diverted from the central focus of this research effort, GUIs were developed on an as-needed basis. The minimum set included a GUI for (1) design database management, (2) system components generation, (3) simulation control flow automation, and (4) evaluation results navigation. Two additional

[14]As and example, see the command line synopsis of listing B.2

GUI's[15] greatly increased the design flow efficiency: (5) a graphical system component editor and (6) a simulation timing analyzer.

In summary, the GUIs were found to increase the design efficiency (1) by improving design database navigation; entry and review, (2) by the encapsulation of complex parameterized design-flows, (3) by the organization of searchable and reviewable exploration histories, and (4) for complex data-structure visualization.

4.5 Chapter Summary

A design environment, motivated by the templated-mapping methodology advocated by this research, has been developed. The details of this environment are presented within this chapter. First, the core infrastructure, including the foundation internal data model and other environment-wide primitives, are introduced. Subsequently, a description of a key design environment tool, the system-level simulator, is presented. The simulator takes a quantitative approach to evaluate and rank system performance. It has been implemented using a framework that facilitates fine-grained design exploration using scripted simulation execution. Finally, a design environment design-flow is proposed that emerges from the overall design methodology. Both a design activity responsibility breakdown and base-case

[15]The graphical system component editor and a simulation timing analyzer were obtained from

design-flow is discussed. Some issues surrounding course-grain design exploration through tool-scripting and design-flow support with graphical user interfaces are briefly introduced.

Chapter 5

Design-flow Evaluation

> The grand aim of all science is to cover the greatest number of empirical facts by logical deduction from the smallest number of hypotheses or axioms. Albert Einstein US (German-born) physicist (1879 - 1955).

In this chapter we demonstrate the design environment and methodology design-flow by composing systems and running simulations with various run-time optimization objectives. Here, rather than focusing on a single design example, we have chosen to generate design families. A basic scheduler is implemented and described along with a set of objective functions for task ordering and template selection. Next we describe a performance metric defined to rank the quality of run-time selections. Finally we present one example from the generated design database and a few simulation report results.

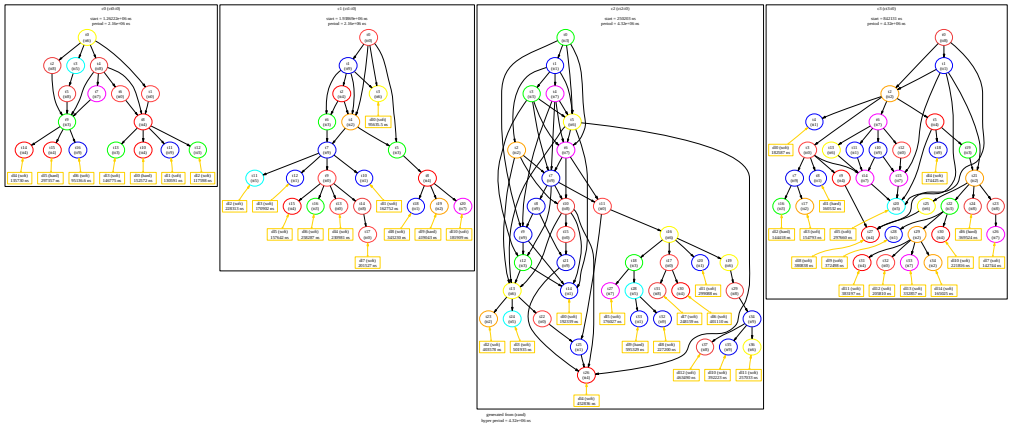
5.1 System Component Composition

A system is composed of one or more algorithms, architectures, candidates, schedulers, and objective functions. In the next several paragraphs we briefly discuss the strategy used to generate system design cases that are subsequently analyzed with simulation.

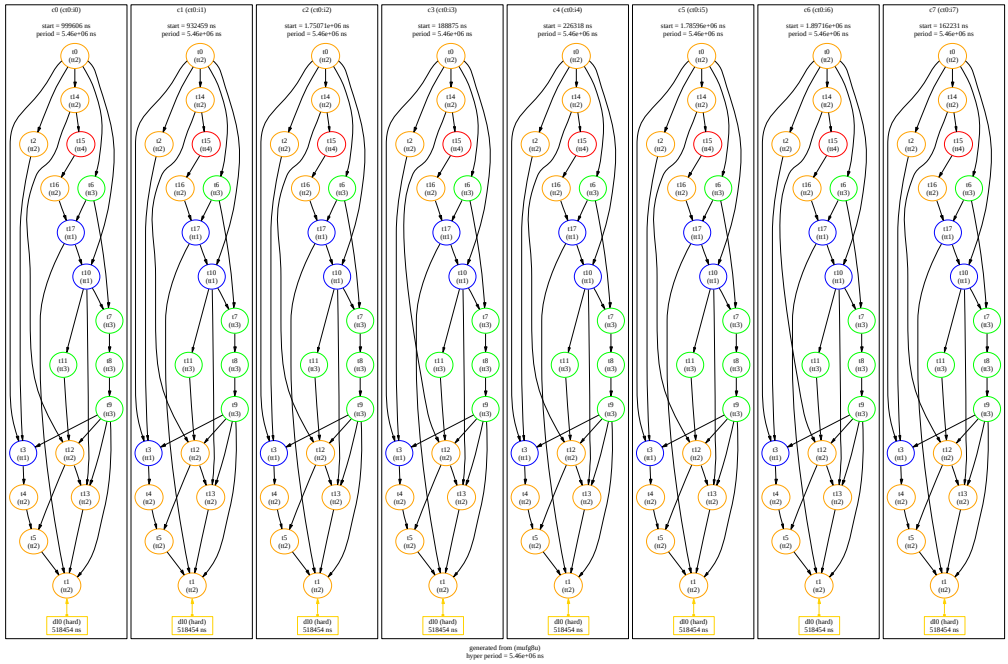
Algorithms: Algorithms are generated with one or more independent graphs. Each graph has one or more instance copies and are assigned activation times. The graph nodes have varying operation counts and are assigned hard and soft deadlines. Three forms of node connectivity are explored: (1) parallel unconnected algorithms, shown in figure 5.1(a), (2) random connected algorithms, shown in 5.1(b), and (3) series-parallel connected algorithms, shown in 5.1(c). Parameters are used to specify the series length, parallel width, fan-in, fan-out, etc. The parameters for each connectivity form are dithered from a median value, one at a time, to create the families of algorithms. Thirteen parallel, fifteen random, and seventeen series-parallel algorithms are generated in total.

Architectures and Candidates: Architectures are composed of globally interconnected reconfigurable units blocks. Each reconfigurable unit block (RUB) is independently configurable with data inputs, data outputs, an execution clock,

(a) parallel unconnected algorithm



(b) random connected algorithm



(c) series-parallel connected algorithm

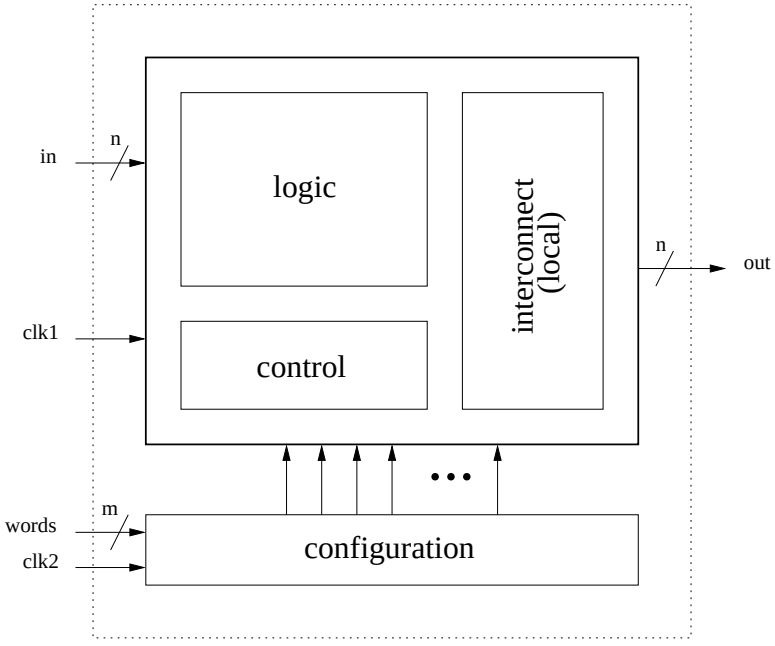
Figure 5.1: Generated algorithm families examples

a configuration clock, and configuration word inputs. The model is shown in figure 5.2(a). Each RUB has logic, control, configuration memory, and local interconnect. The RUB's are tiled, as shown in 5.2(b) and 5.2(c), to compose 36 architectures that each have a fixed area equivalent to 10 million minimum-sized gates. The total area is divided between RUB's and global interconnect with varying RUB granularities. Three independent variables are used; (1) the number of RUB types, (2) the number of instances of each RUB type, and (3) a flexibility index for the resultant structure. From these variables architectures are composed of six dependent parameters; (1) global interconnect and configuration-word routing overhead; (2) total RUB gates, (3) RUB control gates, (4) RUB local interconnect, (5) RUB logic gates, and (6) RUB configuration words.

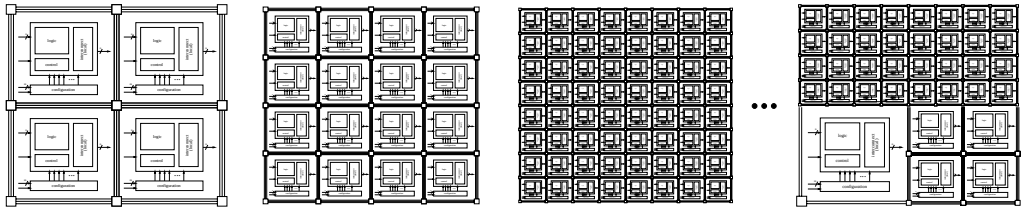
For each algorithm node type[1] and mappable architecture resource, configurations are generated using a four-parameter model; (1) required RUB types, (2) required RUB instances, (3) candidate speed up, and (4) candidate activity factor. An example candidate is shown in 5.3(a).

Each architecture parameter and candidate parameter uses a weighted linear approximation (within a specified range given by a minimum and maximum value) of a weighted polynomial composed of the independent variables terms with constant coefficients and constant powers. In all, there are 1,620 designs (45

[1]This also applies for each algorithm edge type.



(a) Basic reconfigurable unit model



(b) Controlled variation of granularity

(c) Heterogeneity

Figure 5.2: Modeling families of heterogeneous reconfigurable architectures

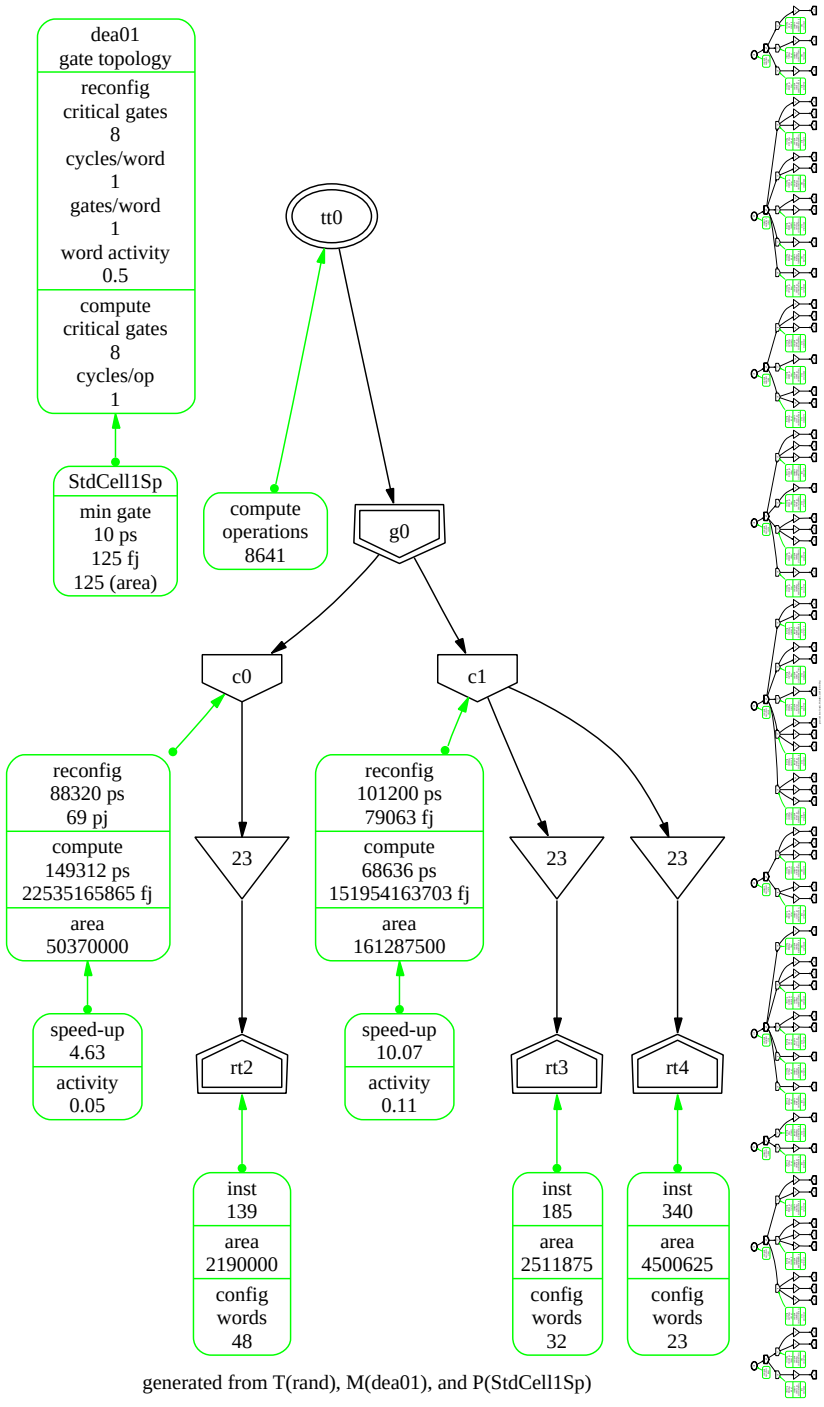


Figure 5.3: Generated candidate and system of candidates example

algorithms and 36 architectures) that can be used to evaluate different scheduling and run-time optimization strategies.

Scheduler: A single-threaded dynamic best effort list scheduler has been composed using five phases. One phase processes new messages from assigned tasks[2] (*message*). Another phase processes the tasks that have completed execution (*done*). A separate phase is used to process tasks that are blocked awaiting resources (*block*). Another phase processes those that have become ready for execution (*ready*). And finally, a separate phase is used to configure resources and dispatch execution (*dispatch*).

The *ready* phase performs task ordering, required when there is more than one task ready, and candidate selection, according to the run-time optimization objectives. If the candidate selection exists and is a valid (cached selection), then the search is truncated. The *dispatch* phase need only configure resources when the configuration cache is invalid. Otherwise, it simply dispatches execution. After execution, the resource reservation and candidate selection mode policy is enforced by the *done* phase. When a policy calls for dynamic modes, the caches are marked indicating that they may be overwritten. They remain valid, however, until they are in-fact overwritten.

[2]Tasks are assigned to a particular scheduler when they are created. They can subsequently be reassigned to another scheduler, however, the current implementation limits such activity to tasks that are in the *waiting* state.

Task Ordering Objective Functions: Task ordering functions have been written to order tasks based on various static and dynamic parameters; such as the task priority in task slack time. Each function takes as arguments two tasks and returns a flag indicating if they are properly ordered according to the objectives of the ordering function. The framework facilitates this with the use of registered predicate functions.

Candidate Selection Optimization Functions: Schedulers select a template from amongst the candidates at run-time based on system-level optimization objectives. When the selection is required, optimizations functions are used to evaluate and rank the candidates. Function have been developed for each of the individual use-cost parameters defined by the methodology. For example by area, configuration energy, or computation time. A parameterizable polynomial relation, of the form in equation 3.1 has also been developed.

Static Design Analysis Routines: A collection of static design analysis routines have been written to use during system evaluations. Some are shown in figure 4.3. Routines to perform bounds checking, such as worst-case and best-case deadline performance, has been developed. Also a collection of routines for assigning priorities, using for example, a deadline monotonic method, is developed.

Table 5.1: Simulation example run key

run	Description
cnfT	minimize configuration time
cmpT	minimize computation time
totT	minimize total time
cnfE	minimize configuration energy
cmpE	minimize computation energy
totE	minimize total energy
area	maximize area use
rand	make random selections

5.2 Example System Simulation

In this section we discuss the results for multiple simulation runs of the random algorithm shown in 5.1(b) on an architecture with five RUB types. The candidate configurations for each unique algorithm behavior are shown in figure 5.3(b). Each system tasks is assigned a priority based on its deadline and a priority-based task ordering scheduling scheme is used wherever task ordering is required. Simulations are run with differing selection optimization objective functions. Table 5.1 describes each run objective.

Four reports are presented that compare the results. The report in figure 5.4 shows a simulation runs summary. Figure 5.6 shows the performance of the run-time candidate selections. Figure 5.7 shows the run-time concurrency and achieved resource utilization. And finally, figure 5.8 shows a breakdown of how

time is consumed during each simulation.

As shown in figure 5.4, it appears prudent to use optimization objectives that focus on minimizing total use-costs rather than use-cost components. For example, objectives that minimize both total energy and total time performed better than those that focus on either the configuration or computation components alone. Total costs oriented objectives also minimized the system delays and response times for this particular design example.

A Quality Measure: Before discussing the selection quality report, a brief introduction to the quality metric slider is in order. As shown in figure 5.5, a selection performance has an upper and lower bound. Worse-case selections are assigned the quality index of zero. Best-case selections are assigned the quality index of one hundred. With this normalized index measure the average performance can be viewed as the fraction of the time that best-case selections are obtained.

As shown in figure 5.6, objective functions that minimize total time (`totT`) and total energy (`totE`) also outperform in terms of selection quality to overhead ratio; refer to the first column of charts for "Time:" performance results. Moreover, although there is slightly more overhead required to make total-cost selections, the quality-to-overhead ratios are also improved. The time overhead and energy overheads are high in this design example. The scheduler is assigned to a general-

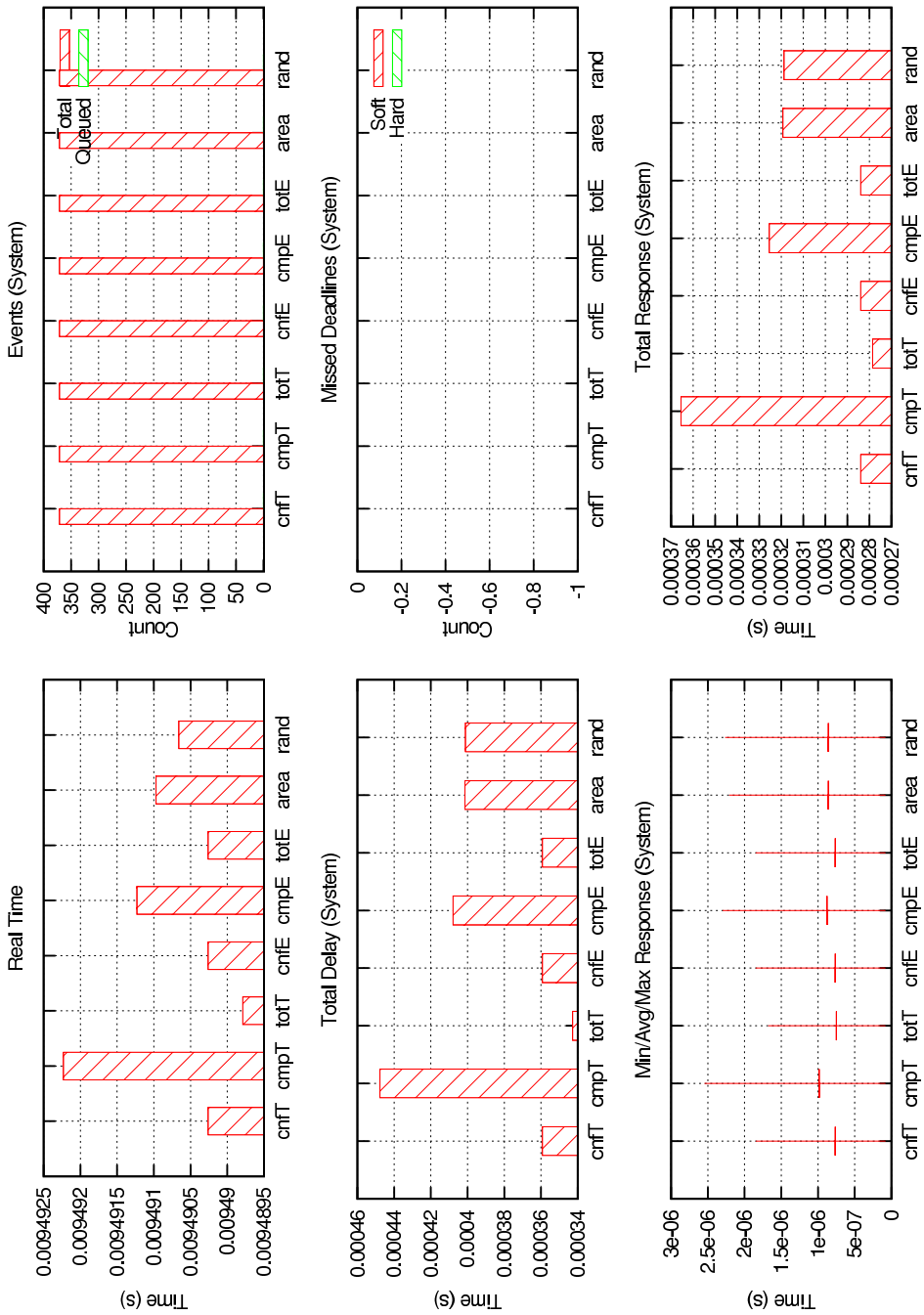


Figure 5.4: Run summary simulation example

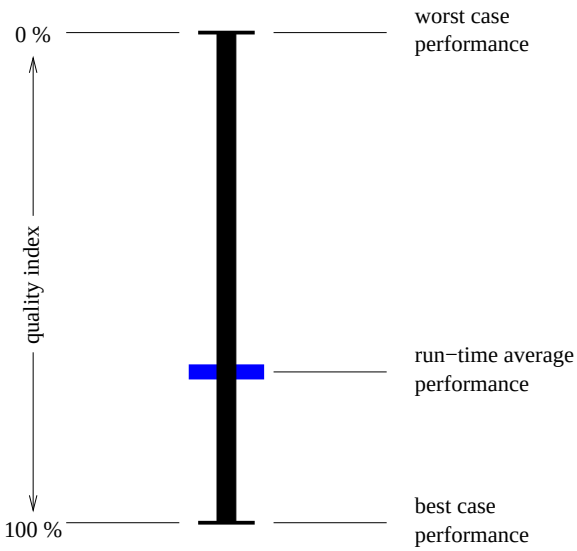


Figure 5.5: Candidate selection performance metric

purpose microprocessor that has high per-operation used-cost. This contribution could be substantially reduced if more specialized hardware were used to carry out the scheduling responsibility.

Based on the simulation report in figure 5.7, we see that each optimization objective achieves equivalent results in terms of concurrency. In each simulation, a maximum of nine concurrent tasks are running and, on average, approximately 1.5 are active over the entire simulation. This result is due to the average resource utilization which remains below 40% in all cases. In no case are there ever blocked tasks that must wait for resource availability. As expected, when the optimization objective is to minimize computation time, greater resource utilization results since one of the assumptive rules of the system generators is that faster mappings

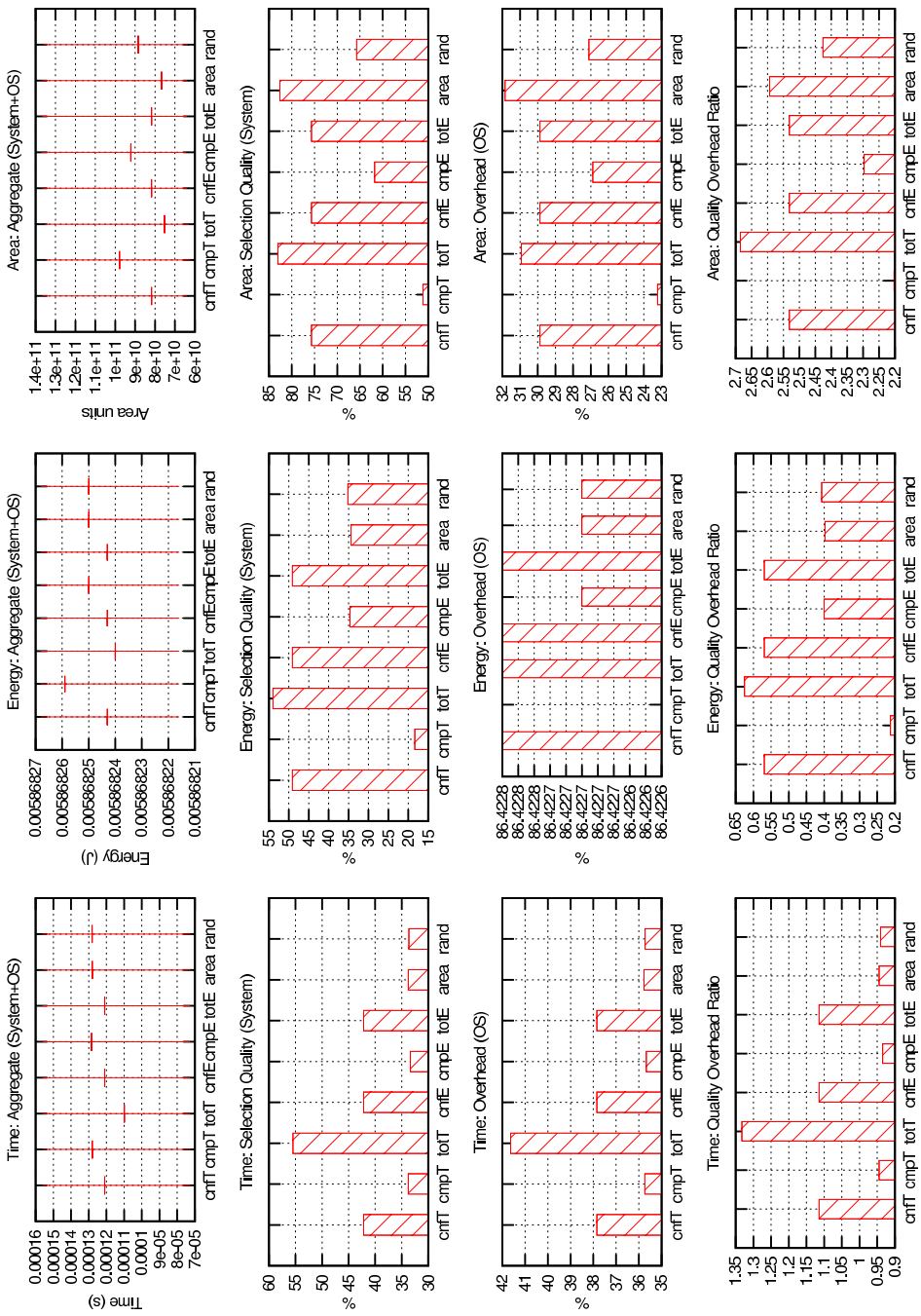


Figure 5.6: Candidate selection quality simulation example

require more resources. And, in the case of maximizing area use, we see that the performance is very similar to minimizing computation energy.

In the final report example, see figure 5.8, is shown a distribution for aggregate system time for each simulation case. Time is broken down into schedulers (RTOS), resource configuration, and execution (computation). Similar reports are available for each system use-costs. Since this one details time, let us focus on the time-based optimization objectives for the moment. We see that the run-time optimization functions do as expected. In the first case, the scheduler chooses candidates that minimize configuration time which is reflected in the report. A similar observation holds for the second case, which aims to minimize computation time. In the third, we see that when the scheduler uses an optimization objective that minimizes total time: (1) greater overheads result; (2) computation time is less than the first case and slightly greater than the second case; (3) the configuration-time is less than the second case and on-par with the first; and (4) the overall time is reduced.

5.3 Chapter Summary

In this chapter we demonstrate the function of the templated-mapping based design methodology, and the implemented design environment, on generated de-

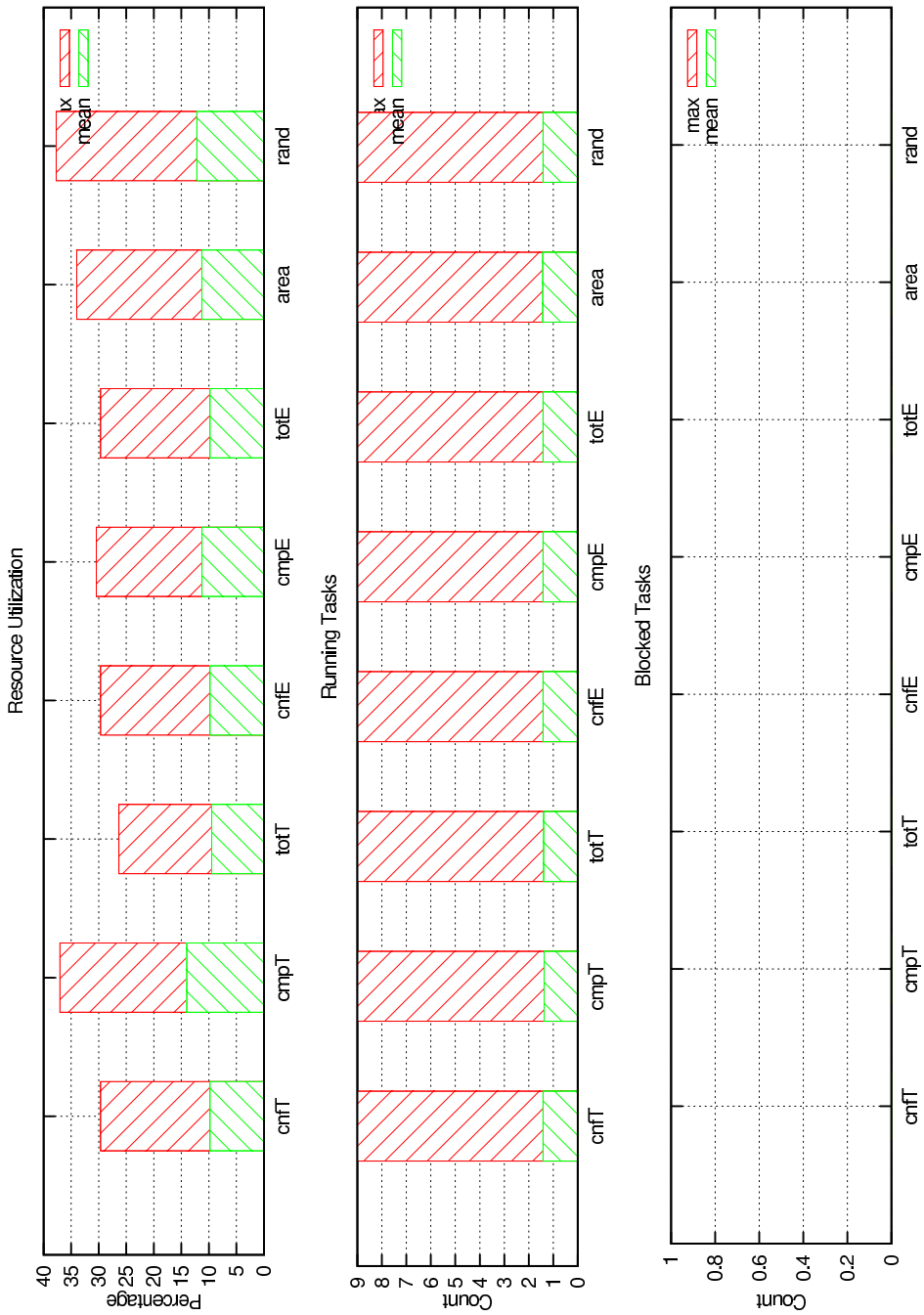


Figure 5.7: Concurrency and utilization simulation example

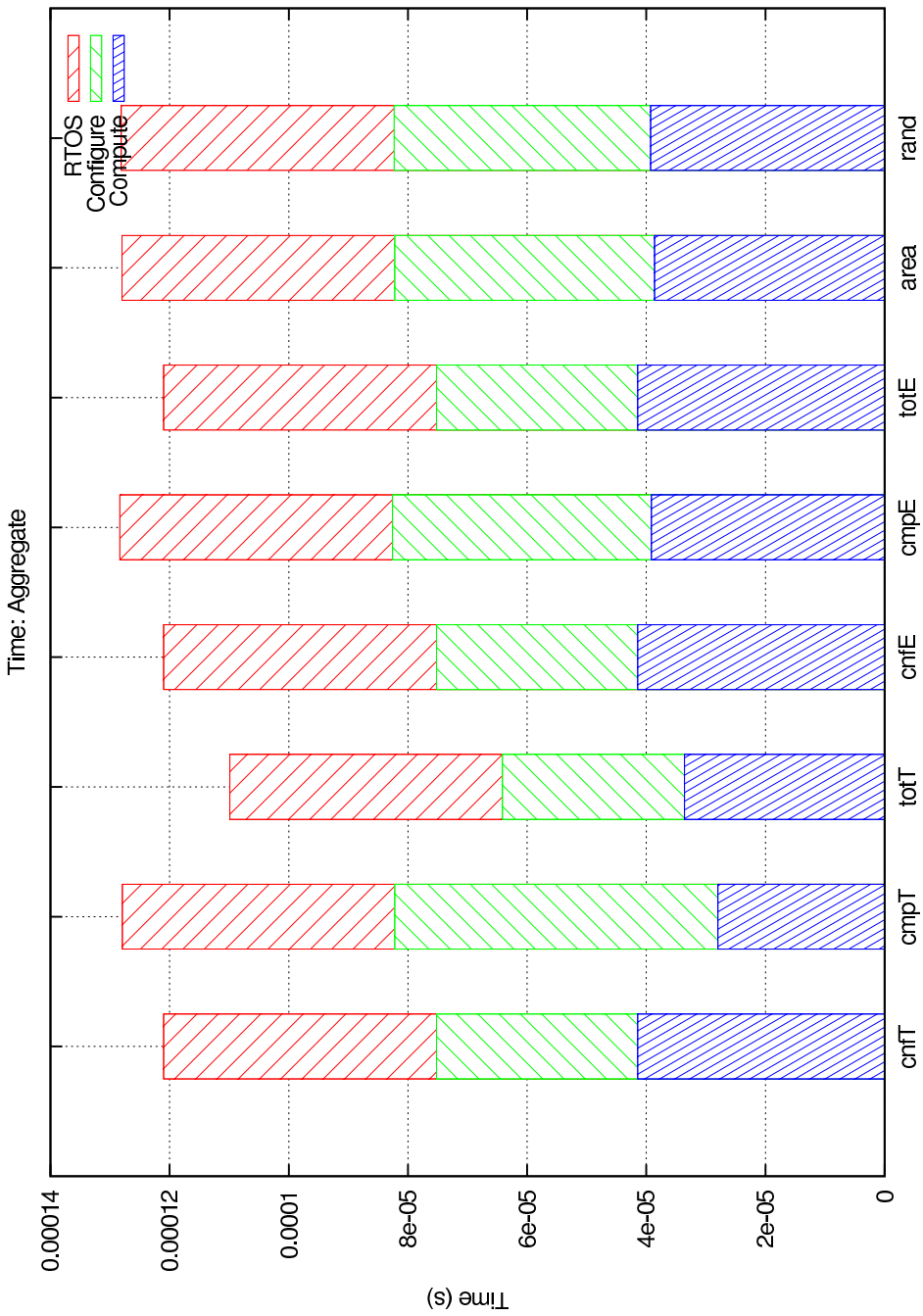


Figure 5.8: Time-use distribution simulation example

sign examples. We describe the strategy that is used to compose thirteen parallel, fifteen random, and seventeen series-parallel algorithms. Next we described an architecture model that is used to compose thirty-six heterogeneous (multi-grained) architectures that are subsequently used to generate candidates for each algorithm. In all 1,620 designs are evaluated.

A system scheduler has been developed and is described along with several task-ordering and candidate selection objective functions. A brief description of some static design analysis routines is presented. Finally, an example simulation run with varied run-time optimization objectives is presented along with some resulting reports. A discussion of theses reports and an introduction to a selection quality measure is presented.

Chapter 6

A Design Exploration Example

> The best thing to give to your enemy is forgiveness; to an opponent, tolerance; to a friend, your heart; to your child, a good example; to a father, deference; to your mother, conduct that will make her proud of you; to yourself, respect; to all men, charity. Francis Maitland Balfour.

This chapter presents a demonstration of the design methodology toward run-time management in a multi-user MPEG decoder system design. This adds to the overall concreteness of the methodology evaluation presented in the prior chapter. Moreover, it is shown that the constructs of the proposed methodology and the analysis possible within the environment, not only enable structured run-time management of reconfigurables, but also facilitates design exploration, refinement, and optimization.

We begin with a description of the decoder algorithm used by each user and its

block-level static and dynamic characteristics. A limited resource heterogeneous architecture, comprised of application specific integrated circuit (ASIC) blocks, two general purpose processors (GPP), a field programmable gate array (FPGA), and a local interconnect switch, is assumed. A reference design mapping to an FPGA is used as a basis and its use-costs are scaled for the GPP and ASIC.

We present an example templated-mapping for the dominant algorithm kernel and show a simulation, obtained directly from the design environment, of the resultant analysis annotated with its' static and dynamic properties. In the remaining portion of this chapter we explore the performance of the design under various run-time scenarios. The chapter concludes with a few highlighted design features and a summary.

6.1 System Description

The system consists of a multi-user simple profile MPEG-4 decoder. A block diagram is presented in figure 6.1. It consists of the following root-level block types: input interface, input vector parser, copy controller, motion compensation, IDCT-based texture processing, texture update, share memory, and FIFOs. A description of these blocks is available elsewhere [Xil05]. Here we focus on how each are mapped to a shared reconfigurable resource set and the costs incurred

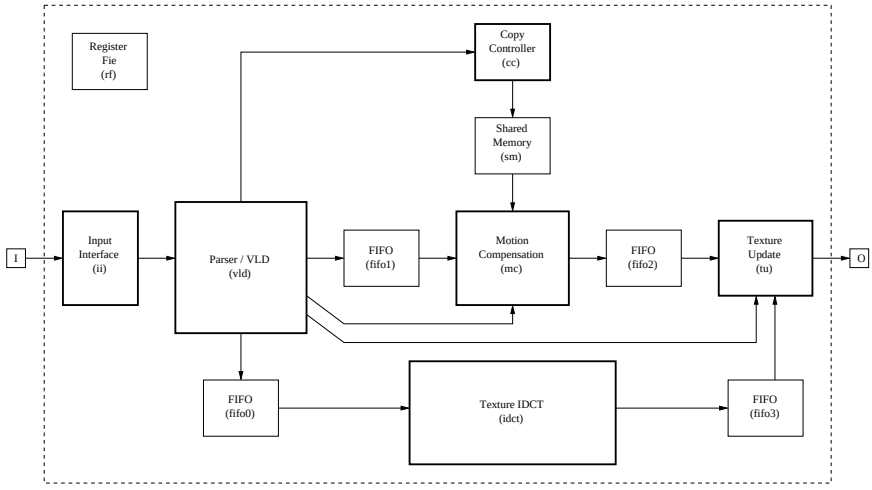


Figure 6.1: MPEG-4 simple profile decoder block diagram

to use a given mapping in terms of resource requirements and the performance metrics of interests (time, energy, and area).

We perform dynamic code profiling of the decoder implementation mapped to a GPP to reveal how the computational complexity is distributed amongst the algorithm blocks. Consider, for example, the blocked labeled texture/IDCT (inverse discrete cosine transform). As shown in table 6.1(b), the IDCT accounts for over 30% of the computation time during image decompression [1]. For a description of algorithm analysis techniques used here, please refer to section 3.3.1. The implementation of this kernel block on a Xilinx Virtex-II Pro FGPA requires 6 brams, 25 multipliers, and 1710 slices; as summarized in table 6.1(a).

[1]Since the profiling is performed on a single threaded general purpose processor, without co-processor accelerators, we can assume the operation count directly correlates with the computation time distribution.

Table 6.1: Static and dynamic MPEG-4 decoder profile

block	bram	slices	mult
ii	1	20	0
vld	0	1700	2
cc	0	340	1
mc	0	340	1
idct	6	1710	25
tu	0	150	2
fifo(0-3)	4	80	0
sm	4	80	0

(a) Xilinx FPGA resource requirements

block	GPP time
ii	1 %
vld	27 %
cc	1 %
mc	27 %
idct	31 %
tu	11 %
fifo(0-3)	1 %
sm	1 %

(b) Dynamic Profile

By design, given the heterogeneity of the system resource, we can implement the IDCT by mapping to other resource structures. This is essential to the methodology since it is the availability of multiple mappings that give rises to run-time flexibility and dynamic reconfiguration. For demonstration, we choose two additional mappings for each algorithm block; one to an ASIC and another to the GPP. We then normalize the use-cost performance of the FPGA reference design and scale for the ASIC and GPP.

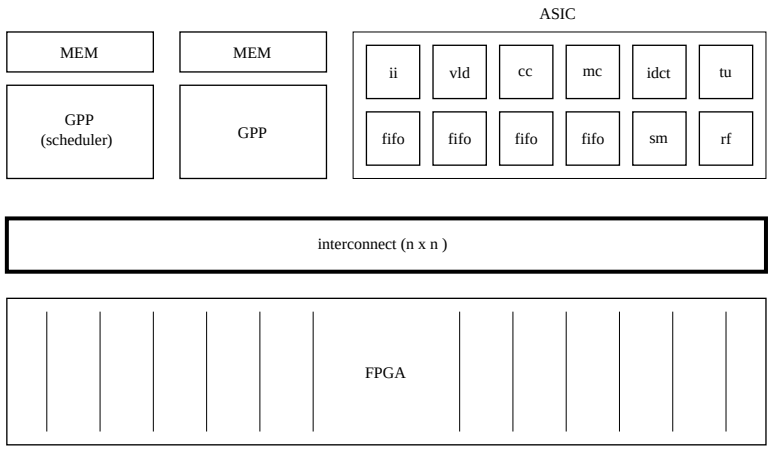


Figure 6.2: Multi-user decoder system resource

6.2 Templated-Mappings

To clarify how templated-mapping are used in real systems, figure 6.3 shows the IDCT algorithm kernel mapping with its candidate implementations. As indicated there are three versions available to the system scheduler for run-time use, each with a corresponding list of resources required by the template. By construction, each version has the same input-to-output functional behavior[2], but performs differently in time, energy, and area. Attributes relevant to the runtime strategy are added as shown in figure 6.4 (use cost, current state, use constraints, etc.).

The templates are constructed for each algorithm block and are available for instantiation as required for each systems user. To construct the template costs,

[2]Care must be given to validate the behavioral equivalency across the candidates.

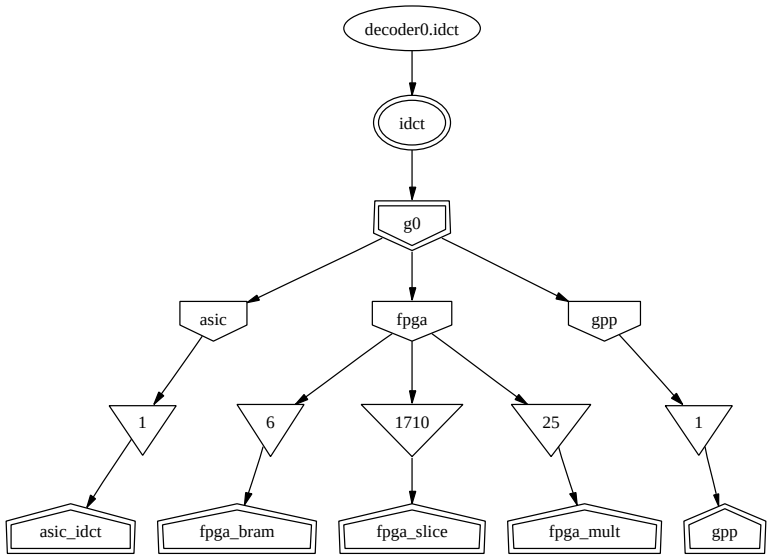


Figure 6.3: IDCT kernel templated-mapping

the following rules were used:

- Base time percentage distribution using GPP-implementation dynamic profiling, normalized to minimum algorithm component, are used as reported in table 6.1(b).

- GPP context switch incurs 10% overhead when used by both the scheduler and the mapping templates.

- FPGA area, configuration time, and configuration energy are estimated using a weighted polynomial function of implementation required resources as reported in table 6.1(a).

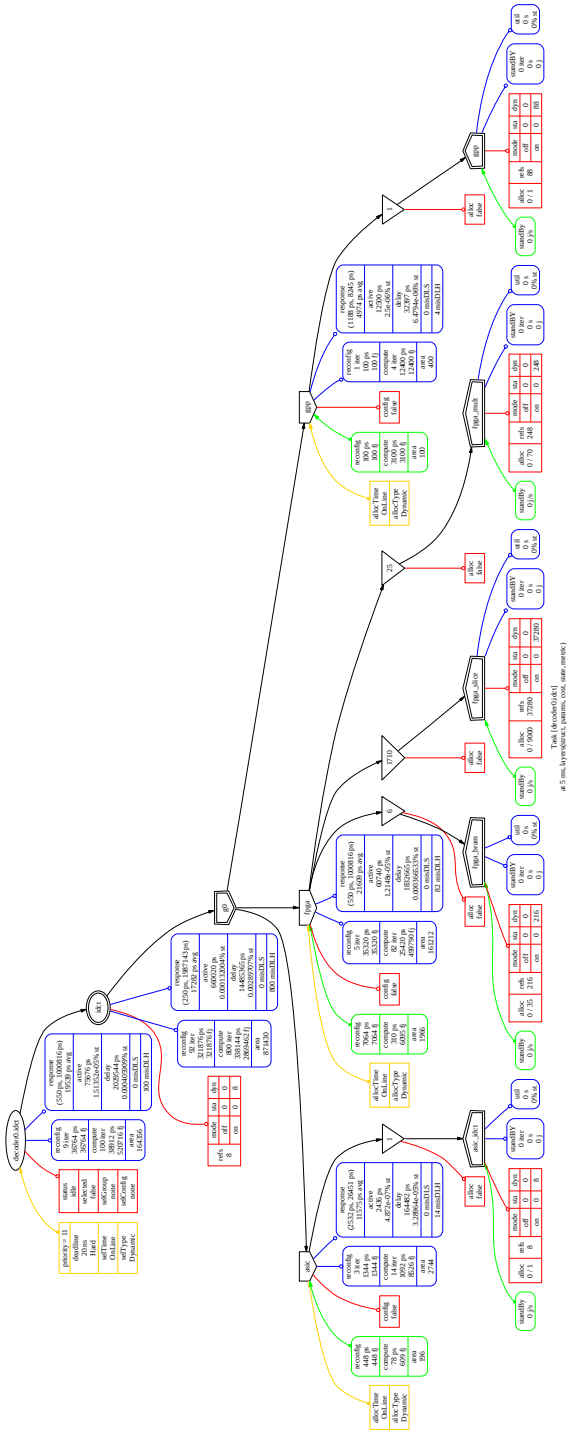


Figure 6.4: IDCT kernel decorated templated-mapping

- FPGA computation time is proportional to operation counts, ignoring possible performance improvements from behavioral transformations.
- FPGA computational energy is estimated using a weighted polynomial function of area and operation counts.
- ASIC & GPP implementation performance is scaled up and down by one order of magnitude respectively.
- ASIC configuration performance is based on weighted polynomial function of GPP context switch FPGA resources requirements.

The parameterizable scheduler described in section 5.1 is used to coordinate system execution. In the following section we explore the performance of this multi-user system of homogenous algorithms under various conditions.

6.3 Design Exploration

Here we present just a few of the results obtained from the system simulation broken down into two categories: (1) system sensitivity to users loading, in section 6.3.1, and (2) system behavior under various scheduling policies, section 6.3.2.

6.3.1 Active User Sensitivity

For each simulation in this section, the scheduler is configured for the following concurrency ordering objectives: (1) ready-tasks based on their priority, (2) configurable-tasks based on their priority, and (3) done-tasks in first-in-first-out order. The schedulers run-time optimization objective is to minimize the total execution time (allowing progress with sub-optimal selections when the resources for the optimal cannot be reserved) while meeting the deadline constraints. Each user has an equivalent input workload and the simulation is configured to run until all have completed[3].

Figure 6.5 shows the normalized run-time scheduler execution time verses concurrent active users. By concurrent, it is meant that each user workload is coincident creating an effective instantaneous workload delta proportional to the number of active users. Since the aggregate workload increases with each user, one would expect a continual increase in scheduler execution with each additional user. It is interesting to note, however, the leveling in scheduler activity that occurs at 3 and the drop at 4 active users. Upon inspection of the behavior, it is observed that there is a transition in scheduler behavior between 3 and 4 active users where there is resource saturation. Here, the resource utilization spikes

[3]The simulation terminates at a maximum simulation time even when all users have not completed execution.

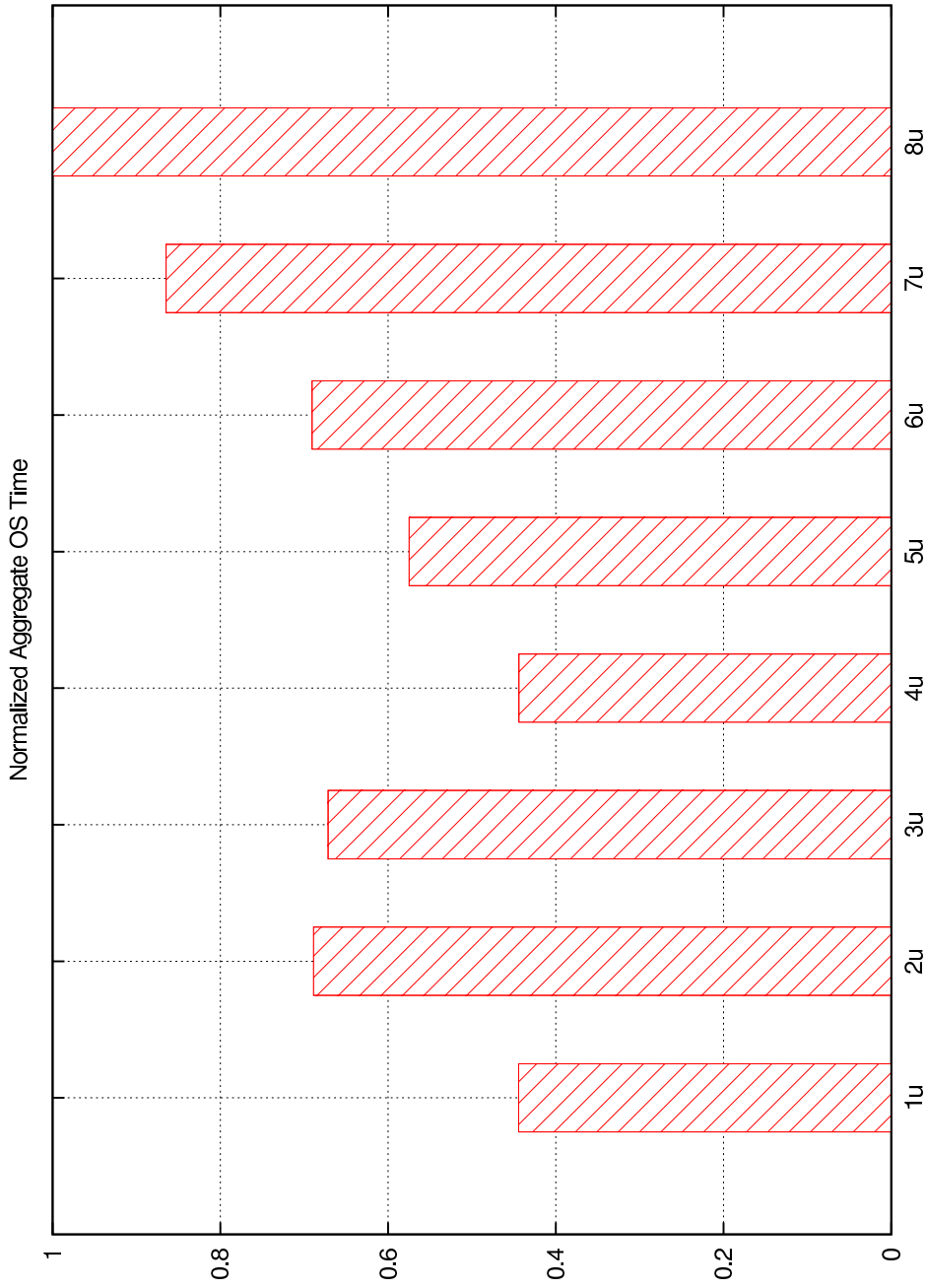


Figure 6.5: Run-time scheduler execution time vs. concurrent active users

to its maximum see figure 6.6(a). With the 4^{th} active user, no excess resources are available for scheduler run-time optimization. The subsequent linear rise in scheduler time is due primarily to the list processing of active tasks; little computation is spent on runtime selection optimization. There are two district regions of operation. One where there is a trade-off in excess system resource that are considered during run-time optimization; $O(n^2)$. At some point, a maximum is reached and the relative balance between excess resources and run-time selection optimization. After this point, the scheduler run-time optimizations are continually diminished due to the reduced search space. The second represents a region of resource saturation, where the scheduler simply attempts to satisfy deadline requirements with any available resource, which reduces to a linear function of the number of users; $O(n)$.

Figure 6.6 shows peak and mean normalized results for resource utilization, running tasks, and blocked tasks. This data further confirms the above findings. For example see the resource saturation point obvious from figures 6.6(a) and 6.6(b).

Figure 6.7 examines the quality of run-time selections in terms of delay, figure 6.7(a), and area, figure 6.7(c). A value of one indicates the highest quality or the minimization of the corresponding use-cost metric. Here it can be observed that as the number of concurrent active users approaches the limits of the available re-

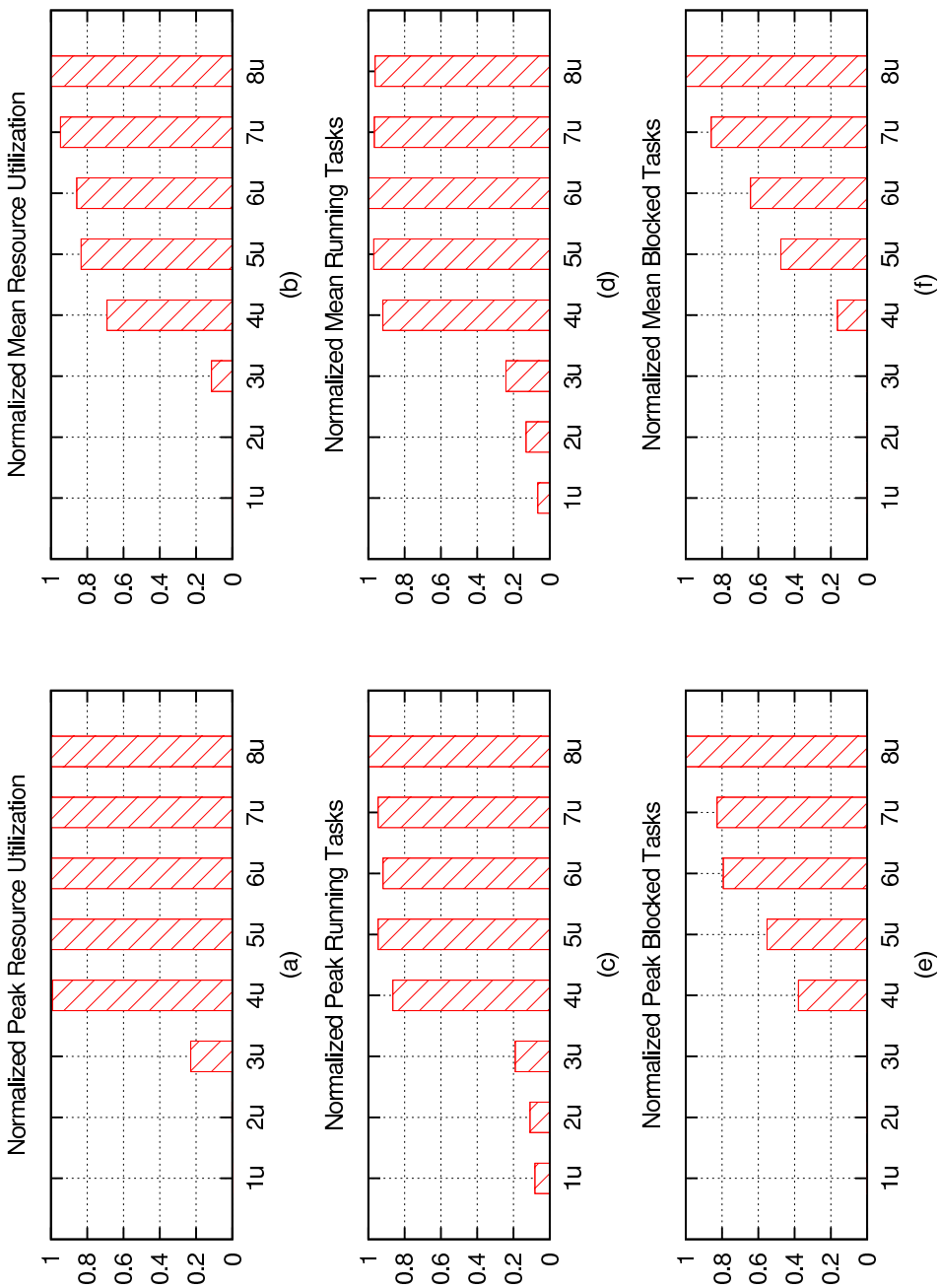


Figure 6.6: Select dynamic statistics vs. concurrent active users

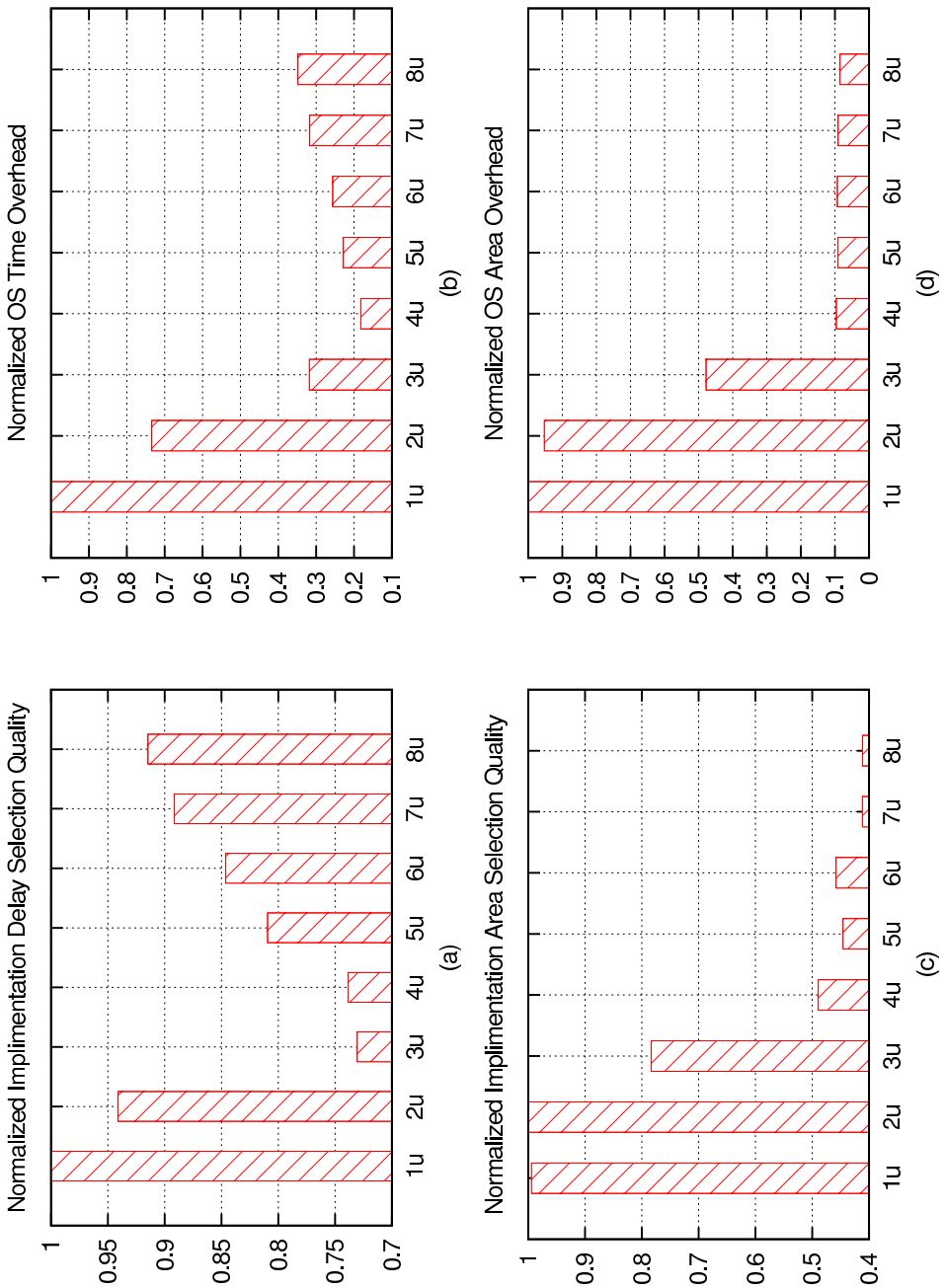


Figure 6.7: Quality and overhead vs. concurrent active users

sources, the scheduler is constrained and increasingly must choose sub-optimal implementations. As for the delay selection quality, a minimum can be seen around 3 concurrent users and subsequently it gradually increases with additional users. This may seem counter-intuitive. However, upon examination, this behavior can be attributed to the following: With 4 users and greater, the system gradually degrades missing more and more deadlines, as shown in 6.8(b). With increasing missed deadlines more and more work is dropped and unprocessed. This creates increasing opportunity for the scheduler to resume run-time optimization (albeit the missed hard deadlines and dropped work clearly indicates system failure).

In the case of area as shown in figure 6.7(c), the selection quality continually decreases as more and more resources are shared in attempt to keep up with the increased concurrency.

User Staggering

To increase the number of users that can share one system, we assign staggered time slots for each user. This additional structural knowledge can be exploited by the scheduler to more than double the number of system users from fewer than 3 to more than 6. Figures 6.8(c) and 6.8(d) shows the new performance with this additional system-level use constraint. As shown, all deadlines are met for up to 7 active users (as opposed to 2 without the system use constraint).

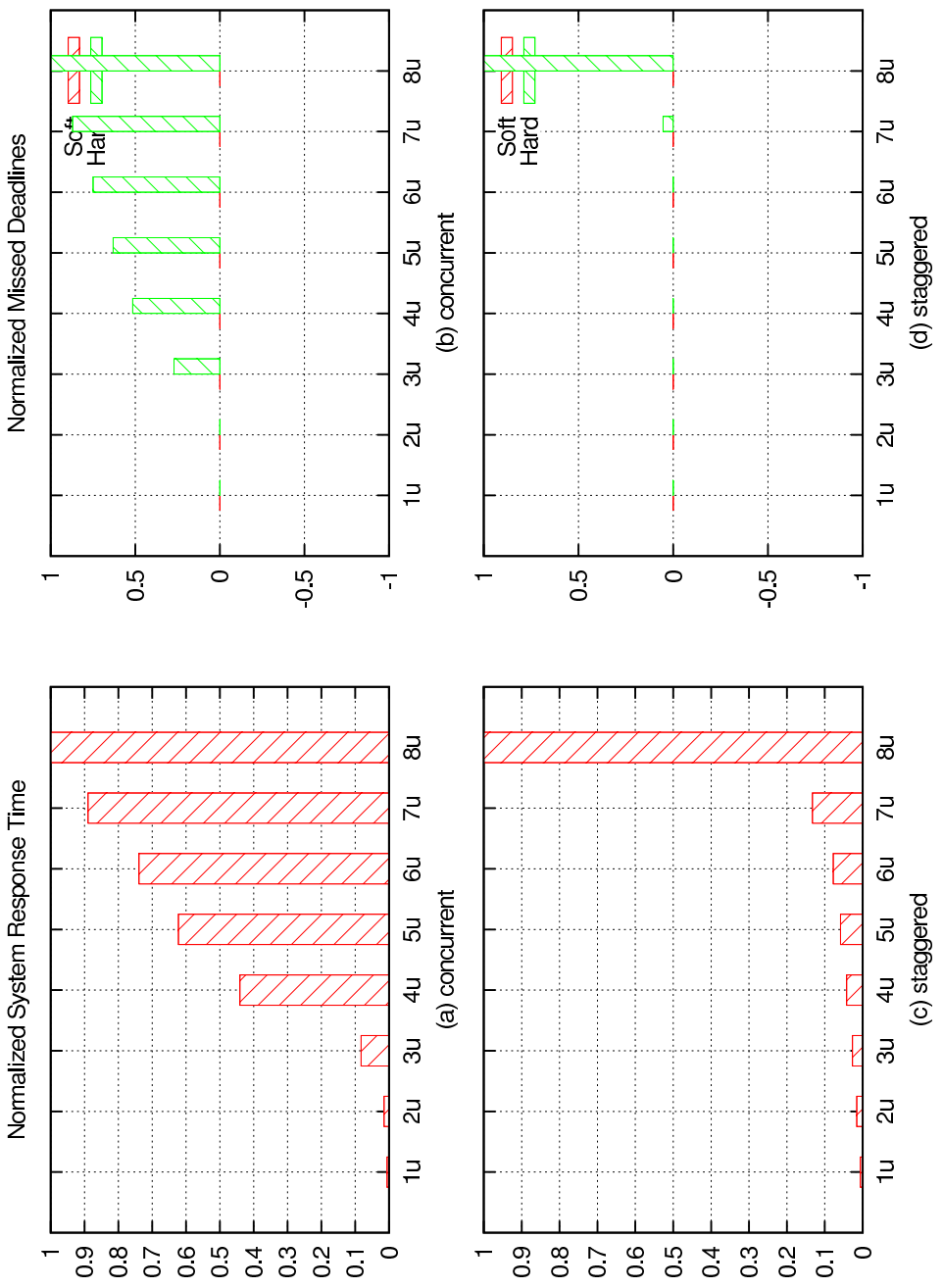


Figure 6.8: Timing performance vs. active users

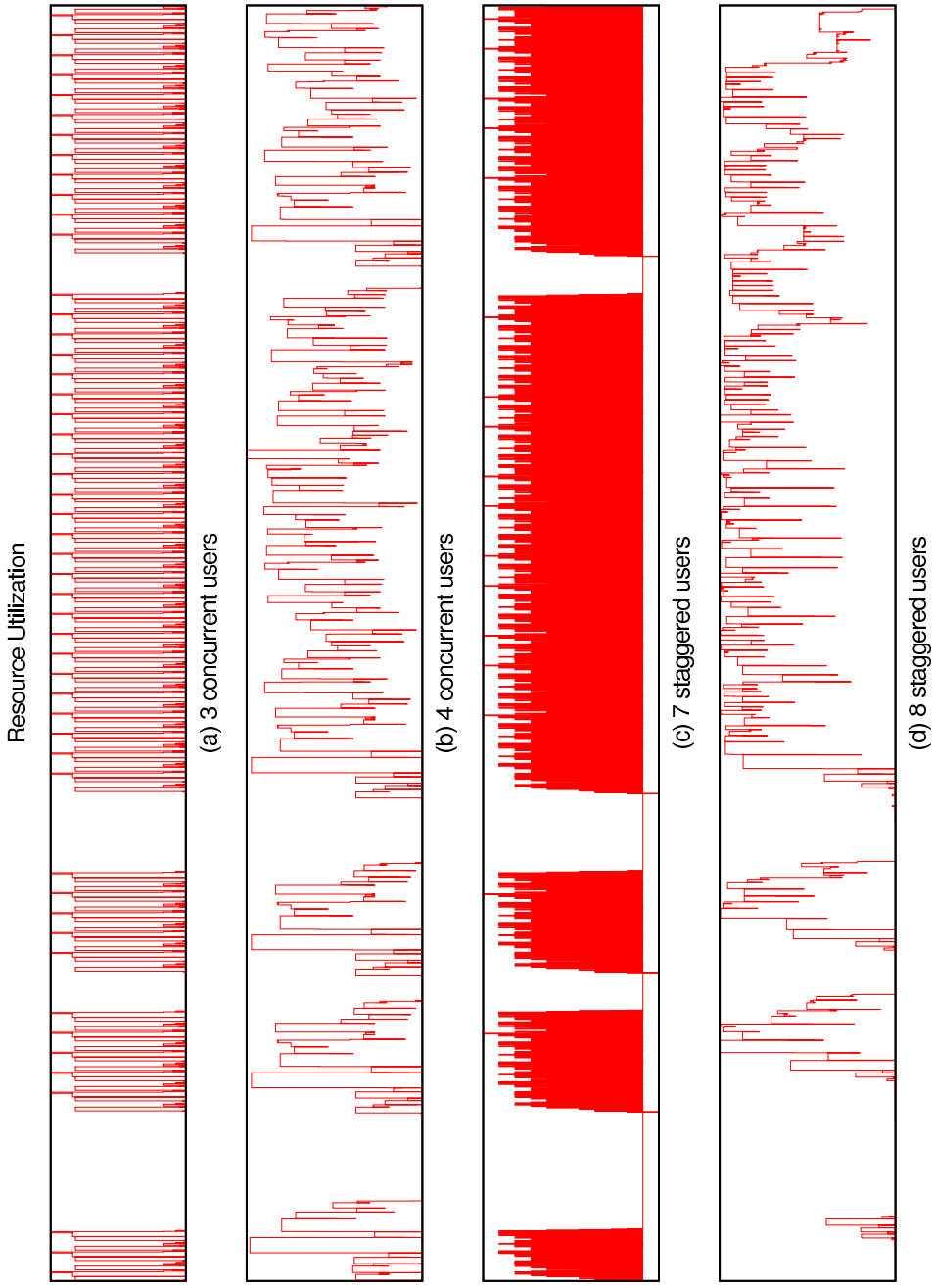


Figure 6.9: Resource utilization vs. time

As a demonstration of the scheduler's ability to exploit the additional structure, see figure 6.9. In figures 6.9(a) and 6.9(b) are shown the run-time resource utilization profile over time for 3 and 4 concurrent users. In figure 6.9(c) and 6.9(d) one can observe that scheduler is able to complete significantly greater work per unit time before running into the limitation of fixed resource pool[4].

6.3.2 Run-time Optimization Objectives

In this section we examine the methodology's capability to support run-time changes in scheduler ordering and optimization objectives to cope with changing system environments. Three concurrent users push the limits of system stability, as has been seen in the presented charts. Therefore it is selected for the following experiments.

Figure 6.10 shows variation in scheduler task ordering functions for the task concurrency in "ready to run tasks". As shown, random ordering (rand) has the lowest run-time overhead, achieves the greatest resource utilization, and greatest number of running tasks[5]. However, for a design such as this, with its' inherent structure, it also creates the greatest problem for the number of blocked

[4]Although not observed by this example, the scheduler itself could also become a bottleneck and not be able to dispatch addition income workload in a timely fashion. In such a case one would want to explore distributed scheduling schemes.

[5]Random selections have the lowest overhead since no run-time comparison need be perform, only one random choice.

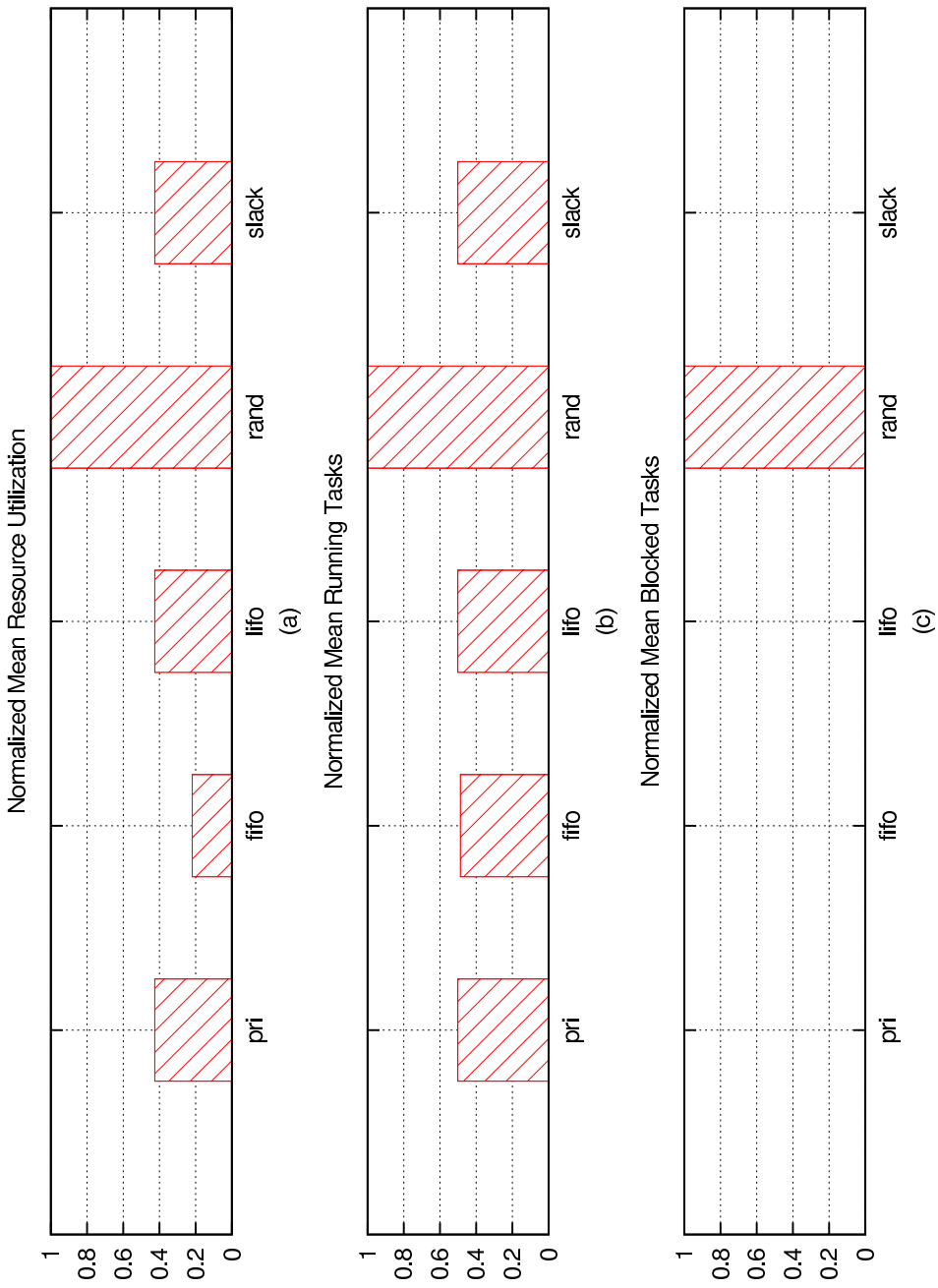


Figure 6.10: Select dynamic statistics vs. scheduler task ordering function

tasks. This should be of little surprise since the algorithm structural knowledge is ignored. Given that each user is instantiated from the same algorithm template, there is little differentiation between the abilities of the other task ordering schemes (pri, fifo, lilo, and slack). The small differences observed can be attributed to the observation that each scheduling choice come with future opportunity cost. In other words, the current choice has an impact on future choices. This would likely be different for heterogeneous user algorithms.

Finally, figure 6.11 presents the mapping selection quality (in terms of delay performance) against variations in run-time optimization objectives. As expected high delay selection quality is archived when either of total time, total energy, or area is used as a run-time optimization objective. With these objectives, the scheduler is biased towards the ASIC implementation since it has the best time, energy, and area performance.

6.4 Chapter Summary

In this chapter the design methodology proposed within this thesis is demonstrated on a practical system design; a multi-user MPEG decoder. The per-user algorithm is introduced along with its static and dynamic characterization. An architecture resource set is specified and templated-mappings are constructed for

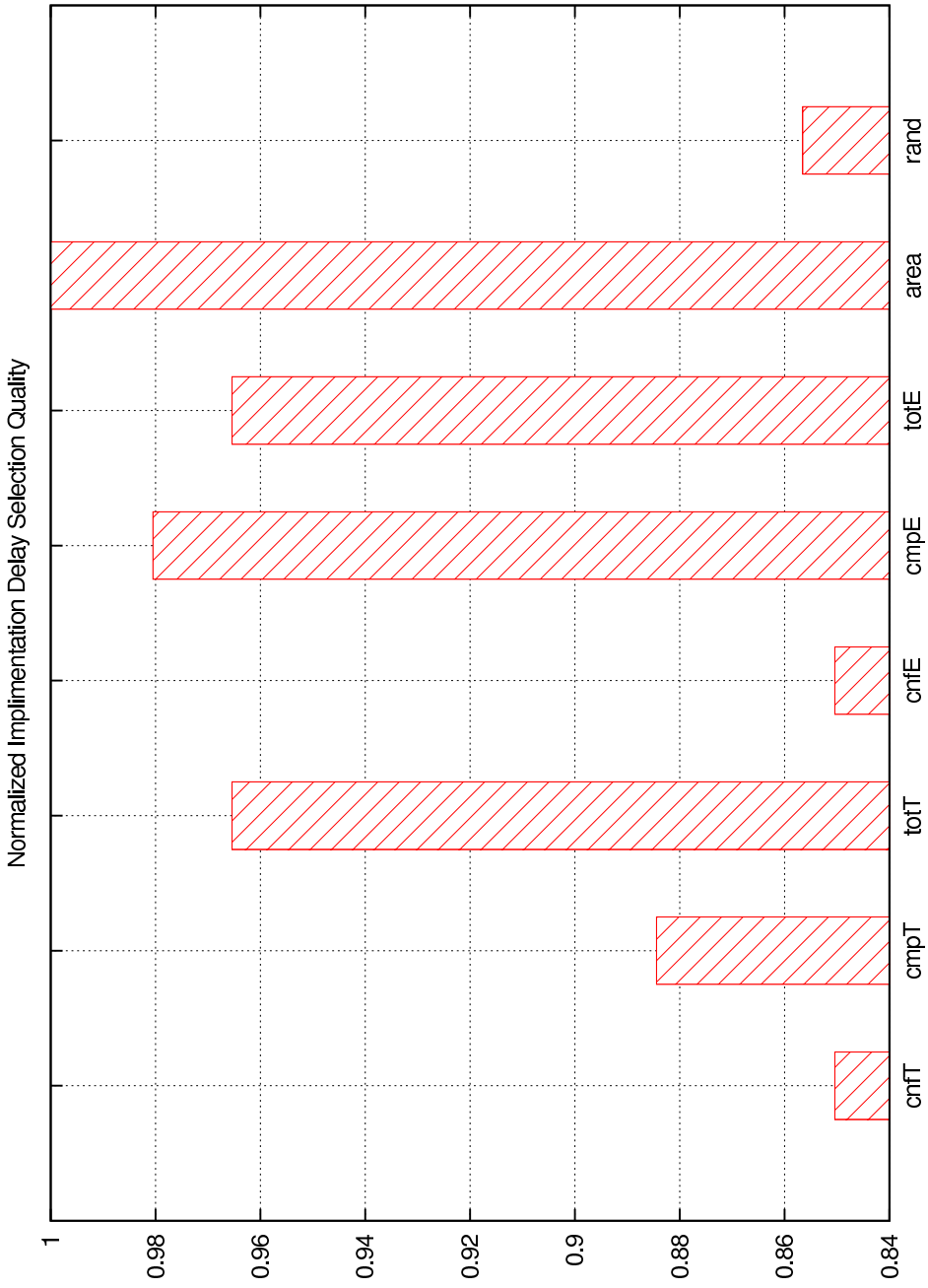


Figure 6.11: Mapping selection quality vs. scheduler optimization function

each algorithm block. A reference FPGA mapping is used to construct the template costs estimates and is scaled for estimate performance on a general purpose processor and application specific integrated circuit. Each algorithm block has three templated-mappings that can be dynamically considered by the run-time scheduler.

The simulation environment is used to explore system behavior under varying workload and a system refinement that more than double the possible workload is proposed and validated. It is shown that the methodology is able to, not only dynamically manage the reconfigure resources according to run-time optimization objectives, but also supports run-time re-optimization of such objectives under varying environmental conditions.

Chapter 7

Conclusions

The best ideas are common property. Seneca, Epistles.

This research addresses the problem of how to design systems that deal with changing algorithm computation at run-time on dynamically reconfigurable hardware, trading the architectural flexibility for efficiency. It is observed that for the applications of interest, the algorithm change can be classified into different forms. However, in each case, the change involves a refocus of the dominant computation effort into new computation kernel regions. A-priori knowledge of the form specific to a system is useful in minimizing its latencies and overheads incurred during reconfiguration in preparation for the change. To limit the resulting run-time remapping overheads, an approach is used that encourages a designer to implement multiple mapping versions off-line at design-time,

each with annotated cost-performance estimates, that are later appropriately reviewed and chosen on-line during run-time scheduling. To increase flexibility, the mappings specify resource type allocations but do not assign specific resource bindings. The step is deferred for the run-time. The design-time allocations are called templated-mappings (or mapping templates) and the collection of multiple alternative versions are called candidates.

With the organization of candidate templated-mappings, an appropriate algorithm mapping can be chosen, with limited overhead, at run-time when system resources need to be shared or when there is change in the algorithm or environment. To guide the way kernels are mapped to and from the reconfigurable hardware, modes are established to assign use policies, which are enforced by schedulers, for the selection of templated-mappings and for the reservation of mapping-specified resources. A set-based formal model is developed that includes a definition for system run-times with multi-threaded and concurrent execution control. A design environment has been developed that implements the methodology and all of its features; including dealing with threaded and concurrent execution. It is applied to a collection of generated system designs to demonstrate the effectiveness of the approach. The methodology and design environment is also demonstrated on a practical system design problem.

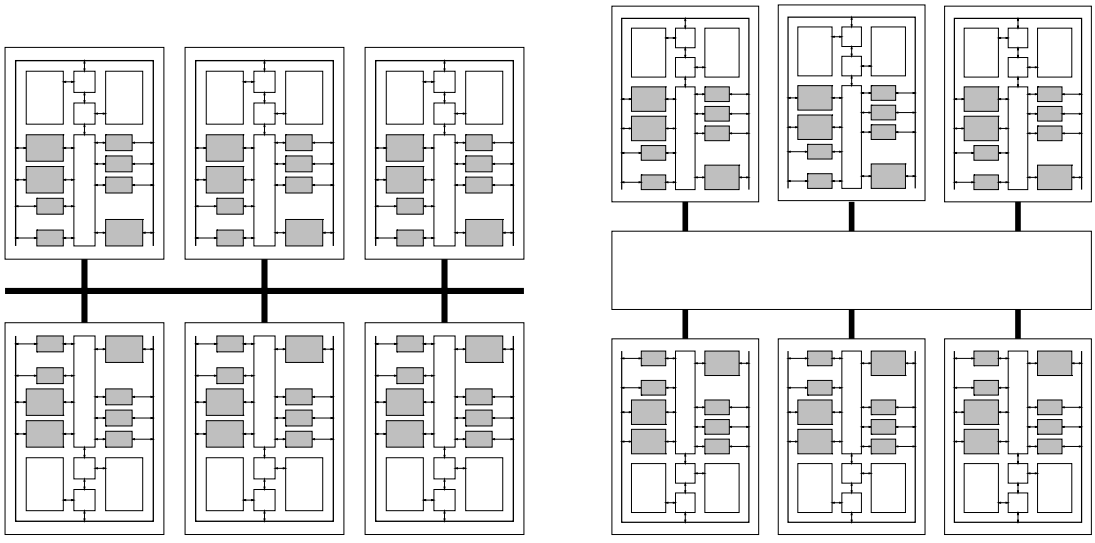
7.1 Directions for Future Work

The methodology is an enabling contribution. Until this point, there lacked the tools to perform system-level design evaluation for dynamically reconfigurable platforms. With this approach, and design environment, further research is possible in the analysis of architecture appropriateness and system-level management strategies. Some of the interesting areas are outline below.

Scheduling Overhead Reductions: Future research in threaded-scheduer[1] and hardware-oriented scheduling would undoubtedly greatly reduce the overhead and response time for run-time execution. Greater effort to exploit the forms of change in the higher control structure of the algorithms with off-line schedules should be explored.

Variable Voltage Architectures: This framework defined by this methodology, and implemented in the design environment, readily supports alternative use-cost models. With minor effort, models that use voltage as a parameter could be incorporated. With this modification, run-time strategies for the control of variable voltage architectures could be explored. This would add yet another dimension where run-time trade-offs could be made. For example, to conserve

[1]This should not be confused with threaded execution.



(a) 2-level: Net/bus-connected

(b) 2-level: Net/net-connected

Figure 7.1: Domain-specific systems: Multi-level networks and bus interconnect

energy under reduced loading condition.

System-level Dynamic Management: One of the most interesting areas for future research is in the exploration of distributed and hierarchical scheduling strategies. The approach defined by this research assigns a scheduler to each behavior independently when it is created. With this, many possible scheduling strategies can be explored. As one proposal, consider the two reconfigurables systems shown in figure 7.1(a) and 7.1(b). They are composed from multiple domain-specific sub-systems that are interconnected by either a bus or a network. Assume each sub-system has been designed from the architecture template presented in figure 2.3 for a specific group of algorithms; for example a video domain

or an audio domain.

From our perspective, each sub-domain can be developed with local schedulers. And as shown in figure 7.2, a system-level scheduler can be developed to deal with system-level algorithm change. The result is a globally dynamic and locally pseudo-static configuration management approach. The system-level scheduler would have the vantage to determine whether to accept new algorithms and could assign them based on the opportunities present as defined by the domains and their run-time utilizations.

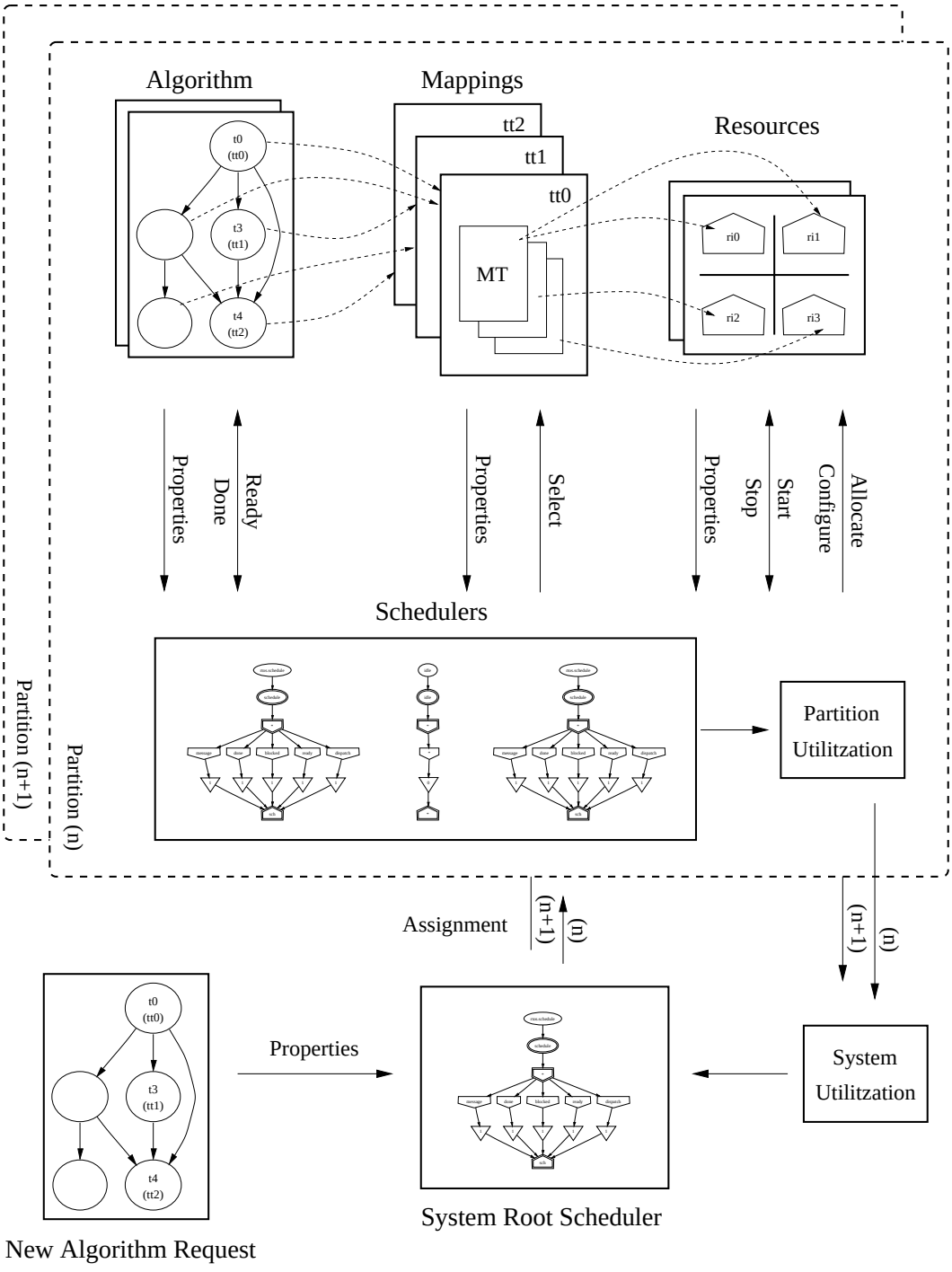


Figure 7.2: Dynamic algorithm schedule management

Bibliography

[ASI+98] Arthur Abnous, Katsunori Senoy, Yuji Ichikawaz, Marlene Wan, and Jan Rabaey. Evaluation of a low-power reconfigurable dsp architecture. In *12th International Parallel Processing Symposium / 9th Symposium on Parallel and Distributed Processing*, Orlando, Florida, USA, March/April 1998. IEEE Computer Society.

[AZW+02] Arthur Abnous, Hui Zhang, Marlene Wan, George Varghese, Vandana Prabhu, and Jan Rabaey. The pleiades architecture. In A. Gatherer and A. Auslander, editors, *The Application of Programmable DSPs in Mobile Communications*, pages 327–360. Wiley, 2002.

[BFS04] Carlo Brandolese, William Fornaciari, and Fabio Salice. An area estimation methodology for fpga based designs at systemc-level. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 129–132, New York, NY, USA, 2004. ACM Press.

[BH01] Peter Bellows and Brad Hutchings. Designing run-time reconfigurable systems with jhdl. *J. VLSI Signal Process. Syst.*, 28(1-2):29–45, 2001.

[BL02] Dag Bjorklund and Johan Lilius. A language for multiple models of computation. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 25–30, New York, NY, USA, 2002. ACM Press.

[BRL97] Ole Bentz, Jan M. Rabaey, and David Lidsky. A dynamic design estimation and exploration environment. In *DAC '97: Proceedings*

of the 34th annual conference on Design automation, pages 190–195, New York, NY, USA, 1997. ACM Press.

[Bro92] R. W. Brodersen. *Anatomy of a silicon compiler*. Kluwer Academic Publishers, 1992.

[Byb] Tony Bybell. Gtkwave, a free electronic waveform viewer. Information Available on the World Wide Web at: http://www.cs.manchester.ac.uk/apt/projects/tools/gtkwave.

[Cas91] Andrea Casotto. *OCTTOOLS-5.1 user guide and reference*. Electronics Research Laboratory, University of California, Berkeley, 94720, 1991. editor.

[CEL+03] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The reconfigurable streaming vector processor (rsvptm). In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 141, Washington, DC, USA, 2003. IEEE Computer Society.

[CG00] S Cacopardi F Frescura M Vagheggini C Gnudi, P Antognoni. A software radio development system for wireless multimedia systems. ICSPAT, 2000.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[CKR+03] Chen Chang, Kimmo Kuusilinna, Brian Richards, Allen Chen, Nathan Chan, and Robert W. Brodersen. Rapid design and analysis of communication systems using the bee hardware emulation environment. In *Proceedings of the IEEE Rapid System Prototyping Workshop*, June 2003.

[CPC+99] Lukai Cai, Junyu Peng, Chun Chang, Andreas Gerstlauer, Hongxing Li, Anand Selka, Chuck Siska, Lingling Sun, Shuqing Zhao, and Daniel D. Gajski. Design of a jpeg encoding system. Technical Report ICS-TR-99-54, UC Irvine, November 1999.

[CPRB03] Anantha Chandrakasan, Miodrag Potkonjak, Jan Rabaey, and Robert Brodersen. Hyper-lp: A system for power minimization using architectural transformations. In A. Kuehlman Ed, editor, *The Best of ICCAD, 20 Years of Excellence in Computer-Aided Design*, pages 107–116. Kluwer Academic Publishers, 2003.

[cyg] Cygwin: A linux-like environment for windows. Information Available on the World Wide Web at: http://cygwin.com/.

[DGM02] Rainer Dömer, Andreas Gerstlauer, and Wolfgang Mueller. The formal execution semantics of specc. In *Proceedings of the 15th international symposium on System Synthesis (ISSS '02)*, pages 150–155, Kyoto, Japan, October 2-4 2002.

[DRW98] Robert P. Dick, David L. Rhodes, and Wayne Wolf. Tgff: Task graphs for free. In *Proc. Int. Workshop Hardware/Software Codesign*, pages 97–101, March 1998.

[DST99] Deepali Deshpande, Arun K. Somani, and Akhilish Tyagi. Configuration caching vs data caching for striped fpgas. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 206–214, New York, NY, USA, 1999. ACM Press.

[ECKM04] Virantha Ekanayake, IV Clinton Kelly, and Rajit Manohar. An ultra low-power processor for sensor networks. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 27–36, New York, NY, USA, 2004. ACM Press.

[EL03] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program.*, 48(1):21–42, 2003.

[FBK98] Josef Fleischmann, Klaus Buchenrieder, and Rainer Kress. A hardware/software prototyping environment for dynamically reconfigurable embedded systems. In *CODES/CASHE '98: Proceedings of*

the 6th international workshop on Hardware/software codesign, pages 105–109, Washington, DC, USA, 1998. IEEE Computer Society.

[GHI+04] Manfred Glesner, Thomas Hollstein, Leandro Soares Indrusiak, Peter Zipf, Thilo Pionteck, Mihail Petrov, Heiko Zimmer, and Tudor Murgan. Reconfigurable platforms for ubiquitous computing. In *CF'04: Proceedings of the first conference on computing frontiers on Computing frontiers*, pages 377–389, New York, NY, USA, 2004. ACM Press.

[GLMS02] Thorsten Grtker, Stan Liao, Grant Martin, and Stuart Swan. *System design with SystemC*. Kluwer Academic Publishers, Boston, hardcover edition, 2002.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[gnu] Gnuplot. Information Available on the World Wide Web at: http://www.gnuplot.info/.

[GNVV04] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of fpgas over processors. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 162–170, New York, NY, USA, 2004. ACM Press.

[Gri04] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal, Elsevier*, 38(2):131–183, December 2004.

[GZD+00] Daniel D. Gajski, Jianwen Zhu, Rainer Dmer, Andreas Gerstlauer, and Shuqing Zhao. *SPEC C: Specification language and design methodology*. Kluwer academic publishers, Boston, MA, March 2000. ISBN 0-7923-7822-9.

[HAA+96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S.

Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[Har01] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[Har04] Reiner Hartenstein. The digital divide of computing. In *CF'04: Proceedings of the first conference on computing frontiers on Computing frontiers*, pages 357–362, New York, NY, USA, 2004. ACM Press.

[Him95] Michael Himsolt. Gml: A portable graph file format. Information Available on the World Wide Web at: http://infosun.fmi.uni-passau.de/Graphlet/GML/, 1995.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[HR97] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, March 1997.

[JHK+05] Alex K. Jones, Raymond Hoare, Dara Kusic, Joshua Fazekas, and John Foster. An fpga-based vliw processor with custom hardware execution. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 107–117, New York, NY, USA, 2005. ACM Press.

[KBP+04] Sami Khawam, Sajid Baloch, Arjun Pai, Imran Ahmed, Nizamettin Aydin, Tughrul Arslan, and Fred Westall. Efficient implementations of mobile video computations on domain-specific reconfigurable arrays. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21230, Washington, DC, USA, 2004. IEEE Computer Society.

[KKP99] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile networking for smart dust. In *ACM/IEEE Intl. Conf. on Mobile Computing and Networking*, pages 17–19, Seattle, WA, August 1999.

[Kri98] Jens Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998. ACM SIGPLAN Notices 33(7).

[LA93] Panos E. Livadas and Scott D. Alden. A toolset for program understanding. In Bruno Fadini and Vaclav Rajlich, editors, *Proceedings of the IEEE Second Workshop on Program Comprehension*, pages 110–118, 1993.

[LB04] John Lach and Kia Bazargan. Editorial: Special issue on dynamically adaptable embedded systems. *Trans. on Embedded Computing Sys.*, 3(2):233–236, 2004.

[LCH00] Zhiyuan Li, Katherine Compton, and Scott Hauck. Configuration caching management techniques for reconfigurable computing. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 22–36, Napa, California, April 2000.

[LSR03] Suet-Fei Li, Roy Sutton, and Jan Rabaey. Low power operating system for heterogeneous wireless communication system. In Luca Benini, editor, *Compilers and Operating Systems for Low Power*, pages 1–16. Kluwer Academic Publishing, Norwell, MA, 2003.

[LZ05] Edward A. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control: 8th International Workshop*, volume 3414 / 2005, pages 25–53, Zurich, Switzerland, March 9-11 2005. Springer-Verlag GmbH.

[MD04] Binu Mathew and Al Davis. A loop accelerator for low power embedded vliw processors. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software*

codesign and system synthesis, pages 6–11, New York, NY, USA, 2004. ACM Press.

[NB02] Juanjo Noguera and Rosa M. Badia. Dynamic run-time hw/sw scheduling techniques for reconfigurable architectures. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 205–210, New York, NY, USA, 2002. ACM Press.

[NR95] Kelvin D. Nilsen and Bernt Rygg. Worst-case execution time analysis on modern processors. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 20–30, New York, NY, USA, 1995. ACM Press.

[Ous90] John Ousterhout. Tcl: An embeddable command language. In *Proceedings of the 1990 Winter USENIX Conference*, pages 133–146, 1990.

[Ous91] John Ousterhout. An x11 toolkit based on the tcl language. In *Proceedings of the 1991 Winter USENIX Conference*, pages 105–115, 1991.

[OZD+03] Vojin G. Oklobdzija, Bart R. Zeydel, Hoang Dao, Sanu Mathew, and Ram Krishnamurthy. Energy-delay estimation technique for high-performance microprocessor vlsi adders. In *ARITH '03: Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, page 272, Washington, DC, USA, 2003. IEEE Computer Society.

[PKB99] K. S. J. Pister, J. M. Kahn, and B. E. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes. *Electronics Research Laboratory Research Summary*, 1999.

[PKK97] Miodrag Potkonjak, Kyosun Kim, and Ramesh Karri. Methodology for behavioral synthesis-based algorithm-level design space exploration: Dct case study. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 252–257, New York, NY, USA, 1997. ACM Press.

[PMC03] Antti Pelkonen, Kostas Masselos, and Miroslav Cupk. System-level modeling of dynamically reconfigurable hardware with systemc. In

IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, page 174.2, Washington, DC, USA, 2003. IEEE Computer Society.

[RCHP91] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of datapath-intensive architectures. *IEEE Design and Test of Computers*, 8(2):40–51, June 1991.

[Rec04] W3C Recommendation. Extensible markup language. Information Available on the World Wide Web at: http://www.w3.org/XML/, February 2004.

[RVN02] Tero Rissa, Milan Vasilko, and Jarkko Niittylahti. System-level modelling and implementation technique for run-time reconfigurable systems. In *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 295–296, Napa, California, September 2002. IEEE Computer Society.

[SC04] Lesley Shannon and Paul Chow. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 190–199, New York, NY, USA, 2004. ACM Press.

[Sie] Jeremy Siek. Boost graph library. Information Available on the World Wide Web at: http://www.boost.org.

[SL99] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, November/December 1999.

[SRB01] Chris Savarese, Jan M. Rabaey, and Jan Beutel. Locationing in distributed ad-hoc wireless sensor networks. In *International Conference on Acoustics, Speech,and Signal Processing (ICASSP)*, Salt Lake City, Utah, May 2001.

[SS05] Alireza Shoa and Shahram Shirani. Run-time reconfigurable systems for digital signal processing applications: A survey. *J. VLSI Signal Process. Syst.*, 39(3):213–235, 2005.

[SSR98] Roy A. Sutton, Vason P. Srini, and Jan M. Rabaey. A multiprocessor dsp system using paddi-2. In *35th annual conference on Design automation*, volume 00, pages 62–65, June 1998.

[Sut98] Roy A. Sutton. An infopad basestation interface using ansi/ieee 488.1-1987. Master's thesis, University of California, Berkeley, 1998.

[sys] The open systemc initiative. Information Available on the World Wide Web at: http://www.systemc.org/.

[TB01] Russell Tessier and Wayne Burleson. Reconfigurable computing for digital signal processing: A survey. *J. VLSI Signal Process. Syst.*, 28(1-2):7–27, 2001.

[Wan01] Marlene Wan. *Design methodology for low power heterogeneous reconfigurable digital signal processors*. PhD thesis, University of California, Berkeley, 2001.

[xer] The apache software foundation xml project. Information Available on the World Wide Web at: http://xml.apache.org/.

[Xil05] Xilinx. *MPEG4 Simple Profile Decoder and Encoder Purchase Instructions*, February 2005. Version 1.0.

[YJ05] Kaushik Ravindran Nadathur Satish Kurt Keutzer Yujia Jin, Will Plishker. Soft multiprocessor systems for network applications. Talk or presentation, February 2005.

[yWo] yWorks. A free java graph editor yed. Information Available on the World Wide Web at: http://www.yworks.com.

[ZPG+00] Hui Zhang, Vandana Prabhu, Varghese George, Marlene Wan, Martin Benes, Arthur Abnous, and Jan M. Rabaey. A 1-v heterogeneous reconfigurable dsp ic for wireless baseband digital signal processing.

IEEE Journal on Solid State Circuits, 35(11):1697–1704, November 2000.

[ZPTB99] Ning Zhang, Ada Poon, David Tse, and Robert Brodersen. Tradeoffs of performance and single chip implementation of indoor wireless multi-access receivers. Available on the World Wide Web at http://bwrc.eecs.berkeley.edu/Publications/1999/tradeoffs_performance_singlechip_implement/index.htm, 1999.

[ZSGR01] Lizhi Charlie Zhong, Rahul Shah, Chunlong Guo, and Jan Rabaey. An ultra-low power and distributed access protocol for broadband wireless sensor networks. In *IEEE Broadband Wireless Summit*, Las Vegas, N.V., May 2001.

Appendix A

Terminology

Algorithm Mapping: the act of assigning a subset of a coded algorithm to physical hardware resources that will carry out the computation.

Candidate (mapping): one or more unique algorithm mappings that are logically equivalent. That is to say, for some given set of inputs, each candidate implements the same input-to-output behavior.

Dynamic Algorithm: not to be confused with self-modifying code, this term is used to reference a growing group of DSP algorithms that operate within changing environments. Changes such as competing and complimentary algorithm "standards," triggered and sequential shifts in system-level algorithm kernel activity, algorithmic computation effort sensitive to resource availability (or some other use objective), and replicated behavior — instantiated on demand — coexisting to share a given finite system resource.

Kernel: an identified region of an algorithm that has relatively higher computation requirements. Usually kernels are associated with iterative loops over relatively small code regions and loosely translates to high repetition of structured behavior.

Off-Line: relates to occurrences in system-time *prior to* the activation of a run-time system-level management kernel responsible for the governance of input-to-output behavior.

On-Line: relates to occurrences in system-time *after* the activation of a run-time system-level management kernel responsible for the governance of input-to-output behavior.

Reconfigurable Architecture: refers loosely to hardware architectures that have functional units where one ore more may be "tailored" and/or the physical interconnection between the aforementioned functional units can changed in a run-time environment to suit dynamic algorithm behavior.

Task: a name given to a group of algorithm codes that perform some desired sequence of operations.

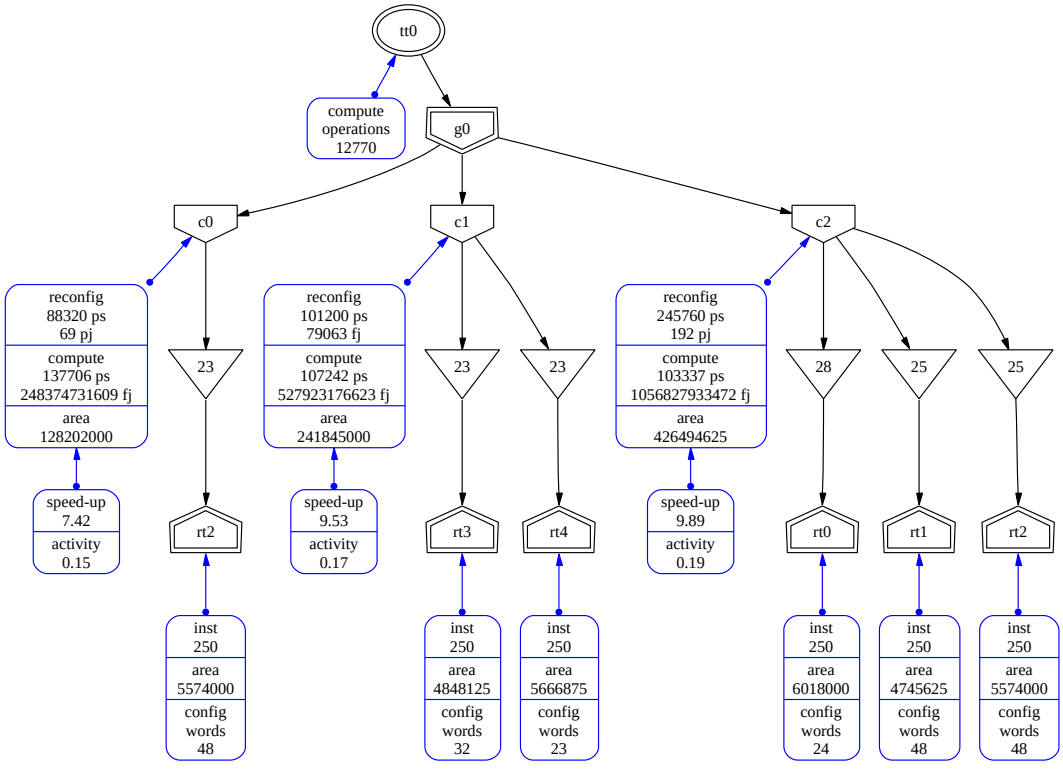
Templated-Mapping: refers to a partial algorithm mapping that completely specifies the behavioral form of the mappings — in terms of identified hardware functional units and their required interconnection (the resources) — but defers specific resource binding to a future point in time.

Appendix B

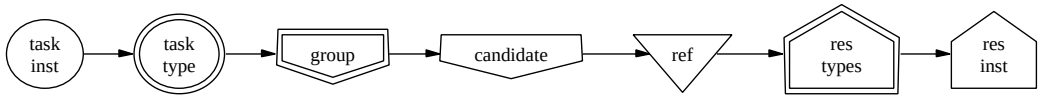
Design Environment Tutorial

> This document describes the usage and input syntax of the Unix Vax-11 assembler As. As is designed for assembling code produced by the "C" compiler; certain concessions have been made to handle code written directly by people, but in general little sympathy has been extended. Berkeley Vax/Unix Assembler Reference Manual (1983).

This chapter presents an introduction to the structure, use, and expansion of the design environment. With the exception of a brief introduction to mapping templates, it does not cover the inner workings of the framework. Please refer to chapters 3 and 4 for that discussion.



(a) Template example with some system property annotations



(b) Computation mapping-level key

Figure B.1: A templated-mapping example

B.1 Templated-Mapping

The notion of a mapping template is a key feature that distinguishes this research from other work. A template is the building block of a structure of candidates that contains one or more logically equivalent partial mappings of some algorithm onto a target architecture. Figure B.1(a) shows an example of a template, $tt0$, with three candidates $c0$, $c1$, and $c2$, with varying resource requirements. Templates are mappings to resource types rather than specific resource instances. For example, instantiation of candidate $c0$ requires 23 instances of resource $rt2$. Templates are members of groups and a candidate-structure has one or more such groups. In practice, templates within a grouping share some classification of similarity. This example has one group named $g0$. The blue boxes in the diagram are not part of the structure. They represent a set of system properties that are relevant to a particular runtime management model.

Task instances, the components of the system algorithms, are mapped to task types. There is a one-to-one bilateral mapping between task-types and candidates. See figure B.1(b) for a key to the mapping relationships.

Candidates facilitate the development of efficient schemes for runtime management of resources in dynamic reconfigurable systems. The design approach begins offline with, the decomposition of an algorithm into types. The algorithm

behavior is then recomposed of instance references of the appropriate task types. We call them task types and task instances. Additionally offline, the candidates of templated-mappings are constructed for each task type. Later, online, an appropriate templated-mapping is selected from amongst the candidates, guided by some runtime optimization objectives. And finally the runtime scheme instantiates a candidate by reserving the required resource types and resource instances.

Templates can be used to dynamically manage resources, trading performance for flexibility in response to changes in the system algorithms and stimulus.

B.2 Tool Types

The design environment has been coded within a Unix-based operating system [cyg] and should build under most modern versions with minimal modification. It consists of a collection of tools in the categories as identified in table B.1(a) coded in one or more of the languages as identified in table B.1(b). The tools operate within database structures that is stored in a file system based hierarchy[1]. The detail of these structures are presented in section B.4.

[1]At the beginning of this research, a file system based database scheme seemed adequate, however, in hindsight, it clearly would have been advantageous to use a dedicated database management system such as that presented in [Cas91].

Table B.1: Design environment tool types

Category
Design tools
Analysis Tools
Graphical user interfaces
Visualization Rendering
Data format converters
Miscellaneous utilities

(a)

Language	Programs	Lines
C/C++ programs	8	29,891
JAVA programs	1	-na-
Tcl/Tk programs	5	7,639
GNUPlot scripts	12	747
Bash scripts	5	223
Makefiles	11	1,624

(b)

B.3 Data Formats

The internal representations for the design tools were structured using the graph library presented in [SL99] and available at [Sie]. This library is a generic template-based C++ graph component library. It support the association of arbitrary parameters to the coded graph structure. This works well in the development of runtime management schemes that deal with various system properties.

At the heart of the design flow is a refinement process whereby appropriate tools are applied to transform system representations into new forms that have properties more like the target implementation. These tools communicate one with another via the file formats identified in table B.2. Most notably are public data formats `.gml` and `.xml`. The former is used to exchange graph-based representations [Him95]. The latter is a popular public markup language for the

Table B.2: Design environment file formats

Extension	Owner	Format Description
.csv	public	comma-separated values
.dat	private	simulation estimates raw data
.dot	public	graph rendering description language
.gml	public	graph modeling language
.gp	public	plot tool scripting language
.gpi	private	plot instance include file
.html	public	hypertext markup language
.idx	private	plot multi-simulation index file
.include	private	plot simulation name-mapping file
.mfropt	private	batch-run multi-simulation meta-file parameters
.osgenopt	private	scheduler cost generation parameters
.ps	public	postscript level-2 conforming
.ps2	public	postscript with PDF annotations
.rcgenopt	private	architecture and candidates generation parameters
.tech	private	base technology data file
.tgffopt	public	computation task set generation parameters
.txt	public	Unix formatted plaint text file
.vcd	public	Vector change dump file
.xml	public	Extensible Markup Language

interchange of structured data [Rec04][xer]. It is a highly flexible scheme that could replaced each of the other formats. However, this would have limited inter-operability with other useful freely-available and open-source tools.

B.4 File Structure

In the next several paragraphs, the components of the design environment directory structure, the design database directory structure, and simulation database directory structure are presented. These components are identified by number in figures B.2, B.3, and B.4. Corresponding numbers preceding each paragraph where they are described.

1: The design environment root directory. A shell environment variable `$MANIFOLD` should point to this location. This established the base reference for relative addressing of other locations within the directory structure.

2, 3, 5, 6, and **7**: The content of these directory sub-trees are as the names suggest within the context of UNIX-based development project. They contain the source code and executables for the tool components of the design environment.

4: An arbitrary name given to a sub-directory that has been established to contain the environment evaluation databases.

8: Arbitrary names given to a design database ($db1, db2, ...$ for example). Each database contains a one or more groups of components that can be assembled into systems that are evaluated via simulation.

9, 13, and **14**: A location for the collection of elements common across databases. For example, in **13** reside the Document Type Definition (DTD) files that

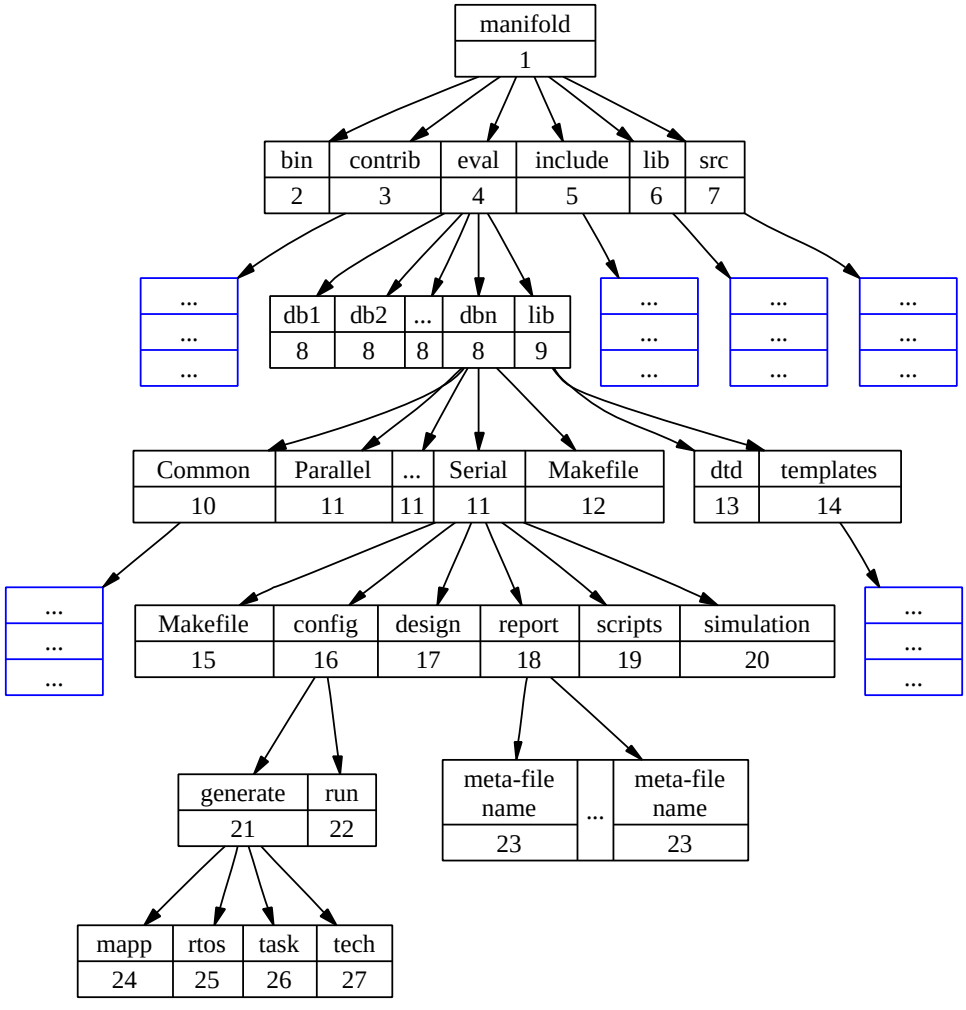


Figure B.2: Design environment directory structure

defined the structure of the eXtensible Markup Language (XML) data file formats used in the environment. In **14** reside a tree of templates structures used throughout the design database sub-trees described in **8**.

10: A tree containing components that are common across groups of a design database. For example descriptions of, architectures, schedulers, or technology can be placed here and subsequently referenced by other pier groups. A special Makefile exist within this directory to manage the unique purpose of this database group.

11: Arbitrary names given to design database groups.

12: A database-level Makefile that allows design database groups to be directed collectively from one location. Here, the simulation control (generation, compilation, simulation, report, etc.) of an all groups can be managed.

15: A group-level Makefile. This file encapsulates the design environment tool invocation automation flow. It contains directives for representation transformation, simulation, and evaluation. The function of this file is spread across numerous other Makefiles that reside in the script sub-directory labeled **19**. A simulation run configuration file, or the so-called meta-file discussed below, is parsed to specify a simulation. See listing B.1 for an example.

Listing B.1: A parsed simulation run metafile example

```
1 ##############################################################################
2 #       SIMULATION RUN META-FILE : (PARSED FROM CONFIG FILE)       #
```

```
3  ######################################################################
4     CONFIG FILE: cf=[ config/run/1_tasks_select_arch.mfropt ]
5
6            TASKS: [ par ser pas rnd ]
7    ARCHITECTURES: [ a123c4x+01 ]
8     TECHNOLOGIES: [ StdCell1Sp ]
9       SCHEDULERS: [ stdbegp1 ]
10     READY ORDER: [ pri+ ]
11      DONE ORDER: [ pri+ ]
12    CONFIG ORDER: [ pri+ ]
13   CONFIG SEARCH: [ ttime++ ]
14     ACTIVATIONS: [ per ]
15       DURATIONS: [ don ]
16
17      TGGEN_OPTS: [ -P ]
18      RCGEN_OPTS: [ -P ]
19      OSGEN_OPTS: [ -P ]
20   CSV2HTML_OPTS: [ --HRCount 2 --mergeDupRows --mergeDupDRColumns ]
21    HTML2PS_OPTS: [ -f ${manifold}/lib/html2ps/Letter -D -L ]
22      OSSIM_OPTS: [ --clusDoneCnt 3 --hypIterCnt 10 ]
23      OSSIM_OPTS: [ --abortAtMaxSimTime --maxSimTimeMul 10 ]
24      OSSIM_OPTS: [ --cycleCnt 10000000 ]
25     SIM OUT DIR: [ ./simulation/ ]
26
27 REPORT OUT DIR: [ ./report/ ]
28
29  simulate: directive configuration options
30       run_sim = yes, missing_only = yes
31       post_process = yes, compress_out = yes
```

16: Within this tree reside the parameter files for case-study system generation and simulation automation described next.

21 and **24-27**: this directory tree contains configuration files that are used by a collection of tools that generate system design components from parameterized descriptions.

22: This directory contains simulation run meta-files that automate the process of design exploration. It uses a simple scheme for enumerating system component

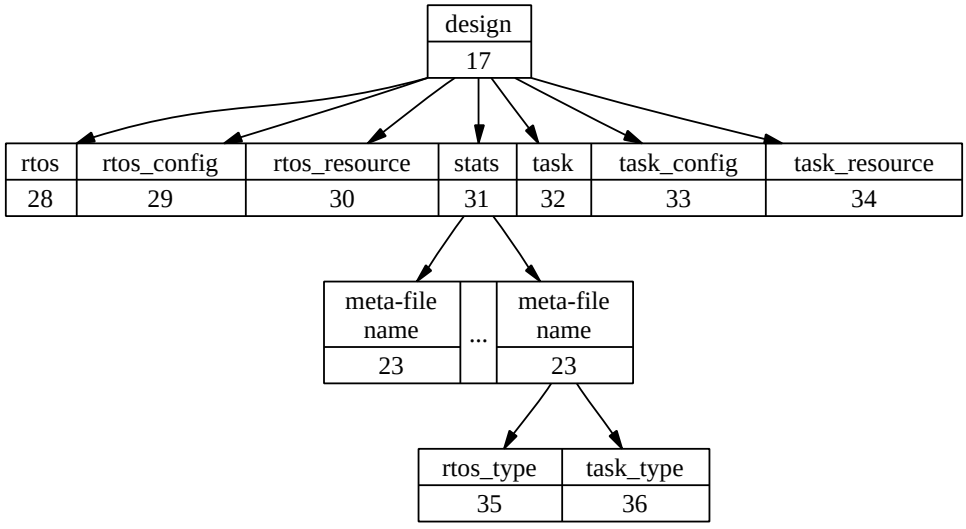


Figure B.3: Design database directory structure

and specifying designed tool options. Using this simple language, automatic evaluation can be performed across structured combinations of the design components with controlled variation of design tool options.

18 and **23**: The reports located at this location are of simulation results compared across the runs specified in a meta-file. Several tools work together to extract, aggregate, and render viewable forms of numerous reports. With minimal effort, new reports can easily be incorporated to the automated flow. Reports are organized by meta-file name in **23**.

17: Each database group has a design directory sub-tree, see figure B.3, that organizes the component objects that can be composed into a system for evalua-

tion; namely schedulers, algorithm "tasks", candidate architecture mappings, and architecture resources. These objects are reside here.

28 and **32**: The structure of schedulers (RTOS) and system algorithms (tasks) reside here. This includes, among others, a graph-base description, an executable specification with a model of the algorithm, stimulus, and performance constraints. The framework can incorporate high-level estimates or cycle accurate implementations. Multiple levels of abstraction may be used concurrently to describe different system components as desired.

29 and **33**: In this research, the notion of candidates and templated architecture mappings is proposed. The candidates reside here. This includes the template structure itself and any properties relevant for a run-time management scheme.

30 and **34**: The descriptions of architecture resources are stored here. This includes resource types, instances, and any properties relevant for a run-time management scheme. (for example, properties might include standby energy, a specification of instance coordinates, etc.).

31, 35, and **36**: To assist in the interpretation of results during evaluation of parameterized component generation, several tools have been developed to generate comparative summaries across the components specified in a simulation run

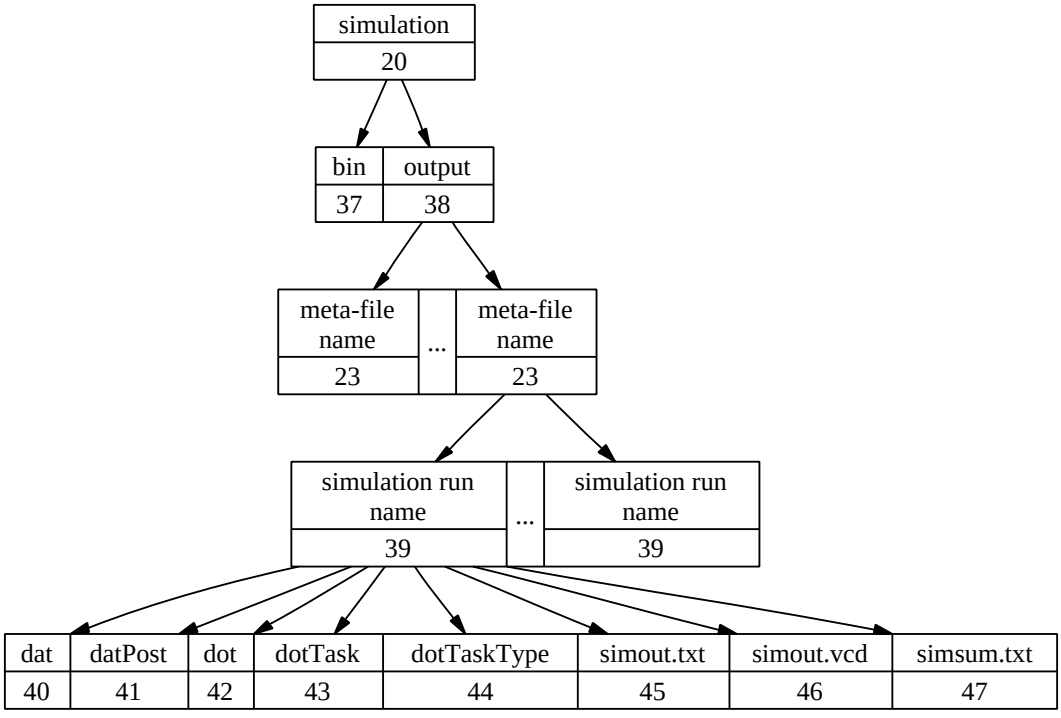


Figure B.4: Simulation results directory structure

meta-file. The compiled statistical summaries are organized within this sub-tree.

20: Tasks, architectures, candidates, schedulers, scheduler objectives, simulation options, and a based technology file are presented in concert to the design environment simulator forming a simulation run. The results of simulation are stored within the tree shown in figure B.4.

37: Due to limitations of the high-level modeling language [GLMS02][sys] upon which the simulator is built[2], a separate executable binary is created for

[2]Some initial effort was expended to work around the lack of dynamic process creation in System C 2.0 with some success. Since it is an open source C++ library, extension is possible.

each algorithm task set. These binaries are placed here. As it turns out, this scheme has the unintended benefit of supporting more general C++-based system models that are refinable and executable.

38 and **39**: The outputs of each simulation run, as specified by a meta-file, are stored, grouped by meta-file name, under **38** and **23**. The specific components that are to participate in the simulation run, as described above in **20**, are aggregated to create the simulation run names at location **39**.

40-44: Within the simulator, there are routines to query, extract, and record-to-file internal data representations of interest. A framework exist that facilitates the construction of queries. In **40-44**, a set of standard queries are performed and recorded as discussed next.

40 and **41**: Simulation internal data representations are recorded here and subsequently processed into charts. Post-simulation analysis routines can be registered with the simulator and invoked after simulation. These routines use location **41** to record results.

42, 43, and **44**: Simulation internal graph representations can be queried as described two paragraphs above. These results are subsequently transformed into more print-friendly versions. In **42**, general system graphs are written. The

However, the path to separate binaries presented significantly less resistance without any loss toward the goals of this research.

graphs for algorithm task instances and task types are recorded to **43** and **44**, respectively.

45: The step-by-step cycle-based detailed operations of each simulation is recorded to text file. Macros are provided to encourage consistent record formatting across different simulator functionality. This facilitates the development of Pre- and Post-simulation optimization/analysis functions and system management routines with consistent time-annotated record of events.

46: System state variables may be registered so that they will be recorded to a vector change dump file (VCD). With this interface, the states of all registered variables are recorded whenever there is a state change in any system variable. This public format is efficient in that only differential changes are recorded and it therefore yields a compact time-annotated history of system state transition. Moreover, numerous freely distributed tools are available to analyze these files structures.

47: In addition to composing the chart data in **41**, the registered post-simulation analysis routines may also write text base summaries to file like the one identified here.

Table B.3: Design environment command line tools

Name	Description
`tggen`	Algorithm task graph generation.
`rcgen`	Architecture and candidates generation.
`osgen`	Scheduler costs estimates generation.
`vgagen`	Heterogeneous multi-grain architecture family generation.
`ossim`	System simulator (see synopsis in Listing B.2).

B.5 Tools

The design environment text-base tools, mostly written in C++, are summarized in table B.3. Each is self documenting (via: `tool_name --help`) with a parameterized command line interfaces. Each are discussed in the following paragraphs.

`osgen`: This tool has been developed to facilitate the process of generating high-level cost estimates for systems schedulers within the proposed framework. A parameter file is used to: (1) Characterize the architectural resources available to schedulers. (2) Describe how schedulers make use of these resources. And, (3) provide technology independent cost estimates for the relationship between the two. This tool refines the scheduler model into representations that are meaningful for the simulator.

`tggen`: This tool has its roots in another program [DRW98] whose purpose

is similar. The task graph generation functionality of this program has been extended to support cluster — both types and instances — and has been heavily modified to (1) automatically generate C++ executable models that fit within the design environment framework, and to (2) produce additional intermediate formats for visualization post processing using the techniques presented in [GN00]. For an example of the C++ code generated by `tggen` to model stimulus and task behavior see the listings B.3 and B.4. As with the other generation tools, task sets are described using parameter text files[3].

`rcgen`: Parameterized architectural model are processed by this tool. These parameter files are divided into three sections. In the first are parameters that model architecture gate topology for both configuration and computation structures. In the second section are parameters that control stochastic generation of resources — types, instances, area, and other properties of interest to a run-time scheme. The last section has parameters that drive stochastic generation of candidate templates for each task type in the system.

`vgagen`: This tool was written to generate structured sets of architecture parameter files that may be subsequently refined by `rcgen`. It attempts to model the complex relationships within an architecture and between one and its use. Specifically, it focuses on (1) the relationships that exist between gate topology

[3]The parameter naming scheme found here differs substantially from that of the others generation tools since the naming originated with the work of others.

and its impact on resource structures, and (2) the relationship that exist between these resources, and their use by algorithms. In essence, this tool generates a set of architectures, using a first-order model of the relationships (1) and (2) assuming some fixed overall gate count for each architecture in the set.

`ossim`: This, the system simulator is, by far, the most complex piece of code developed for this designed environment. It has a highly flexible interface, see listing B.2. As described in the listing synopsis, it is a simulator for high-level performance estimation of dynamic reconfigurable digital signal processing systems. Command line arguments exist to specify the system algorithms, architecture, base-technology, schedulers, scheduler objectives, input stimulus, and simulation control options.

Listing B.2: Simulator command line synopsis

```
1
2  USAGE:
3
4      ./ossim  -s <name string> -m <name string> -p <name string> [-d
5               <directory name string>] [-x <directory name string>] [-o <directory
6               name string>] [--schTPFready <predicate function name>] [--schTPFconfig
7               <predicate function name>] [--schTPFdone <predicate function name>]
8               [--schCPFsearch <objective function name>] [-a <one of (per|rwd)>] [-r
9               <one of (cyc|hyp|don)>] [-c <integer count (>0)>] [--clusDoneCnt
10              <integer count (>0)>] [--hypIterCnt <integer count (>0)>]
11              [--maxSimTimeMul <integer multiplier (>=1)>] [--abortAtMaxSimTime]
12              [--hypPerExt] [-l <integer in (0-100)>] [--] [-v] [-h]
13
14
15 Where:
16
17     -s <name string>,  --schName <name string>
18       (required)  (value required)  scheduler name
19
20     -m <name string>,  --mappName <name string>
21       (required)  (value required)  architecture name
```

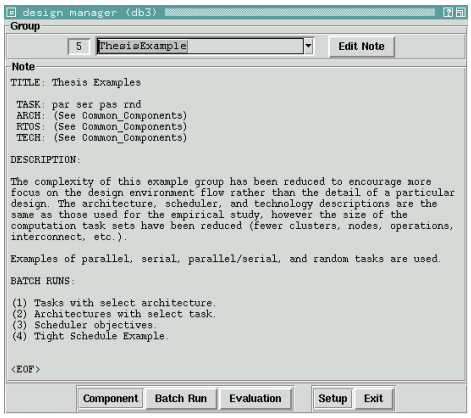
```
22
23      -p <name string>, --techName <name string>
24        (required) (value required) technology name
25
26      -d <directory name string>, --designDir <directory name string>
27        (value required) design directory
28
29      -x <directory name string>, --dtdDir <directory name string>
30        (value required) XML DTD directory
31
32      -o <directory name string>, --simOutDir <directory name string>
33        (value required) simulation output directory
34
35       --stimBaseDir <directory name string>
36            (value required) simulation stimulus input base directory
37
38      --schTPFready <predicate function name>
39        (value required) scheduler task ordering predicate function for ready
40        tasks
41
42      --schTPFconfig <predicate function name>
43        (value required) scheduler task ordering predicate function for
44        configuration tasks
45
46      --schTPFdone <predicate function name>
47        (value required) scheduler task ordering predicate function for done
48        tasks
49
50      --schCPFsearch <objective function name>
51        (value required) scheduler candidate configuration search objective
52        function
53
54      -a <one of (per|rwd)>, --cluActMethod <one of (per|rwd)>
55        (value required) cluster input stimulous activation model
56
57      -r <one of (cyc|hyp|don)>, --simDurMethod <one of (cyc|hyp|don)>
58        (value required) simulation duration/termination scheme
59
60      -c <integer count (>0)>, --cycleCnt <integer count (>0)>
61        (value required) cycle count for (cyc)-based scheme
62
63      --clusDoneCnt <integer count (>0)>
64        (value required) cluster completion count for (don)-based scheme
65
66      --hypIterCnt <integer count (>0)>
67        (value required) hyper period iterations for (hyp)-based scheme
68
69      --maxSimTimeMul <integer multiplier (>=1)>
70        (value required) maximum simulation time multiplier for abort monitor
71
72      --abortAtMaxSimTime
```

```
73      abort monitor terminates simulation if time limit is reached
74
75    --hypPerExt
76      extend hyper period by delay of latest cluster start
77
78    -l <integer in (0-100)>, --verbosity <integer in (0-100)>
79      (value required)  output verbosity level
80
81    --, --ignore_rest
82      Ignores the rest of the labeled arguments following this flag.
83
84    -v, --version
85      Displays version information and exits.
86
87    -h, --help
88      Displays usage information and exits.
89
90
91    SYNOPSIS: A candidate-centric simulator for high-level performance
92    estimation of dynamic reconfigurable DSP systems. Command line arguments
93    exist to specify the system algorithms, architecture, base-technology,
94    schedulers, scheduler objectives, and simulation control options.
```

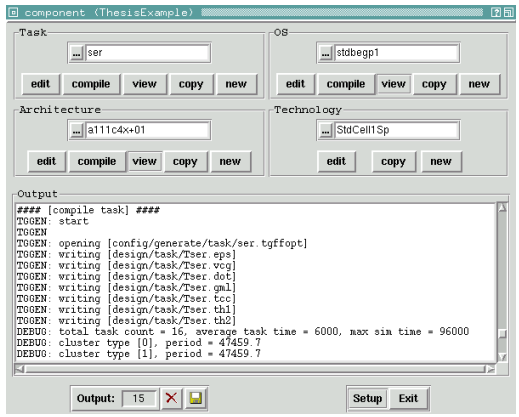
Although the core design tools are command-line based, a collection of Tck-/Tk [Ous90]/[Ous91] graphical user interface front ends have been developed to manage the design flow and large data sets that result from case-study design generation and simulation[4]. Snapshots of these graphical front ends are shown in figure B.5(a)-(d) and are discussed in the next several paragraphs. For each interface, irrelevant operations, within various contexts of the flow, remain ghosted until they become relevant.

`mdm`: The (manifold) design manager, shown in figure B.5(a), is a graphical interface that provides the main starting point to select the a working group of

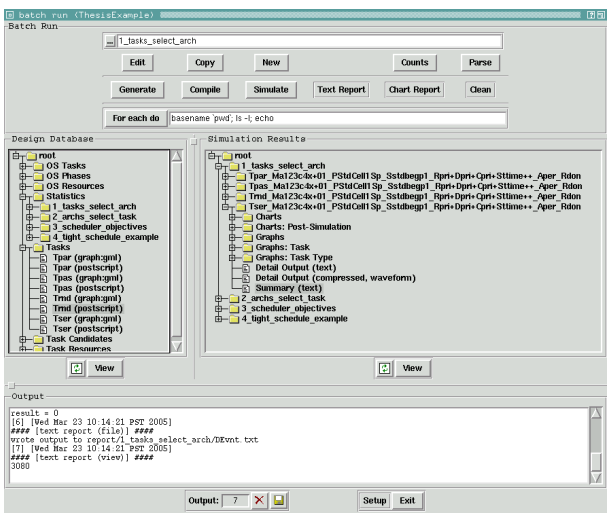
[4]For example, for a system of s_t scheduler types, s_i scheduler instances, t_t task types, and t_i task instances, $f_n = 4 \times (s_t + s_i + t_t + t_i + 1) + 42$ files are produced during each simulation run in the base-case setup for analysis and report generation.



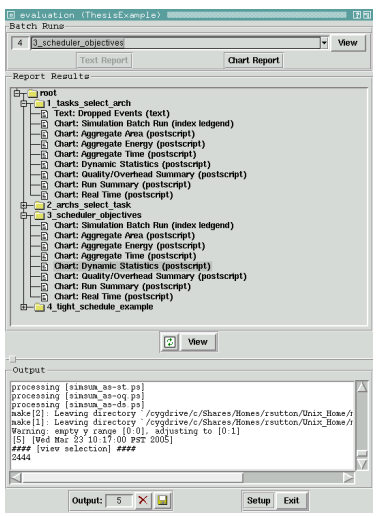
(a) Design manager



(b) Component generation



(c) Simulation batch run



(d) Multi-run evaluation

Figure B.5: Design-flow graphical user interfaces

a design database (see **17** above). It maintains a description of each group and provides shortcuts to launch other tools.

`mdm_comp`: The component designer, shown in figure B.5(b), is a graphical interface that orchestrates system object generation. Algorithm task sets, schedulers, and architectures, may be generated from parameterized descriptions. This facilitates empirical study of basic relationships in dynamic reconfigurables. This tool manages the creation of component parameter files and their compilation into other representations. Architecture generation for both the task sets and schedulers use a common technology file where based costs in time, energy, and area are established.

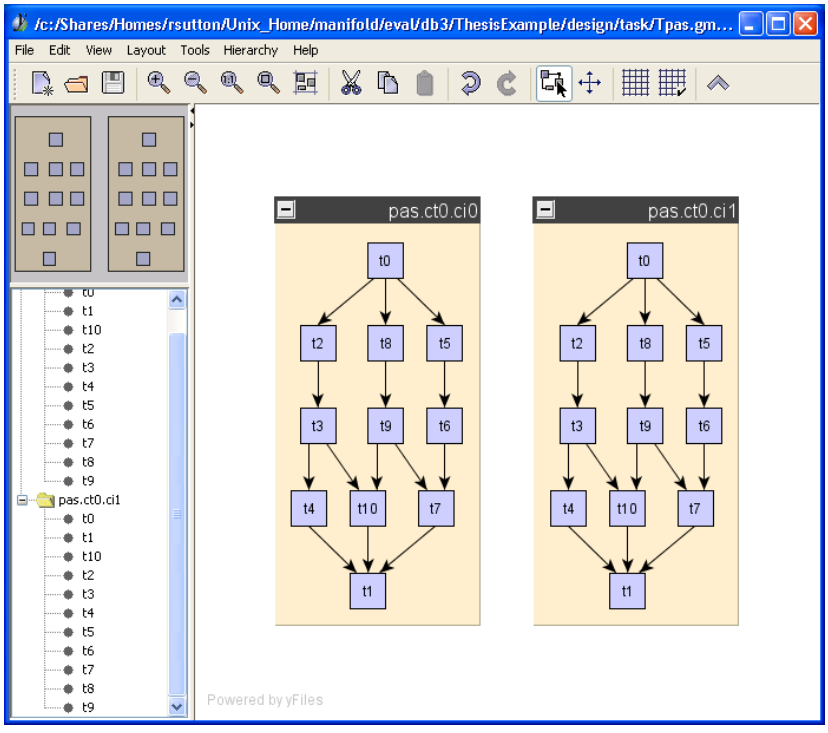
`mdm_batch`: Each simulation can be run directly from an interactive shell. However, a quick look at the simulator command line synopsis in listing B.2 and it is obvious that this strategy would be tedious and error prone. Inasmuch, a simulation run meta-file scheme has been established, as described prior in section B.4 (see listing B.1), that allows multiple runs to be submitted in batch. This user interface, shown in figure B.5(c), manages the creation of these meta-files. It also may be used to generate the components, compile the executable models, and perform each simulation run specified within a selected meta-file. Subsequent to simulation, various text-base and chart-based reports may be generated from respectively named menus. The design database and simulation results are

presented using a hierarchal tree representation with expandable and collapsible nodes. Objects within each tree have multiple representations. The setup menu provides control over the component views presented in these trees.

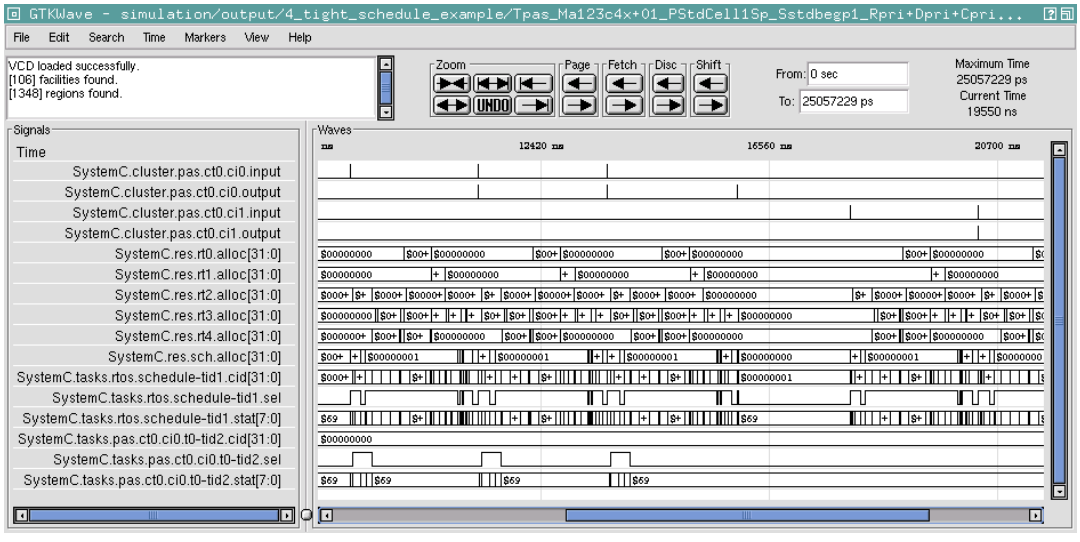
`mdm_eval`: To help expose relationships across simulation sets, the evaluation graphical interface, shown in figure B.5(d), has been developed. It drives the tools that aggregates and process data across multiple stimulation runs for comparison purpose. A setup menu allows selection of component views as described in the preceding paragraph. The component, batch run, and evaluation graphical interface each use output logs that detail each step of the exploration process. These logs are often invaluable records of an exploration path and may be saved to text files for future reference.

One of the big challenges in the development of any simulation-based designed environments is the "capture" of meaningful system descriptions and the exposure and evaluation of the complex, often latent, relationships hiding in the resulting data. Both of these problems are outside of the scope of this research but essential to its practical success. Two open-source graphical tools were selected to provide this functionality. A synopsis of these tools follows.

`Yed`: Is a free Java graph editor application from [yWo], shown in figure B.6(a), that provides sophisticated and well behaved layout and/of visualization functionality for graph-based data structures. It can read and write graph representations



(a) yEd: Component designer



(b) GTKWave: Vector change dump analysis

Figure B.6: Graphical tools from other sources

in various formats including the public `.gml` format used in this environment.

`GTKWave`: Is a mature feature rich waveform viewer written by [Byb] that is open source and supports numerous data formats including the popular vector change dump format (VCD). It is shown in figure B.6(b), and is very useful during the analysis of system timing behavior.

B.6 Reports

To provide feedback during design exploration, notable effort went toward the development of mechanisms for report generation in the design flow. Inasmuch, the tools of the environment, most importantly, the simulator, have been developed in a manor that facilitates the extraction of structures, static state, and dynamic state from internal representations of system objects.

The design environment currently has reports implemented within the five categories presented in table B.4[5]. Reports are generated on a per-simulation and multi-simulation bases. Additional reports flows may be implemented as discussed in the next few paragraphs.

"Well behaved" routines within the framework make use of standard mecha-

[5]Graph reports for system component types and instances are produced during each simulation. Therefore this report category count varies with each system.

Table B.4: Implemented report flows

Report	Simulation	
Category	Single-run	Batch-run
Text	2	27
Chart	11	8
Graph	7+	—

nisms for text-based output logging. These mechanisms establish the structure that is later relied upon during the creation of text-based reports. Routines register unique names and use these names to submit text message logging requests. The names, along with other relevant information — such as current simulation time — are recorded using a standard format to a simulation log files, see (**45**)(**47**) of figure B.4.

The simulator has a reporting framework that includes data structures for use in specifying (1) the desired set of system objects, search schemes, and (2) the report-dependent properties that configure how these objects are presented, style schemes. Functions are developed that assist in the population of these data structures. Sets of objects can be specified by ID, name, range, type, for example to complete (1). And groups of standard report properties are specified and assigned names to complete (2). Of course, both can be completed directly should more fine-level control be desired, at the cost of increased coding effort.

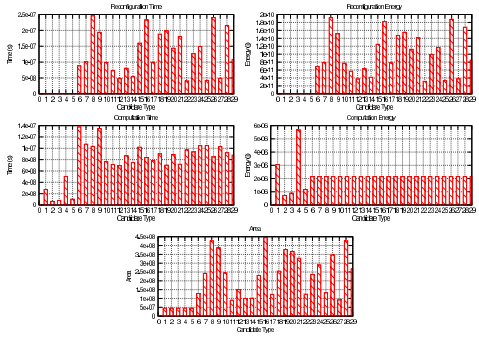
With this framework, data extraction is expressed as $\{searchscheme, stylescheme\}$

and one could, for example, request a view of the internal data representations for "the scheduler tasks objects using the fully decorated report view style" to be gathered and written to a file. This step produces an intermediate file, currently either `.dat` or `.dot`, that is subsequently processed for visualization. Makefiles and scripts are used to sequence this post-processing.

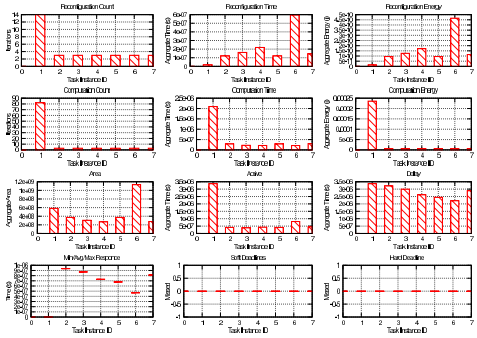
The environment makes use of a freely available graphing tool [gnu] to render charts. This tool is highly configurable and has a rich scripting language that allows data to be rendered in many forms and styles. It is a good fit for the chart based visualization of static data sets produced by the design flow. Examples of these charts are presented in figures B.7(a)-(f). Graph reporting proceeds in similar fashion, except that the sets of specified system objects are written in a format that maintains the graphical relationships between the objects. The environment makes use of [GN00] to render graphs. Examples of these graphs are presented in figures B.8(a)-(e).

B.7 Framework Expansion

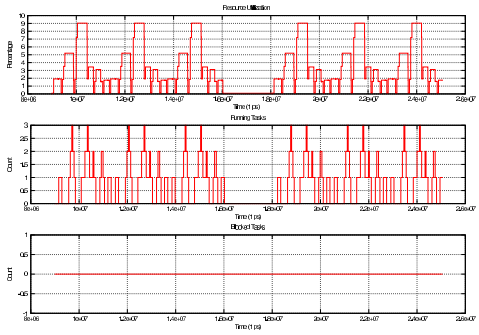
Expansion of the framework typically falls in one or more of the following four categories. (1) Customized stimulus models. (2) Customized task sets. (3) Customize system management schemes — schedulers and the like. (4) Attachment



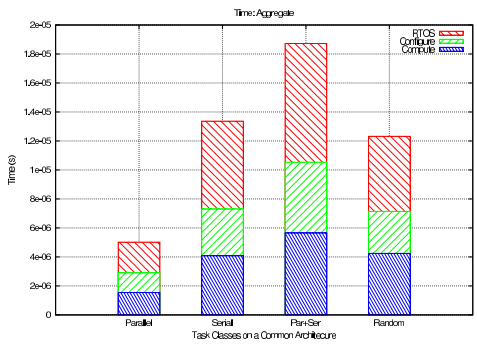
(a) Candidate costs estimates



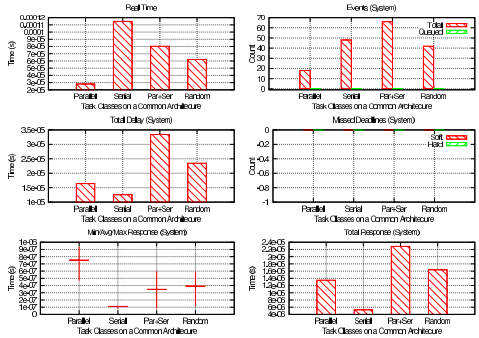
(b) Task instance performance



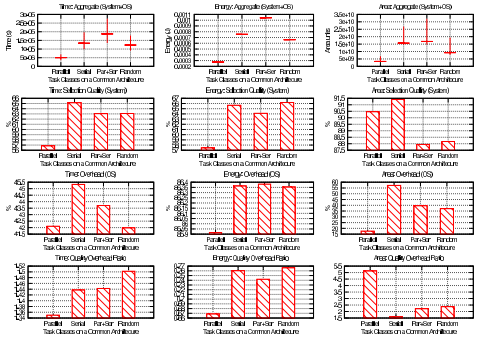
(c) Utilization traces



(d) Aggregate cost breakdown

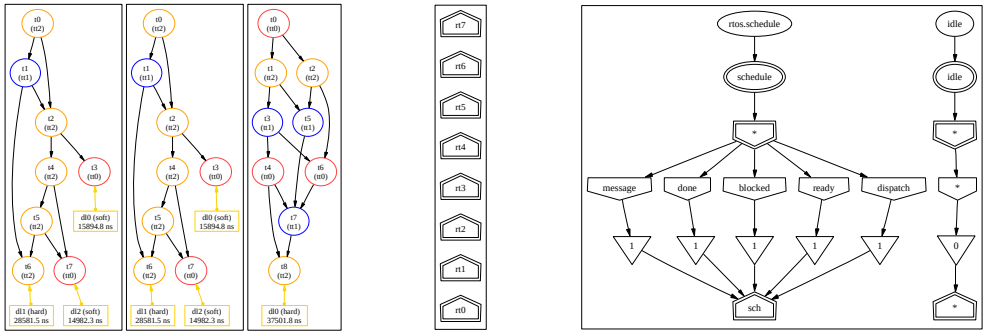


(e) Timing metric summaries

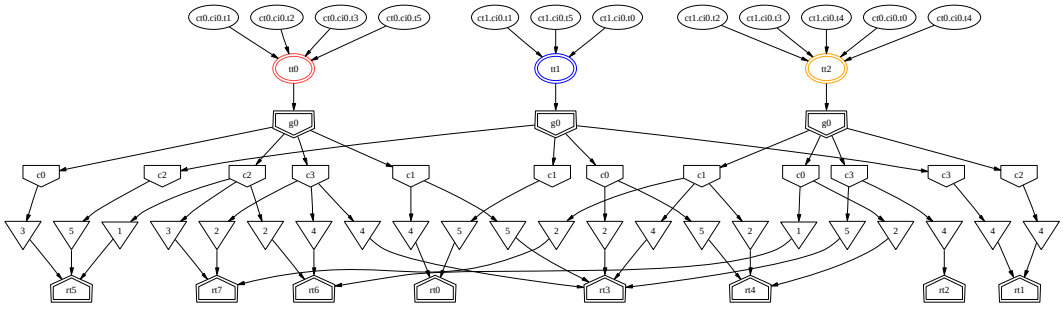


(f) Performance quality measures

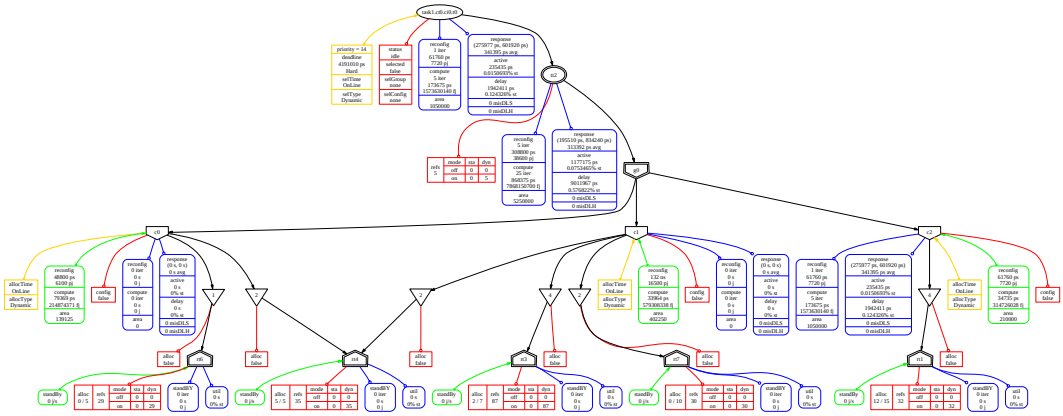
Figure B.7: Chart report examples



(a) Algorithm task graph (b) (c) System management scheduler



(d) Candidate-centric view of task instance to resource type mapping



(e) A task instance and its template, decorated with properties for runtime management.

Figure B.8: Graph report examples

of additional properties to system objects.

Properties may be attached to tasks, candidates, resources, etc. to provide increased awareness/scope for new management schemes. These properties may then be evaluated and manipulated as desired within the framework. For example, the registered routines for pre-simulation and post-simulation analysis, in addition to the management initialization and runtime code, are within the scope of these system object properties.

The current management scheme implements object properties that fall into one of the following categories: parameters, state, costs, and metrics (where the latter two are specific forms of the former two, respectively).

B.7.1 Stimulus Modeling

Stimulus is implemented as a separate task or collection of tasks. Each cluster[6] within a system of tasks has its own stimulus. According to some desired data pattern behavior, the stimulus "activates" one more tasks within its cluster using the mechanisms of the implemented model of computation. There is no other constraint on how stimulus models are constructed. Data can be read from a file or generated using some algorithm. An example stimulus task is shown in listing

[6]A cluster contains one or more graphs and essentially adds a level of hierarchy and/or abstraction to algorithm composition.

Listing B.3: Stimulus function modeling example

```
1  ////////////////////////////////////////////////////////////////////
2  // cluster stimulous
3  ////////////////////////////////////////////////////////////////////
4  void sim_sys::t_pasct0ci0start() {
5      TM_CLUSTER_INIT( "pas.ct0.ci0", 0, true, 30000, 8988.29, SC_NS );
6      wait ( 8987.29, SC_NS );
7
8      do {
9          TM_CLUSTER_START;
10         TM_CLUSTER_START_TRACE( 1, SC_NS );
11
12         notify ( e_pasct0ci0start );
13
14         if ( TM_PERIODIC_SIM )
15             wait ( 29999, SC_NS );
16         else
17             wait( e_pasct0ci0done );
18
19     } while ( true );
20 }
```

B.3.

Macros are provided so that stimulus behavior can be reported back to the framework. This allows the stimulus behavior to be recorded to the simulation time-line. Additionally, it becomes a property that can be observed during system-level dynamic management.

B.7.2 Task Modeling

Tasks can be constructed with significant latitude. Many models of computation can be implemented and multiple models can be intermixed within a

design. There are essentially two constraints. (1) They must adhere to the constructs of the underlying high-level modeling language. (2) They must conform to the framework imposed by the design environment. Given the current high-level language[GLMS02] upon which the framework is implemented, the full expressiveness of C++ is available for task modeling.

An example task skeleton is shown in listing B.4. It uses a buffered-input concurrent sequential process model of computation. In the listing, the upper task models the behavior and the lower task models the input queue.

To conform to the constructs of the design environment framework, tasks must (1) indicate that they are ready for computation, (2) indicate that the computation cost estimation is to be performed, and (3) indicate that the task has completed. These actions are synchronized with primitives: `TM_TASK_READY`, `TM_SIMULATE_COMPUTATION`, and `TM_DONE`. The runtime system uses these events to dynamically management resources and to evoke the appropriate estimators for selected candidate.

B.7.3 Scheduler Modeling

The design environment framework is used to assign tasks to registered schedulers. This mapping is referenced when a task requests service as described in

Listing B.4: Task modeling example

```
22 ////////////////////////////////////////////////////////////////////////
23 // task [pas.ct0.ci0.t0] structure
24 ////////////////////////////////////////////////////////////////////////
25 void sim_sys::t_pasct0ci0t0S() {
26     TM_CREATE_TASK ("pas.ct0.ci0.t0", "tt0", "pas.ct0.ci0");
27     TM_PEND_EVENT;
28
29     while ( true ) {
30         TM_TASK_READY;
31
32         ////////////////////////////
33         // perform computation here //
34         ////////////////////////////
35
36         TM_SIMULATE_COMPUTATION;
37
38         notify ( e_pasct0ci0t0t2 );
39         notify ( e_pasct0ci0t0t5 );
40         notify ( e_pasct0ci0t0t8 );
41
42         TM_TASK_DONE;
43         TM_PEND_EVENT;
44     }
45 }
46
47 ////////////////////////////////////////////////////////////////////////
48 // task [pas.ct0.ci0.t0] input event queue
49 ////////////////////////////////////////////////////////////////////////
50 void sim_sys::t_pasct0ci0t0E() {
51     TM_GET_TASK_ID ("pas.ct0.ci0.t0");
52     wait( e_pasct0ci0start );
53
54     while ( true ) {
55         TM_POST_EVENT;
56
57         wait( e_pasct0ci0start );
58     }
59 }
```

Listing B.5: Scheduler example: Pre-static analysis initialization

```
1  ////////////////////////////////////////////////////////////////////////
2  // scheduler routine called prior to pre-simulation static analysis
3  ////////////////////////////////////////////////////////////////////////
4  void sim_sys::stdbegp1_initPreSA( const char * funName )
5  {
6      SM_SIM_MSG_LOG( 2, << funName << ": queing static analysis" << endl; )
7
8      rtosSimParam.sAnalysisQ.push( "OffLineSelection" );
9      rtosSimParam.sAnalysisQ.push( "OffLineAllocation" );
10     rtosSimParam.sAnalysisQ.push( "RefResUpperBounds" );
11     rtosSimParam.sAnalysisQ.push( "RefResLowerBounds" );
12     rtosSimParam.sAnalysisQ.push( "HyperPeriod" );
13
14     rtosSimParam.sAnalysisQ.push( "SetSchAttr_Pri_DLMon" );
15     rtosSimParam.sAnalysisQ.push( "SetSchAttr_Pri_Compress" );
16
17     rtosSimParam.sAnalysisQ.push( "BestWorstCaseDL" );
18
19     return;
20 }
```

the previous section. In this manor many scheduling schemes can be explored. One could, for example, devise a scheme that uses a " higher-level" scheduler that re-maps tasks across scheduler boundaries according to some global management objective.

The behavior of each scheduler must be coded to implement the detail of a system management scheme. Schedulers must implement at least three tasks. (1) A task called prior to system pre-simulation static analysis, see listing B.5. (2) A task called after pre-simulation static analysis, see a listing B.6. And, (3) at least one task called when tasks needs service. The framework uses two index mappings that separate the "task ready" and "task done" services. These

Listing B.6: Scheduler example: Post-static analysis initialization

```
22 ////////////////////////////////////////////////////////////////////////
23 // scheduler routine called after pre-simulation static analysis
24 ////////////////////////////////////////////////////////////////////////
25 void sim_sys::stdbegp1_initPostSA( const char * funName )
26 {
27     SM_SIM_MSG_LOG( 9, << funName << ": reviewing task attributes" << endl; )
28
29     BEGIN_FE_TASK( rtosTasks, task, tid, false )
30         SM_SIM_MSG_LOG( 9, << funName << ": task = [" << task.name
31                            << "] schAttr.pri = [" << task.schAttr.priority
32                            << "] schAttr.DL = [" << task.schAttr.deadline
33                            << "]" << endl; )
34     END_FE_TASK
35
36     return;
37 }
```

mappings can point to a single scheduler task object if desired. An example code fragment is shown in listing B.7.

Listing B.5 shows how registered analysis routines are called within the framework. This mechanism can be used to analyze and manipulate system properties prior to simulation.

Listing B.7: Scheduler example: Done task processing code fragment

```
39 ////////////////////////////////////////////////////////////////////////////
40 // single thread dynamic best effort scheduer routines (FRAGMENT)
41 ////////////////////////////////////////////////////////////////////////////
42 void sim_sys::stdbegp1_schedule()
43 {
44     // done task list processing
45     if ( int list_size = d_SL.size() ) {
46         ostName = " SCHED-DL";
47 
48         // select cost model for this routine
49         SM_TASK_NEW_CANDIDATE( ostName, SCHE_TID, 0, 1 );
50 
51         SM_SIM_LOG_TASK_LIST( ostName, d_SL, sli );
52         int alloc_freed = 0, select_freed = 0, select_kept = 0;
53 
54         for( sli = d_SL.begin(); sli != d_SL.end(); ) {
55 
56             alloc_freed += rtosDeallocate ( *sli );
57             if ( rtosConfigDeselect( *sli ) )   select_freed++;
58             else                                select_kept++;
59 
60             SM_STATUS_ADVANCE( d_SL, sli, i_SL, i );
61 
62             sli = d_SL.begin();
63 
64             // execute cost model advancing time, energy, and area
65             SM_TASK_LOOP_ITER( SCHE_TID );
66         }
67 
68         SM_SIM_LOG( ostName )
69             << "allocations freed = " << alloc_freed
70             << ", selections freed = " << select_freed
71             << ", selections kept = " << select_kept
72             << ", for " << list_size << " done task(s)" << endl;
73     }
74 
75     // log que status & update trace file
76     SM_SIM_LOG_TASK_STATUS_SUMMARY;
77     SM_TRACE_FILE_ENTRIES( SM_TFWM );
78 
79     return;
80 }
```

Index

adaptation, 18

adaptive schemes, 63

algorithm analysis, 47

algorithm estimates, 49

algorithm profiling, 49

analysis dependency, 117

arbitrary selections, 63

architecture resource, 43

architecture template, 22

binding, 35

binding mode, 68

candidate, 37, 46

coding languages, 31

composite relations, 62

concurrency, 97

configuration caching, 25

configuration management, 65

control treads, 97

cost dimensions, 23

cost distribution, 23

cost estimation, 58

deferred binding, 53

difference-based, 25

dynamic adaptation, 23

dynamic algorithms mapping, 32

estimation abstracts, 59

exploration, 48

hard deadline, 39

management properties, 55

mapping, 25, 32

mapping mode, 40

mapping time, 33

metrics, 57

mode use, 42

multi-use, 16

multi-user, 16

objective function, 60

offline, 33

online, 33

opportunity, 48

overheads, 24

partially-mapped, 38

pipelined, 19

post-design, 34

QoS, 16

reconfigurable processor, 22

refinement, 50

resource reservation, 41, 64

run-time, 33

run-time selection, 60

scheduling, 60, 84

search ordering, 62

selection quality, 138

sequenced regions, 19

simple relations, 61

soft deadline, 39

standards, 5, 18

statistical approaches, 64

strictness, 62

system model, 81

system properties, 78

system-level, 61

template instantiation, 64

template selection, 40

templated-mapping, 35, 44

trade-off, 20

triggered regions, 19

Colophon

This dissertation has been compiled by the open source typesetting tool *LaTex2e* under the Open Source unix-like environment *cygwin* running within a commercial operating system. The following macro packages were used to assist in the typesetting: *graphicx*, *relsize*, *dropping*, *caption*, *subfig*, *epsfig*, *amssymb*, *amsmath*, and *letterpaper*. The figures were drawn with one of *Xfig*, *ipe*, or *UMLet*; all Open Source tools. The screen dumps of the CAD tools were produced with *xwd*, for tools running under the unix environment, and *Gadwin PrintScreen*, for tools running in the commercial operating system host environment. The graphs are direct output from the CAD tools developed during this dissertation research. These tools made extensive use of AT&T's Open Source visualization graphic library *Graphviz*. The charts are all generated by routines that utilize *GNUPlot*, an Open Source plotting system. The following two free Java-based applications were used: *Freemind*, a "mindmapping" application, to organize the content structure and *JabRef* to maintain the bibliography database. The text of this dissertation was dictated using speech recognition technology developed by dragon systems.

Typography, when done properly, is a thoughtful blend of art and science and I hope that you enjoyed a balance of both. However, if you did not enjoy the science contemplated within, at the least, I hope you enjoyed the art.

▽ ▽ ▽

Except for the commercial speech recognition system, and the commercial operating system it required, this entire research effort has been completed using freely available Open Source software.

Openness levels the playing field.

▽