MULTIOBJECTIVE SYNTHESIS OF LOW-POWER REAL-TIME DISTRIBUTED EMBEDDED SYSTEMS

Robert P. Dick

A DISSERTATION

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE

BY THE DEPARTMENT OF

ELECTRICAL ENGINEERING

November 2002

© Copyright 2002 by Robert P. Dick.

All rights reserved.

Abstract

This dissertation presents methods for automating the synthesis of embedded systems, i.e., special-purpose computers. In addition, it describes a method for analyzing the manner in which real-time operating system use influences embedded system power consumption.

After introducing the embedded system synthesis problem and summarizing previous work in the field, we present four evolutionary algorithms that simultaneously optimize the different costs of embedded systems, e.g., price, power consumption, response time, and area, while ensuring that hard real-time constraints are met. These algorithms generate multiple solutions that present tradeoffs between different architectural costs. Each algorithm targets a different embedded system domain. The first algorithm synthesizes distributed embedded systems. The second synthesizes systems-on-chip composed of intellectual property cores that may come from different vendors. It does clock selection, floorplanning block placement, and bus topology generation. The third synthesizes distributed client-server systems in which the bandwidth of client-server communication is tightly constrained, e.g., wireless embedded systems. It incorporates a novel scheduling method tailored to embedded systems with multiple clients for each server. The fourth synthesizes embedded systems that may contain reconfigurable processors. In addition, we present a method of analyzing the effects of real-time operating system usage on the overall performance and power consumption of embedded systems.

Acknowledgments

First, I would like to thank my advisor, Niraj Jha. We closely collaborated on all the work presented in the body of this dissertation. He has all the traits of an excellent research advisor; he is intelligent, diligent, methodical, careful, imaginative, and even-tempered. I appreciate the corrections and suggestions offered by my dissertation readers: Niraj Jha, Sharad Malik, and Anand Raghunathan. In addition, Keith Vallerio helped me with numerous administrative and research problems while I was writing this dissertation.

I would like to thank Zhen Luo for modifying his design rule checking software [1] in order to collect metal density information from a number of layouts. This information was used to build the system-on-chip synthesis benchmarks described in Sections 6.8 and 7.10.2. It also reinforced my view that one can relate global routing layer metal density to floorplanner quality. I appreciate Zhigang Pan's help in answering the questions I asked while implementing his, and Jason Cong's, wiring delay model [2] for use in the work described in Section 7.7. I thank David Dobkin for his suggestions during design of the bus topology generation algorithm described in Section 7.8. He helped me to look at the problem from an unconventional perspective. I-Jong Lin's suggestions helped in developing the clock frequency selection algorithm (Section 7.10.1).

In addition to Niraj Jha, Anand Raghunathan and Ganesh Lakshminarayana collaborated on the real-time operating system power consumption analysis work described in Chapter 10. Much of this work was done during my term as an employee of NEC Computer and Communications Research Laboratories. I would like to thank Dr. Leslie French, from NEC C&C Research Labs, for helpful discussions on real-time operating systems and his assistance with the Ethernet interface example. David Rhodes, Wayne

Wolf, and I collaborated on the parametric task graph and resource database project described in Appendix A.

I would like to thank André Tits for correcting a multiobjective optimization terminology error in Section 4.5 and Forrest Brewer for encouraging me to describe a method of supporting streaming data communication within the task set model in Section 3.5.

Niraj Jha, Sun-Yuan Kung, Margaret Martonosi, Sharad Malik, Larry Peterson, and Andrew Yao gave competent instruction and advice about numerous engineering problems. I was glad to have the opportunity to discuss research with Jiong Luo, Li Shang, Tat Kee Tan, Shaojie Wang, Keith Vallerio, Yuan Xie, and Lin Zhong. Sarah Griffin, Sheila Gunning, and Karen Williams helped me deal with the university's bureaucracy. In addition, Sheila and Karen advised me as if I were a younger brother.

Financial support for my work at Princeton University was provided by an NSF Graduate Fellowship, NSF Grant Number MIP-9423574, a grant from NEC C&C Research Labs, Princeton University's George Van Ness Lothrop Fellowship in Engineering, Army CECOM, and DARPA under contract number DAAB07-00-C-L516.

Contents

Abstract **iii**

Acknowledgments **iv**

1 Introduction **1**

1.1 Embedded system design automation 3

1.2 Multiobjective embedded system design 5

1.3 Power consumption 6

1.4 Dissertation overview 8

2 Past work **11**

2.1 Hardware-software co-design research 12

2.2 Hardware-software co-synthesis 15

3 Definitions **21**

3.1 Hardware-software co-synthesis decisions 22

3.2 Problem taxonomy 22

3.3 Constraint specifications: Multi-rate task sets 24

3.4 Multi-rate task sets for real-time systems 26

3.5 Modification of the task set model for pre-computation and streaming 29

3.6 Processing elements (PEs) 30
3.7 Communication resources 31

4 Optimization algorithms 33
4.1 Solving NP-hard problems 33
4.2 Simulated annealing 38
4.3 Genetic algorithms 40
4.4 Parallel recombinative simulated annealing (PRSA) 43
4.5 Multiobjective optimization 43

5 Synthesis of Low-Power Heterogeneous Distributed Systems 49
5.1 Requirements for the optimization algorithm 50
5.2 Specialized hardware resources 51
5.3 Solution representation 53
5.4 Optimization algorithm 56
5.5 Clusters 57
5.6 Initialization and genetic operators 61
5.7 Solution evaluation 65
5.7.1 Scheduling 65
5.7.2 Task graph copies 69
5.7.3 Cost calculation 71
5.7.4 Constraint violation 72
5.8 Ranking and reproduction 73
5.9 Experimental results 75
5.9.1 Price optimization 76
5.9.2 Multi-objective power and price optimization 82
5.10 Conclusions 86

6 Enhanced Low-Power Heterogeneous Distributed Systems Synthesis 87

6.1 Communication and memory model 88

6.2 Optimization infrastructure 89

6.3 Multidimensional locality preserving crossover 91

6.4 Guided task assignment mutation 94

6.5 Initialization 100

6.6 Cost calculation 101

6.7 Solution cache 102

6.8 Benchmarks 103

6.9 Experimental results 106

6.9.1 Multiobjective optimization for the E3S benchmarks 106

6.9.2 Price-only optimization for examples from the the literature . . 107

6.10 Conclusions 113

7 Intellectual Property Core-Based System-on-Chip Synthesis 115

7.1 Motivation 116

7.2 IP core model 117

7.3 Algorithm overview 118

7.4 Clock selection 120

7.5 Tie prioritization 128

7.6 Floorplan block placement 129

7.7 Wiring delay and power consumption model 131

7.8 Bus topology generation 132

7.8.1 Motivation 132

7.8.2 Definitions and assumptions 133

7.8.3 Overview 134

7.8.4 Efficiency 137

7.9 Cost calculation 138
7.10 Experimental results 139
7.10.1 Clock selection 140
7.10.2 Feature comparisons 141
7.10.3 Multiobjective optimization for the E3S benchmarks 146
7.11 Conclusions 148

8 Wireless Low-Power Client-Server System Synthesis 149
8.1 Problem formulation 151
8.2 Motivating example 153
8.3 Scheduling and client-server pipelining 157
8.4 Cost calculation 166
8.5 Experimental results 167
8.5.1 Multiobjective optimization for the E3S benchmarks 167
8.6 Conclusions 172

9 Synthesis of Dynamically Reconfigurable Embedded Systems 173
9.1 Motivation 174
9.2 FPGA model 175
9.3 Scheduling 177
9.4 Experimental results 181
9.5 Conclusion 186

10 Analysis of Energy Consumption in Embedded Operating Systems 187
10.1 Introduction 188
10.2 Related work and contributions 191
10.3 Motivation for RTOS energy analysis 193
10.3.1 Anti-lock braking example 194

10.3.2 Commodity trading agent example 196
10.3.3 Ethernet interface example 199
10.4 Energy analysis infrastructure 201
10.4.1 Inputs and outputs 201
10.4.2 System overview 205
10.4.3 System details 209
10.4.4 Extending our approach to other embedded systems 210
10.5 Results and case studies 212
10.6 Conclusions and recommendations 220

11 Comparisons with Related Work 223

12 Contributions and Conclusions 227

A Task Graphs for Free 231
A.1 Introduction 231
A.2 Task set generation 233
A.3 Database generation 239
A.4 Conclusions 241

B Implementation 243

Bibliography 245

Chapter 1

Introduction

An embedded system is a computer within a host device, when the host device, itself, is not generally considered to be a computer. For example, the computers within automobiles, medical devices, and range finders are embedded systems. In most applications, well-designed, correctly functioning embedded systems are almost invisible to their users. Although consumers might be pleased that their cars automatically adjust their engine timing to achieve the best non-pinging performance possible with the currently available gasoline, they are unlikely to consider the fact that an embedded system makes this possible. It is also unlikely that most realize embedded systems are responsible for 30% of the price of the average car [3] and that microprocessors, alone, account for 10% of the price [4].

We are surrounded by embedded systems. When I wake up in the morning, the first thing I hear is the speaker of my digital alarm clock, activated by an embedded system. I get out of bed and put my breakfast in the microwave, allowing its water vapor sensing embedded system to perfectly cook my cereal. I call my parents and sister on my cell phone, a wireless client-server embedded system. The call is routed through a telecommunications infrastructure composed of numerous high-performance embedded systems. I get into my car, filled with embedded systems (15 in the average car [5]), and drive to my office, in which I am surrounded by embedded systems (in the copier, in my

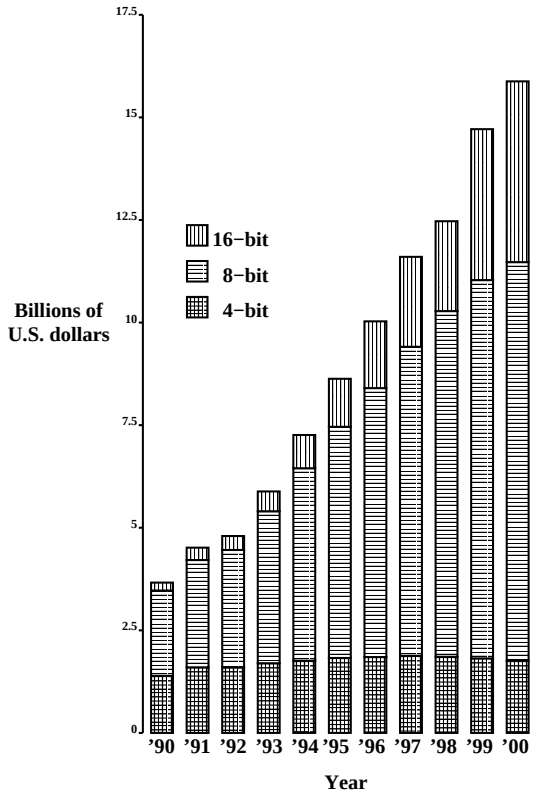

Figure 1.1: Estimated global microcontroller sales in billions of U.S. dollars [4].

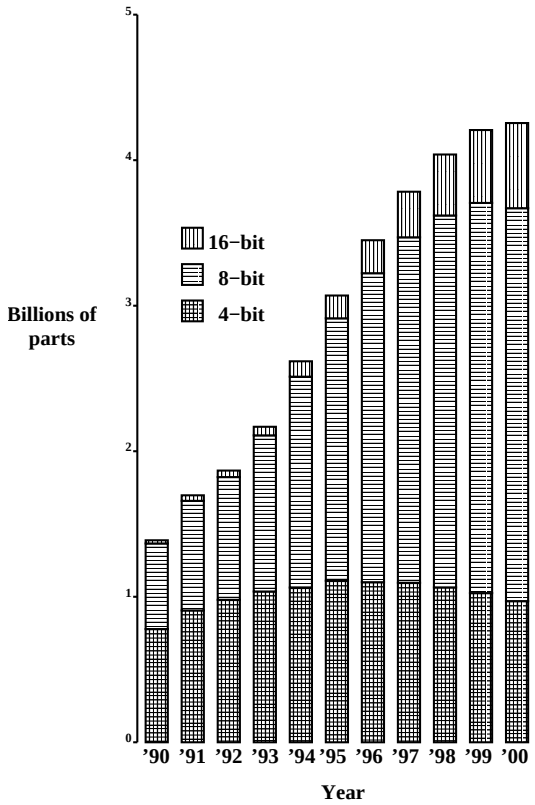

Figure 1.2: Estimated global microcontroller sales in billions of parts [4].

calculator, in the printer, and in the vending machine). I use these embedded systems because they make my life better: they make it easier to communicate with those I love, they help me to schedule my time, they improve the speed and safety of travel, they help me to manage information, and they assist me with hundreds of other tasks. Embedded systems make it possible for me to carry out these tasks perfectly without investing the time and energy necessary to become a specialist in hundreds of different skills.

Many consumers value embedded systems. It naturally follows that the size of the embedded systems market is large. Microcontrollers are processors typically used in embedded systems. They may differ from general-purpose processors by having more input/output support integrated on-chip or by having smaller caches. They may be less

expensive, have lower power consumption, or be designed for specialized tasks. Although embedded systems are composed of many other electronic components in addition to microcontrollers, information about the microcontroller market gives some insight into the embedded systems market. Figures 1.1 and 1.2 give dollar and part volumes for sales of 4-bit, 8-bit, and 16-bit microcontrollers. These figures indicate that the size of the microcontroller market is substantial (approximately $16 billion in the year 2000) and growing rapidly. Note that microcontrollers only account for a portion of the costs of embedded systems; the embedded system market is substantially larger than the microcontroller market.

In 1998, approximately 250 million 32-bit and 64-bit embedded microprocessors were sold [5], [6]. Even though this number is much higher than that for personal computers, workstations, and supercomputers (100 million [5]), it is dwarfed by the number of 4-bit, 8-bit, and 16-bit microcontrollers sold that year (approximately four billion as shown in Figure 1.2).

1.1 Embedded system design automation

Embedded system designers have difficult jobs. Customers have stringent expectations for embedded systems, some of which are listed in Figure 1.3. The first two characteristics in Figure 1.3 imply that embedded system designers need to carefully test their designs to confirm that the designs contain no errors and that every hard deadline is met. Any error has the potential to reduce the profitability of a product. The rigorous design required to meet customer expectations is time-consuming and expensive. A designer might be able to decrease the probability that an embedded system will miss real-time deadlines by using faster and more expensive processors. However, this conflicts with the third and fourth characteristics: people want inexpensive and cool

1. Software or hardware errors are not acceptable. Although many people tolerate it when a general-purpose operating system crashes, this sort of behavior is not acceptable for an anti-lock brake system. Many embedded systems have tasks with hard real-time deadlines. Missing a hard deadline is an error.

2. Embedded systems should not require bug fixes or upgrades. Expecting customers to change the software or hardware in their cars is unreasonable. Embedded systems are generally more difficult to upgrade than general-purpose applications. If a designer discovers a software error in an embedded system after it has been shipped, correcting that error in the field is likely to be more difficult than upgrading the software of a general-purpose application. Correcting hardware errors is even more difficult.

3. An embedded system should be sold at a lower price than competing products, i.e., price competition can be intense. Numerous competitors exist in many embedded systems market segments, e.g., mobile communication devices, home appliances, automobiles, and consumer electronics.

4. Power consumption should be low. High power consumption increases the price, weight, and volume of the cooling systems and energy sources used by an embedded system. This is particularly important for portable battery-powered embedded systems.

Figure 1.3: Customer expectations for embedded systems.

products with long battery lifespans. Each favorable attribute of an embedded system design conflicts with other favorable attributes, making it necessary to consider tradeoffs between them.

The incompatible expectations listed above conspire to make an embedded system designer's job difficult and unpredictable. A CMP Media LLC survey of 1,100 embedded system developers in 2001 indicated that the majority of their projects were running late, with a four-month lag the norm. The majority also failed to achieve even half of their expected performance [7]. We advance the following conjecture:

> The unpredictability of the embedded system design process is due to the predominance of manual, ad-hoc embedded system design.

According to Napper, "Embedded system design is largely the same as it was 20 years ago, when 8-bit microcontrollers were the state of the art" [8]. Consumer expectations have increased the demands on embedded systems. However, design automation software has not kept pace with the resulting increase in embedded system complexity.

Automation has the potential to help designers keep pace with increasing problem complexity. Its value has been demonstrated in a number of lower-level disciplines. As shown in Table 1.1, design automation has followed the historical trend from automation of low-level stages of the design process toward automation of increasingly high-level stages of the design process. High-level stages of the design process generally have more ambiguous problem definitions than low-level stages. Embedded system synthesis and hardware-software co-synthesis are still open problems. As they become increasingly well-defined and solved, embedded system designers will finally have a practical alternative to ad-hoc design.

1.2 Multiobjective embedded system design

As noted in the previous section, embedded systems have numerous attributes designers attempt to optimize, e.g., power consumption, price, and speed. It is frequently possible to improve one attribute only at the cost of another. This implies that, in order to understand the interplay between different embedded system costs, a designer needs to consider different alternative architectures.

The process of exploring the embedded system design space is equivalent to designing and analyzing numerous different embedded systems. Doing this manually would be time-consuming and expensive. Co-synthesis algorithms automatically synthesize

embedded system architectures. However, the vast majority of existing co-synthesis algorithms are capable of optimizing only one system cost: price. The few co-synthesis algorithms that attempt to minimize other costs either do so in an informal way, replace all but one cost with constraints, or combine multiple costs into a single cost with a weighting sum. Each of these approaches has significant disadvantages, as discussed in Section 4.5. In our research, we have taken care to avoid these disadvantages by developing a truly multiobjective approach to embedded system synthesis.

1.3 Power consumption

Although embedded system power consumption is only one cost among many, it has become increasingly important in recent years. In the past, embedded system power consumption was frequently ignored, or modeled in extremely coarse and inaccurate ways. However, proliferation of portable embedded systems during the past few years has focused attention on the reduction of power consumption. Numerous embedded systems (e.g., cellular phones, personal digital assistants, clocks, and games) are portable. It is important that they be light and compact. High power consumption implies high heat dissipation. Embedded systems with high heat dissipation require bulky cooling devices, e.g., heat sinks or fans. In addition, a portable embedded system with high power consumption requires bulkier batteries in order to have the same run time as a lower-power embedded system. Although portability is not a factor for stationary embedded systems, high power consumption is still a disadvantage. It increases the price of running and cooling the embedded system.

Table 1.1: Design tools development [9]

1950-1965	Manual design
1965-1975	Layout Editors Automatic routers (for PCB) Efficient partitioning algorithm
1975-1985	Automatic placement tools Well-defined phases of circuit design Significant theoretical development in all phases
1985-1990	Performance-driven placement and routing tools Parallel algorithms for physical design Significant development in underlying graph theory Combinatorial optimization problems for layout
1990-present	Over-the-cell routing tools Three-dimensional interconnect based physical design Synthesis tools mature and gain widespread acceptance

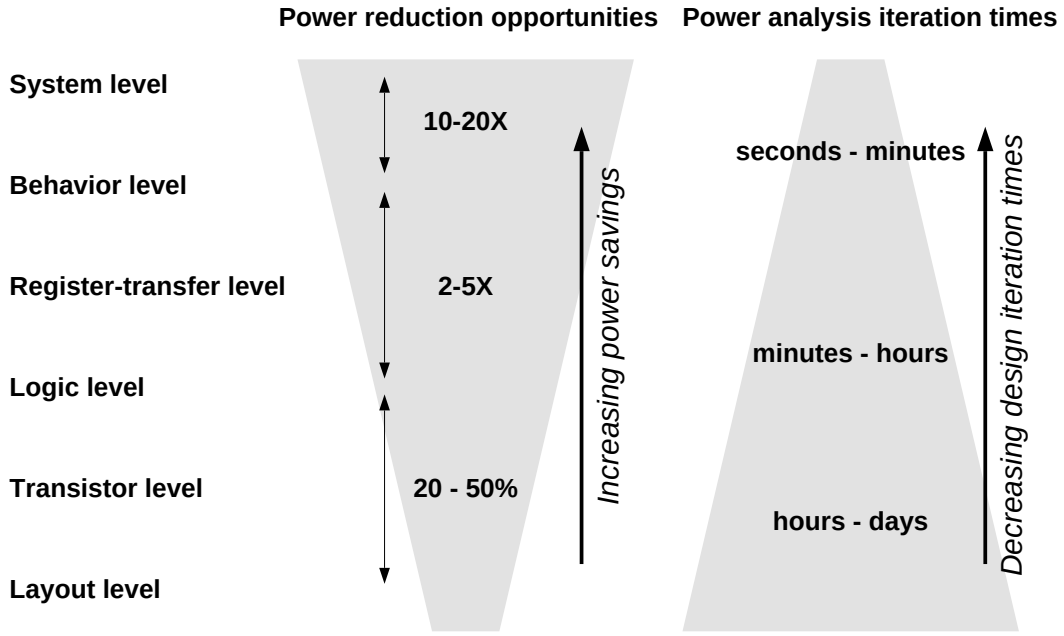

Figure 1.4: Benefits of high-level power analysis and optimization. Used with permission [10].

Power optimization has followed the same general trend as design automation (see Table 1.1 and Figure 1.4). Initially, power consumption was optimized only at the transistor and gate levels. As time progressed, researchers began considering power consumption at the register-transfer and behavioral levels. A few researchers have recently started optimizing power consumption at the system level. Although formulating concrete problem definitions at this level of the design process is more difficult than at the lower levels, the potential for power consumption reduction is greater [10]. In our work, we have taken care to consider the impact of system-level design decisions on power consumption.

1.4 Dissertation overview

In Chapter 2, we survey the work of other embedded system synthesis research groups. In Chapter 3, we provide definitions that are useful when discussing hardware-software co-synthesis and embedded system synthesis, formalize the basic hardware-software co-synthesis problem, and describe the evolutionary optimization framework we use to solve this problem. Chapter 4 describes and classifies optimization algorithms that may be used for hardware-software co-synthesis and embedded system synthesis.

In Chapters 5–9, we describe our algorithms to synthesize low-power heterogeneous distributed systems (5 and 6), embedded systems-on-chip (7), client-server embedded systems (8), and distributed embedded systems containing dynamically reconfigurable hardware (9). We present experimental results produced by the software implementations of each of these algorithms. In Chapter 10, we describe our real-time operating system power analysis framework. Chapter 11 contrasts our work with closely related work of other researchers. We summarize our contributions and present conclusions in Chapter 12. Appendix A describes TGFF, a tool that may be used to automatically

generate task sets and resource databases. Appendix B gives details about the software implementations of our hardware-software co-synthesis and embedded system synthesis algorithms.

Chapter 2

Past work

Hardware-software co-design is the concurrent design of the hardware and software portions of a computer system. Most of the work in this field targets embedded systems [11]. Hardware-software co-synthesis is the automated design of a hardware-software computer system. Hardware-software co-synthesis algorithms generally target embedded systems. A number of authors have surveyed the co-design and co-synthesis fields [12]–[18].

Although our work relies on and draws from, or advances, research in numerous fields (e.g., asynchronous design, computer graphics, evolutionary algorithms, interface synthesis, physical design, real-time operating systems, reconfigurable computing, scheduling, simulated annealing, wireless communication), it is most strongly tied to hardware-software co-design and hardware-software co-synthesis. This chapter provides a survey of previous work in these fields. Summaries of related research in other fields are deferred until the relevant sections in the following chapters of this dissertation.

2.1 Hardware-software co-design research

This section provides a survey of previous work in the field of hardware-software co-design. Although the software described in this section does not automatically synthesize embedded systems, it assists designers in determining the resources used in an embedded system, and the ways in which they are used. Although it requires a designer in the loop, some of the following software assists in the management of implementation details that are often ignored or modeled in an abstract way in hardware-software co-synthesis systems. In other words, many hardware-software co-design tools tackle less ambitious problems than hardware-software co-synthesis tools but solve them in a way that may sooner be practical for designers to use.

Some researchers have examined the manner in which constraints on an embedded system are specified. Dasarathy described a way to represent and validate timing constraints in embedded systems [19]. Gong et al. developed an algorithm that automatically refines constraint specifications after manual partitioning [20].

Others have focused on performance analysis of hardware-software systems. Calvez and Pasquier developed an event monitor to analyze the performance of an existing hardware-software system [21]. It is often useful to evaluate the performance of an embedded system that has been designed but not yet built. The simulation of hardware-software systems can be time-consuming. As a result, a number of researchers have focused on simulator acceleration. Benner et al. devised a way of rapidly simulating application-specific integrated circuits (ASICs) by using field programmable gate arrays (FPGAs) [22]. Coumeri and Thomas accelerated the simulation of hardware-software systems by running hardware and software simulators on separate processors [23]. Hines and Borriello designed a hardware-software co-simulator that changes its level of detail in order to speed up simulation when accuracy is not essential, while

maintaining accuracy when necessary [24]. Kuttner described a method of rapidly prototyping hardware-software systems by synthesizing and simulating processors [25]. Rowson and Sangiovanni-Vincentelli designed an event-based simulator that sacrifices superfluous details to improve simulation speed [26]. [26]. Thomas et al. used separate Unix processes to simulate hardware and software described in Verilog [27].

Researchers have worked on automating the communication-oriented portions of embedded system design. Castelluccia et al. developed software to automatically compile efficient protocol code from an abstract specification [28]. Freund et al. automated the assignment, bus scheduling, and protocol optimization of communication events [29]. Jirachiefpattana and Lai provided utilities to verify protocol descriptions and translate them between different languages [30]. Smith and De Micheli automated the generation of synchronous interfaces between different hardware elements described with a hardware description language [31].

Some work does not cleanly fit into any of the major co-design categories. Adé et al. developed an algorithm to compute the minimal buffer memory required for deadlock-free satisfaction of multi-rate data flow graph specifications [32]. Coelho et al. automated the process of determining whether old software will function correctly on a new, modified, version of hardware [33]. Gogniat et al. developed a parametric hardware architecture template that is sufficiently general for some special-case applications [34]. Hadjiyiannis et al. developed software that automatically generates an assembler, given a high-level description of a machine [35].

Many researchers have built hardware-software co-design infrastructures that assist a designer in partitioning an embedded system implementation between hardware and software. The ASAR project is a collection of tools and languages that assist a designer

in a number of tasks, e.g., specifying the behavior of real-time systems, generating systolic arrays, describing reactive systems, and generating pipelined signal-processing architectures [36]. Bolsens et al. developed a co-design tool that allows interactive refinement of an embedded system specification [37]. The design may be partitioned among multiple heterogeneous processors. Buck et al. built a general signal processing co-design infrastructure [38]. Chiodo et al. automated the analysis of manually partitioned embedded system specifications [39]. Chou and Borriello described transformations and partitioning of embedded system specifications written in a hierarchical state transition language [40]. Hu et al. described the hierarchical refinement of an automotive powertrain architecture [41]. They used increasingly detailed analysis during refinement, as the number of potential solutions decreased. Ismail et al. developed an interactive hardware-software partitioning tool that represents architectures with an extended finite-state machine model [42]. Kalavade and Lee built a manual partitioning, automatic analysis, system for digital signal processing applications [43]. Passerone et al. developed a virtual prototyping infrastructure in which they represented tasks with finite-state machines [44]. They used a homogeneous timing model for hardware and software to allow rapid migration of tasks between the two.

A number of companies sell tools to assist in hardware-software co-design. Most of these are co-simulation software packages for use in co-verification. Co-simulation or, more formally, hardware-software co-simulation, is the process of simulating the interacting hardware and software portions of an embedded system. Generally, the software's behavior is specified using a programming language and the hardware's behavior is specified using a hardware description language (HDL). Although it is possible to use conventional simulators running on general-purpose processors to do co-simulation, this approach is sometimes too slow to be used for co-verification. Co-verification is the process of confirming that the hardware and software portions of an embedded system

function correctly together. Co-simulation tools can be used for co-verification only if they are fast enough to allow a significant portion of an embedded system's functionality to be exercised in a reasonable amount of time. As a result, a number of companies accelerate co-simulation by using special-purpose hardware, e.g., field programmable gate array (FPGA) based hardware emulation engines.

The Virtual Component Codesign (VCC) tools, from Cadence Design Systems, Inc., allow the simulation and modeling of hardware components described in VHDL or Verilog, and algorithms described in C, C++, or a specialized signal processing description language. The N2C Design System, from CoWare, Inc., provides a co-simulation environment for hardware described in Verilog or VHDL. In addition, hardware described in their C and C++ variants can be rapidly simulated through the use of an executable specification. These hardware simulators interface with processor simulators from other vendors. The Seamless hardware-software co-verification environment, from Mentor Graphics, integrates a collection of instruction set simulators for popular embedded processors and logic simulators. Mentor Graphics also sells the Platform Express system-on-chip co-simulation environment. Virtual-CPU Pro, from Summit Design, Inc., allows processor descriptions to be written in C or C++ and rapidly simulated without the use of a logic simulator. The Eagle tools, from Synopsys, Inc., link together hardware simulators from numerous vendors. However, these products suffered from performance problems and Synopsys recently discontinued their sale.

2.2 Hardware-software co-synthesis

In this section, we survey past work in the field of hardware-software co-synthesis. Work in this field tackles the ambitious problem of automatically synthesizing embedded systems without guidance from a designer.

A substantial amount of research has focused on a version of the co-synthesis problem in which only a few processing elements (PEs) are allowed (typically one general-purpose processor and one ASIC), or the communication model is too simple to represent many real embedded systems. Ambrosio and Hu developed a hardware-software partitioning algorithm that uses very high-level estimates of the probability that an architecture can be scheduled [45]. Chatha and Vemuri developed an iterative improvement algorithm to partition task graphs between a single hardware coprocessor and a general-purpose processor [46]. Eles et al. developed simulated annealing and tabu search hardware-software partitioning algorithms [47]. Ernst et al. developed an algorithm that iteratively migrates embedded system functionality from software to hardware [48]. Gajski et al. developed an algorithm that allows a designer to manually or automatically partition an embedded system specification between different processors [49]. Gupta and De Micheli developed an iterative improvement algorithm to partition real-time embedded systems between a co-processor and a general-purpose processor [50]. Henkel and Ernst developed a dynamic granularity simulated annealing algorithm for hardware-software partitioning [51]. Kalavade and Lee designed a constructive algorithm that partitions a system specification between hardware and software by traversing a list of tasks [52]. It dynamically changes the relative weights of speed and area in the optimization criterion. Karakehayov developed an algorithm to automatically partition embedded system specifications among homogeneous distributed processors [53]. Knudsen and Madsen used dynamic programming to minimize the execution time of a single general-purpose processor, single ASIC embedded system under an area constraint or minimize the area under an execution time constraint [54]. Koroušić-Seljak and Cooling optimized task assignments with a genetic algorithm [55]. Lee and Shin solved the homogeneous task assignment problem, with consideration of communication, for homogeneous arrays or trees [56]. They transformed this problem to the minimum-cut,

maximum-flow problem and solved it in polynomial time. Liu and Wong developed an iterative improvement algorithm that integrates hardware-software partitioning and scheduling [57]. The algorithm migrates tasks from up to two general-purpose processors to an ASIC. Potkonjak and Rabaey formulated the ASIC algorithm selection problem in a fashion similar to the classical task assignment problem [58]. They used a constructive algorithm to optimize throughput or price. Saha et al. developed a genetic algorithm for hardware-software partitioning [59]. Towlsey efficiently solved the heterogeneous distributed system assignment problem for execution time minimization in the absence of hard real-time deadlines [60]–[62]. Vahid et al. explained the advantages of functional partitioning over structural partitioning [63]. These authors also described a way of functionally partitioning a system-level specification between hardware and software [64].

Recently, researchers have started to consider the heterogeneous distributed embedded system problem without tight limits on resource allocations. Axelsson compared the solutions produced by three different types of algorithms when run on a simplified version of the hardware-software co-synthesis problem: a tabu search algorithm, a simulated annealing algorithm, and a genetic algorithm [65]. See Chapter 11 for a critique of this work. Bender solved this problem with mixed integer linear programming (MILP) [66]. He used a linear weighting sum to combine execution time, processor prices, and communication resource prices. He claimed optimality. However, his approach must use a sub-optimal heuristic pre-processing stage to have any chance of solving complicated (realistic) problems in a reasonable amount of time. Dave et al. used a constructive algorithm to solve the classical multi-rate distributed system co-synthesis problem. This work was extended to target low-power embedded systems [67],

hierarchical embedded systems [68], and embedded systems containing dynamically reconfigurable processors [69]. Hsiung developed a hardware-software co-synthesis algorithm for massively parallel homogeneous software applications in which a small set of solutions is enumerated [70]. Solutions that do not satisfy the specified constraints are eliminated. Jeong et al. developed a hardware-software co-synthesis algorithm that allows the use of incrementally, dynamically reconfigurable hardware [71]. Karkowski and Corporaal allocated and partitioned an ANSI-C specification among homogeneous processors on a single chip [72]. Kuchcinski used constraint logic programming to minimize the price of an embedded system under time constraints [73]. The computational complexity of his algorithm may be arbitrarily reduced, as long as one is willing to tolerate sub-optimal solutions. We believe this is the most formal existing approach to solving a problem similar to the hardware-software co-synthesis problem that, at the same time, remains computationally tractable. Lee et al. developed an A$\star$ search algorithm in order to optimize embedded system resource allocations [74]. This algorithm uses earliest deadline first scheduling integrated with a load balancing assignment algorithm borrowed from behavioral synthesis. It does not model inter-task dependencies. Oh and Ha developed an iterative algorithm targeting the heterogeneous distributed system co-synthesis problem [75]. See Chapter 11 for a more detailed discussion of this work. Prakash and Parker developed a MILP solver for the distributed hardware-software co-synthesis problem [76]. Schwiegershausen and Pirsch developed a MILP solver for the heterogeneous distributed system synthesis problem [77]. Srinivasan and Jha developed a heuristic constructive algorithm that synthesizes fault-tolerant real-time distributed embedded systems [78]. Teich et al. applied an evolutionary algorithm to the co-synthesis problem. They repaired bad solutions instead of avoiding their creation [79]. Their algorithm optimized period and price in the absence of hard real-time constraints. See Chapter 11 for a more detailed comparison between our work and this

algorithm. Wolf developed a fast greedy iterative improvement for the classical co-synthesis problem [80]. His algorithm may be used to model communication. However, in the presence of non-zero communication times, this algorithm is no longer guaranteed to produce the minimal cost solutions that meet deadlines. Yen and Wolf also developed an iterative improvement algorithm for the hardware-software co-synthesis problem [81].

A few researchers have focused on improving the way embedded system constraints are expressed to hardware-software co-synthesis systems. Xie and Wolf described an iterative improvement hardware-software co-synthesis algorithm that allows the use of conditionals within task graphs [82]. Kordon and Kaim developed a hierarchical communicating state machine model for large distributed systems [83]. They generated code and Petri nets from this model in order to facilitate implementation and verification of the embedded system.

Some researchers have found ways of preprocessing embedded system specifications in order to allow the co-synthesis algorithm to arrive at better results in less time. Hou and Wolf described a task clustering method to speed co-synthesis and, under some circumstances, improve solution quality [84]. Knudsen and Madsen used a coarse-grained control/data flow graph to build a specification suitable for hardware-software partitioning [85]. They did transformations, selected a task granularity, and conducted communication analysis on the graph.

Others have worked on specifying and computing timing information. Dasdan, et al. explained a method for determining maximum execution rates starting from a generalized task graph specification [86]. Gupta described a sophisticated method for specifying timing constraints for embedded systems [87]. Hu and Sambandam developed a co-synthesis algorithm that treats timing behavior as a multi-valued approximation and produces multiple solutions that trade off different components of the timing behavior,

e.g., feasibility probability and price [88]. Rhee et al. developed a timing tool that computes conservative execution times of tasks on a reduced instruction set computer, taking into account pipelining and cache effects [89].

Communication between PEs, or between the hardware and software portions of an embedded system, has drawn the attention of a few authors. Benner and Ernst developed a simulated annealing co-synthesis algorithm that focuses on the communication process timing model and protocol [90]. Chou et al. synthesized the interface between the hardware and software portions of an embedded system [91]. They simulated the embedded system before, during, and after synthesis. Rhodes and Wolf developed a co-synthesis algorithm featuring a detailed communication model that takes into account real-time operating system preemption costs [92].

Others have worked on analyzing the performance and cost of implementing tasks on different processors and ASICs. Li and Malik devised an algorithm to derive the worst-case performance of software [93]. Rabaey and Guerra described a method for making rough estimates of the complexities of implementing simple digital signal processing algorithms on ASICs [94]. Xie and Wolf developed a way of automatically estimating the performance of different ASICs used as PEs in co-synthesis, and optimizing their implementations [95].

Chapter 3

Definitions

This chapter provides definitions that will be useful when describing our hardware-software co-synthesis and embedded system synthesis algorithms. In this dissertation, we use the units conventions of the International System of Units (SI) and International Electrotechnical Commission (IEC), i.e., *b* is the symbol for bits, *B* is the symbol for bytes, *k* is the symbol for 10^3, *Ki* is the symbol for 2^{10}, and *Mi* is the symbol for 2^{20}.

Section 3.1 explains the architectural decisions a hardware-software co-synthesis algorithm must make. Section 3.2 gives a taxonomy of the different classes of problems within the hardware-software co-synthesis and embedded system synthesis research areas. Sections 3.3, 3.4, and 3.5 define, and explain enhancements to, multi-rate task sets. This type of specification is used to represent problem constraints in our algorithms, and by other researchers working in hardware-software co-synthesis. Sections 3.6 and 3.7 describe basic models for the computation and communication resources used within embedded systems. This chapter should give the reader a more formal understanding of the co-synthesis problem, and our models for embedded system problem constraints and resources. The definitions presented in this chapter will later be expanded to suit different problem domains.

3.1 Hardware-software co-synthesis decisions

There are three decisions that must be made during synthesis of distributed heterogeneous systems. This section describes these decisions.

- **Allocation**: Determine the quantity of each type of resource, e.g., processing elements or communication resources, to use.
- **Assignment**: Select a resource to execute each task and communication event.
- **Scheduling**: Determine the time at which each task and communication event occurs.

In addition to making these three decisions, a hardware-software co-synthesis algorithm must evaluate embedded system performance. The costs of an architecture, e.g., price, speed, area, and power consumption, must be computed.

Each of the three decisions, listed above, influences others. Therefore, attempting to consider a decision in isolation, or without feedback from subsequent decisions, is likely to result in poor quality solutions. We have taken care to allow incremental feedback in our algorithms.

Some authors use terms that differ from ours when referring to the decisions made by embedded system synthesis algorithms. *Allocation* is sometimes used to refer to the decisions we separate into *allocation* and *assignment*. Others use *binding* or *mapping* as synonyms for what we call *assignment*.

3.2 Problem taxonomy

In this section, we define and classify embedded system synthesis problems. To solve the homogeneous system synthesis problem, one must decide upon an allocation

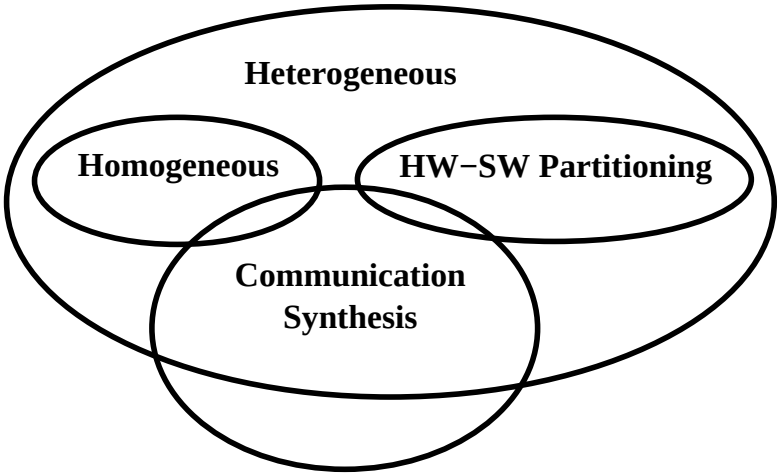

Figure 3.1: Taxonomy of hardware-software co-synthesis problems.

of identical processors, an assignment of tasks to processors, and a schedule for these tasks. The heterogeneous system synthesis problem is similar to the homogeneous system synthesis problem. However, the types of processors in the allocation may differ from each other. The hardware-software partitioning problem allows only two different processors, of different types, in the allocation. To solve the communication synthesis problem, it is necessary to determine a communication resource allocation and assign communication events to communication resources.

As shown in Figure 3.1, algorithms capable of solving the heterogeneous system synthesis problem are also generally capable of solving the homogeneous system synthesis problem and the hardware-software partitioning problem. Some communication synthesis algorithms also tackle the heterogeneous system synthesis problem, the homogeneous system synthesis problem, or the hardware-software partitioning problem, although communication synthesis algorithms need not do processing element allocation, assignment, and scheduling.

In the multi-rate system synthesis problem, cyclic specifications may contain tasks with different intervals of time between subsequent executions. Algorithms that can

handle the multi-rate problem can also handle a single rate problem, in which all tasks have the same period. Synthesis algorithms that can handle problems containing data dependencies are also capable of handling problems without data dependencies, i.e., problems containing only independent tasks.

Our algorithms solve problems that are supersets of the classical hardware-software co-synthesis problem or, more formally, the heterogeneous multi-rate distributed system synthesis problem with communication synthesis and data-dependent tasks.

3.3 Constraint specifications: Multi-rate task sets

There are a number of methods that may be used to specify the behavior of real-time embedded systems. Many of these representations constrain the allocations and assignments that will ultimately be used for implementation, and are therefore not suitable for use in a synthesis algorithm that needs the freedom to generate its own allocations, assignments, and schedules. State machine based representations can naturally represent the behavior of reactive embedded systems that do not execute complicated algorithms. In addition, they're suitable for use in synthesis because they do not constrain implementation. However, they have historically had the problem of state explosion, e.g., in order for a finite state machine (FSM) to represent two concurrent n-state subsystems, n^2 states are required. Some FSM variants have been developed to reduce this problem [96], [97]. However, timing analysis is often difficult for these variants. Many other representations exist, each of which has its own advantages and disadvantages [98]–[101]. In our system synthesis work, we have modeled embedded system behaviors and timing constraints with muti-rate task sets. This is a natural model for data flow and signal processing behaviors. It is amenable to detailed timing analysis.

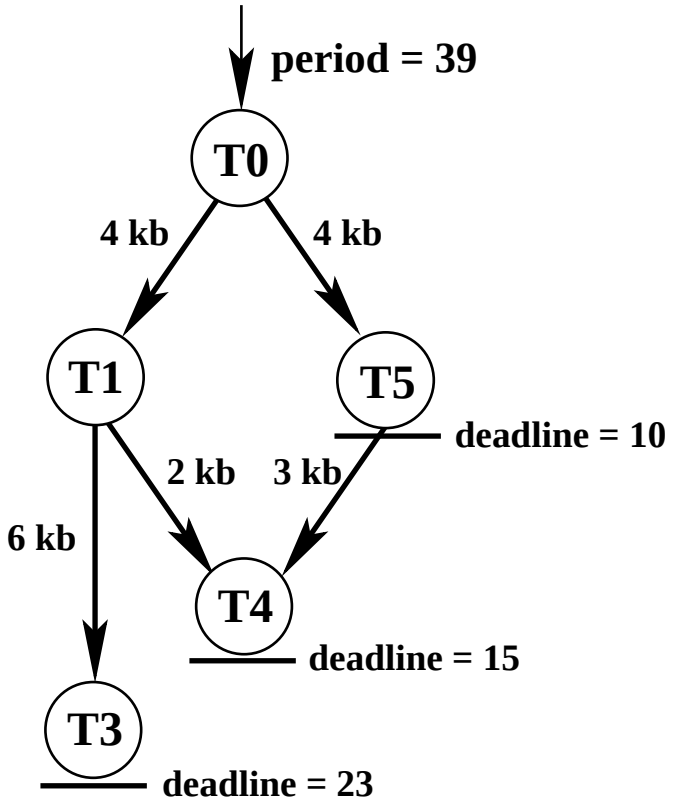

Figure 3.2: Task graph.

This is especially useful for synthesis of hard real-time embedded systems: it allows a synthesis algorithm to guarantee that hard real-time constraints will be met.

Multi-rate task sets may be used to specify some of the requirements a designer places upon an embedded system. A task graph, as shown in Figure 3.2, is a directed acyclic graph in which each node is associated with a task and each edge is associated with a scalar indicating the amount of data that must be transferred between the two connected tasks. Each task may only begin executing after all of its data dependencies have been satisfied. Thus, in Figure 3.2, task T4 may only begin execution after tasks T1 and T5 have each completed execution and transferred two and three kilobits of data, respectively, to task T4.

Embedded system synthesis research generally assumes coarse-grained tasks, i.e., each task is complicated enough to require numerous microprocessor instructions. The *period* of a task graph is the amount of time between the earliest start times of its consecutive executions. A node with no outgoing edges is called a *sink* node. A *deadline*, the time by which the task associated with the node must complete its execution, may exist for any node. The deadline of a task graph is the maximum of all the deadlines specified in it. Our task graph model supports hard and soft deadlines. Hard deadlines may not be violated. Soft deadlines need not be met, but violating them is undesirable. If a task should finish executing as soon as possible, it may be given a soft deadline at time zero. A multi-rate task set contains one or more task graphs, each of which may have a different period. The *hyperperiod* of a multi-rate task set is the least common multiple of the periods of the task graphs within the task set.

3.4 Multi-rate task sets for real-time systems

In this section, we consider the advantages and disadvantages of using representations that are closely related to multi-rate task sets for constraint specification in real-time distributed embedded systems.

Some researchers are dissatisfied with multi-rate tasks sets for specifying constraints in real-time distributed embedded systems [102]. It is sometimes claimed that multi-rate task sets are too simple and need to be more expressive in order to allow embedded systems to be realistically modeled. Although the claim that they are simple is correct, researchers sometimes propose adopting extensions from constraint representations that are used to represent problems without hard real-time constraints. However, many of these extensions have significant drawbacks when applied to hard real-time systems. In

this section, these drawbacks are described. In addition, we note one extension that has value when used to specify real-time systems.

Some constraint specification formats are more general than task sets [39], [103], [104]. Such formats might better represent constraints without hard real-time deadlines. However, many of these representations add nothing to a synthesis system targeting hard real-time problems. In other words, when the multi-rate task set representation is extended to handle many apparently interesting generalizations, the extensions must be bounded to allow synthesis of embedded systems that are guaranteed to meet hard real-time constraints. This results in a representation that can be losslessly converted back to a classical multi-rate task set.

If one starts from a multi-rate task set and introduces unbounded loops between start and deadline nodes, it becomes impossible to guarantee that down-stream deadlines in the graphs are met. If one constrains the specification by requiring a bound on the number of times a loop may be executed, then it becomes possible to guarantee that deadlines are met. However, one must guarantee that sufficient time is available, in the schedule, to host the worst-cast number of executions. This leaves one with a representation that is more complicated than a multi-rate task set but provides no advantage. One could achieve the same goal by unrolling loops in a pre-pass transformation into conventional task sets. This transformation would increase the time required for explicit scheduling by approximately the same amount as using a bounded loop representation.

If one starts from multi-rate task sets, introducing a representation for fine-grained parallelism internal to tasks may seem useful in order to allow some tasks to be dynamically split among numerous processors. However, one could pre-process such a graph, splitting nodes conservatively based on the properties of the resources available and thereby convert it, in a mildly lossy way, back into a conventional task graph.

If one starts from multi-rate task sets and introduces communication between nodes in task graphs with different periods, one can losslessly transform such a modified task set into a conventional task set by unrolling the task graphs into the hyperperiod and adding the new communication events. There is another straightforward change required involving start node offsets. For scheduling algorithms that unroll to the multi-rate hyperperiod [105], the resulting increase in scheduling time would not be dramatic.

If one starts from multi-rate task sets and introduces tasks that may start executing before their incoming data have arrived, this format can be transformed into a conventional task set by splitting each such task into a zero-duration parent task that accepts the pre-start arc, and a normal-duration child task that must be assigned to the same processing element as the parent task and that accepts the post-start arc.

If one starts from multi-rate task sets and introduces internal conditionals, it is necessary to reserve sufficient time in the schedule to ensure that the more time-consuming task, and down-stream tasks, can execute. We know of no lossless transformation from a task set with conditionals into a classical task set. Unlike a number of other task set enhancements, conditionals make task sets more expressive, even for hard real-time systems. Eles et al. describe scheduling of conditional task graphs in embedded system synthesis [103]. Their specifications support conditionals. Xie and Wolf developed an embedded system synthesis algorithm that allows task sets to contain conditionals [82]. Their algorithm detects mutually exclusive tasks instead of enumerating all condition combinations. Ziegenbein et al. described a way of representing correlations between the execution of tasks and a small set of execution modes [104]. Although their approach may allow some types of embedded systems to be represented more directly than would be possible with conditionals, it does not substantially reduce the complexity of conducting real-time system synthesis; constraint specifications to which this approach

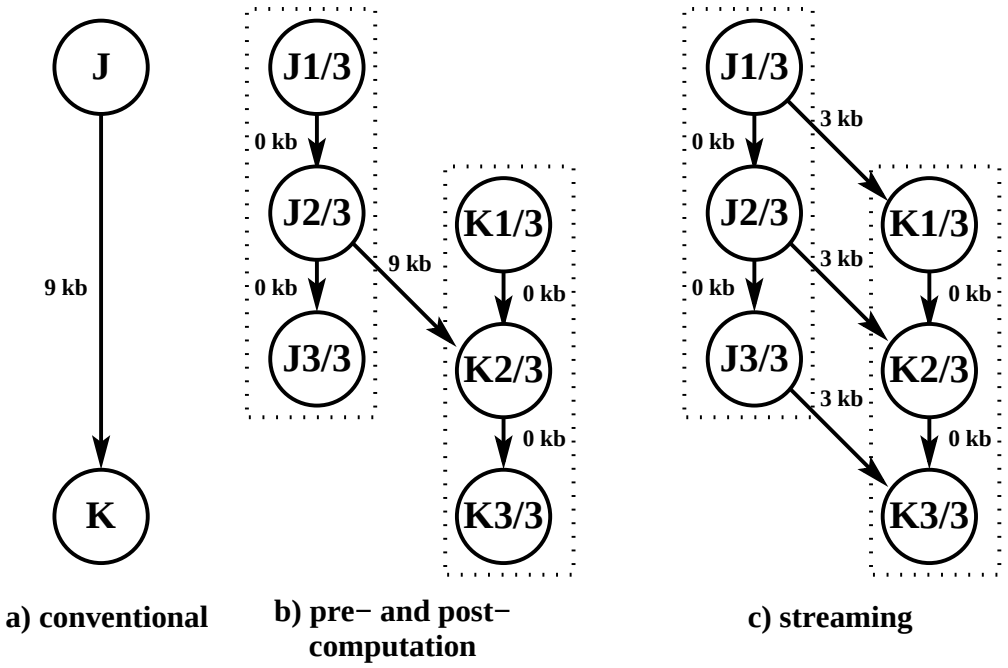

Figure 3.3: (a) conventional model and task assignment grouping to model (b) pre-computation and post-computation, as well as (c) streaming data communication.

may practically be applied cannot contain a number of condition combinations that is exponential in the number of graph nodes.

3.5 Modification of the task set model for pre-computation and streaming

Our algorithms support a task graph model that allows the assignments of multiple tasks within the same task graph to be tied to the same processing element. This makes it possible to model pre-computation and post-computation [76], as well as course-grained streaming data communication. Figure 3.3a illustrates a conventional communication event between task J and task K. Task J completes execution, 9 kb of data are transferred from task J to task K, then task K executes. In Figure 3.3b, tasks J and K are

each split into three sub-tasks. All a task's sub-tasks have their assignment tied to the same processing element, as indicated by the dashed box containing them. This enables pre-computation and post-computation. For example, sub-task K1/3 can execute before sub-task K2/3's input data have arrived (pre-computation), and sub-task J2/3 can transmit output data before sub-task J3/3 has completed execution (post-computation). Figure 3.3c shows a similar use of task assignment tying to model streaming data communication. In this example, instead of first completing task J's execution before conducting one 9 kb communication event, the communication is split into three 3 kb chunks that occur during task J's execution. After each of task J's subtasks (J1/3, J2/3, and J3/3) completes execution, a 3 kb communication event may occur. Note that this streaming data model can easily be made more fine-grained by breaking a task into more subtasks. However, making the streaming data representation more fine-grained will result in an increase in the CPU time required for scheduling.

3.6 Processing elements (PEs)

A PE executes tasks. Our hardware-software co-synthesis and embedded system synthesis algorithms model many types of PEs: general-purpose processors, digital signal processors (DSPs), application-specific integrated circuits (ASICs), multiprocessor integrated circuits (ICs), intellectual property (IP) cores, and FPGAs. However, in this definition, we will describe only the general model for general-purpose processors, DSPs, and ASICs. The features peculiar to multiprocessor ICs, IP cores, and FPGAs will be described in the chapters devoted to the algorithms that use them: Chapters 5, 7, and 9.

A solution may contain multiple instances of the same type of PE. Our algorithms require databases that describe the relationships between tasks and PEs. Characterizing

a PEs in this manner requires that the designer know the input vectors that elicit the worst-case execution time for each task-PE pair. Alternatively, one may use worst-case performance analysis tools to determine an upper-bound on execution time, without requiring a specific input vector [89], [106]. In addition, the average power consumption for each task-PE pair must be known or estimated. The power consumption of general-purpose and application-specific processors can be estimated by using models, simulation, and explicit analysis [10], [107]–[111].

The following information establishes the relationships between tasks and PEs:

- A two-dimensional array indicating the worst-case execution time of each task on each PE.
- A two-dimensional array indicating the average power consumption of each task on each PE.

In addition to these arrays, each PE has a price, I/O energy per communicated bit, and idle power consumption. PEs may be buffered, in which case they can communicate and compute at the same time, or unbuffered, in which case communication and computation may not overlap in time. In the case of buffered communication it is, of course, still necessary for a task's incoming data to arrive before it can begin execution.

3.7 Communication resources

Communication resources have the following attributes: controller price, bits per packet, transmission time per bit, power consumption during operation, number of contacts, and idle power consumption. The number of contacts a communication resource supports is the number of integrated circuits (ICs) it can connect, i.e., a communication

resource with two contacts is a point-to-point communication resource and a communication resource with more than two contacts is a bus. The method of deriving these attributes depends on the type of architecture being synthesized. For example, in a distributed system, the parameters of a communication bus can be determined from the bus specifications, as well as the controller datasheets. If one is targeting a system-on-chip, the behavior of the on-chip busses can be derived from the fabrication process parameters and floorplan using a wire delay and power consumption model, as described in Section 7.7. Each task graph edge must be assigned to a communication resource. The worst-case communication time and average power consumption of an edge are linearly dependent on the integer number of packets transferred via its communication resource. There may be more than one communication resource connected to a PE instance. In previous distributed computing work, it is commonly assumed that communication between tasks that are assigned to the same IC consume an insignificant amount of time and power. We also make this assumption in our distributed system synthesis algorithms. However, we use a detailed wire delay model in our system-on-chip (SOC) synthesis algorithm. If an architecture contains two communicating tasks that execute on separate ICs, the architecture is invalid if there are no communication resources connecting the ICs.

Chapter 4

Optimization algorithms

This chapter presents preliminary concepts that will later be used in the description of our evolutionary optimization framework. Section 4.1 gives a brief survey of methods for solving NP-hard problems. Sections 4.2 and 4.3 describe two problem-solving heuristics that are closely related to the optimization infrastructure used in our evolutionary optimization algorithms. This class of algorithms, parallel recombinative simulated annealing algorithms, are described in Section 4.4. In Section 4.5 we explain some of the challenges of solving multiobjective problems, and explain how parallel recombinative simulated annealing algorithms can be adapted to simultaneously optimize multiple costs.

4.1 Solving NP-hard problems

This section gives an introduction to the classes of algorithms that may be used to solve the hardware-software co-synthesis problem and system synthesis problem.

Hardware-software co-synthesis, and embedded system synthesis, contain multiple NP-complete, and therefore NP-hard, problems within them. Worse yet, these problems are interdependent. Allocation/assignment and scheduling are each known to be NP-complete for distributed systems [112]. Any instance of an NP-complete problem can

be converted to an instance of any other NP-complete problem in an amount of time that is, at worst, polynomially dependent on the size of the instance. For decades, clever people have searched for solutions to NP-complete problems that requite time polynomially dependent on the size of the problem instance. However, nobody has ever published an algorithm that optimally solves an NP-complete problem in, at worst, polynomial time. As a result, most algorithm designers operate under the conjecture that finding a guaranteed-optimal solution to an NP-complete problem requires an algorithm that may take an amount of time exponentially dependent on the size of the problem instance. If one can be sure to encounter only very small problem instances, it is practical to use potentially exponential-time algorithms, e.g., A* [113], dynamic programming [114], mixed integer linear programming [115], branch-and-bound [116], or one of the trivial exhaustive searches. However, if one might encounter large problem instances, the above conjecture implies that one must settle for an algorithm producing solutions that are not guaranteed to be optimal.

Approximation algorithms may be used to produce solutions to NP-complete problems in polynomial time [114], [117]. Although the solutions produced may sometimes be optimal, optimality is not guaranteed. However, approximation algorithms are guaranteed to produce solutions with bounded deviations from optimal cost. Devising approximation algorithms for NP-complete problems remains an area of vigorous research; there are still numerous problems for which approximation algorithms have not yet been developed and analyzed. When attempting to solve such problems, researchers often resort to heuristic algorithms. Good heuristic algorithms can be empirically demonstrated to produce high-quality solutions to important problems most of the time, although no formal proof bounding the deviation of solution costs from optimality may be known. Many effective heuristics fall within one of seven algorithm classes:

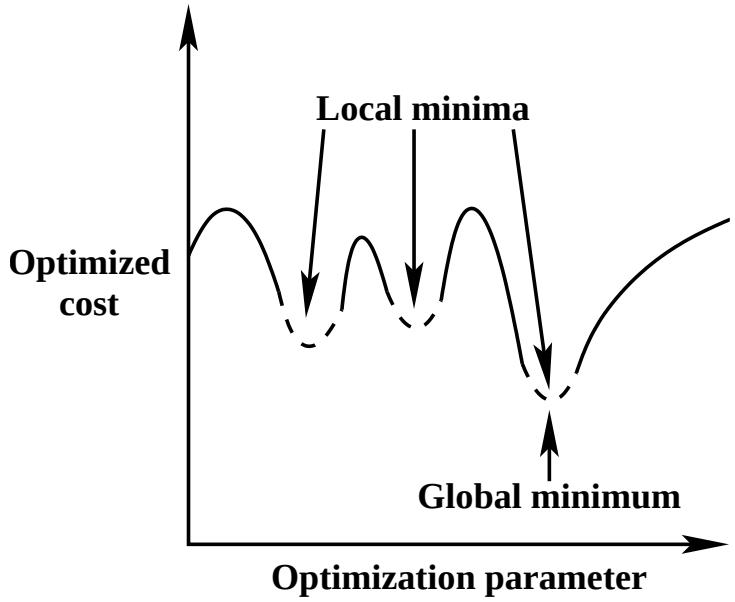

Figure 4.1: Local and global minima.

constructive, greedy iterative improvement, variable depth iterative improvement, tabu search, simulated annealing, genetic, and parallel recombinative simulated annealing.

An *optimized cost* is a value optimization algorithms attempt to minimize, e.g., price. For any problem instance, the optimized cost is a function of the *optimization parameters*. An optimization algorithm attempts to find a value of the optimization parameters such that the optimized cost is minimized. The *neighborhood* of a solution to an optimization problem is the set of other solutions that can be reached in one discrete step of the optimization algorithm. A *local minimum* is a solution for which no other solution within that solution's neighborhood has a lower optimized cost. This implies that the local minima in a solution space are based on the optimization algorithm-dependent definition of neighborhood. The *global minima* are those solutions for which the optimized cost is the lowest value possible for the problem instance. In general, optimization algorithms attempt to locate global minima and avoid becoming trapped in local minima. This concept may be illustrated with an example. The curved line in Figure 4.1 shows

the relationship between a problem's optimization parameter and optimized cost. The dashed portions of the line are the neighborhoods surrounding the solutions at the local and global minima. As indicated, although the two local minima on the left are not global minima, there are no lower-cost solutions within the neighborhoods of these solutions. An optimization algorithm with the indicated neighborhoods could become trapped in one of the sub-optimal local minima.

Constructive algorithms generate finished solutions that are not later improved upon. Generally, constructive algorithms are fast. However, they usually follow a fixed set of rules that result in a tightly constrained exploration of the solution space. The qualities of solutions produced by a constructive algorithm are strongly dependent on the amount of problem-specific knowledge built into the algorithm. Greedy constructive algorithms are prone to becoming trapped in local minima.

Iterative improvement algorithms generate a solution and make changes to it in an attempt to improve its quality. We do not require that an algorithm evaluate and rank all possible local moves during each iteration in order classify it as an iterative improvement algorithm. Formally, an iterative improvement algorithm must contain a constructive algorithm within it in order to generate the initial solution. However, most problem-specific knowledge in an iterative improvement algorithm is usually incorporated in the improvement portion of the algorithm, instead of the constructive portion.

Greedy iterative improvement algorithms repeatedly make incremental changes to a solution. If a change results in an improvement, it is accepted. If it results in a degradation, it is rejected. As soon as the solution reaches a position from which no incremental change results in an improvement, the algorithm halts. As a result, this type of algorithm is liable to become trapped in a local minimum.

Variable-depth search algorithms are a type of iterative improvement that is capable of backing out of local minima of arbitrary, but bounded, depth [117]. Of course, run

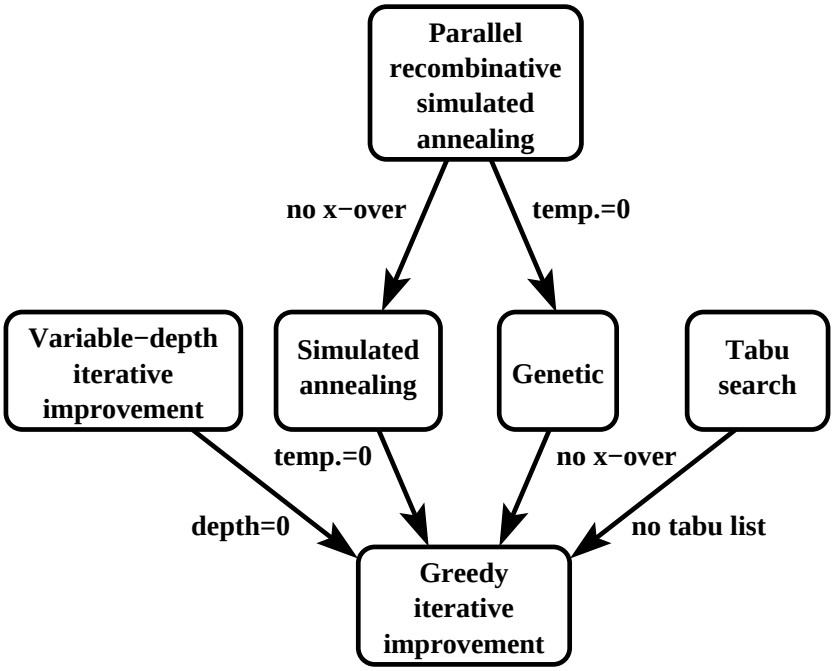

Figure 4.2: Iterative improvement taxonomy.

time depends on the backtracking depth selected. A variable-depth search algorithm with an infinite backtracking depth may be optimal, assuming its repertoire of changes is sufficient to explore the entire solution space. A variable-depth search algorithm with a backtracking depth of zero is equivalent to a greedy iterative improvement algorithm. Therefore, variable-depth search algorithms are a superset of greedy iterative improvement algorithms, as shown in Figure 4.2.

Tabu search is a form of iterative improvement in which some moves are dynamically prohibited [118]. A tabu list is maintained. This list prevents recently visited solutions from being revisited. A tabu search algorithm with a tabu list of length zero is equivalent to a greedy iterative improvement algorithm. Therefore, tabu search algorithms are a superset of greedy iterative improvement algorithms, as shown in Figure 4.2.

4.2 Simulated annealing

In an iterative improvement algorithm, we define *greediness* as the probability that a cost-decreasing change to a solution will be preferred instead of a cost-increasing change. Simulated annealing algorithms are iterative improvement algorithms in which greediness increases during the run of the algorithm [119]. Simulated annealing algorithms conduct *Boltzmann trials* between solutions before and after modifications; changes are accepted with probability

$$\frac{1}{1 + e^{(N-P)/T}}$$

where T is the global temperature, P is the cost of the old solution and N is the cost of the modified solution.

Boltzmann trials are more easily described with the use of an illustration. Note that, in order to start a simulated annealing algorithm with approximately equal probabilities of selecting changes that decrease or increase cost, it is necessary to initially have a global temperature near infinity. In order to smoothly illustrate the behavior of a Boltzmann trial as the global temperature varies from infinity to zero, we introduce

$$U = 1 - \frac{1}{T + 1}$$

U gradually varies from a value near one to zero during the run of a simulated annealing algorithm, causing T to gradually vary from a value near infinity to zero.

When a simulated annealing algorithm begins execution, the global temperature is set to a high value, i.e., U is approximately one. As a result, changes that increase the cost of a solution are selected with approximately the same probability as changes that decrease its cost, as can be seen in Figure 4.3. When U is near one, the acceptance probability is approximately 0.5, independent of the difference between the cost of the current solution, P, and the cost of the modified solution, N. In this explanation, we

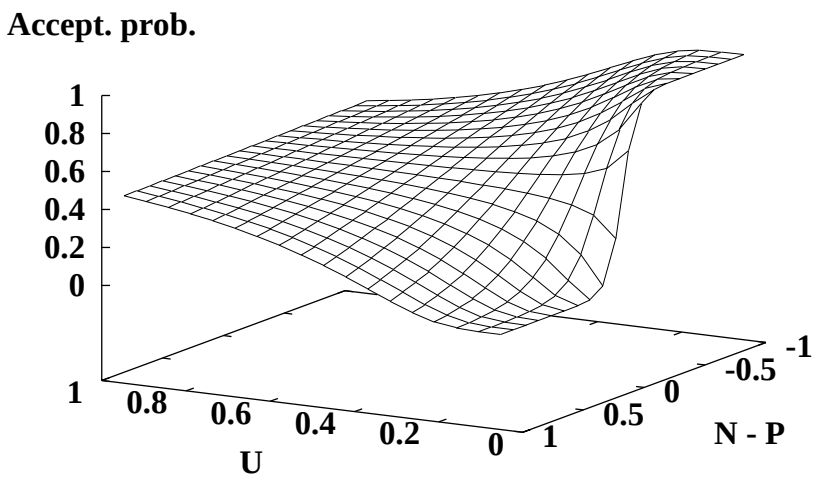

Figure 4.3: Boltzmann trial acceptance probability.

assume that $P, N \in [0, 1]$. At this stage, the algorithm is capable of easily escaping local minima. However, it is not particularly effective at reducing the value of the optimized cost. As time elapses, the global temperature is reduced, gradually changing the behavior of Boltzmann trials until changes that decrease the cost of a solution are always selected and changes that increase the cost are always rejected. In other words, as the global temperature approaches zero, a simulated annealing algorithm degrades to a greedy iterative improvement algorithm. This greedy behavior is depicted by the portion of Figure 4.3 at which U is zero; an improvement in solution quality will always be accepted and a degradation in solution quality will always be rejected. Greedy iterative improvement is a special case of simulated annealing. Therefore, simulated annealing is a superset of greedy iterative improvement, as shown in Figure 4.2.

4.3 Genetic algorithms

Genetic algorithms maintain a pool of solutions that evolve in parallel over time. During each *generation*, genetic operators that allow randomized local changes and the exchange of information between solutions are applied to the solutions in the current pool in order to improve them. The lowest quality solutions are then removed from the pool [120]. Genetic algorithms have the ability to escape local minima and communicate information among solutions. A genetic algorithm with a solution pool containing only a single solution is equivalent to a greedy iterative improvement algorithm. Therefore, genetic algorithms are a superset of greedy iterative improvement algorithms, as shown in Figure 4.2.

In a conventional genetic algorithm, a solution is represented by a one-dimensional array, or *string*, of values. All changes to strings are made with two operators: mutation and crossover. *Mutation* randomly changes part of a solution's string. *Crossover* swaps portions of different solutions. Two different types of crossover are commonly used: one-cut and two-cut. In one-cut crossover, a pair of strings is selected and the portions to the left of a randomly selected offset into the strings are swapped. In two-cut crossover, a pair of strings is selected and the portions between two randomly selected offsets into the strings are swapped. Figure 4.4 shows an example of two-cut string crossover. In this illustration, crossover occurs between strings L and M. Two-cuts are made and the portions of L and M between these cuts are swapped, producing strings L+ and M+. Crossover is the operator that gives genetic algorithms their strength; it allows different solutions to share information with each other.

An *independent sub-solution* is a portion of a solution for which the optimal configuration is not influenced by the configurations of other sub-solutions. It is important that the string encoding used to represent a solution maintain *locality*, i.e., it is important for data representing each independent sub-solution to be located contiguously, instead of

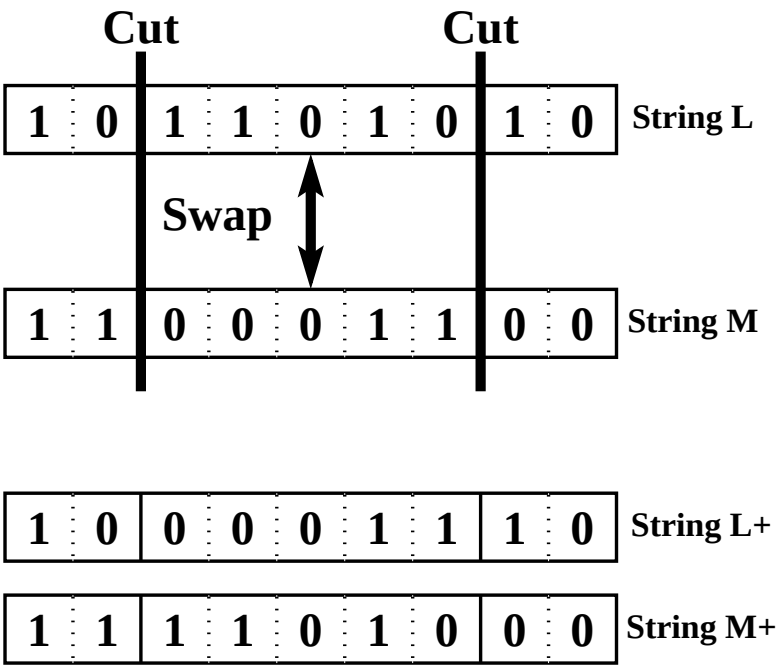

Figure 4.4: Crossover.

being interleaved with data representing other sub-solutions. The reason for this requirement is most easily illustrated with an example. In Figure 4.5, A, B, and C represent variables associated with different independent sub-solutions. When crossover occurs, strings are cut into sections. Independent sub-solutions are interleaved in the bottom two strings. As a result, data representing independent sub-solutions are likely to be split into separate solutions when crossover occurs. However, in the top two strings, independent sub-solution representations are contiguous. As a result, data describing most independent sub-solutions won't be split between different strings when crossover occurs. If a string contains a good independent sub-solution, it is important for the encoding of that sub-solution to remain intact. The practical effect of using a string encoding and crossover method that effectively maintains locality is a genetic algorithm

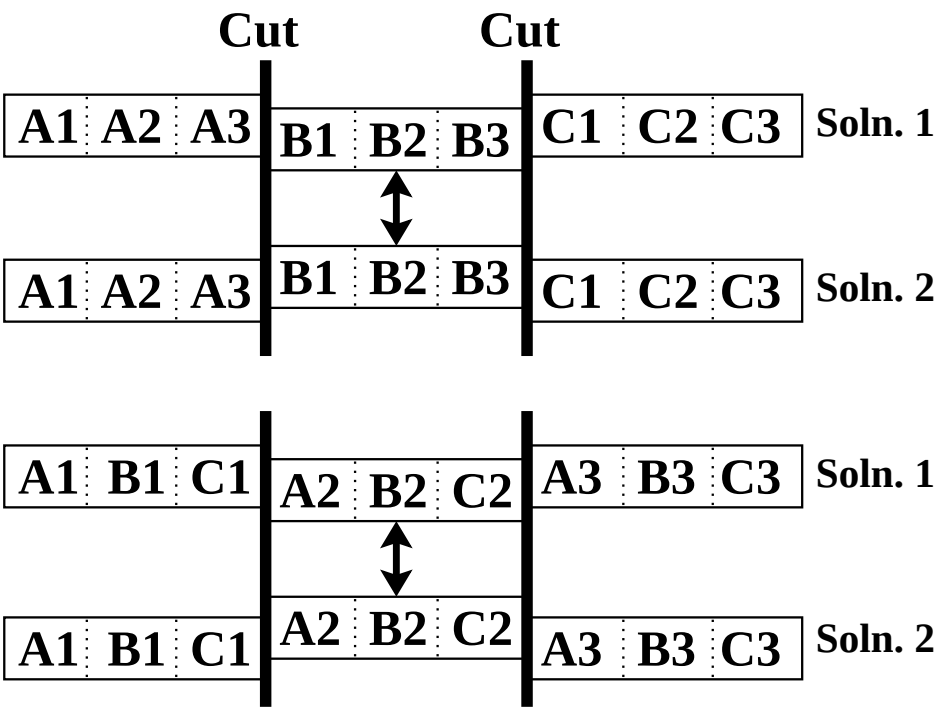

Figure 4.5: String locality.

that takes advantage of *implicit parallelism*, i.e., n function evaluations implicitly examine approximately n^3 string configurations [120], [121], where n is the size of the solution pool.

Genetic algorithms are, in general, difficult to design, implement, and analyze. They should be considered for solving optimization problems that are multiobjective or difficult to analyze and decompose, e.g., problems composed of multiple inter-dependant NP-hard problems, each of which has huge solution spaces.

4.4 Parallel recombinative simulated annealing (PRSA)

PRSA algorithms have some of the best attributes of both genetic algorithms and simulated annealing algorithms [122]. This class of algorithms is best understood as genetic algorithms that use Boltzmann trials between modified and existing solutions, in order to select the solutions that will exist in the next generation. The greediness of a PRSA algorithm starts low and increases during an optimization run, allowing it to escape local minima in a fashion similar to simulated annealing. A PRSA algorithm in which the global temperature is always zero is substantially equivalent to a genetic algorithm. A PRSA algorithm, in which there is only one solution, is a simulated annealing algorithm. Therefore, PRSA algorithms are a superset of genetic algorithms and simulated annealing algorithms, as shown in Figure 4.2.

4.5 Multiobjective optimization

This section describes a method of solving problems for which the solutions have multiple conflicting costs.

Real-world hardware-software co-synthesis problems are inherently multiobjective. Embedded systems have numerous costs and improving one cost often results in the degradation of others. In the past, the few co-synthesis algorithms that attempted to optimize multiple costs either did so in an informal way, replaced all but one cost with constraints, or combined the multiple costs into a single cost with a weighting sum and optimized this sum. For this method to be successful, the weighting vector used must be appropriate for the problem instance as well as the designer's desired solution. Unfortunately, the hardware-software co-synthesis problem is too complicated for an instance's

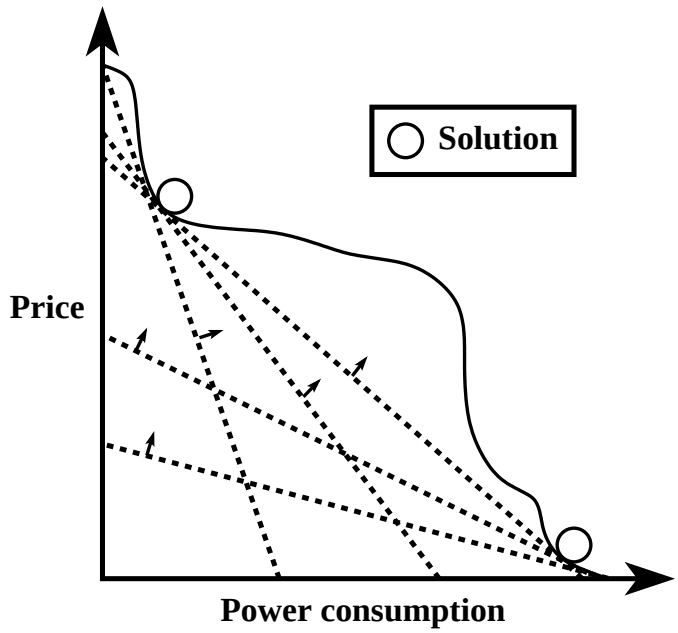

Figure 4.6: Linear weighted sum cost function.

best weighting vector to be known without first exploring that instance's *Pareto-optimal* set of solutions, i.e., those solutions that can only be improved in one area by being degraded in another. It is impossible, however, to explore the Pareto-optimal set of solutions if a weighting vector has been used to combine all costs into a single value.

Assume that a designer is trying to optimize two conflicting features of an embedded system: price and power consumption. If the designer uses a conventional optimization algorithm that can only deal with one cost function, it is necessary to combine the two costs into one value. In Figure 4.6, the curved line is the Pareto-optimal curve. It approximates the Pareto-optimal set of solutions, i.e., those solutions that can only have one cost improved at the expense of degrading another cost. Each of the vectors is associated with a linear weighting sum a designer might potentially select. These vectors point in the direction of ascent for the gradient defined by the associated weighted sum of costs. The dotted lines perpendicular to these vectors represent sets of points with

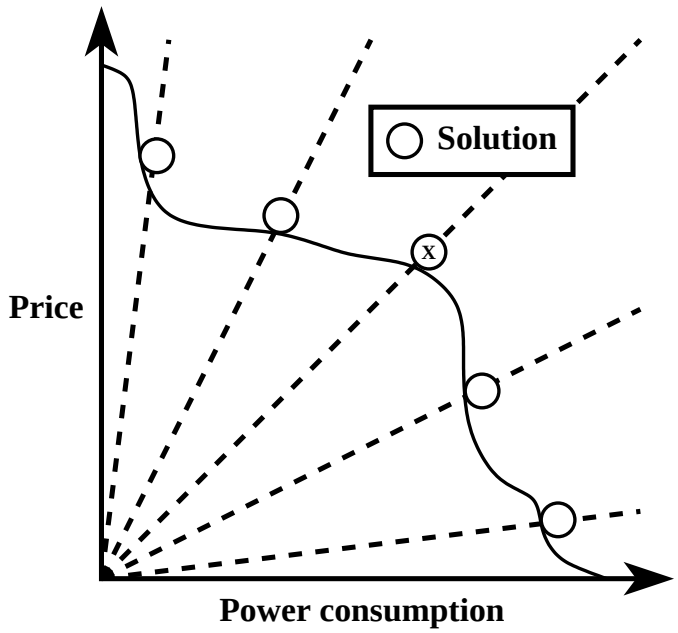

Figure 4.7: Non-linear directed cost function.

the same linear weighting sum costs. As illustrated by the figure, for this Pareto-optimal set of solutions, a perfect optimization algorithm is only capable of generating solutions at or near the two solutions shown. The minimum cost solutions will lie near the first intersection of a line (hyperplane) that is swept outward from the origin, and the Pareto-optimal curve. Using a linear weighting sum forces numerous, and potentially preferable, solutions to be neglected.

Alternatively, a designer may use a non-linear directed cost function to find solutions near arbitrary intersections between lines that intersect the origin and Pareto-optimal curve. In Figure 4.7, each line represents a trough in the following 2-D cost function:

$$\text{cost}(x, y, w) = \max\left(x \cdot w, y \cdot 1 - w\right)$$

where w is a weight with a value ranging from 0 to 1. In the general multidimensional case, this function is defined as:

$$\text{cost}\left(\vec{v}, \vec{w}\right) = \max_{i=1}^{n} \left(v_i \cdot w_i\right)$$

where n is the number of dimensions, v is an arbitrary n-dimensional vector, and w is an n-dimensional weighting vector. This cost function generally decreases as component costs decrease. The lowest-cost legitimate solution for a given cost function will lie, approximately, at the intersection of the line approximating the Pareto-optimal set and the inverse of the cost function's weighting vector.

Although an apparently reasonable weighting array can be selected, the designer has no way of knowing the shape of the Pareto-optimal curve ahead of time. As a result, a designer might easily select a weighting vector leading to the production of the solution marked with an X. However, this is probably not the weighting vector a designer would have selected if the shape of the Pareto-optimal curve were known. For a small power consumption penalty, the price of the system can be significantly decreased; for a small price penalty, the power consumption of the system can be significantly decreased. Unfortunately, the designer will never know about those valid solutions because the weighting vector prevents this portion of the Pareto-optimal curve from being explored. Although the limitations of single objective optimization can be seen from this simple example, the problem of selecting an appropriate weighting array becomes even more severe as the number of costs in a system increases.

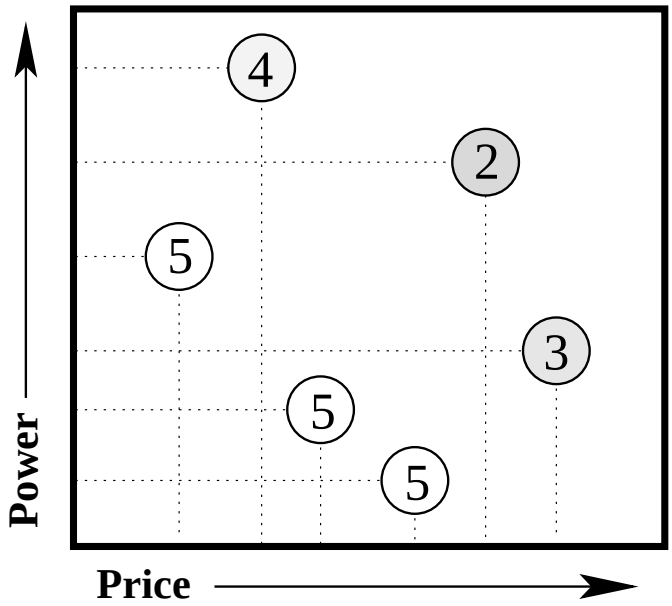

Figure 4.8: Pareto-rank.

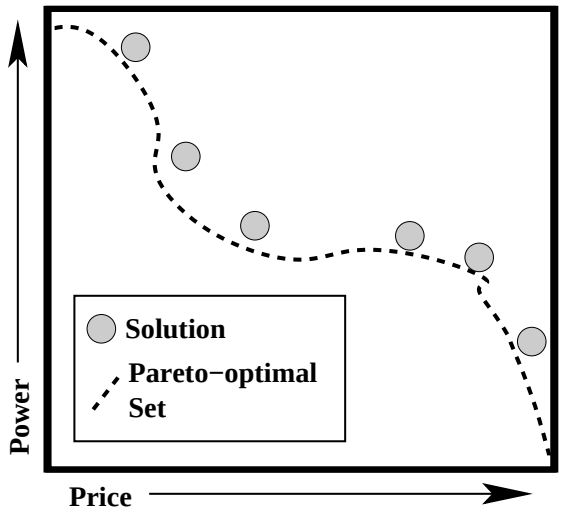

Figure 4.9: Pareto-rank based multiobjective optimization.

Let us digress, for a moment, to provide a definition. A solution *dominates* another if all of its features are better. A solution's *Pareto-rank* is the number of other solutions, in the solution pool, that do not dominate it. Given a solution pool of size p, calculating Pareto-rank is an $\mathcal{O}\left(p^2\right)$ operation; each solution must be compared with every other solution. In Figure 4.8, each circle represents a solution. Each solution's price and power consumption are indicated by the position of its circle in the graph. The number in each circle indicates the Pareto-rank of the associated solution.

At the end of a multiobjective optimization algorithm's run, the designer is presented with one or more non-inferior solutions (see Figure 4.9), i.e., those solutions that are not dominated by any other solutions. Although the non-inferior solutions are not guaranteed to be the Pareto-optimal set of solutions for the problem instance (the heterogeneous distributed system co-synthesis problem contains multiple interdependent NP-complete problems), they do form an upper bound on the Pareto-optimal set, giving the designer insight into the shape of the problem's Pareto-optimal solution set. The tradeoffs available between solution costs in these non-inferior solutions are made clear.

In order to carry out Pareto-rank multiobjective optimization, it is necessary for multiple solutions to be compared with each other. Algorithms in which a pool of solutions exists concurrently, e.g., genetic algorithms and PRSA algorithms, excel at Pareto-rank optimization [123].

Chapter 5

Synthesis of Low-Power Heterogeneous Distributed Systems

In this chapter, we present MOGAC, an adaptive multiobjective genetic algorithm for hardware-software co-synthesis of distributed embedded systems. This algorithm, and its associated software implementation, solves the hardware-software co-synthesis problem or, more formally, the heterogeneous multi-rate distributed system synthesis problem with communication synthesis and data-dependent tasks. Our solution to this classical problem provides a basis for the embedded system synthesis algorithms described in the remaining chapters of this dissertation. MOGAC partitions and schedules embedded system specifications consisting of multiple periodic task graphs. Its adaptive multiobjective genetic algorithm is designed to avoid becoming trapped in local minima. Price and power consumption are optimized while hard real-time constraints are met. MOGAC places no limit on the number of hardware or software processing elements in the architectures it synthesizes. Our general model for bus and point-to-point communication resources allows multiple types of communication resources to be used in an architecture. Heuristics are used to tackle multi-rate systems, as well as systems containing task graphs with hyperperiods that are large relative to their periods. The application of a multiobjective optimization strategy allows a single co-synthesis run

to produce multiple designs that trade off different architectural features. Experimental results indicate that MOGAC has advantages over previous work in terms of solution quality and running time.

This chapter relies on definitions provided in Chapters 1, 2, and 3. In Section 5.1, we state the requirement that must be met by an algorithm that synthesizes low-price, low-power, heterogeneous, distributed systems. Section 5.2 describes the model MOGAC uses for integrated circuits containing multiple concurrently executing processing elements. Section 5.3 describes MOGAC's method of representing solutions. In Sections 5.4, 5.5, 5.6, and 5.7, we describe the optimization algorithm used by MOGAC. Section 5.7 explains our method of evaluating solution quality and ensuring that timing constraints are met. Section 5.8 describes solution ranking and reproduction. We present experimental results and conclusions in Sections 5.9 and 10.6.

5.1 Requirements for the optimization algorithm

Given constraint specifications in the form of a multi-rate task set (defined in Section 3.3), and a resource database describing the types of processing elements (PEs) and communication resources available, it is our goal to generate embedded system architectures consisting of allocations, assignments, and schedules. These architectures must meet hard real-time constraints. In addition, they should have low prices and power consumptions. This problem contains multiple NP-complete problems within it. The problem instances may be large. Operating under the conjecture that solving NP-complete problems optimally requires, in the worst case, an amount of time exponential in the size of the problem instance, we were forced to resort to a potentially sub-optimal algorithm. We needed an optimization infrastructure with a number of attributes.

- **Scalable**: It should be easy to trade off optimization time for solution quality.

- **Multiobjective**: It should excel at simultaneously optimizing different costs.
- **Robust**: It should resist becoming trapped in local minima.
- **Problem-specific**: It should be straightforward to incorporate problem-specific heuristics into the optimization framework.

We implemented a multiobjective genetic algorithm that has all the attributes listed above.

5.2 Specialized hardware resources

In addition to the single processor PEs and communication resources described in Section 3.6, MOGAC models another type of PE: multiprocessor integrated circuits (ICs). Multiple cores, each of which has the attributes associated with a PE, may be located on the same IC, allowing multiple tasks to execute simultaneously on the IC. This provides a model for application specific integrated circuits (ASICs) that are capable of carrying out different tasks at the same time.

MOGAC accepts a database that specifies the performance of each task on each available PE and core type, as well as providing other information about the PEs and cores available, e.g., a list of tasks that are incompatible with each type of PE and core, the price of each resource, and the number of devices provided by each IC and consumed by each core.

The relationship between tasks and PEs was described in Section 3.6. The following information establishes the relationship between tasks and cores:

- A two-dimensional array indicating the relative worst-case execution time of each task on each core.

- A two-dimensional array indicating the relative average power consumption of each task on each core.
- A two-dimensional array indicating the peak power consumption of each task on each core.

In MOGAC, cores do not have inherent prices. However, each core is assigned to an IC that does have a price. The following variables are associated with ICs: price, device count, pins available, idle power consumption, peak power dissipation, speed, and power efficiency. Each core places a device count requirement, e.g., number of transistors or configurable logic blocks (CLBs), on the IC to which it is assigned. For an architecture to be valid, each IC must meet device count requirements of the cores assigned to it and the pin count requirements of the communication resources attached to it. In addition, each IC must meet the peak power dissipation requirements of the tasks assigned to the cores implemented on it. Tasks do not have pin count, device count, or peak power dissipation requirements. However, tasks may be carried out by cores that place such requirements on their host ICs.

The worst-case execution time for a task assigned to a core is equivalent to its relative worst-case execution time divided by the speed of the IC on which the core is implemented. The task's average power consumption is its relative average power consumption divided by the power efficiency of the IC on which the task's core is implemented. Thus, in the current implementation of the algorithm, we assume that there is a linear relationship between core worst-case execution time and task relative worst-case execution time. Similarly, there is a linear relationship between core average power consumption and task relative average power consumption. This model could trivially be generalized to use a full lookup table, similar to the approach used to determine the execution time of a task running on any given PE.

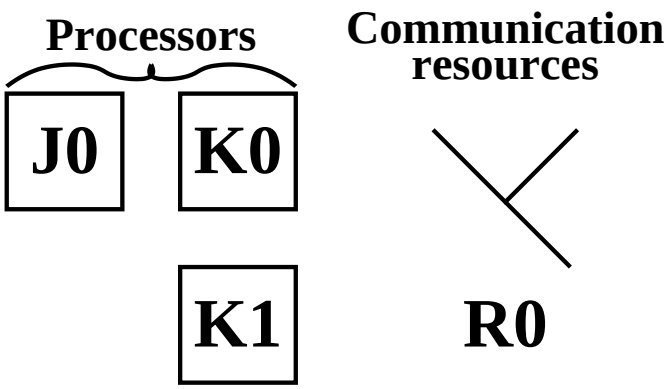

Figure 5.1: Example allocation.

5.3 Solution representation

Although we discuss the optimization infrastructure used by MOGAC in conventional terms, each solution is represented by a collection of multidimensional data structures; primitive linear strings are never computed. The PE allocation is held in a one-dimensional array of integers. The offset into this array corresponds to PE type. The integer at each offset represents the number of PEs of that type in a solution. The communication resource allocation is represented by a similar one-dimensional array. Assuming that three types of PEs (J, K, and L) are available, the PE allocation shown in Figure 5.1 would be represented by the following array:

$$[1, 2, 0]$$

This array indicates that the allocation contains one PE of type J, two PEs of type K, and zero PEs of type L.

Task assignment is represented by a two-dimensional array in which the first dimension corresponds to the task graph that a task belongs to, and the second dimension corresponds to the task's index within the graph. Each entry in the array holds a one-dimensional array with two entries. The first entry is the type index of the PE to which

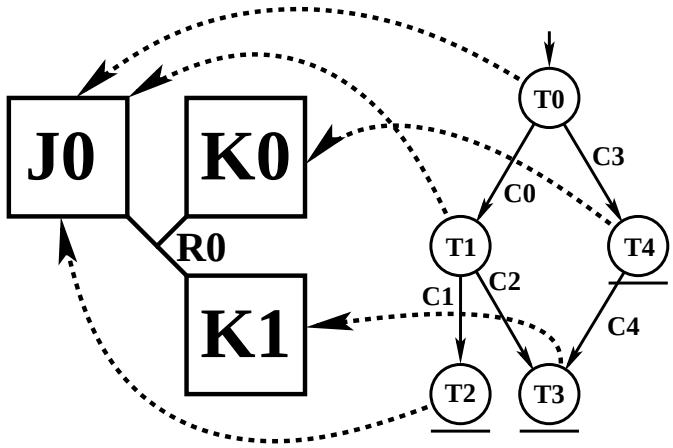

Figure 5.2: Example assignment.

the task is assigned. The second entry is the instance index of the PE to which the task is assigned. The example assignment for the single graph shown in Figure 5.2 would be represented by the following one-dimensional array of two-entry one-dimensional arrays:

$$\begin{bmatrix} [0,0] \\ [0,0] \\ [0,0] \\ [1,1] \\ [1,0] \end{bmatrix}$$

The five rows correspond to task T0 to task T4. PEs J0, K0, and K1 are encoded as $[0,0]$, $[1,1]$, and $[1,0]$, respectively. Note that an assignment with multiple task graphs would require a multidimensional array with a column for each graph. In addition, an additional task copy dimension exists in multirate systems, as described in Section 5.7.2.

As mentioned in Section 4.3, a genetic algorithm's strings should preserve locality, i.e., representations of interdependent portions of the solution should be located closer to each other in a string than disparate entries [120]. The allocation string ordering algorithm places PEs such that those with similar characteristics, e.g., price, have

a higher probability of being located close together in the string than those with disparate characteristics. The order of PE types in the PE allocation string is determined in the following way. The relationship between tasks and PEs is defined by a collection of two-dimensional arrays (see Section 3.3). For the purpose of characterizing a PE type, the one-dimensional arrays corresponding to that PE type are selected from these two-dimensional arrays. Thus, each PE can be characterized by a collection of one-dimensional arrays and some scalars. The first step in determining the order of PE types in the PE allocation string is to collapse each PE type's arrays into scalars. This conversion is done by taking a sum of each array's entries and weighting each entry with the number of tasks, of the type corresponding to that entry's position, which exist in the embedded system specification. After this step, each PE is described by a collection of scalars, i.e., a vector. An approximation algorithm is used to impose an order on these vectors that, in general, places vectors that are close to each other in the n-dimensional space, close together in the PE allocation string. The *communication resource allocation string* and *IC allocation string* are similar to the PE allocation string and they are ordered using similar algorithms.

The *communication resource connectivity string* is an array of IC and processor instance references specifying the ICs and basic PEs to which each communication resource is connected. An example communication resource connectivity string is shown in Figure 5.3. In this illustration, communication resource G's two contacts are connected to PE instances P and Q. Communication resource H connects P, Q, and R. More than one communication resource may be connected to the same PE instance. In Figure 5.3, PE instance Q is an example of a PE connected to two communication resources. The order of communication resource types in the communication resource connectivity string is equivalent to their order in the communication resource allocation string.

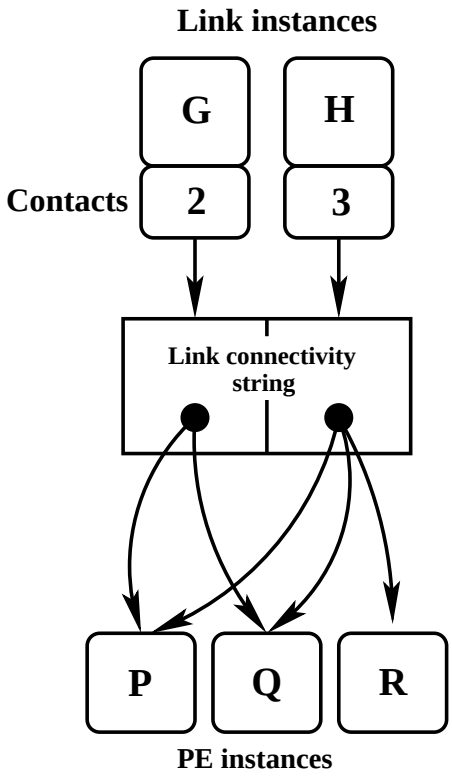

Figure 5.3: Example communication resource connectivity string.

5.4 Optimization algorithm

In this section, we give an overview of MOGAC's optimization infrastructure. This algorithm maintains a pool of solutions that evolve in parallel. Figure 5.4 illustrates MOGAC's core algorithm. After initializing each solution with randomized algorithms, MOGAC enters a loop that repeats until the halting condition, the passage of a number of generations without improvement in the solution pool, is met. During evaluation, a solution's costs, e.g., price and power consumption, are determined. The costs are then compared to the designer-supplied constraints to determine how severely the constraints are violated. At this point, the solutions are ranked using the multiobjective criterion described in Section 4.5. If the halting conditions have not yet been reached, low-rank

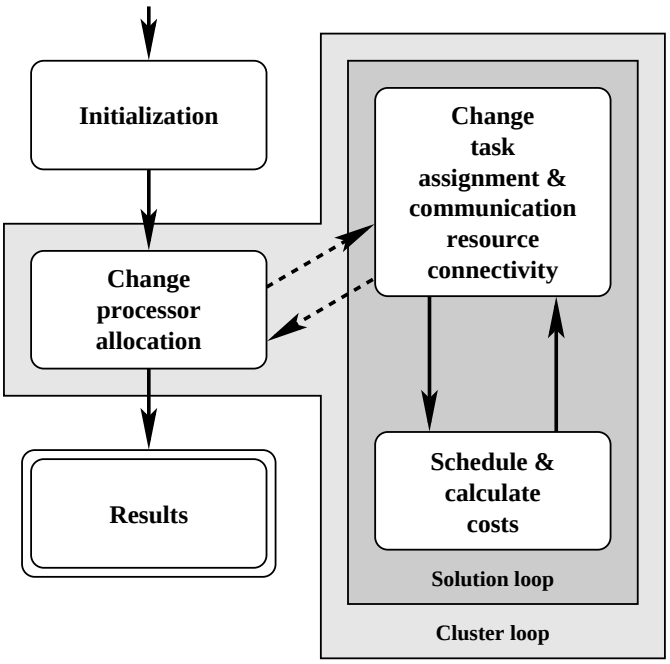

Figure 5.4: Optimization algorithm overview.

solutions are terminated and high-rank solutions reproduce to take their places. The newly born solutions are then modified via crossover and mutation. At this point, the generation has completed and another begins. Eventually, enough generations pass without improvement in the solution pool to trigger the halting condition. Before halting, MOGAC prunes any invalid and inferior solutions from its solution pool and presents the remaining solutions to the designer.

5.5 Clusters

Clusters of solutions are used to prevent crossover from producing *structurally incorrect* solutions, i.e., solutions that are physically impossible. If it were possible for crossover to occur between arbitrary solutions, structurally incorrect solutions would

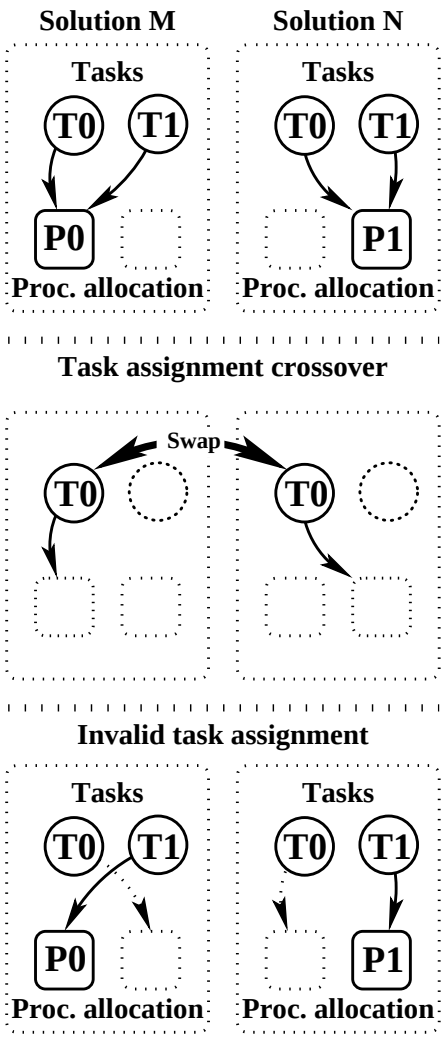

Figure 5.5: Bad crossover.

sometimes be produced. Assume the existence of two solutions: N and M. As illustrated in Figure 5.5, N's PE allocation contains only one PE instance, of type P0. M's PE's allocation contains only one PE instance, of type P1. Therefore, all tasks in M are assigned to the PE of type P0 and all tasks in N are assigned to the PE of type P1. If a crossover were to occur between the task assignment strings in the two solutions, the result might be the existence of some tasks in M that are assigned to a PE of type P1.

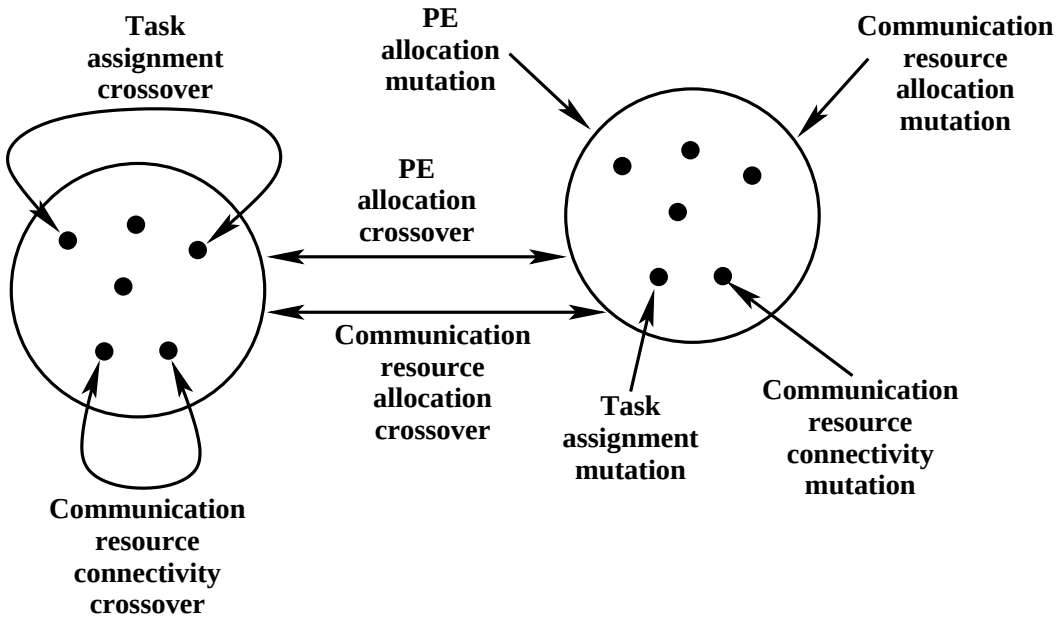

Figure 5.6: Solution clusters.

However, no PEs of type P1 exist in M's PE allocation. Similar problems may be caused by an indiscriminate crossover of other types of strings.

It would be possible to detect structurally incorrect solutions and repair, or immediately terminate, them. However, examining every solution and modifying or terminating those that are structurally incorrect would be costly in terms of computation time. More importantly, the post-processing would destroy the locality of the crossover operator, i.e., this step would disrupt the partial solutions that were swapped during crossover.

MOGAC, unlike past work in the research area, uses the concept of solution clusters to prevent structurally incorrect solutions from being created in the first place, i.e., MOGAC considers the interdependence between different portions of a solution's representation when carrying out genetic operators. As shown in Figure 5.6, solutions are grouped into clusters. In this figure, each dot is a solution and the circles around groups of dots are clusters. Solutions within a cluster all share the same PE allocation and communication resource allocation strings. Thus, each solution in the single cluster has the

same PE and communication resources available to it. However, the task assignment, core assignment, and communication resource connectivity strings of solutions in the same cluster may differ. Crossover of assignment and communication resource connectivity strings occurs between solutions in the same cluster. Individual communication resources (busses or point-to-point links) are treated as atomic during communication resource connectivity crossover. Mutation of these strings can be applied to individual solutions. Solutions resulting from these operations are guaranteed to be structurally correct. Crossover of allocation strings occurs between entire clusters. Similarly, when one of a cluster's allocation strings mutates, all of the solutions within the cluster are updated so that they share the cluster's new allocation string. Inter-cluster crossover and mutation of allocation strings occurs less frequently than intra-cluster crossover and mutation. Every time crossover or mutation are applied to clusters, instead of individual solutions, the information contained in the assignment and communication resource connectivity strings of the involved solutions is no longer valid. These strings are, therefore, re-initialized.

There are three advantages to the use of solution clusters. The overall algorithm is simplified because it is not necessary to detect or repair structurally incorrect solutions after solution crossover. The algorithm's execution time is decreased because it is not necessary to deal with structurally incorrect solutions and because locality is not destroyed by repair operations, thus allowing more implicit parallelism in the genetic algorithm (see Section 4.3). Finally, using clusters makes MOGAC easier to parallelize. The solutions within a cluster can evolve independently of solutions in other clusters, except when crossover between clusters occurs. There is no need for solutions in different clusters to communicate with each other except during the infrequent application of inter-cluster crossover.

```
Start from an empty PE allocation.
For each task t:
    If there exist no PEs capable of executing t:
        Randomly select a PE type, pe, that
         is capable of executing t.

        Add an instance of pe to the solution's
         PE allocation string.
```

Figure 5.7: PE allocation string initialization.

5.6 Initialization and genetic operators

In this section, we describe the manner in which solutions are initialized and modified by genetic operators.

PE allocation strings are initialized with the simple constructive algorithm shown in Figure 5.7. If the solution contains any cores, its IC allocation string is initialized to contain a single, randomly chosen IC. Otherwise, the IC allocation string is initially empty. Initially, a solution's communicate resource allocation string is empty, i.e., the solution contains no communication resources. Communication resources are introduced by subsequent mutations. The intention of these initialization algorithms is to set up minimal valid solutions that will be improved via mutation and crossover.

An allocation string's mutation operator selects a PE, IC, or communication resource type at random; each PE, IC, or communication resource type has the same probability of being selected. The number of instances of the selected PE, IC, or communication resource type is either incremented or decremented, with equal probability. When the crossover operator is applied to two allocation strings, the strings are cut at the same two random offsets and the portions between the cuts are swapped. After the crossover

Randomly select a task instance, t, in the task assignment string.

pe is the position, in the allocation string, of the PE to which t is assigned.

g is a Gaussian random variable with $\mu = 0$ and $\sigma^2 = 1$.

Set $pe^+ := \lceil g \cdot a + pe \rceil$.

If there are no PEs of type pe^+ allocated or t may not execute on pe^+:

Select the nearest neighbor of pe^+ in the allocation that can execute t.

Figure 5.8: Task assignment string mutation.

or mutation of a PE allocation string, the constructive algorithm shown in Figure 5.7 is applied to the participating string. This enforces the condition that, for each task, there exists at least one PE capable of executing it. Usually, it is not necessary for this post-processing step to make any changes to the PE allocation string. Similarly, if a crossover or mutation causes a solution that contains one or more cores to have no ICs, a single, randomly selected, IC is introduced.

Initially, each task is randomly assigned to a PE instance in the PE allocation string that is capable of executing it. The constructive algorithm used to initialize a solution's PE allocation string guarantees that there is at least one PE capable of executing each task (see Figure 5.7). Similarly, each core in the core assignment string is randomly assigned to an IC.

The task assignment string mutation operator selects a task at random and changes the PE type used to carry out that task (see Figure 5.8). An analogous algorithm is used for the mutation of core assignment strings. MOGAC maintains a PE aggressiveness variable, a, that decreases during the run of the algorithm. If the value of this variable is small, a nearby PE type will probably be used to carry out the task. If a is large, it is likely that pe^+ will be far from pe in the PE allocation string. The PE allocation string

---

Generate an array, l, of PE locations.

Set each PE location in l to a unique location in the PE allocation string.

For each array, r, of PE references in the commun. resource connectivity string:

  Randomize the order of the entries in l.

  c is the number of PEs to which l may connect.

  For $i := 0$ to $\min(c - 1, \text{length}(r))$:

    Set $r[i] := l[i]$.

Figure 5.9: Communication resource connectivity string initialization.

is ordered in a locality preserving way. Hence, there is an inverse correlation between distance on the PE allocation string and PE type similarity. Decreasing a during a run allows MOGAC to initially mutate task assignment strings in a way that is likely to cause large jumps across the solution space. As a run nears its end, task assignment mutation makes only small changes to the task assignment string, fine-tuning it.

When the crossover operator is applied to two task assignment strings, the strings are cut at the same random offset and the portions following the cut are swapped. The two participating strings always come from solutions that have the same PE allocations because task assignment string crossover is an intra-cluster genetic operator. The mutation operation for core assignment strings is analogous.

Initially, each communication resource is randomly connected to PEs in the PE allocation string (see Figure 5.9). The communication resource connectivity mutation operator selects a location in the string at random and applies the inner loop of the initialization algorithm shown in Figure 5.9 to it. In other words, it connects a communication resource to PEs randomly. The communication resource connectivity string's crossover operator cuts the participating strings at the same random offset and swaps

the portions following the cut. The two participating strings always come from solutions that have the same communication resource allocations because communication resource connectivity string crossover is an intra-cluster genetic operator.

During mutation, randomized local changes are made to solutions and clusters. Note that randomized changes need not be entirely random, i.e., mutation may be directed toward more promising results by heuristics as described in Section 6.4. Cluster mutations occur a random number of times, varying from zero to the number of clusters, during each iteration of the cluster loop. Similarly, solution mutations occur a random number times, varying from zero to the number of solutions, during each iteration of the solution loop. During cluster PE allocation mutation, a PE of a randomly selected type is either added or removed. Similarly, during communication resource allocation mutation, a randomly selected type of communication resource is either added or removed. During task assignment mutation, a task is randomly selected and its assignment is re-initialized as described in Section 6.5. During communication resource connectivity mutation, the PEs to which a communication resource is connected are re-initialized as described in Section 6.5.

Sometimes, due to PE allocation crossover or mutation, a PE is removed from a cluster's allocation. When this happens it is necessary to adjust the task assignments of the solutions within the cluster. Any tasks assigned to the lost PE must be immediately reassigned for the solutions in the cluster to remain structurally correct (see Section 5.5). The assignments of the affected tasks are re-initialized using the algorithm described in Section 6.5.

5.7 Solution evaluation

Performance evaluation consists of calculating a solution's costs and determining how severely they violate the constraints imposed by the designer. If one of the system's costs is higher than its *hard constraint*, the system is invalid. For example, the times at which tasks are scheduled cannot exceed their hard real-time constraints. Valid systems may have costs that are higher than their *soft constraints*, although it is desirable to reduce a cost until it is lower than its soft constraint. In this section, we will explain how MOGAC does performance evaluation and then describe the process by which raw performance metrics are converted into hard and soft constraint violation values.

5.7.1 Scheduling

The PE allocations, communication resource allocations, task assignment, core assignment, and communication resource connectivities of MOGAC's solutions are optimized by its genetic algorithm. Scheduling, however, is carried out by a conventional algorithm during each solution evaluation. MOGAC uses a slack-based list scheduling algorithm to generate static PE and communication resource schedules. Static scheduling makes it possible to guarantee that hard real-time constraints will be met [124]. MOGAC's scheduling algorithm assigns a priority to a task based upon the difference between its latest possible start time and its earliest possible start time. The relative priorities of tasks in different task graphs, as well as different copies of the same task graph, are based on the periods and deadlines of the different graphs. The scheduler is capable of dealing with embedded system specifications in which task graphs have periods less than their deadlines.

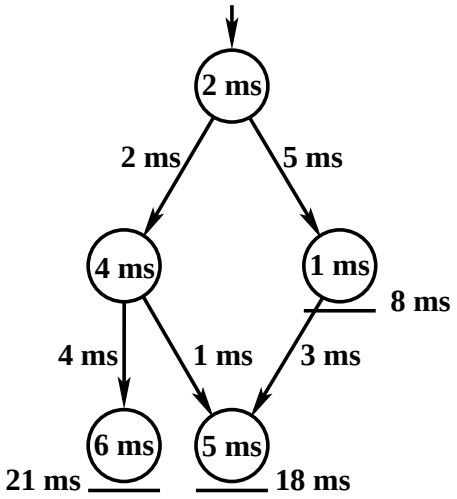

Figure 5.10: Task durations, communication event durations, and deadlines.

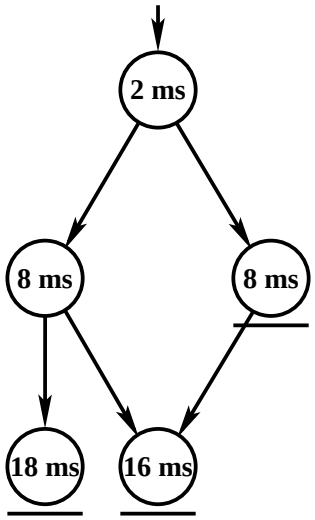

Figure 5.11: Earliest finish times.

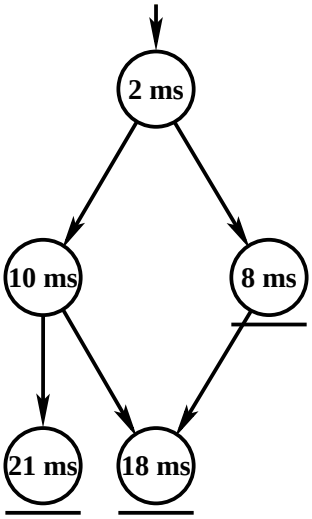

Figure 5.12: Latest finish times.

Slack computation is best illustrated with an example. Figures 5.10, 5.11, 5.12, and 5.13 show the major steps in slack computation. The tasks and communication events in Figure 5.10 are labeled with their durations. Deadlines are also specified (21 ms, 18 ms, and 8 ms). Figure 5.11 shows the earliest finish times of each task. Earliest finish times are computed by conducting a topological search of the task graph, starting from the node with no incoming edges. Each communication event duration is assumed to be the duration required by the slowest communication resource connecting the PEs to which the communicating tasks are assigned. It is commonly assumed, in distributed computing research, that communication between tasks assigned to the same PE is effectively instantaneous, relative to inter-PE communication. We also make this assumption. Figure 5.12 shows the latest finish times of each task. Latest finish times are computed by conducting a backward topological search of the task graph, starting from the nodes that have deadlines. Figure 5.13 shows the slack of each task. Slack is the different between a tasks's latest finish time and its earliest finish time. A task's priority is equal to its negated slack. Priority is static, i.e., it is computed before scheduling begins and

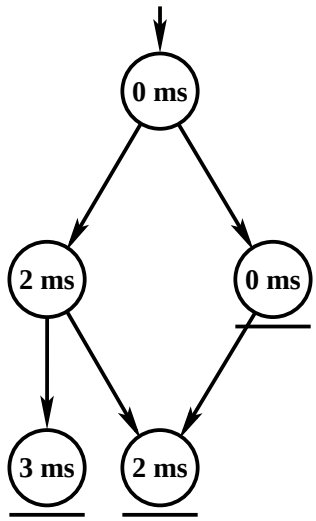

Figure 5.13: Slacks.

is not adjusted during scheduling. Priority computation, therefore, takes $\mathcal{O}(e + t)$ time, where e is the number of edges and t is the number of tasks.

An example schedule is shown in Figure 5.14. This schedule corresponds to the allocation shown in Figure 5.1 and the assignment shown in Figure 5.2. In this example, a task graph is shown on the right and one possible schedule is shown on the left. For the sake of simplicity, a single rate schedule, in which the period of the task graph is greater than its highest deadline, is shown. Task T0 is scheduled to PE J0 at time zero. After this, communication event C1 is scheduled to communication resource R0. Note that communication event C0 need not be scheduled because tasks T0 and T1 are both assigned to PE J0. The schedule continues, similarly, with time ordering partially determined by data dependencies.

Co-synthesis systems that use a straightforward application of the LCM scheduling method [105] are forced to repeatedly schedule each task graph until the hyperperiod of the system has elapsed. This can be computationally expensive for systems in which the hyperperiod is large, relative to the periods of individual task graphs. MOGAC uses

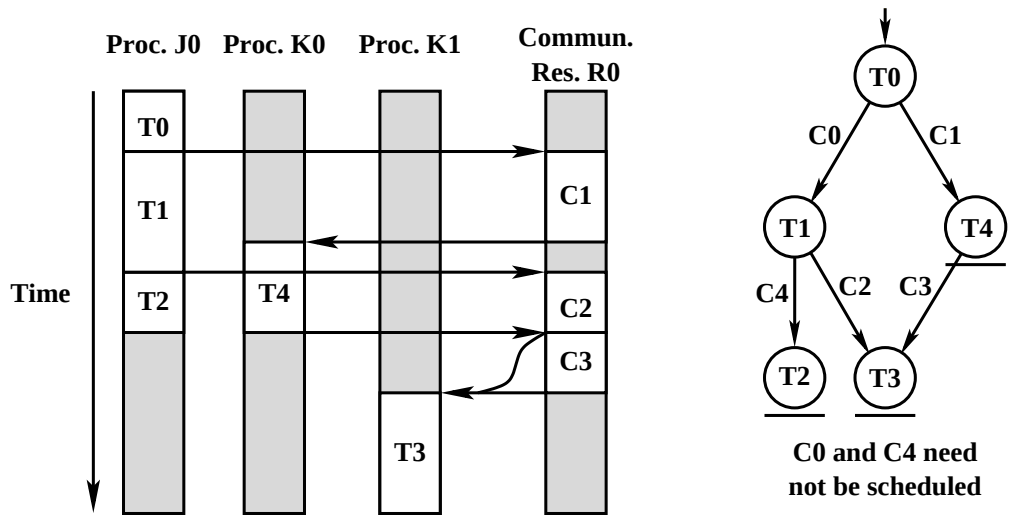

Figure 5.14: Example schedule.

heuristics to tackle system specifications with a large hyperperiod. One of these is an extension of a method used in real-time computing [125]. The problem caused by a large hyperperiod can be reduced by tightening the periods of some task graphs. Consider a system consisting of two periodic task graphs, in which the first has a period of 12, and the second has a period of 13. The hyperperiod is, therefore, 156. If we tighten the period of the second task graph to 12, however, the system's hyperperiod reduces to 12. The designer has full control over the aggressiveness with which the hyperperiod contraction heuristic is applied. MOGAC allows the designer to specify the maximum and minimum acceptable periods for each task graph in the system. Subject to these constraints, a period for each task graph is calculated such that the number of task graph copies needed for LCM scheduling is minimized.

5.7.2 Task graph copies

When a task graph's period, p, is less than the hyperperiod, h, of a task set, $\frac{h}{p}$ copies of the graph are scheduled (see Section 3.4). Figure 5.15 shows an example schedule

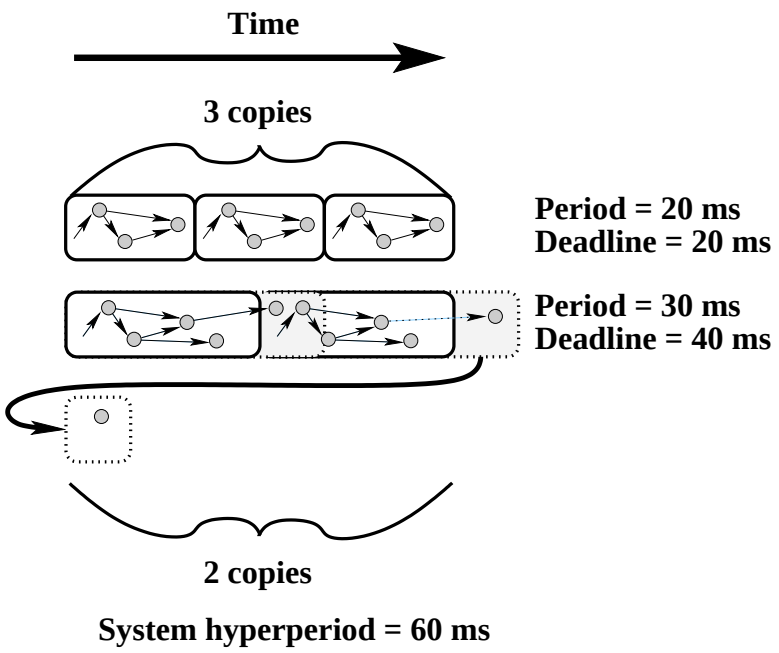

Figure 5.15: Task graph copies.

containing two task graphs. The upper task graph has a period of 20 ms, the lower task graph has a period of 30 ms. As a result, the system hyperperiod is 60 ms. Three copies of the upper task graph must be scheduled and two copies of the lower task graph must be scheduled. Intuitively, scheduling new copies of task graphs until the system hyperperiod is reached ensures that all inter-task graph interactions that may ever be encountered will have valid schedules. Somewhat less intuitively, some graphs may have periods that are less than their maximum deadlines. The lower task graph has a period of 30 ms but it has a maximum deadline of 40 ms. As a result, some tasks from the first (left) task graph copy may continue to execute after the second (right) task graph copy begins execution. Some of the tasks in the second (right) task graph copy may execute after the first task graph copy has started executing again, thereby wrapping past the end of time to the beginning of time. This can be more intuitively understood by viewing time as circular.

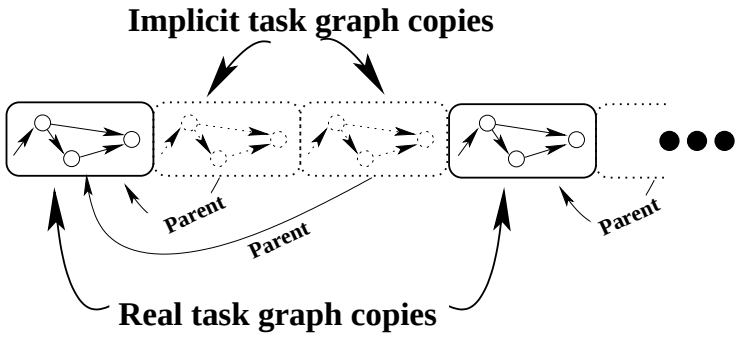

Figure 5.16: Implicit task copy assignment.

We have developed a method in which some of the task graph copies in the hyperperiod are *implicit* and some are *real* (see Figure 5.16). Each implicit copy has a real *parent*. Implicit copies are not entered in a solution's task assignment string; they share the assignment strings of their parents. Although it is necessary to schedule implicit task graph copies, there is no need to prioritize the nodes of these copies; the implicit task graph node priorities are equivalent to the parent task graph node priorities. Additionally, the absence of implicit copies from a solution's task assignment string reduces the size of the genetic algorithm's solution space, thus speeding optimization. Selecting a ratio of the number of real task graph copies to the total number of task graph copies involves making a trade-off between potential solution quality and MOGAC's run-time. This decision is left to the designer. For the examples in Section 5.9, a low ratio (< 0.2) rapidly produced high-quality results.

5.7.3 Cost calculation

Hard real-time deadline violation, price, and power consumption are computed during cost calculation. The completion time of each node in a task graph is recorded during scheduling. Therefore, the completion times of all nodes with deadlines are available for inspection. All schedules span the system's hyperperiod. Price is determined by taking

the sum of the prices of all PEs and communication resources in a solution's PE and communication resource allocations. System power consumption is computed by stepping through each PE's and communication resource's hyperperiod schedule, obtaining the system energy required, including the idle PE or communication resource energy, and dividing the energy by the hyperperiod [68].

5.7.4 Constraint violation

A system's constraint violations are derived from its costs and the constraints imposed by the designer. Solutions have a number of hard constraints. Although solutions in which one or more hard constraints have been violated are invalid, MOGAC treats them no differently than other solutions during its run. Solutions that violate their hard constraints are removed only at the end of a co-synthesis run. It may seem counter-intuitive to allow invalid solutions to survive. However, doing so is beneficial when solving constrained problems [126], for there are significant disadvantages associated with the alternatives. If one terminates invalid solutions immediately, one wastes a significant amount of computation time in identifying such solutions. The solutions most likely to eventually evolve into high-quality valid solutions are those that are near the boundary between valid and invalid. By immediately terminating all invalid solutions in each generation, one destroys many solutions that are likely to ultimately evolve into high-quality valid solutions. One could, instead, attempt to repair invalid solutions. However, in general it is difficult to formulate a repair operation that is guaranteed to repair all solutions [79]. Thus, one will often be forced to terminate solutions even after expending computation time attempting to repair them. More importantly, a repair operation applied to a solution that was made invalid by crossover disrupts a portion of that solution, effectively changing the crossover operator such that it no longer preserves

locality. These problems are analogous to the problem with terminating or repairing invalid solutions discussed in Section 5.5.

Every task graph has one or more nodes with specified deadlines. A system's hard real-time constraint violation is the sum of the time constraint violations of all such nodes in all the task graph copies in the system.

5.8 Ranking and reproduction

In this section, we explain the manner in which solutions and clusters are selected for reproduction. The number of clusters and solutions maintained by MOGAC is conserved during one run of the algorithm. For each cluster or solution created via reproduction, another is terminated. The number of solutions and clusters maintained during a run can be chosen at the start of the run. We typically use 20 clusters, each of which contains 20 solutions.

Solutions within a cluster are ranked using the method presented in Section 4.5. In each generation, a pre-specified number of solutions within each cluster are eliminated to make space for the reproduction of other solutions. MOGAC maintains a solution selection elitism variable, e, that controls the probability of high-rank solutions being selected for reproduction. This variable increases during the run of the algorithm. This feature has the practical effect of allowing MOGAC to more easily escape local minima during the start of an optimization run. Near the end of a run, however, MOGAC becomes greedier to allow its solutions to converge on local minima. Solutions are selected for reproduction by indexing inward from the highest-ranking solution with a Gaussian random variable whose variance is the inverse of e. The pseudo-code for MOGAC's reproduction algorithm is shown in Figure 5.17.

e is the solution selection elitism.

G is a Gaussian random variable with $\mu = 0$ and $\sigma^2 = \frac{1}{e}$.

n is the number of solutions to be replaced via reproduction.

S is the array of solutions, with L_S entries.

Sort solutions in the order of increasing Pareto-rank.

For i := 0 to $n - 1$:

    Select a random instance, g, from G.

    Set $o := L_S - 1 - g/e$.

    Set $S[i] := S[o]$.

Figure 5.17: Solution reproduction algorithm.

After reproduction, crossover and mutation are carried out on the solutions that were copied. The number of crossovers and mutations per generation, for each type of string, are specified by user-defined parameters. Crossover is applied to randomly selected solution pairs that are selected from the solutions created by reproduction. Mutation is applied to randomly selected solutions that are also selected from the solutions created by reproduction.

Ranking clusters is more complicated than ranking solutions. Each solution has one set of costs. Thus, determining whether it dominates another solution is straightforward. Clusters, however, contain numerous solutions; each cluster is associated with many sets of costs. We extend the concept of domination, in a straightforward way, to take partial domination into account. Cluster domination is represented by a scalar instead of a Boolean value. The definition of rank must also be adjusted when it is applied to clusters. Let x and y be clusters. $\mathrm{Nis}\,(x)$ is the set of non-inferior solutions in x.

$\text{dom}\,(a, b)$ is 1 if a is not dominated by b and 0 otherwise. Cdom is a function of two clusters. Then,

$$\text{cdom}\,(x, y) = \max_{a \in nis(x)} \sum_{b \in nis(y)} \text{dom}\,(a, b)$$

and,

$$\text{rank}[x] = \sum_{y \in \substack{all \\ clusters}, \, y \neq x} \text{cdom}\,(x, y)$$

Once cluster ranks have been determined, cluster reproduction is analogous to solution reproduction. A pre-specified number of clusters is removed to make room for high-rank clusters to reproduce. Clusters are selected for reproduction in the same manner as solutions. Cluster crossover and mutation are also analogous to solution crossover and mutation.

5.9 Experimental results

MOGAC is a prototype consisting of approximately 18,000 lines of C^{++} and Bison code. Our results were obtained on a 200 MHz Pentium Pro system with 96 MB of memory running the Linux operating system. We compare our results with those of Yen [127], Hou [84], and COSYN [128], which were obtained on a SPARCstation 20, as well as those of SOS [76], which were obtained on a Solbourne Series5e/900 (similar to a SPARC 4/490). The CPU times are given in seconds.

MOGAC's input consists of two ASCII files. The first file specifies the attributes of each PE, IC, and communication resource type that may be used to implement an architecture. In addition, this file specifies the relationships between PEs and tasks, i.e., for each PE it contains arrays specifying the worst-case execution times, average power consumptions, and peak power consumptions of each task on that PE. The second file specifies the topologies, periods, deadlines, tasks, and communication events

associated with all the task graphs comprising the system specification. MOGAC runs without designer intervention and, upon halting, outputs one or more solutions. Each solution is a system architecture consisting of a price, power consumption, PE allocation, IC allocation, communication resource allocation, core assignments, task assignments, communication resource connectivities, task schedules for each PE, and communication event schedules for each communication resource.

5.9.1 Price optimization

MOGAC has numerous parameters that can be modified to tune its performance. Although every problem has its own optimal parameter settings, it would be inappropriate to only report the CPU time necessary to achieve a given solution if significantly more time was spent finding a good set of parameters. We, therefore, use the same set of parameters for all the examples presented in this section. In addition, the same value is used to seed MOGAC's random number generator for every result presented in this paper, with the exception of Table 5.4.

It was necessary to trade off run-time against solution quality when selecting a general parameter set for the examples in this section. Using a smaller solution pool and cluster pool would allow MOGAC to produce low-cost solutions for simple examples more rapidly. However, the solution quality for more complicated examples would suffer. For illustrative purposes, run-times achieved by tuning MOGAC's parameters to an individual problem's complexity, as well as the run-times that resulted from using the general parameter set, are shown in the price optimization tables.

Table 5.1 compares MOGAC's performance with that of COSYN [67] and Yen's system [127] when each is run on the clustered and unclustered versions of Hou's task graphs [84]. Task clustering is the process of using a pre-pass to collapse multiple tasks into a cluster of tasks. This cluster is treated like a single task during assignment, i.e.,

Table 5.1: Hou's examples

Example	No. of Tasks	Yen's system		COSYN		MOGAC		
		Price	CPU time (s)	Price	CPU time (s)	Price	CPU time (s)	Tuned time (s)
Hou 1&2 (unclustered)	20	170	10,205	170	5.1	170	5.7	2.8
Hou 3&4 (unclustered)	20	210	11,550	n.a.	n.a.	170	8.0	1.6
Hou 1&2 (clustered)	8	170	16.0	n.a.	n.a.	170	5.1	0.7
Hou 3&4 (clustered)	6	170	3.3	n.a.	n.a.	170	2.2	0.6

all the tasks in a cluster are executed on the same PE. Clustering reduces the complexity of the co-synthesis problem by decreasing the number of tasks that must be assigned. Hou ran Yen's system on the clustered and unclustered versions of his graphs. We use the same clusters as Hou when comparing our results with his, and those of COSYN. For the example upon which it was possible to make a comparison between MOGAC and COSYN, COSYN's performance was similar to that of MOGAC. The only existing implementation of COSYN is solely owned by Lucent. We relied on results reported in the literature to compare with COSYN.

It is interesting to observe the impact of an increase in problem complexity upon MOGAC and Yen's system. MOGAC's CPU time increases slightly when it solves the unclustered versions of Hou's examples instead of the clustered versions. In contrast, Yen's system takes approximately 1,000 times as long to produce solutions. Despite consuming significantly less CPU time, in one case MOGAC produces a lower-price architecture than Yen's system. The difference in solution quality between Yen's system and MOGAC is likely to be a result of differences in their respective optimization infrastructures. The run-time of Yen's system is significantly influenced by the method used to guarantee schedule validity. In addition, Yen uses an optimization algorithm in

Table 5.2: Prakash and Parker's examples

Example ⟨Performance⟩	No. of tasks	SOS		COSYN		MOGAC		
		Price	CPU time (s)	Price	CPU time (s)	Price	CPU time (s)	Tuned time (s)
Prakash & Parker 1 ⟨4⟩	4	7	28	n.a.	n.a.	7	3.3	0.2
Prakash & Parker 1 ⟨7⟩	4	5	37	5	0.2	5	2.1	0.1
Prakash & Parker 2 ⟨8⟩	9	7	4,511	n.a.	n.a.	7	2.1	0.2
Prakash & Parker 2 ⟨15⟩	9	5	385,012	5	0.4	5	2.3	0.1

which a single solution is iteratively improved. Although the search is not blind, only a single stage of look-ahead is used. For each real evaluation, only a single solution is implicitly evaluated. Invalid solutions are terminated instantly instead of being improved upon. The use of a locality preserving crossover operator allows MOGAC's genetic algorithm to implicitly evaluate more than one solution for each explicit evaluation (see Section 4.3). Instead of maintaining a single solution that moves across the solution space, MOGAC maintains multiple solutions that spread out across the solution space. These solutions share information with each other. MOGAC attempts to improve invalid solutions, which are otherwise of high quality, instead of terminating them immediately. We believe that these features allow MOGAC to tackle large problem instances without a prohibitive increase in execution time.

The hyperperiod contraction heuristic described in Section 5.7 was applied to the clustered and unclustered versions of the task graphs called Hou 3&4. The period of one of the task graphs in these examples was contracted by 5%. We were able to decrease MOGAC's CPU time, without decreasing solution quality, by tuning the size of MOGAC's solution pool and making its halting conditions less tolerant.

Table 5.3: Yen's large random examples

Example	No. of Tasks	Yen's system		MOGAC		
		Price	CPU Time (s)	Price	CPU Time (s)	Tuned (s)
Yen's Random 1	50	281	10,252	75	6.4	0.2
Yen's Random 2	60	637	21,979	81	7.8	0.2

Table 5.2 compares MOGAC's performance with that of SOS [76] and COSYN when they are applied to Prakash and Parker's task graphs. The performance number shown by each task graph is the worst-case finish time for the task graph. For instance, "Prakash & Parker 1 $\langle 4 \rangle$," refers to Prakash and Parker's first task graph with a worst-case finish time of 4 time units. In these graphs, an unconventional model for communication is used [76]. A task may begin executing before all of its input data have arrived. We converted their specifications into graphs that conform to the conventional communication model, i.e., a task can only begin execution when all of its input data have arrived. Their model implies that part of each task is independent of the task's input data. This is expressed by splitting each task into a portion that depends on input data and a portion that is independent of its input data. We ensure that each task's subtasks are assigned to the same PE. It is not surprising that SOS requires significantly more CPU time than MOGAC. The mixed integer-linear programming algorithm used in SOS has the potential to take exponential time, relative to the problem instance complexity. It guarantees optimality, while MOGAC makes no such guarantee. However, in practice, MOGAC also obtained optimal results.

Table 5.3 compares MOGAC's performance with that of Yen's system when each system is applied to Yen's large random task graphs [127]. Random 1 consists of six task graphs, each of which contains approximately eight tasks. There are eight PE types

available in this example. Random 2 consists of eight task graphs, each of which contains approximately eight tasks. There are 12 PE types available in this example. Neither of these examples contains communication resources; all communication costs are zero. The observations comparing MOGAC to Yen's system, in the discussion of Table 5.1, apply to these examples as well. However, the price savings achieved by MOGAC are even more substantial.

The task graph periods in these systems are co-prime. Therefore, the hyperperiod contraction heuristic presented in Section 5.7 significantly reduces the number of task graph copies that MOGAC is required to schedule. The heuristic was prevented from specifying task graph periods to be less than the corresponding deadlines, or greater than their original periods [127]. MOGAC's performance depends on the seed given to its pseudo-random number generator. Each problem instance has a different random seed for which MOGAC produces the best results most rapidly. However, MOGAC is able to arrive at a high-quality solution given suboptimal seeds, if its solution pool size or cluster pool size are increased, or its halting conditions are made more lenient.

Table 5.4 shows the average results of optimizing each of the price optimization examples thirty times, given random seeds ranging from one to thirty. In this table, *reported price* is the price reported for a single run of MOGAC with a fixed seed (see Tables 5.1, 5.2, and 5.3). *Effort* corresponds to the computing resources MOGAC is allowed to dedicate to the problem. The meaning of each effort value is given in Table 5.5. The *average price* column shows the average price of the solutions. MOGAC was run in single-objective optimization mode for these experiments. Therefore, each run produces only one non-dominated solution. When MOGAC is given the same parameters as were used in the previous tables in this section, there are a small number of example-random seed combinations for which it does not arrive at valid solutions. Slightly more liberal parameters were used for Table 5.4 than for the preceding tables. This ensures that

Table 5.4: Effect of varying random seed

Problem	Reported Price	Effort	Average Price	Average CPU Time (s)
Hou 1&2 (unclustered)	170	1	183.3	23.1
		2	175.0	56.2
		3	176.7	89.4
		4	171.7	156.8
Hou 3&4 (unclustered)	170	1	176.0	41.8
		2	177.7	80.9
		3	171.0	125.5
		4	171.7	226.0
Hou 1&2 (clustered)	170	1	176.3	11.9
		2	176.7	26.3
		3	170.7	39.7
		4	170.7	73.3
Hou 3&4 (clustered)	170	1	176.6 [1]	12.6
		2	175.7	30.5
		3	174.0	41.6
		4	178.7	72.4
Prakash & Parker 1 ⟨4⟩	7	1	7.0	11.4
		2	7.0	31.7
		3	7.0	49.9
		4	7.0	89.6
Prakash & Parker 1 ⟨7⟩	5	1	5.0	8.0
		2	5.0	24.8
		3	5.0	40.1
		4	5.0	73.3
Prakash & Parker 2 ⟨8⟩	7	1	7.3	10.1
		2	7.1	27.2
		3	7.0	42.3
		4	7.0	72.1
Prakash & Parker 2 ⟨15⟩	5	1	5.0	6.0
		2	5.0	18.0
		3	5.0	29.5
		4	5.0	54.1
Yen's Random 1	75	1	75.0	18.7
		2	73.7	80.1
		3	74.4	125.2
		4	74.4	225.6
Yen's Random 2	81	1	81.0	32.1
		2	81.0	91.1
		3	81.0	148.0
		4	81.0	266.4

Table 5.5: Effort definitions

Effort	Solutions	New solutions	Clusters	New clusters	Generations before halting
1	26	10	33	17	5
2	34	14	40	20	10
3	36	14	44	22	14
4	44	18	45	23	20

average price is meaningful. Note that, when allowed a modest increase in run-time, MOGAC robustly deals with varying random seeds.

Table 5.5 shows the parameter settings corresponding to each effort setting in Table 5.4. *Solutions* is the total number of solutions per cluster and *new solutions* is the number of solution reproductions that occur per generation, per cluster. Similarly, *clusters* and *new clusters* are the total number of clusters and the number of cluster reproductions per generation. *Generations before halting* is the number of generations that must pass without improvement in MOGAC's solution pool before MOGAC halts.

5.9.2 Multi-objective power and price optimization

Table 5.6 displays the results of simultaneously optimizing the price and power consumption of system architectures based on examples presented in past work. The database for the example called Yen's Random 2 contains two IC types and two core types in addition to the processor types specified by Yen, for a total of 14 PE types. The values shown in the "Ignoring Power" column indicate the results of running MOGAC, in single objective price optimization mode, on the same embedded system specifications. MOGAC was given the same parameters for all of the examples in this section, although the parameter set used for price optimization in Section 5.9.1 differs from the parameter set used in this section. The database files used for these examples are

Table 5.6: Power consumption examples

Example	No. of Tasks	MOGAC Ignoring Power			MOGAC Optimizing Power		
		Price	Power	CPU time (s)	Price	Power	CPU time (s)
Hou 1&2 (unclustered)	20	170	60.6	16.9	170	51.8	89.6
Hou 3&4 (unclustered)	20	170	62.4	30.7	170	48.6	26.3
Hou 1&2 (clustered)	8	170	75.3	7.8	170	62.5	9.5
Hou 3&4 (clustered)	6	170	47.1	3.9	170	43.3	5.1
Prakash & Parker 1 ⟨4⟩	4	7	75.4	10.1	7 15	75.4 64.2	20.1
Prakash & Parker 1 ⟨7⟩	4	5	44.4	8.5	5 7 10	44.4 35.1 21.5	16.9
Prakash & Parker 2 ⟨8⟩	9	7	49.8	8.4	7 12	49.8 40.0	17.2
Prakash & Parker 2 ⟨15⟩	9	5	48.0	6.32	5 7 12	48.0 26.8 21.8	22.4
Yen's Random 1	50	75	25.6	43.1	75 151 225 301	17.7 8.2 6.8 3.3	453.1
Yen's Random 2	60	81	39.8	59.2	81 153 158 214 338	34.4 25.4 15.7 9.9 7.0	268.8

available at ftp://ftp.ee.princeton.edu/pub/dickrp/Trans/Mogac. These examples do not contain soft deadlines.

The advantage of multiobjective optimization, over the use of a non-linear directed cost function, can clearly be seen in Table 5.6. When MOGAC simultaneously optimizes power and price, it provides a designer with its entire set of non-inferior solutions. For each system specification, only a single co-synthesis run was necessary to produce all the corresponding architectures whose costs are listed in Table 5.6.

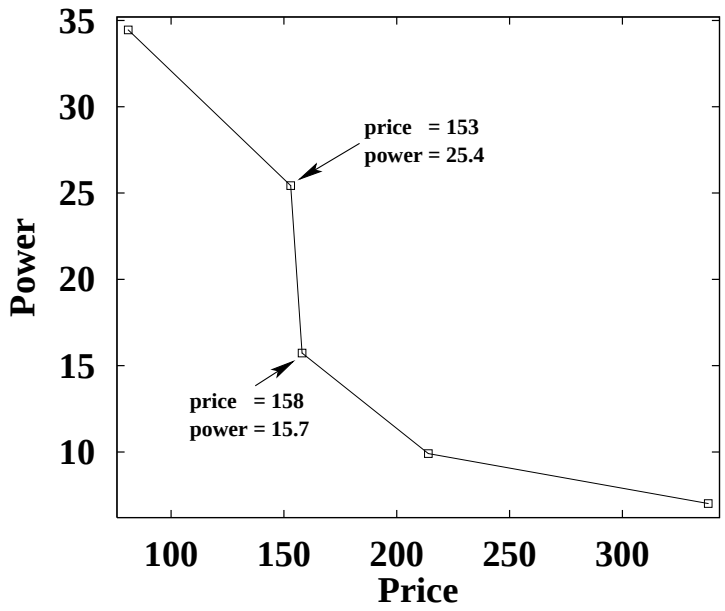

Figure 5.18: Yen's Random 2 example.

MOGAC provides an upper bound on a problem's Pareto-optimal solution set instead of merely producing a single solution. This approach allows a designer to see the relationship between the costs of different architectures that satisfy the same system specification. Figure 5.18 illustrates the danger of selecting a solution without knowing the shape of a system's non-inferior solution curve. Although all of MOGAC's solutions for Yen's Random 2 example are non-inferior, a designer would rarely select the solution with a price of 153 and a power consumption of 25.4 when, for a price penalty of only 5, a solution with a power consumption of 15.7 can be obtained. Presenting a non-inferior solution set shows the designer the cost tradeoffs available between different solutions.

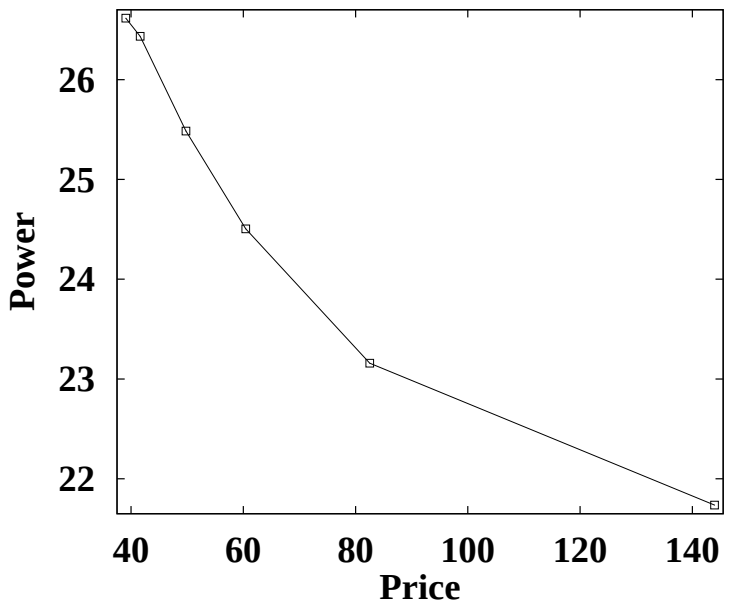

Figure 5.19: Very Large Random 1 example.

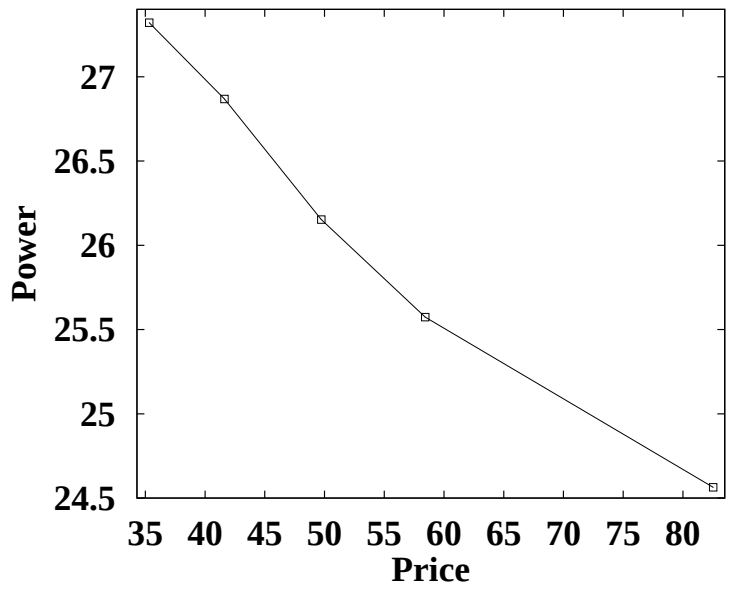

Figure 5.20: Very Large Random 2 example.

Figs. 5.19 and 5.20 show the results of optimizing very large multi-rate examples that require communication resource synthesis. These pseudo-random examples were generated with the TGFF system [129]. They are available via anonymous FTP. The first very large example contains 8 task graphs, each of which has 62 or 63 tasks. There are 8 PE types and 5 communication resource types available. MOGAC took 40.9 CPU minutes to arrive at the non-inferior solution curve shown in Figure 5.19. The second very large example contains 10 task graphs, each of which has 99 tasks. There are 20 PE types and 10 communication resource types available. MOGAC took 203.5 CPU minutes to arrive at the non-inferior solution curve shown in Figure 5.20. The primary purpose of these examples is to demonstrate that MOGAC can rapidly solve extremely large problem instances. We hope that others will use these examples for comparative purposes.

5.10 Conclusions

In this chapter, we have presented a method for the co-synthesis of low-power real-time multi-rate heterogeneous hardware-software distributed embedded systems. A novel multiobjective genetic algorithm that allows exploration of the Pareto-optimal set of architectures instead of providing a designer with a single solution, has been developed and applied to a number of examples found in the literature. MOGAC has been shown to rapidly synthesize architectures with costs that are lower than or equal to those presented in previous work. For large examples upon which comparisons with other systems are possible, MOGAC produces significantly lower-cost solutions, despite requiring orders of magnitude less run-time. It has been demonstrated that adaptive multiobjective PRSA algorithms are well suited to solving the co-synthesis problem.

Chapter 6

Enhanced Low-Power Heterogeneous Distributed Systems Synthesis

In this chapter, we introduce a number of improvements to the optimization algorithm described in Chapter 5. We refer to the resulting optimization infrastructure as EMOGAC. Section 6.1 describes changes made to our hardware resource model to increase its accuracy. Section 6.2 describes changes to the optimization infrastructure that make it conform more closely to the definition of parallel recombinative simulated annealing and improve its performance. Section 6.3 describes the changes we made to crossover in order to better preserve locality. Section 6.4 describes a new task assignment mutation algorithm that takes problem-specific information into account. Section 6.5 describes the constructive algorithm used to initialize solutions. Section 6.6 describes EMOGAC's cost calculation algorithms. Section 6.7 explains a method of caching solutions that improves EMOGAC's performance. Section 6.8 introduces a new embedded synthesis benchmark suite we have developed. We present experimental results and conclusions in Sections 6.9 and 6.10.

6.1 Communication and memory model

In this section, we describe changes made to the communication resource model presented in Section 3.7 in order to increase its accuracy. In addition, we present an enhanced memory model.

Processing elements (PEs) that have had their performance characterized for the Embedded Microprocessor Benchmark Consortium benchmarks suite (described in Section 6.8) are representative of those commonly used to implement embedded systems. The differences between bus protocols for these PEs motivated us to make a change in our communication resource model. We have augmented the simpler classical model described in Section 3.7 with a new cost, price per contact, to represent bus bridge and/or interface circuit price. The augmented communication resource model may be used to model a bus that requires a protocol translator, or bridge, for each connected processor by assigning the bus an appropriate controller price and a contact price equal to the price of a bus bridge.

We use a memory model in which each PE has a dedicated memory used by the tasks assigned to it. It might, at first, seem desirable to allow shared external memories in order to reduce the total quantity of memory, and number of packages, required in the embedded system. Unfortunately, using shared external memory requires that communication with memory be scheduled in a way that avoids contention between memory access requests by tasks assigned to different PEs. This would require detailed information about the exact times at which different tasks access memory. Gathering this information would be difficult; it would be processor-dependent and data-set dependent. In the absence of this information, in order to guarantee that hard real-time deadlines are met, it would be necessary to assume each task constantly accesses the

shared memory during its execution. This would prevent multiple tasks from executing concurrently on different PEs, eliminating one of the major advantages of having multiple PEs. Therefore, we associate dedicated memory with each PE.

We compute the quantity of memory associated with each PE based upon code and data memory requirements. For each PE, we require an entry in the PE database giving the code size of each task type that may execute on that PE. The code memory for a PE in a solution's allocation is the sum of the code memory requirements of the tasks assigned to that PE. We do not currently have access to any benchmarks in which the data memory requirements of each task are given. Therefore, we make the assumption that each task requires an amount of data memory equal to the sum of the data quantities of its incoming and outgoing communication events. Although this is a reasonable assumption for many dataflow tasks, it should be noted that this method of computing memory requirements could easily be changed in the presence of more detailed information about task data memory requirements. In order to compute the memory requirements for a PE, we take the maximum of the data memory requirements of all the tasks assigned to it, and add the sum of the code memory requirements of the tasks assigned to it. Note that task code could initially be stored in electrically programmable read-only memories (EPROMs), and transferred to PE local memories during system initialization. Memories commonly have sizes that are integer powers of two. In order to be conservative, we ensure that each PE has a quantity of memory that is an integer power of two.

6.2 Optimization infrastructure

In this section, we describe changes made to our optimization infrastructure causing it to more closely conform to the definition of a parallel recombinative simulated annealing (PRSA) algorithm.

During our design of MOGAC, described in Chapter 5, we wanted to build an optimization infrastructure with the multiobjective optimization strength of a genetic algorithm and the resistance to becoming trapped in local minima of a simulated annealing algorithm. We believe MOGAC meets these criteria. However, it was not, exactly, a PRSA algorithm. We have subsequently changed our solution reproduction method from that described in Section 5.8 to that described in Section 4.4. Changing from a method that does randomized weighted ranking to a method that conducts Boltzmann trials between the old and new solutions after mutation and crossover makes EMOGAC conform more closely to the definition of a PRSA. The main remaining exception stems from the multiobjective nature of the embedded system synthesis problem. In order to use Pareto-ranking for multiobjective optimization, we rank all solutions relative to each other. This requires a comparison between every pair of solutions. Based on empirical observations of algorithm performance, we have chosen not to constrain Boltzmann trials such that they only occur between direct participants in the same crossover or mutation operation. Constraining Boltzmann trials in this manner has the advantage of easing parallel implementation of PRSA algorithm. However, for a multiobjective optimization algorithm using Pareto-ranking, this advantage is illusory.

At the end of a solution loop, after new solutions have been created and changed as described in Sections 5.6 and 6.3, solutions within each cluster are ranked based on the costs described in Section 5.7.3, using the method described in Section 4.5. Boltzmann trials (see Section 4.2) are then used to eliminate solutions until the cluster contains the same number of solutions as it did at the start of the loop iteration. At the end of a cluster loop, Boltzmann trials are used to eliminate clusters in a similar way.

In EMOGAC, the global temperature used in Boltzmann trials decreases linearly (subtractively) during execution. We experimented with multiplicative and linear cooling schedules and found that using a linear cooling schedule generally resulted in better

solution quality and optimization time. When a user-selected number of generations pass without improvement in solutions, the temperature is decreased.

6.3 Multidimensional locality preserving crossover

In a conventional PRSA algorithm, each solution is represented by a string, i.e., a linear array of values. Information is traded between different solutions by conducting crossover of the strings representing them. Unfortunately, the solutions to many real problems cannot be cleanly represented by one-dimensional strings of values. Each solution in EMOGAC is represented by a collection of values, each of which is associated with a multidimensional vector. For example, each type of PE in a solution's PE allocation is associated with a scalar value indicating the number of PEs of the corresponding type existing in the solution, as described in Section 5.3. In addition, each type of PE is associated with a multidimensional vector describing its attributes, i.e., price is one dimension, weighted average power consumption is another dimension, and weighted average execution time is a third dimension, etc. Conventionally, researchers who use genetic algorithms impose a linear order on the information representing a solution. However, there are problems with this approach. For genetic algorithms to operate efficiently, it is necessary for their crossover operations to preserve locality [120], [121].

When solutions are represented by multidimensional data structures, the complexity of imposing a good locality-preserving order on a solution is increased. Consider the problem of imposing a linear order on a set of n-dimensional vectors. If it is possible to make the assumption that locality is inversely proportional to Euclidean distance, i.e., the attributes that are closest together in space are components of the same mostly independent sub-solutions, or *building blocks*, then imposing a linear optimally locality preserving order on these attributes is equivalent to the n-dimensional Euclidean

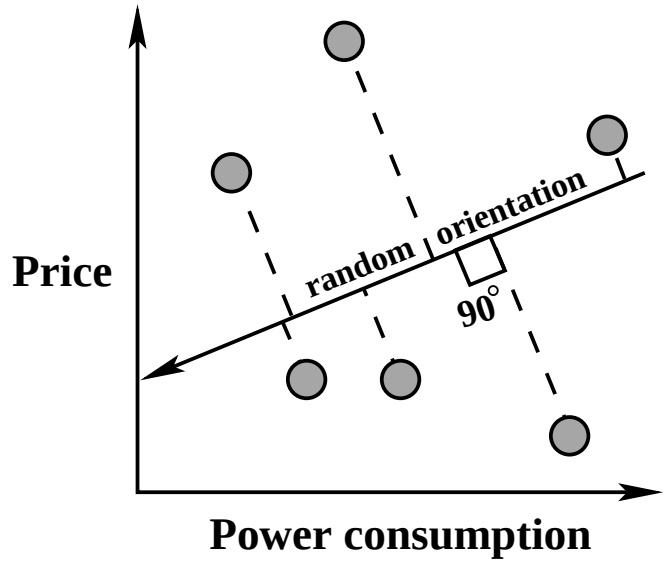

Figure 6.1: Selection of random orientation in crossover.

traveling salesman problem, which is NP-complete [112]. Even if it were computationally feasible to find an optimal solution to this problem, in general, reducing the dimensionality of information from n to one would result in a distortion of space and, consequently, bias the exploration of the solution space. In order to encode some building blocks contiguously, it is necessary to disrupt others. EMOGAC attempts to mitigate the effects of this disruption by dynamically imposing a linear order on the elements of a solution's allocations and assignments. This allows the preservation of locality for a different combination of dimensions during each crossover.

EMOGAC's dynamic locality preserving linearization algorithm assumes an inverse correlation between the distances between resources, e.g., PEs, in a multi-dimensional cost space and membership in the same building blocks. This algorithm has two stages. An illustration of the first stage is shown in Figure 6.1. For the sake of simplifying this example, we will discuss only two PE dimensions: price and power consumption. However, in general, this method may be applied to n-dimensional elements, where n

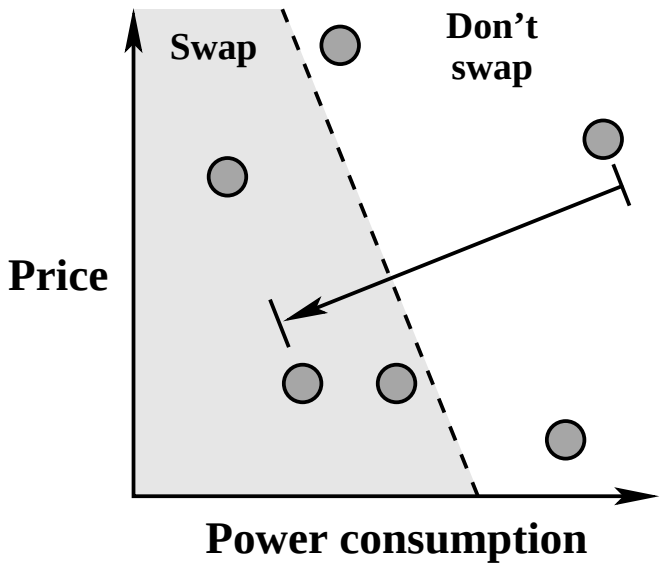

Figure 6.2: Selection of bounded random position in crossover.

is an arbitrary positive integer. Each circle in Figure 6.1 is an element in a solution's PE allocation, i.e., each circle corresponds to a PE. Initially, a randomly oriented, unit length vector (labeled *random orientation* in the illustration) is placed in the PE allocation hyper-space (a plane in this example). The dot product of each element and this vector is taken, i.e., the relative offsets of the intersections between the $n-1$ dimension hyper-planes that intersect the elements and are perpendicular to the randomly oriented vector are computed. As shown in Figure 6.2, an offset between the minimum and maximum dot products is then randomly selected using a uniform random variable. An $n-1$ dimension hyper-plane perpendicular to the randomly oriented vector is then placed at this offset and the values associated with elements on one side of the hyper-plane are swapped between solutions while the values associated with elements on the other side remain in their original solutions.

When carrying out crossover, we take care not to disrupt solution components, e.g., during PE allocation crossover, solution task assignments adapt to the loss of PEs via re-assignment, as described in the next section.

It is our belief that the effects of locality preservation during crossover between solutions with multidimensional representations is an area within evolutionary optimization theory that can have great impact on solution quality but is still poorly understood. Pelikan, Goldberg, and Canú-Paz have noted the difficulty of automatically identifying solution building blocks in order to better preserve locality during crossover and developed an adaptive method of building block identification based on Bayesian networks [130]. After comparing it with a number of alternative methods of preserving locality during crossover, we are satisfied with the performance of the heuristic we have described in this section. However, in the future we hope to consider the problem of locality preserving crossover for multidimensional solution representations in more detail.

6.4 Guided task assignment mutation

A desire for improved performance, especially on problems in which the bandwidth of communication resources is tightly constrained, motivated us to incorporate problem-specific knowledge within EMOGAC's task assignment mutation algorithm. This change also resulted in improved performance for other problem domains. In this section, we describe this guided task assignment mutation algorithm.

As described in Section 5.6, mutation makes randomized changes to task assignments. However, these changes need not be random: they may be guided by problem-specific heuristics. We have developed a guided task assignment mutation algorithm that attempts to minimize PE over-use, task execution time, and communication time. After randomly selecting a task to be reassigned, this heuristic generates an array of

PEs capable of executing it. Three costs are associated with each PE in the solution's allocation: communication time, execution time, and loading.

Communication time is a metric that takes into account the impact of a change to a task's assignment upon the amount of time required to transmit incoming and outgoing data. A task's neighbors are the other tasks with which it communicates, i.e., the tasks connected to it by arcs as shown in Figure 3.2. Let $Q_{a,b}$ be the quantity of data, in bits, transferred along the edge between a task, a, and one of its neighbors, b. Let function $\text{ctime}\,(q, P_a, P_b)$ give an estimate of the amount of time required to transmit q bits of data between the PE, P_a, to which task a is assigned and the PE, P_b, to which task b is assigned. In a distributed system, we approximate the amount of time required to transmit information between a pair of PEs based on the average data transmission rate of the communication resources in that solution's allocation. We previously computed the set of communication resources between each pair of PEs to more accurately approximate communication time. However, the CPU time required for this operation was too costly to justify the potential for improved estimation. In our wireless client-server system synthesis algorithm, described in Chapter 8, we maintain separate average data transmission rates for the communication resources in the client, the communication resources in the server, and the wireless communication resource. In the system-on-chip synthesis algorithm described in Chapter 7, we do not use expected communication time to guide task assignment. Instead, when high-priority communication occurs between a pair of PE's, we position them close together on the integrated circuit to reduce communication time. The communication time $C_{P,T}$ for each PE, P, a task, T, might potentially be assigned to is the sum of the communication times for communication between that task and all of its neighbors, set N_T, i.e.,

$$C_{P,T} = \sum_{i \in N_T} \text{ctime}\,(Q_{T,i}, P, P_i)$$

We attempted defining communication time as the maximum communication time for any neighbor of the task under consideration. However, using a sum instead of a maximum resulted in better solution quality.

In addition to communication time, C_T, we use execution time to prioritize PEs to which a task might potentially be assigned. Execution time is the amount of time required to execute the task on the PE under consideration. Our final prioritization metric is loading, the proportion of a PE's time, in the system hyperperiod, that has already been occupied by the other tasks assigned to it, i.e., if h is the system hyperperiod, T_j is the set of all tasks assigned to PE P, and function etime (P, T) is the time required to execute task T on PE P, then the execution time $E_{P,T}$ for each PE, P, a task, T, might potentially be assigned to is defined as follows:

$$E_{P,T} = \sum_{i \in T_P} \frac{\text{etime}\,(P, i)}{h}$$

Unless all PEs are overloaded, i.e., have a loading greater than or equal to one, overloaded PEs are not considered legitimate targets for task assignment.

Note that we have three metrics for the quality of PEs to which a task's assignment might potentially mutate. We rank candidate PEs by using the Pareto-ranking method described in Section 4.5. We considered using only two costs in this Pareto-ranking: loading and the sum of communication time and execution time. However, we found that leaving communication time and execution time separate until Pareto-ranking resulted in better solutions. After ranking, PEs are sorted by their ranks. We empirically determined that better results were produced when PEs of the same rank were randomly ordered, i.e., EMOGAC does not allow solution encoding to bias task assignment decisions. Once the PEs are ordered, we select one by indexing into the array of PEs using a random variable with a probability density function (PDF) that favors PEs with the highest rank. We tried using a number of different indexing functions but settled on a mathematically elegant approach that produces good results.

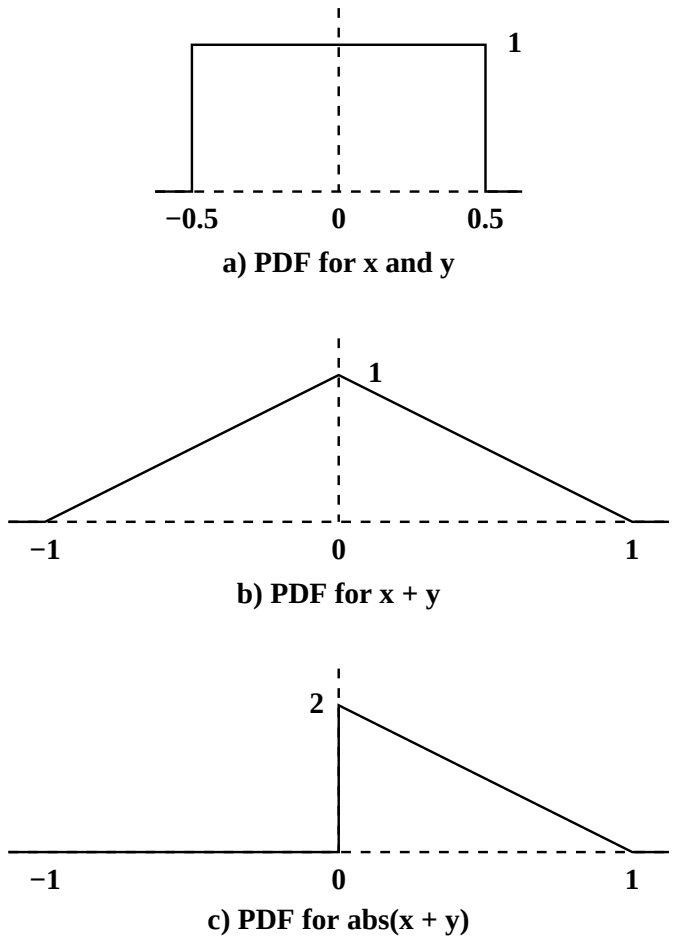

Figure 6.3: Probability density functions

We wanted a function that, given uniform random variable (URV) instances, produces random variable instances with a biased PDF. In addition, we wanted to be able to control the degree of bias toward selecting high-rank PEs. We will now present a function meeting the first requirement. Let x and y be uncorrelated URVs in the range $[\frac{1}{2}, \frac{1}{2})$ with the PDF's shown in Figure 6.3a. Recall that the PDF of the sum of two URVs is the convolution of their PDFs. Therefore, the PDF of two URVs with equal ranges is a pyramid peaking at the sum of their means, as shown in Figure 6.3b. By taking the absolute value of $x + y$ we get the triangular PDF shown in Figure 6.3c. The random

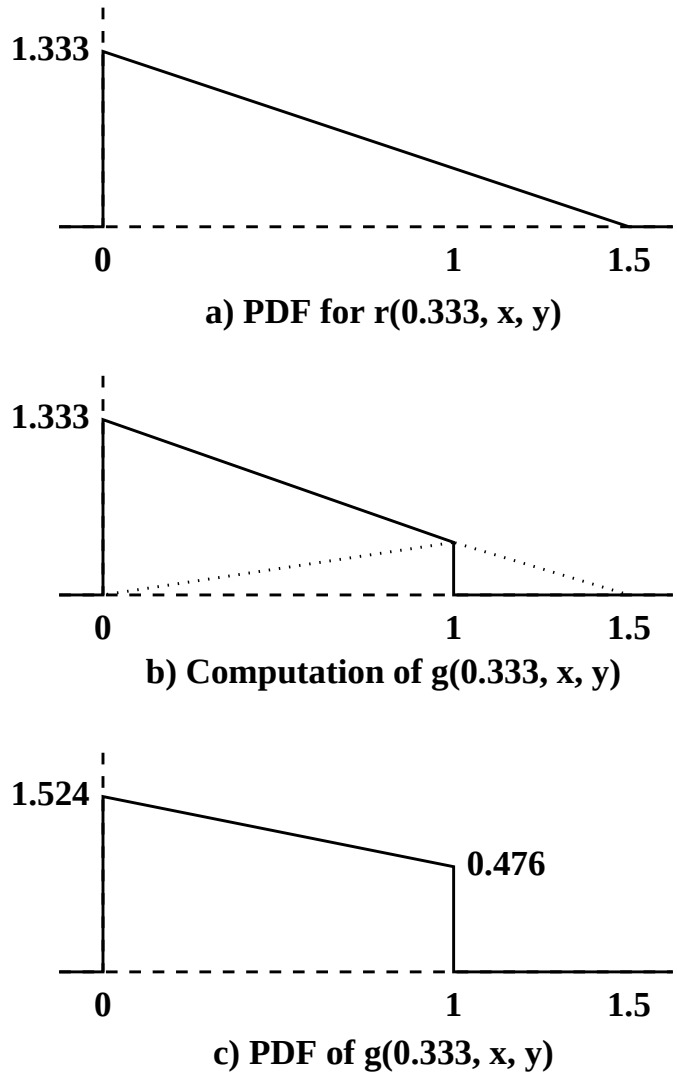

Figure 6.4: Calculation of $\mathrm{g}(\frac{1}{3}, x, y)$

variable instances produced in this manner range from zero to one and are biased toward zero. Thus, the following function satisfies our first requirement:

$$\mathrm{f}(x, y) = \mathrm{abs}\,(x + y)$$

In order to satisfy our second requirement, we introduce a slope control variable, s, with a range of $[0, 1)$. When $s = 0$ we would like function $\mathrm{g}(s, x, y)$ to produce random

variable instances that have a PDF with a slope of 0 in the range $[0, 1)$. When $s = 1$ we would like function $\mathrm{g}(s, x, y)$ to produce random variable instances that have a PDF with a slope of -2 in the range $[0, 1)$. Let

$$\mathrm{smod}(s) = \frac{s+1}{2}$$

and

$$\mathrm{r}(s, x, y) = \frac{\mathrm{f}(x, y)}{\mathrm{smod}(s)}$$

Figure 6.4a shows the PDF for $\mathrm{r}(0.333, x, y)$. Note that the maximum value of this function is greater than one. Now let

$$\mathrm{g}(s, x, y) = \begin{cases} \mathrm{r}(s, x, y) \leq 1 & \textit{if } \mathrm{r}(s, x, y) \\ \mathrm{r}(s, x, y) > 1 & \textit{if } 1 - \frac{\mathrm{f}(x,y) - \mathrm{smod}(s)}{1 - \mathrm{smod}(s)} \end{cases}$$

As illustrated for the case in which $s = 0.333$ in Figure 6.4b, the case in which $\mathrm{r}(s, x, y)$ is greater than one can be intuitively understood as reflecting the tail of the PDF, i.e., the portion for values greater than one, back upon the remainder of the PDF and scaling the tail such that it covers the range $[0, 1)$. Figure 6.4c shows the PDF for $\mathrm{g}(0.333, x, y)$. Note that, given URVs x and y, $\mathrm{g}(s, x, y)$ produces random variables with a PDF bias controlled by s. We use the function $\mathrm{g}(s, x, y)$, multiplied by the number of PEs to which a task might be assigned, to index downward into the rank-sorted array of PEs. In practice, setting $s = \frac{1}{2}$ results in high-quality solutions.

In addition to guiding task assignment mutation, we also probabilistically constrain differences in task assignment mutation between different copies of the same task in the hyperperiod. We allow tasks in different copies of a task graph to be assigned to different PEs, as described in Section 5.7.2. However, we have developed a more flexible way of integrating control of these task assignment probabilities into the evolutionary algorithm. We allow the user to provide a parameter specifying the probability, per task

assignment mutation, that the mutation will affect all of a task's copies instead of only one task copy. This allows arbitrary combinations of task assignments to be explored while making it possible to focus the search on promising area of the solution space in which most copies of a task are assigned to the same PE. The designer may specify the proportion (a value greater than 0.9 works well in practice) of task assignment changes that are made to all copies of a task, and the proportion of the changes that are made to only a single copy.

6.5 Initialization

At the start of the EMOGAC's run, the initial solution pool must be populated. A user-defined number of clusters is created, each of which contains a user-defined number of solutions. Constructive algorithms are used to initialize cluster allocations, communication resource allocations, task assignments, and communication resource connectivities.

In the first step of PE allocation initialization, it is ensured that, for each type of task in the task set, there is at least one PE capable of executing the task. This is accomplished by iteratively finding a task that cannot be executed by any of the PEs in the allocation, and adding a randomly selected PE of a type capable of executing the task. Note that, even after this step, it is still possible that there are too few resources to execute all the tasks in the system before their hard deadlines. In the next step, additional PEs are randomly added until there are sufficient hardware resources to execute all tasks within an amount of time equal to the hyperperiod multiplied by a scalar, h. The value h is proportional to twice the ratio of the index of the cluster to the total number of clusters, i.e., some clusters will have few PEs in their allocation and others will have many. This allocation diversity in the initial solution pool improves optimization.

After a PE allocation has been decided, task assignments are initialized by a two-stage algorithm. In the first stage, information is not yet available about communication times. Therefore, a modified version of the algorithm described in Section 6.4 is used to assign each task to a PE. This algorithm considers all the criteria of the guided task mutation algorithm, with the exception of communication times. After the first stage of task assignment initialization is complete, the second stage reassigns each task using the full guided task assignment mutation algorithm, i.e., it considers the communication times associated with different potential task assignments. Communication resource connectivity is initially random, i.e., each contact of a communication resource is attached to a randomly selected PE.

6.6 Cost calculation

In this section, we describe EMOGAC's cost calculation algorithms. Before a solution's cost may be calculated it is necessary to generate its schedule. The first step in scheduling is task prioritization. Now that task execution times and communication times are known, it is possible to use slack, earliest start time (EST), and latest finish time (LFT) based prioritization. The method of prioritization and scheduling is similar to that used in Section 5.7.1. As in MOGAC, we use negative slack, the difference between a task's EFT and its LFT, to prioritize tasks. However, in EMOGAC, we also use other priority metrics if slack-based prioritization does not produce a solution that meets all its hard real-time deadlines. If slack-based scheduling does not work, EMOGAC also attempts to produce a valid schedule using negative EST, and then negative LFT, as priority metrics. We found that using multiple priority metrics improved solution quality, for some problems.

Although EMOGAC uses multiple prioritization metrics when there is some hope of producing a valid schedule, it is careful to avoid needless scheduling. A PE is overloaded if the sum of the execution times of the tasks assigned to it is greater than the system hyperperiod. EMOGAC does not spend time generating schedules for solutions in which some PEs are overloaded. Instead, it notes the degree to which PEs are overloaded and uses this cost for comparison with other solutions. Avoiding needless scheduling decreases the amount of run-time the synthesis algorithms require without decreasing solution quality.

The other aspects of cost calculation are similar to those described in Section 5.7.3. However, price computation differs slightly. The prices of PEs to which no tasks are assigned, and communication resources that carry no communication events, are not included in the price of the embedded system. Such unused resources play a role in optimization. However, they should not appear in a manufactured embedded system. In addition, each PE has a local memory with a size computed in the manner described in Section 6.1. The price of this memory is determined based on a price per bit value read from the resource database. Each solution has a soft deadline violation proportion cost, defined as the sum of the times by which every copy of every task in the architecture misses its soft deadline, divided by the hyperperiod.

6.7 Solution cache

Every time a solution is changed, it is necessary to determine its new cost. Carrying out cost evaluation every time a solution changes would be the most straightforward approach. However, solution evaluation, which requires scheduling, and might require floorplan block placement (see Section 7.6) and bus topology generation (see Section

7.8), is the most time-consuming operation undertaken by our hardware-software co-synthesis and embedded system synthesis algorithms. In order to avoid needless solution evaluations, EMOGAC maintains a cache of solution cost sets to prevent the re-evaluation of solutions after every modification. In our algorithms, scheduling, floorplanning, and bus topology generation are deterministic. Therefore, for any PE allocation, task assignment, link allocation, and link connectivity, there exists exactly one system cost set. Thus, any solution is characterized by a small amount of information, relative to the amount of information computed during cost evaluation.

Sometimes, solution mutation and crossover produces a solution identical to one for which cost calculation was previously done. In these cases, the solution's cost set is retrieved from a cache, making it unnecessary to carry out cost evaluation. We use a least-recently used (LRU) replacement policy. The cache size is dynamically controlled based on EMOGAC's total memory usage, i.e., we allow more entries to exist if the entries consume little memory. Our experimental results indicate that the cache is usually hit 50% of the time. Its use generally cuts synthesis time in half.

6.8 Benchmarks

In this section, we describe our motivations for constructing an embedded system synthesis benchmarks suite based on the Embedded Microprocessor Benchmark Consortium (EEMBC) benchmarks suite [131], and briefly describe this benchmarks suite.

Our hardware-software co-synthesis and embedded system synthesis algorithms can be viewed as functions that take an embedded system synthesis problem specifications, i.e., a resource databases as well as behavioral and constraint specifications, as their input, and produce embedded system architectures as their output. In order to demonstrate the operation of our synthesis algorithms, it is necessary to provide them with embedded

system synthesis problem specifications. Making these problem specifications public allows other researchers to compare the results produced by their algorithms with those produced by ours.

Acquiring realistic embedded system synthesis problem specifications is difficult. Ideally, we would have access to large industrial problem specifications. A colleague of ours was employed by a company that is heavily involved in embedded systems design. He requested the release of some old specifications to us under a non-disclosure agreement. However, even in this nearly ideal situation, the legal department of his company refused to give us access to the specifications. As a result of the difficulty of getting access to industrial examples, we were left with three other options.

It is possible to find examples in the embedded systems design literature. This approach has the advantage of allowing easy comparison of our algorithms with those designed by other researchers. However, it also has a number of disadvantages. Most examples in the literature are simple, small, and somewhat unrealistic. In addition, they typically assume synthesis software that solves the most basic of hardware-software co-synthesis problems, i.e., power consumption, and issues related to single-chip synthesis are neglected. Nonetheless, we run our algorithms on examples from the literature.

We could hand-generate our own examples. This would potentially allow us to produce larger and more realistic problem specifications than those common in the literature. However, there is a reason for the scarcity of large and realistic problem specifications. Accurately characterizing a large set of resources and specifying the constraints on an embedded system takes a lot of time. For each substantial example, we would be required to do a large portion of the work required to design an embedded system. This would have diverted a great deal of our resources away from research and toward useful but mundane design projects. However, we do use some manually produced embedded system problem specifications for illustrative purposes. In addition, we have gathered

information about hardware resources. This information serves as a starting point for automatically generated examples.

In collaboration with David Rhodes, an algorithm was developed for the automatic generation of embedded system problem specifications (see Appendix A). This algorithm is parametric, i.e., it allows the user to control the general attributes of the resource databases and task sets it generates. This allows the use of some of the characteristics of real processors, real communication resources, and industrial task sets in automatically generated embedded system problem specifications. Automatically generated examples have a number of advantages over examples from the literature and hand-generated examples. One can rapidly generate numerous large, differing, problem instances with similar structural attributes.

In order to ease collaboration in finding and building embedded system synthesis benchmarks, we established a mailing list [102]. We considered the discussions on this list when collecting benchmarks for this dissertation. Good benchmarks motivate better problem formulations and algorithms. We have developed an embedded system synthesis benchmarks suite, called E3S, based on data from EEMBC [131]. The first release of E3S contains 17 processors, e.g., the AMD ElanSC520, Analog Devices 21065L, the Motorola MPC555, and the Texas Instruments TMS320C6203. These processors are characterized based on the measured execution times of 47 tasks, power numbers derived from processor datasheets, and additional information, e.g., die sizes, some of which were necessarily estimated, and prices gathered by emailing and calling numerous processor vendors. In addition, E3S contains communication resources modeling a number of different busses, e.g., CAN, IEEE1394, PCI, USB 2.0, and VME. Our task sets follow the organization of the EEMBC benchmarks. There is one task set for each of the five application suites: automotive/industrial, consumer, networking, office automation, and telecommunications. This benchmark suite has been publicly released

and are available via the E3S link on the http://www.ee.princeton.edu/˜cad/projects.html web page. We make heavy use of E3S in this dissertation.

6.9 Experimental results

In this section, we present the results produced by running EMOGAC on the E3S benchmarks suite, introduced in the previous section, and a number of examples from the literature.

6.9.1 Multiobjective optimization for the E3S benchmarks

Table 6.1 gives the results of running EMOGAC on the E3S benchmarks and simultaneously optimizing price, power consumption, and soft deadline violation proportion. Note that multiple solutions trading off these costs were produced for each benchmark. In Table 6.1, the first column gives the name of the benchmark, the second column gives the price of each solution, the third column gives the average power consumption of each solution, and the fourth column gives the soft deadline violation proportion (see Section 6.6) of each solution. We rounded prices and power consumptions up to the nearest dollar and milliwatt. This table demonstrates that EMOGAC is capable of running on examples containing 47 tasks representing a wide range of embedded application, and 17 commonly used embedded processors. For these problems, multiobjective optimization was valuable, i.e., there were dramatic differences between the prices and power consumptions of different non-dominated solutions. For example, among the different solutions to the Automotive-Industrial problem, prices varied from $169 to $652, power consumptions varied from 167 mW to 184 mW, and soft deadline violation proportions varied from 1.13 to 2.08.

Table 6.1: Multiobjective optimization for the E3S benchmarks

Example	Price ($)	Average power (mW)	Soft DL viol. prop.
Automotive-Industrial	169	167	2.08
	453	140	1.50
	530	316	1.05
	652	182	1.24
	652	184	1.13
Networking	57	72	1.31
	70	101	1.23
Telecom	291	1569	4.58
	291	1666	4.57
	378	2098	3.18
	379	1974	3.44
Consumer	155	298	1.57
	176	378	1.46
	229	351	1.52
	365	355	1.42
Office Automation	66	55	0.02
	127	449	0.01
	184	440	0.01
	215	273	0.01

6.9.2 Price-only optimization for examples from the the literature

Table 6.2 shows the results of running EMOGAC on all of Prakash and Parker's SOS examples [76]. The first column shows the names of the examples. The second column shows the prices of the solutions found by EMOGAC. The other columns show the prices of the solutions found by algorithms developed by other researchers. The third column is for SOS, Prakash and Parker's mixed integer-linear programming (MILP) algorithm [76]. This algorithm has the advantage of guaranteeing optimality. However,

Table 6.2: Prices for Prakash and Parker's examples

Example ⟨performance⟩	EMOGAC	SOS	COSYN	Oh & Ha's algorithm
P&P 1 ⟨2.5⟩	14	14	n.a.	n.a.
P&P 1 ⟨3⟩	13	13	n.a.	n.a.
P&P 1 ⟨4⟩	7	7	n.a.	7
P&P 1 ⟨7⟩	5	5	5	5
P&P 2 ⟨5⟩	14 (15) †	15	n.a.	n.a.
P&P 2 ⟨6⟩	12	12	n.a.	n.a.
P&P 2 ⟨7⟩	7 (8) †	8	n.a.	n.a.
P&P 2 ⟨8⟩	7	7	n.a.	7
P&P 2 ⟨15⟩	5	5	5	5
P&P 3 ⟨6⟩	10	10	10	n.a.
P&P 3 ⟨7⟩	6	6	n.a.	n.a.
P&P 3 ⟨15⟩	5	5	5	n.a.
† See Figure 6.5.				

its run-time increases dramatically with increasing problem complexity. The fourth column is for COSYN, a constructive algorithm, [128]. The fifth column is for Oh and Ha's heuristic [75]. Entries of n.a. indicate that a result for the corresponding problem and optimization algorithm was not reported in the literature. The P&P 2 ⟨5⟩ and P&P 2 ⟨7⟩ entries are explained in the next few paragraphs.

Prakash and Parker's examples contained no soft deadlines or power information. Therefore, we ran EMOGAC in single-objective price optimization mode. We used the same optimization parameters for each of these examples, and for those in the next subsection. A 1.4 GHz AMD Athlon Thunderbird CPU was used to solve these problems. Each example took between 12 and 35 minutes of CPU time. Note that it is possible for EMOGAC to produce good solutions to the simpler Prakash and Parker examples in significantly less than 12 minutes of CPU time. However, we wanted to use the same optimization parameters for all of Prakash and Parker's examples, as well as all of Hou

- Deadline 5: For this example, SOS's optimal solution to the problem contains exactly one pair of back-to-back links. A bidirectional communication model might allow one of these links to be removed, thereby reducing the solution price by, at most, one unit. Our algorithm arrived at such a solution. If one were to re-insert that link, the solution price would be 15, which we have shown in parenthesis in Table 6.2.
- Deadline 6: SOS's optimal solution to this problem contains no back-to-back links. A bidirectional communication model will not make a link redundant, allowing it to be removed.
- Deadline 7: SOS's optimal solution to this problem contains exactly one back-to-back pair of links. A bidirectional communication model might allow one of these links to be removed, reducing the solution price by, at most, one unit. Our algorithm arrived at such a solution. If one were to re-insert that link, the solution price would be 8, which we have shown in parenthesis in Table 6.2.
- Deadline 8: SOS's optimal solution to this problem contains no back-to-back links. A bidirectional communication model will not allow any improvement to the solution.
- Deadline 15: SOS's optimal solution to this problem contains no point-to-point communication links. A bidirectional communication model will not make a link redundant, allowing it to be removed.

Figure 6.5: Impact of difference on communication model for P&P 2 example.

and Wolf's examples. Therefore, we selected a solution pool size and halting conditions sufficient for more complicated problems, i.e., we granted the optimization algorithm more CPU time than was necessary for simple problems so that it would have good performance on complicated problems. The dependence of EMOGAC's CPU time requirements upon problem complexity stand in contrast with the requirements of SOS. Although SOS took only 11 CPU seconds, on a Solbourne Series5e/900 (similar to a

SPARC 4/490), for a simple problem, P&P 1⟨2.5⟩, its run time increased dramatically with increased problem complexity; it took 106.7 hours of CPU time for P&P 2 ⟨15⟩. That there isn't any particular problem in taking a large amount of CPU time to solve a problem well. However, dramatic increases in optimization time with increasing problem complexity imply that an algorithm may not halt in an acceptable amount of time for large problems.

Prakash and Parker's behavioral specifications are somewhat unconventional. They contain tasks with pre-computation and post-computation. We used the method described in Section 3.5 to precisely model this. Our model does vary from that used by Prakash an Parker in one way: our point-to-point communication links are bidirectional and theirs are directed, i.e., they allow communication to occur over a point-to-point link in only one direction during the life of an embedded system. In our model, communication via a point-to-point link can occur in either direction, although only one communication event can be carried by the point-to-point link at a time. This results in some apparently unusual results for the P&P 2 ⟨5⟩ and P&P 2 ⟨7⟩ examples. Our slightly different model for point-to-point communication links allows our algorithm to get lower prices than SOS in a few instances. Although a price comparison is still legitimate, it requires some explanation. We will now describe the impact of this difference on the solutions to each of the Prakash and Parker examples.

None of the solutions produced by SOS for any of the deadlines associated with the P&P 1 examples contain a pair of point-to-point communication links that connect the same pair of PEs and have different directions, i.e., *back-to-back links*. Therefore, a bidirectional communication model will not allow back-to-back links to be merged, thereby reducing price, in any of these solutions. For every P&P 1 example, EMOGAC arrived at a solution with the same price as SOS. The P&P 3 example resource database does not contain directed point-to-point communication links. As a result, the difference

Table 6.3: Optimization for Hou and Wolf's examples

Clustering	Example ⟨performance⟩	EMOGAC	COSYN	Yen's algorithm	Oh & Ha's algorithm
Unclustered	H&W 1&2	140	170	170	170
	H&W 1&3	170	170	240	170
	H&W 3&4	140	n.a.	210	170
Clustered	H&W 1&2	140	n.a.	170	170
	H&W 1&3	170	n.a	170	n.a.
	H&W 3&4	170	n.a.	170	n.a.

in communication models has no impact on the results for this example. EMOGAC arrived at a solution with the same price as SOS for every P&P 3 example. In Figure 6.5, we describe the impact of the difference between our communication model and that used by SOS upon the results for the P&P 2 problems.

EMOGAC produced a solution with the same price as SOS's optimal solution for every example in which a bidirectional communication model would not allow a pair of back-to-back links to be merged in the optimal solution produced by SOS. In cases for which our communication model could potentially allow point-to-point communication links to be merged, thereby reducing price, our algorithm arrived at a solution that had a price exactly equal to that of SOS, minus the savings that might result from merging of back-to-back links. In practice EMOGAC finds solutions that are substantially equivalent to those produced by SOS, an optimal algorithm, although EMOGAC has CPU time requirements that do not increase rapidly with increasing problem complexity.

Table 6.3 compares the results produced by running EMOGAC on Hou and Wolf's examples [84] with those produced by other hardware-software co-synthesis algorithms. The first column states whether or not the example in question is clustered. Clustering is described in Section 5.9.1. Clustered graphs have a similar structure to unclustered

graphs but contain fewer tasks. The second column contains the names of Hou and Wolf's examples. The third column shows the prices of the solutions produced by EMOGAC. The other columns show the prices of the solutions found by algorithms developed by other researchers. The fourth column is for COSYN [128]. The fifth column is for Yen's iterative improvement algorithm [81]. The sixth column is for Oh and Ha's heuristic [75]. Entries of n.a. indicate that the algorithm's developers did not report a result for the given problem and optimization algorithm.

These examples contained no soft deadlines or power information. Therefore, we ran EMOGAC in single-objective price optimization mode. We did not contract the periods and and deadlines of these examples in order to reduce the hyperperiod: these are precisely Hou and Wolf's example. We used the same optimization parameters for each of these example, and for the examples in the previous subsection. These examples were run on a 1.4 GHz AMD Athlon Thunderbird CPU. Each example took approximately 10 CPU minutes, with the exception of H&W 1&3 unclustered, which took 74 CPU minutes. For all of Hou and Wolf's problems, EMOGAC arrived at solutions with prices that are equal to or lower than those produced by past work.

It is interesting to note the implications of these results for clustering research. Task clustering converts a task graph into another task graph with fewer nodes by grouping some nodes together and treating them as a single node. This has the potential to improve the solutions produced by a co-synthesis algorithm by eliminating unpromising areas from the search space. For example, if all of a problem's promising solutions assign two tasks to the same PE and schedule them concurrently, converting them into a single task will concentrate a search on the most promising areas of the solution space. However, although task clustering can simplify a hardware-software co-synthesis problem and eliminate unpromising potential solutions from the search space, it can also

eliminate promising solutions from the search space. Note that for the Hou 3&4 example, EMOGAC was able to find a superior solution to the unclustered version of the problem. For the unclustered version of this problem, EMOGAC found a solution with a price of 140. However, for the clustered version, such a solution is not possible. In this example, clustering forced tasks that would ideally be assigned to different PEs to be assigned to the same PE. It is important for a clustering algorithm not to eliminate the possibility of finding a good solution in its attempts to simplify a problem. Task clustering is a method of simplifying a behavioral specification and pruning unpromising areas from the solution space. It is our opinion that the best place to carry out such pruning and simplification is within a synthesis algorithm, when additional information is available about allocation and assignment, not as a pre-pass.

6.10 Conclusions

In this chapter, we have described a number of enhancements to our evolutionary optimization algorithm and hardware resource models. We presented a new embedded system synthesis benchmarks suite containing realistic models of 17 processors running 47 different types of embedded system tasks. Finally, we gave the results of running our optimization infrastructure on these benchmarks and compared the quality of solutions produced by our optimization algorithm with solutions presented in past work. When run on the specification in the E3S benchmark suite, EMOGAC produces multiple solutions that trade off different architectural costs. When run on problems from the literature, EMOGAC meets, and very often beats, the results produced by other hardware-software co-synthesis algorithms, without requiring an long run-time for difficult problems.

Chapter 7

Intellectual Property Core-Based System-on-Chip Synthesis

In this chapter, we present a system synthesis algorithm, called MOCSYN, that partitions and schedules embedded system specifications to intellectual property cores in an integrated circuit. Given a system specification consisting of multiple periodic task graphs as well as a database of core and integrated circuit characteristics, MOCSYN synthesizes real-time heterogeneous single-chip hardware-software architectures using an adaptive multiobjective genetic algorithm that is designed to escape local minima. As shown in the previous chapter, the use of multiobjective optimization allows a single system synthesis run to produce multiple designs that trade off different architectural features. Integrated circuit price, power consumption, and area are optimized under hard real-time constraints. MOCSYN differs from previous work by considering problems unique to single-chip systems. It solves the problem of providing clock signals to cores composing a system-on-chip (SOC). It produces a bus structure that balances ease of layout with reduction of bus contention. In addition, it carries out floorplan block placement within its inner loop, allowing accurate estimation of global communication delays and power consumption.

7.1 Motivation

It is possible to implement some embedded systems using a single integrated circuit (IC), thereby reducing cost and improving performance [132]. Economic and time pressures frequently make it impractical to do an in-house design for each component in a single-chip system. Fortunately, the number of intellectual property (IP) cores available from the industry has dramatically increased in the past few years. Numerous companies and non-profit organizations offer a wide range of IP cores, e.g., protocol processors, general-purpose processors, micro-controllers, digital signal processors (DSPs), memory, and application-specific hardware (e.g., Data Encryption Standard engines) [133].

MOCSYN, which stands for multiobjective core-based single-chip system synthesis, differs from past work on system synthesis by considering a number of problems unique to core-based single-chip systems. MOCSYN determines the clock frequencies supplied to different cores. It generates priority-based bus structures of arbitrary topology, balancing ease of routing and bus contention minimization. In addition, it conducts floorplan block placement [134] within its inner loop, thereby determining the location of each core and allowing estimates of global wiring delays and power consumption to be used during scheduling and cost calculation. Experimental results demonstrate that a global bus is, in general, inferior to the use of a priority-based arbitrary bus topology. Conducting block placement in the inner loop generally results in an improvement in solution quality when compared with worst-case or best-case communication delay estimates.

The rest of this chapter is organized as follows. In Section 7.2, we describe the model MOCSYN uses for IP cores. Section 7.3 gives an overview of the SOC synthesis algorithm. Section 7.4 describes MOCSYN's clock selection algorithm. Section 7.5 describes the way we determine how important it is for each pair of IP cores to be placed near each other on the SOC. In Section 7.6, we describe the algorithm we use

to determine floorplan block placements. Section 7.7 describes the model MOCSYN uses for wire delay and power consumption estimation. Section 7.8 describes our bus topology generation algorithm. In Section 7.9, we describe our method of calculating a SOC's costs. We give experimental results and conclude in Sections 7.10 and 7.11.

7.2 IP core model

In this section, we describe MOCSYN's model for IP cores. A core is a processing element (PE) that is capable of executing one or more types of tasks. Multiple cores may be located on the same IC, upon which multiple tasks may execute simultaneously. Note that MOCSYN uses a model for cores and ICs that is substantially more complicated, and powerful, than the one used by MOGAC (described in Chapter 5. The following information establishes the relationship between tasks and MOCSYN's cores:

- A two-dimensional array indicating the relative worst-case number of execution cycles of each task on each core.
- A two-dimensional array indicating the energy consumption per cycle of each task on each core.
- A two-dimensional array indicating the core types upon which each task type may be executed.
- A two-dimensional array indicating the amount of code memory required for each task type executed on each core.

In addition, each core has a price that corresponds to the royalties paid to the IP producer for each fabricated instance. This price is zero for royalty-free IP cores. If

IP has a one-time fee instead of, or in addition to, a per-use royalty, the price is equivalent to the one-time fee divided by the expected production volume. Each core has a width, a height, a maximum clock frequency, a variable indicating whether or not its communication is buffered, an energy consumption per cycle spent in communication, an idle energy consumption per cycle, and a global routing layer density that is used when estimating routability.

7.3 Algorithm overview

In this section, we give a high-level description of the MOCSYN algorithm. This algorithm carries out the following tasks:

1. Determine a *clock frequency* for each core type, subject to tradeoffs between execution time and power consumption.

2. Determine the *allocation* of cores to use.

3. Determine the tasks to *assign* to each core, subject to tradeoffs between ease of routing and minimization of bus contention.

4. Determine a *bus structure* to use on the IC.

5. Derive a *block placement* for the cores, allowing an estimation of wire delay, wire power consumption, and silicon area.

6. *Assign* each communication event to a bus.

7. *Schedule* the tasks on the cores and the communication events on the communication links.

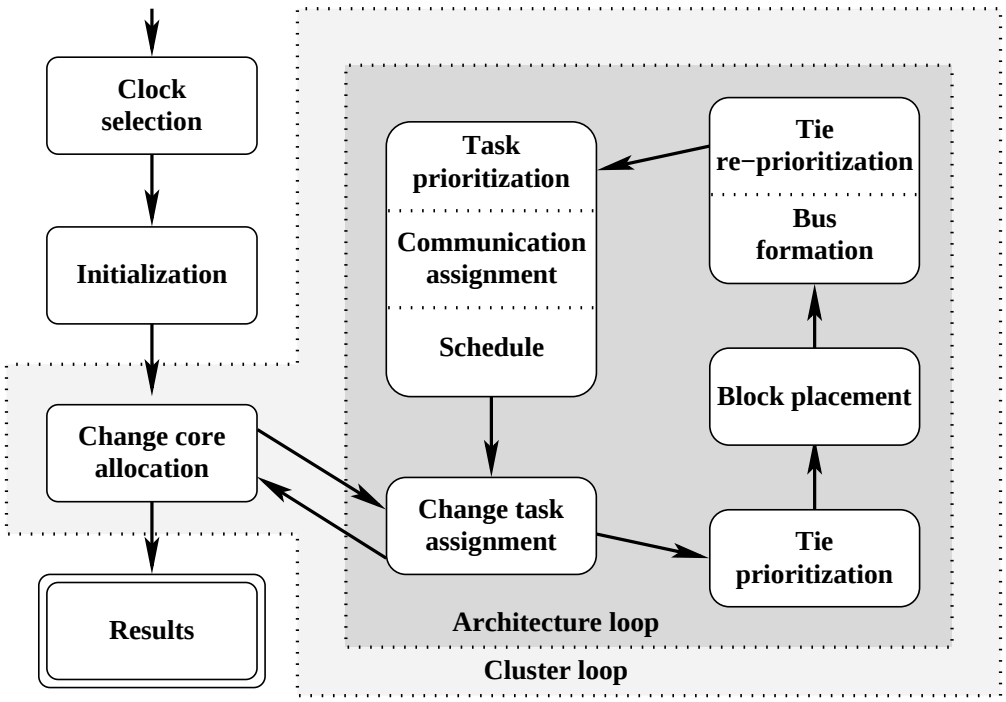

Figure 7.1: MOCSYN overview.

MOCSYN uses a parallel recombinative simulated annealing (PRSA) algorithm to optimize embedded system architectures. An overview of this algorithm is shown in Figure 7.1. Initially an optimal, but potentially slow, algorithm determines the clock frequency to provide to each core. Basic data structures are then initialized. MOCSYN is a hierarchical algorithm. After this phase has been repeated an arbitrary (user-selectable) number of times, an attempt is made to improve the core allocation of a cluster of architectures, i.e., a collection of architectures that share the same core allocation but may have different task assignments (see Section 5.5). Within the architecture optimization loop, a number of deterministic algorithms are used to concurrently evaluate the core allocation and task assignment of each architecture. First, a priority is assigned to each *communication tie*, i.e., the communication carried out between each pair of cores. These priorities are used to generate a block placement for the cores, ensuring that core

pairs for which communication priority is high are located near each other. Ties are re-prioritized based on global wiring delay information that is extracted from the block placement. During embedded system design, a synthesis-time or run-time scheduler prevents multiple communication events from being scheduled to the same communication resource at the same time. Contention occurs when one communication event's transmission blocks the transmission of another communication event during scheduling. A bus structure that trades off potential bus contention for ease of routing is produced. After bus structure generation, tasks are prioritized and a schedule is generated for the tasks assigned to each core. Communication events are concurrently assigned to, and scheduled on, busses. At the completion of each architecture optimization loop, changes are made to the task assignments in an attempt to improve them. At the completion of each cluster optimization loop, changes are made to the core allocations in an attempt to improve them. Initialization, changes to processing element (core) allocation, changes to task assignments, and scheduling were described in Chapters 5 and 6. The remaining internal algorithms shown in Figure 7.1 are described in the following sections.

7.4 Clock selection

In this section, we discuss the problems associated with selecting a clock frequency for each core in an IC and describe the algorithm used in MOCSYN to solve these problems.

An IC's global clocking can be single-frequency synchronous, multi-frequency synchronous, or asynchronous [31], [135]. Single-frequency synchronous global clocking has the potential to keep communication overhead at a minimum. However, its use requires that all the cores that communicate with each other be clocked at the same frequency. When different cores have different maximum frequencies, all cores must

be clocked at a frequency less than or equal to the maximum frequency of the slowest core. Thus, using a single-frequency synchronous communication protocol will generally force sacrifices in core speed. Multi-frequency synchronous communication allows cores with different clock periods to communicate with each other at a rate proportional to the LCM of the communicating core's periods. Unfortunately, when cores have different minimum periods and efforts are made to allow each core to run near its maximum frequency, the LCM of the periods of communicating cores can be significantly higher than the period of any individual core, e.g., LCM$(5, 7) = 35$. This generally results in slow communication. It is possible to significantly accelerate semi-synchronous communication between clock domains if their frequencies are related by rational numbers with small integer numerators and denominators, i.e., the LCM of the frequencies is small [136]. Finally, one may use asynchronous communication, clocking cores at arbitrarily different frequencies and relying on asynchronous circuits to facilitate communication between them. Although it has a reputation for increasing communication overhead, we believe that it is a good option for systems in which different cores are clocked at mostly unrelated frequencies. Using asynchronous communication, speed is bounded only by bus bandwidth, the rate at which communicating cores can transmit and receive information, and some protocol overhead; constraints need not be placed on the relative frequencies and phases of different cores . Using asynchronous communication has the additional advantage of making inter-core clock skew irrelevant. Past work has developed asynchronous communication approaches using clock pausing [137], and first-in first-out memories (FIFOs) [138]. Other work has provided a framework for automatically synthesizing asynchronous interface protocols [135].

If one decides to use asynchronous communication, the selection of clock frequencies for the cores comprising a single-chip system need not be constrained by communication considerations. However, there are a number of other problems that must be dealt

with. Supplying each core with an arbitrary clock frequency would require a large number of frequency generators, e.g., analog timers based on RC delay or crystal oscillation. These components are difficult to integrate with conventional CMOS IC processes. Using discrete components is a poor option because each additional external component increases the price and area of an embedded system. Thus, a clocking approach that requires only one frequency source but allows nearly arbitrary frequencies to be delivered to each core would be advantageous.

We use an approach in which a single external oscillator is used to supply a base frequency. A cyclic counter or interpolating clock synthesizer associated with each core is used to divide this frequency by an integer, in the case of a cyclic counter, or multiply the frequency by a rational number, in the case of an interpolating clock synthesizer [139]. Note that frequencies generated in this manner can easily satisfy the requirements for semi-synchronous communication [136] as well as asynchronous communication. A description of the clock selection algorithm used in MOCSYN follows. This algorithm is capable of dealing with interpolating clock synthesizers. The cyclic counter clock selection problem is a special case of the interpolating clock synthesizer clock selection problem. Therefore, the algorithm used in MOCSYN is capable of solving either problem.

Given: A maximum external clock frequency, $Emax$, and a maximum frequency associated with each of the n cores $\{Imax_1, Imax_2, \ldots, Imax_n\}$.

Each core's clock frequency multiplier is a rational number, $M_i = N_i/D_i$, with a positive integer numerator N_i less than or equal to a user-supplied maximum, $Nmax$, and a positive integer denominator, D_i. A core's internal frequency, I_i, is equal to the external frequency, E, multiplied by its multiplier, M_i.

MOCSYN maximizes the average of the ratios of the core frequencies, I_i, to the core frequency maxima, $Imax_i$, i.e.,

$$\sum_{i=1}^{n} I_i / Imax_i$$

Our quality metric, $q \in [0, 1]$, is the average, over all cores, of the ratio of each core's actual frequency to its maximum frequency, i.e.,

$$q = \frac{\sum_{i=1}^{n} I_i / Imax_i}{n}$$

Thus, when each core is running at its maximum frequency, $Imax$, quality is one. When each core is running at an extremely low frequency, quality approaches zero. We initially considered using a quality metric that weights the relative contribution of each core type by its number of instances in the core allocation, or by how heavily it is used by the tasks eventually assigned to it. However, as shown in Section 7.10.1, we found that it is possible to clock all cores at nearly their maximum frequencies before deciding core allocation or task assignment. This allows us to remove clock selection from MOCSYN's inner loop and conduct it in a pre-pass.

It is simple to determine an optimal external frequency, E, if the value of each multiplier, M_i, is known. Given that $E < Emax$, for an optimal E, $\exists i \in \{1, 2, \ldots, n\}$ such that $I_i = Imax_i$. Thus, one need only consider a small set of E's.

For a given set of multipliers, at least one core's internal frequency will be equal to that core's maximum frequency if the external frequency is optimal. This is obviously the case because, if each internal frequency is below its corresponding maximum, it would be possible to increase the external frequency until one internal frequency is equal to its maximum; this would be guaranteed to result in an increase in *quality*. The maximum external frequency that does not result in any core's internal frequency exceeding that core's maximum internal frequency is the optimal external frequency. It is guaranteed that there exists at least one external frequency that does not result in any

core's internal frequency exceeding the core's maximum internal frequency, because the first external frequency selected by the clock selection algorithm is less than the maximum internal frequency of any core. It is still necessary to determine an optimal set of multipliers. It is obvious that, for any given pair of internal frequencies, if the first is greater than or equal to the second, then the optimal multiplier associated with the first is greater than the multiplier associated with the second. This observation allows the solution space to be pruned.

We now restate the preceding paragraph more formally. Given that $Eopt_i$ is the optimal E for core i, for $\forall i \in \{1, 2, \ldots, n\}, Eopt_i = Imax_i/M_i$. The $\max_{i=1}^{n} Eopt_i$ for which $\nexists_{j=1}^{n} I_j > Imax_j$ is the optimal E for a given set of M's. It is guaranteed that at least one E for which $\nexists_{j=1}^{n} I_j > Imax_j$ exists because the first E chosen by the clock selection algorithm is less than or equal to the $Imax$ value of every core. The only remaining problem is to determine an optimal set of M's. It is obvious that, for any given pair of $Imax$'s, $Imax_a$ and $Imax_b$, if $Imax_a \geq Imax_b$ then an optimal $M_a \geq M_b$. This observation allows the solution space of M's to be pruned.

Initially, all D's are equal to 1 and all N's are equal to $Nmax$. Therefore, all M's are equal to $Nmax$. To maximize the average of core frequency to maximum frequency ratios, one need only repeatedly execute a simple algorithmic kernel, while keeping track of the best set of M's, until $E > Emax$. This kernel is shown in Figure 7.2. Although, given that the maximum and minimum of the set $\{Imax_1, Imax_2, \ldots, Imax_n\}$ are $Imax_a$ and $Imax_b$, respectively, this algorithm takes $\mathcal{O}\left(n \cdot Nmax \cdot Imax_a/Imax_b\right)$ time, in practice it is fast (see Section 7.10).

Linear interpolating clock synthesizers are compatible with standard digital design tools and processes. Their use provides a significant advantage: one can distribute a base global clock frequency that is below the maximum local clock frequencies, thereby reducing power consumption in the global clock distribution net. However, interpolating

For each i between 1 and n, inclusive,
there is an array, A_i, of size $Nmax$,
that contains integers.

Each of these integers is the current denominator
for the numerator equivalent to its index.

Optimize E for the current M's.

For all i's between 1 and n, inclusive, if $I_i = Imax_i$:
    j ranges from 1 to $Nmax$, inclusive
    Find the j for which $j/(A_{ij} + 1)$ is maximal
    Increment A_{ij}
    Set $D_i \leftarrow A_{ij}$
    Set $N_i \leftarrow j$

Figure 7.2: Clock selection kernel.

clock synthesizers are more complicated than cyclic counters. In addition, they are likely to require more area [139]. If one chooses to use cyclic clock division counters, instead of linear interpolating clock synthesizers, the same clock selection algorithm is used. However, $Nmax$ is set to 1.

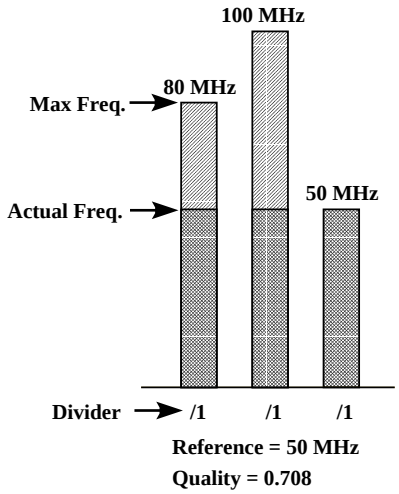

Figure 7.3: Clock selection example initial condition.

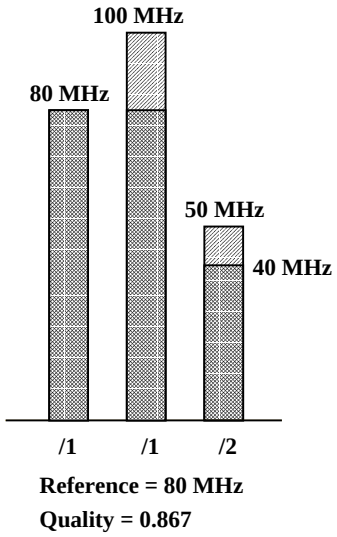

Figure 7.4: Clock selection example first iteration.

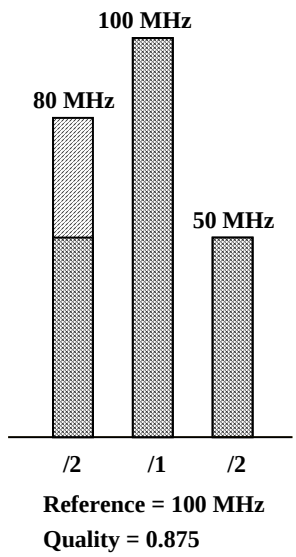

Figure 7.5: Clock selection example second iteration.

For the sake of example, consider a set of three cores that have maximum internal frequencies ($Imax_1, Imax_2, Imax_3$) of 80 MHz, 100 MHz, and 50 MHz. For the sake of simplifying this example, we will consider the use of counter dividers only, not interpolating clock synthesizers. This implies that each core's multiplier (M_i) is equal to the reciprocal of its counter divider value (D_i), i.e., $M_i = \frac{1}{D_i}$. Initially, the counter dividers for each core (D_1, D_2, D_3) are set to one and the external frequency (E) is set to the minimum of the cores' internal maximum frequencies, i.e.,

$$E \leftarrow \min_{i=1}^{3} Imax_i$$

Therefore, as shown in Figure 7.3, the quality is initially

$$\frac{\frac{50MHz}{80Mhz} + \frac{50MHz}{100MHz} + \frac{50MHz}{50MHz}}{3} = 0.708$$

In the next step, the cores for which $I_i = Imax_i$ are located and their dividers (D_i) are incremented. As shown in Figure 7.4, the divider of the core on the far right is incremented. The external frequency (E) is then increased until $\exists i \in \{1, 2, 3\}$ such that $Imax_i = I_i$. In this case, E is increased to 80 MHz, resulting in a quality of 0.867.

The iterative kernel in Figure 7.2 is executed again, incrementing the divider of the core at the far left in Figure 7.5. E is increased to 100 MHz, resulting in a quality of 0.875. When E reaches $Emax$, the algorithm terminates and returns the highest-quality configuration encountered.

7.5 Tie prioritization

This section describes the algorithm used by MOCSYN to prioritize communication ties between pairs of cores. These priorities are used by the floorplanner to determine which pairs of cores should be placed closes to each other. In addition, it is re-calculated after floorplanning and used in the generation of a bus structure. Ties of high priority are likely to get their own point-to-point links. During tie prioritization, task assignments, and therefore task execution times, are known. Tie priority determination is conducted before block placement and bus structure generation. Therefore, exact communication times are not yet known during tie prioritization. For communication between different cores, communication time estimates are based on the average expected separation between pairs of cores in a square grid containing the same number of cores as the solution for which tie priorities are calculated. Each core in the grid is a square with height and width equal to the average of the heights and widths of the cores in the solution. For cases in which the square root of the number of cores is not an integer, the average core separation distance is interpolated. These estimates are used to guide floorplanning, ensuring that cores between which a large amount of low-slack communication occurs are placed near each other. The communication delays are re-computed after floorplanning block placement has been completed in order to estimate them more accurately.

Slack is the difference between the earliest finish time and latest finish time of a task. Thus, it is the amount of time by which a task's execution may be delayed, from its earliest possible execution time, without causing any other tasks to miss their deadlines. Earliest finish times are computed by considering task execution times and estimated communication delays during a topological search of the task graph, starting from the node with no incoming edges. Latest finish times are computed by conducting a backward topological search of the task graph, starting from the nodes that have deadlines.

Task graph edges, which signify communication, have a slack equivalent to the minimum of the slacks of the tasks they connect. A tie's priority is the average of the negative slacks of the task graph edges (communication events) along it, or zero if there are no edges along it. Given that p is tie priority, C is the set of communication events along the tie with size n, and s is slack, then

$$p = \begin{cases} \sum_{i \in C} \frac{-s_i}{n} & if\ n > 0 \\ 0 & if\ n = 0 \end{cases}$$

We did not settle on this definition of priority easily. Numerous experiments revealed that negative slack average, with zeros for unused ties, resulted in better quality solutions than prioritizing ties by their communication event data quantities, communication times, negative slacks, communication durations minus slack times, and numerous other metrics. In some cases this was counter-intuitive, e.g., using a large-magnitude negative priority, instead of zero, for unused ties resulted in a degradation in architecture quality.

7.6 Floorplan block placement

This section describes the block placement algorithm used within MOCSYN's inner loop. This algorithm is built upon the work of other researchers.

Initially, it is necessary to determine the layout shape of each core. Although the raw IP core layout shapes are specified in the resource database input to MOCSYN, the area of memory required by each core must be determined during the run of the algorithm. The memory requirements of each core depend on the tasks assigned to the core. We calculate the code and data memory requirements of each core, using the method described in Section 6.1. After determining a core's memory requirements, we compute the area required to implement this memory and generate a new core layout shape that has the same aspect ratio as the core and an area equal to the sum of the core's area and its associated memory area, thereby ensuring that there is sufficient area reserved for each core's memory to be located near or within it. These core-memory layouts shapes are used in floorplan block placement.

It is possible for some of the cores within a cluster's allocation to be unused by some of the solutions within the cluster. During floorplanning, a solution's unused cores are given insignificant sizes so that they do not interfere with floorplanning.

After core areas have been calculated and unused cores have been omitted, a balanced binary tree of cores is formed, based on tie priorities, i.e., the priorities of communication between core pairs. Accounting for the priority of communication is an extension of the historical algorithm, which considered only the binary presence or absence of communication [140]. As a result, the time complexity of the partitioning algorithm is increased from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^2 \cdot \log n)$ where n is the number of cores. Cores that are adjacent in the binary tree will be adjacent in the final block placement. After forming the binary tree, MOCSYN optimally determines the orientations of all of the cores, under the constraint that the aspect ratio, i.e., the ratio between width and height, does not exceed a value specified by the user. Padding is added to orientations that do not conform to the desired aspect ratio. However, the additional area resulting from padding is counted in the block placement area; this discourages the selection of

block placements with poor aspect ratios. Under these conditions, IC area is minimized. This algorithm is based on past work and takes $\mathcal{O}(n \cdot \log n)$ time where n is the number of cores [141].

7.7 Wiring delay and power consumption model

We have used the approach proposed by Cong and Pan to model communication delay and energy [2]. In this section, we summarize the parameters required for the buffered wire model. We follow the parameter naming conventions used by Cong and Pan, although we use basic units instead of scaled units, i.e., we use 10^{-6} m instead of 1 μm.

- $width_{max}$: the maximum buffer width multiplier
- V_{hi}: the voltage for a logical one (logical zero is 0 V)
- W_{min}: the minimum wire width in meters
- S_{min}: the minimum wire spacing in meters
- r: the sheet resistance in $\Omega/\square$
- c_a: the unit area capacitance in Farads/meter^2
- c_f: the unit effective-fringing capacitance in Farads/meter
- t_g: the intrinsic gate delay in seconds
- c_g: input capacitance of a minimum-sized gate in Farads
- r_g: output resistance of a minimum-sized gate in Ω

Using these parameters, and our own implementation of Cong and Pan's algorithms, we model optimal buffer insertion and wire sizing under the constraints supplied by the designer. We compute delay, as well as driven wire and buffer capacitance, as linear functions of wire length. Wire and buffer capacitances are used to compute power consumption. The linear model allows rapid delay and energy estimation during cost calculation [2].

7.8 Bus topology generation

This section describes the algorithm used by MOCSYN to produce an arbitrary bus structure. Before bus formation, MOCSYN recalculates tie priorities using an algorithm similar to that described in Section 7.5. The global wiring delay information extracted from the floorplan block placement, however, is now available, allowing an accurate estimation of communication time during this recalculation.

7.8.1 Motivation

The objective, during bus formation, is to minimize the probability of bus scheduling contention under the constraint that the bus structure is routable. If routability were not a concern, point-to-point communication resources could be used between every pair of cores, eliminating contention. However, this solution might be un-routable, especially in SOCs containing numerous cores. Therefore, it is necessary to establish some metric of routability. There has been little research on the problem of estimating routability, although some have looked at related problems. Wang and Sarrafzadeh do block placement in a way that minimizes the number of crossings between rectilinear layout regions and nets [142]. Unfortunately, their approach requires fairly precise knowledge of global routing paths. There are widely used commercial implementations of global

and detailed routing tools. Therefore, we defer precise global routing, and detailed routing, to other software instead of re-implementing mature algorithms. As a result, we cannot safely make assumptions about the detailed paths taken by global wires.

7.8.2 Definitions and assumptions

We assume that rectilinear global routing is used. In addition, two-layer routing is assumed. Our algorithm's parameters may be adjusted to account for the availability of additional metal layers for global routing.

In order to make an estimation of congestion that does not depend on the precise paths taken by global wires, we define the term density. *Density* is the ratio between the area of routing metal in a region to the total area of the region. We use the maximum density of a layout and bus topology as a proxy for routability. This allows us to take into account flexibility in routing paths.

One can estimate the wire length of a bus by taking the rectilinear minimal Steiner tree of the points connected by the bus. However, computing a rectilinear minimal Steiner tree is an NP-hard problem [112]. Therefore, we approximate the rectilinear minimal Steiner tree with a rectilinear minimal spanning tree (RMST). The length of an RMST for a set of points is at most one and a half times the length of the rectilinear minimal Steiner tree for the same set of points [143]. Computing the RMST is of time complexity $\mathcal{O}\left(e \cdot \log e\right)$ [114]. We estimate the amount of routing metal in a region by multiplying the length of the RMST by the number of wires in the bus and the average width of each wire.

A bus's *contention estimate* is the sum of the priorities of the communication ties it serves. It is necessary to determine a contention estimate before bus structures are generation and a schedule is produced. Therefore, we do not yet know the times at which communication events will occur. However, minimizing this contention estimate guides

1) Create point-to-point links for communicating pairs of cores. $\mathcal{O}(l \cdot \lg l)$

2) While maximum density $>$ density bound: $\mathcal{O}(l)$

    3) Find the most congested position, *congest*. $\mathcal{O}(l \cdot \lg l)$

    4) For each bus, i, intersecting with the *congest*: $\mathcal{O}(l^2)$

        5) For each bus, j: $\mathcal{O}(l^3)$

            6) Tentatively merge i and j. $\mathcal{O}(l^4)$

            7) Evaluate the density, *new_dens*, of *congest*. $\mathcal{O}(l^3)$

            8) Evaluate new maximum contention estimate, *cont_est*. $\mathcal{O}(l^4)$

    If *new_dens* decreased for any tentative merge:

        9) Merge the pair with greatest *new_dens* decrease. $\mathcal{O}(l^2)$

        Break ties by selecting merge with least *cont_est* increase.

    Else if *new_dens* increased for any tentative merge:

        10) Merge the pair for which the *new_dens* increase is least. $\mathcal{O}(l^2)$

        Break ties by selecting merge with least *cont_est* increase.

    Else halt: no valid topologies were found.

Figure 7.6: Bus formation kernel.

the bus formation algorithm to generate bus structures in which high-priority communication events are likely to occur on point-to-point links and low-priority communication events are likely to occur on large busses.

7.8.3 Overview

An overview of the bus formation algorithm is shown in Figure 7.6. The order notes to the right of the figure will be explained in more detail in Section 7.8.4. This algorithm merges pairs of busses until the maximum density on the chip is lower than the density bound. A heuristic is used to rapidly evaluate the quality of different potential merges.

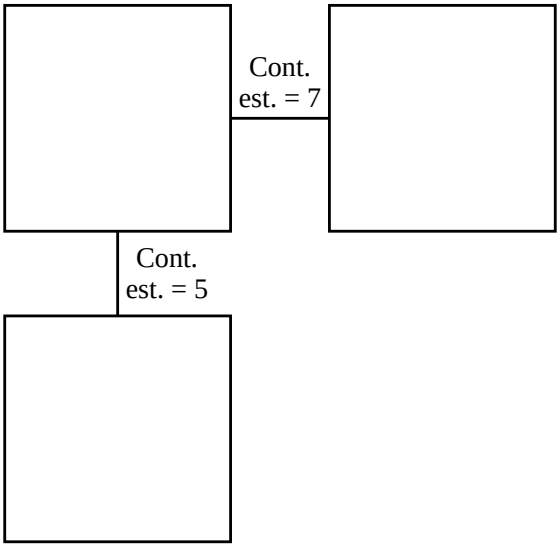

Figure 7.7: Bus formation example, step 1.

In order to reduce computational complexity, this heuristic considers only changes to the maximum density for points within the pair of busses that are merged, not changes in density for other points contained within the tentatively created bus. The algorithm accepts the move that reduces estimated maximum density the most. Ties are broken by accepting a move that increases the maximum contention estimate the least. If all moves increase estimated maximum density, the move that results in the smallest increase is taken. This has the potential of allowing the algorithm to escape local minima in some special circumstances. However, as described in Section 7.8.4, computational complexity was a more significant factor in the design of the algorithm than thorough solution-space exploration. Recall that bus formation is carried out in the inner loop of MOCSYN.

We describe this algorithm with an example. Figure 7.7 shows the starting conditions for a simple instance of the bus formation problem. In this image, every square is a core, all of which are the same size in this example. In order to simplify this example, we assume that the cores have densities of zero, i.e., one need only consider the densities

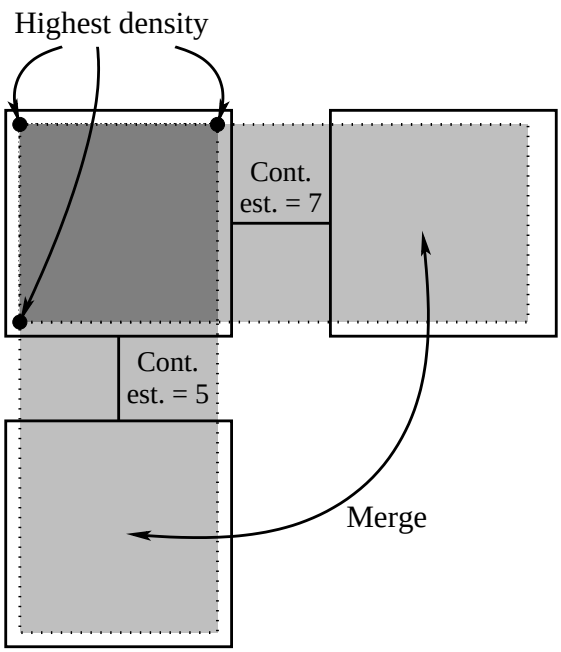

Figure 7.8: Bus formation example, step 2.

of busses. However, our algorithm does take into account core densities. The top tie has a priority, and therefore a contention estimate, of seven. The left tie has a priority, and therefore a contention estimate, of five. Figure 7.8 shows the bus structure after creating a point-to-point bus, i.e., link, between each communicating pair of cores. In this figure, shaded rectangles depict busses. The darkness of the shading at a point indicates the wiring density at that point. As shown in the figure, the highest density points are located at the intersections of the two busses. In this example, this density is higher than the maximum acceptable density; a merge will be attempted. The points of highest density, i.e., the corners of each bus that intersect with the other bus, are located. These points are indicated by black dots. One of these points is randomly selected and pairs of busses intersecting with that point are tentatively merged. In this example, only one such pair exists. After the busses in the pair are merged, only one bus remains, as shown in Figure 7.9. The new bus has a contention estimate equal to the sums of the

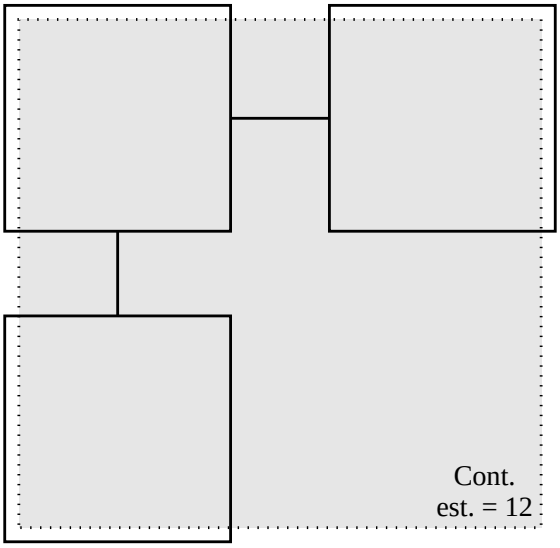

Figure 7.9: Bus formation example, step 3.

contention estimates of the two busses that were merged to form it. This new bus has a lower density than the maximum acceptable density; the algorithm halts.

7.8.4 Efficiency

We use red-black trees to store information about positions, densities, and busses. This allows the lookup and storage of these data to be accomplished in time $\mathcal{O}(\lg n)$. In Figure 7.6, information about the time complexity of the most important portions of the algorithm is given in Figure 7.6. In this figure, l is the number of communicating pairs of cores in the algorithm's input. During step 1, it is necessary to create l point-to-point links. Creating each of these links requires $\mathcal{O}(\lg l)$ time, for a time complexity of $\mathcal{O}(l \cdot \lg l)$. During each execution of loop 2, the number of busses must be reduced by one. Therefore, the contents of loop 2 may be executed a maximum of l times. Step 3 requires a search in a red-black tree of density values and may be executed a maximum of l times, for a time complexity of $\mathcal{O}(l \cdot \lg l)$. The contents of loop 4 may be executed

once per bus, per execution of loop 2, for a time complexity of $\mathcal{O}(l^2)$. Similarly, loop 5 has a time complexity of $\mathcal{O}(l^3)$. Step 6 results in the bus configuration being copied, $\mathcal{O}(l)$, for a time complexity of $\mathcal{O}(l^4)$. Step 7 requires a simple addition, for a time complexity of $\mathcal{O}(l^3)$. Step 8 requires a summation over all busses intersecting with the old highest density position, *congest*. In the worst case, there are l intersecting busses, for a time complexity of $\mathcal{O}(l^4)$. Steps 9 and 10 may be executed l times and each merge may take $\mathcal{O}(l)$ time, for a time complexity of $\mathcal{O}(l^2)$. Therefore, the overall time complexity of the bus formation algorithm is $\mathcal{O}(l^4)$.

A heuristic is used to decrease the time complexity of the algorithm. It would be most straightforward to fully evaluate changes in the densities of all points affected by tentatively merging two busses in order to compute the new highest density. However, this would require new density calculations to be carried out for $\mathcal{O}(l)$ points, in the worst case. Instead, the change in density is evaluated only at the point that previously had the highest density. Although this may result in a suboptimal merge being chosen, correctness is maintained by doing a density calculation at all affected points after a merge is completed. This heuristic reduced the time complexity of the algorithm from $\mathcal{O}(l^5)$ to $\mathcal{O}(l^4)$.

7.9 Cost calculation

As mentioned before, MOCSYN optimizes architecture price, area, and power consumption under hard real-time constraints. An architecture is invalid if any task with a deadline violates that deadline. Total hyperperiod energy is the sum of the energy consumptions of all of an IC's tasks executed on all its cores, throughout the hyperperiod, in addition to the sum of the idle energy consumption of all the cores, plus the energy consumed in the global clock distribution and communication networks. This value is

divided by the hyperperiod to get the power consumption. As described in Section 7.7, we assume the presence of buffers in the global communication network. In addition, the clock network is assumed to be constructed with buffered segments. Leakage current is assumed to be negligible. This allows delay and energy consumption to be estimated as linear functions of wire length and transition count, with constant factors derived from the process parameters and V_{DD}. Ultimately, three such constant factors are computed: communication wire delay factor, communication wire energy factor, and clock energy factor. The energy consumed by the global clock network is determined by estimating the total wire length of this network, multiplying this value by the number of signal transitions occurring during a hyperperiod, and also multiplying by the clock energy factor. The wire length estimate is derived from an RMST of the core positions in the block placement [114]. This provides an approximation of wire length. A Steiner tree may be used in the final post-optimization routing operation, possibly resulting in a lower total wire length. However, as mentioned earlier, computation of minimal Steiner trees is time-consuming (NP-hard) [114]. Hence, it is not used in inner-loop routing estimates. Energy consumption in the global communication networks is similarly computed. A separate RMST is computed for each bus. The transitions required for the communication events occurring on each bus are used to compute the bus energy consumptions.

An architecture's price is the sum of the prices of all the cores on the IC. The area of the IC is equivalent to the total rectangular area required for its block placement.

7.10 Experimental results

In this section, we present experimental results. Previous hardware-software co-synthesis systems do not target the single-chip synthesis problem. As a result, there is no body of work by other researchers with which MOCSYN's performance can be

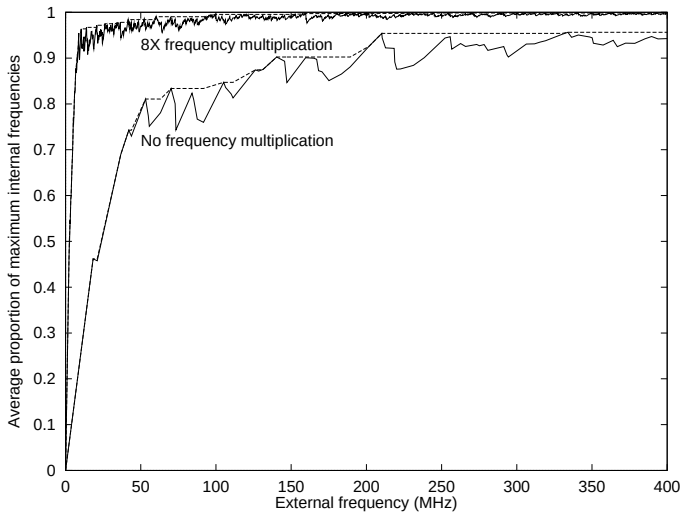

Figure 7.10: Clock selection quality as a function of external frequency.

compared. It is, however, possible to experimentally determine the effects of the algorithms comprising MOCSYN. The examples discussed below attempt to determine how clock selection, block placement, and bus topology generation affect the solution of the single-chip synthesis problem. Section 7.10.1 shows the results produced by the clock selection algorithm when run on a difficult example, i.e., one in which core maximum frequencies varied widely. In Section 7.10.2, we empirically determine the influence of a number of MOCSYN's specialized algorithms. Section 4.5 shows the result of running MOCSYN on the E3S benchmarks suite in the multiobjective optimization mode.

7.10.1 Clock selection

MOCSYN automatically selects clock frequencies for each core using the algorithm described in Section 7.4. In this section, we examine the results produced by this algorithm when run on an example problem.

In the interest of decreasing the power consumed in the global clock distribution network, one may reduce the frequency of the base clock. There is a tradeoff between

power consumption and execution time. However, this relationship is not linear. Figure 7.10 shows the relationship between maximum reference clock frequency and the average proportion of maximum internal clock rates at which the cores are clocked for a set of eight cores, each of which has a random maximum internal frequency ranging from 2 MHz to 100 MHz. Each sample point lies at the optimal reference clock frequency for a set of core multiplier values. The top solid line shows the average ratio of actual core frequencies to maximum core frequencies for linear interpolating clock synthesizers with a maximum numerator of eight. The bottom solid line corresponds to a cyclic counter clock divider. The dotted lines indicate the maximum ratio encountered before or at each frequency. The increase in power consumed by the clock reference frequency distribution network is approximately a linear function of frequency, although this function will be superlinear if voltage scaling is simultaneously carried out. As shown in Figure 7.10, the quality of the internal clock frequencies is a sub-linear function of the reference clock frequency. If one were using an interpolating synthesizer with a maximum numerator of eight for the cores in this example, choosing a maximum reference frequency greater than 100 MHz would not result in a significant increase in execution speed but may have a negative impact on system power consumption.

In summary, it is possible to clock each core at nearly its maximum frequency despite using a fairly conservative global clock frequency and widely varying core maximum frequencies.

7.10.2 Feature comparisons

This section empirically shows the influence of a number of the core-based synthesis algorithms used in MOCSYN.

Table 7.1 shows the results of synthesizing a number of ICs using MOCSYN with various sets of features enabled. These results indicate that it is extremely important

Table 7.1: Feature comparisons

Example	MOCSYN price	Worst-case commun. price	Best-case commun. price	Single bus price
1	219	137	n.a.	n.a.
2	254	n.a.	n.a.	n.a.
3	343	n.a.	n.a.	n.a.
4	279	n.a.	n.a.	n.a.
5	243	n.a.	n.a.	254
6	254	138	254	254
7	n.a.	n.a.	n.a.	245
8	n.a.	n.a.	n.a.	162
9	306	n.a.	n.a.	n.a.
10	354	n.a.	n.a.	n.a.
11	269	n.a.	n.a.	206
12	382	n.a.	n.a.	n.a.
13	283	n.a.	n.a.	n.a.
14	n.a.	n.a.	n.a.	218
15	357	n.a.	n.a.	n.a.
16	343	n.a.	n.a.	n.a.
17	304	n.a.	n.a.	n.a.
18	231	231	n.a.	n.a.
19	254	n.a.	n.a.	n.a.
20	255	n.a.	n.a.	n.a.
21	320	n.a.	n.a.	n.a.
22	216	n.a.	n.a.	n.a.
23	n.a.	n.a.	n.a.	254
24	227	n.a.	n.a.	n.a.
25	280	n.a.	n.a.	280

Continued on next page.

Table 7.1: Feature comparisons (continued)

Example	MOCSYN price	Worst-case commun. price	Best-case commun. price	Single bus price
26	219	n.a.	n.a.	n.a.
27	226	n.a.	n.a.	277
28	n.a.	n.a.	n.a.	137
29	302	n.a.	n.a.	n.a.
30	n.a.	n.a.	n.a.	300
31	216	n.a.	n.a.	n.a.
32	380	n.a.	n.a.	n.a.
33	489	n.a.	n.a.	n.a.
34	267	n.a.	n.a.	n.a.
35	269	n.a.	n.a.	n.a.
36	256	n.a.	n.a.	n.a.
37	291	n.a.	n.a.	n.a.
38	324	n.a.	n.a.	n.a.
39	301	n.a.	n.a.	n.a.
40	239	n.a.	n.a.	239
41	304	n.a.	n.a.	n.a.
42	332	n.a.	n.a.	n.a.
43	213	n.a.	n.a.	213
44	382	n.a.	n.a.	n.a.
45	227	n.a.	n.a.	n.a.
46	n.a.	n.a.	n.a.	256
47	243	n.a.	n.a.	375
48	137	n.a.	n.a.	165
49	213	n.a.	n.a.	213
50	n.a.	n.a.	n.a.	338
MOCSYN better		39	41	36
MOCSYN worse		2	0	9

to consider low-level details such as floorplan block placement and bus topology generation when making allocation, assignment, and scheduling decisions during system-on-chip synthesis for difficult problem instances. For the examples in this table, price was optimized under hard real-time constraints. If multiobjective optimization were used, it would be difficult to compare the solutions produced by different versions of MOCSYN because multiple solutions might have been produced for each problem (see Section 4.5). We used the 17 processors from the E3S benchmarks suite derived from the EEMBC benchmarks as described in Section 6.8. For each processor, we used two layout shapes: square and rectangular. We required numerous task sets for these experiments. Therefore, it was not possible to use the E3S task sets built from the EEMBC benchmarks. The task sets were produced with the aid of TGFF [129], a randomized parametric task set generator. Each example contains five task graphs with ten tasks, each. For each task with a deadline, the deadline is equal to $(d + 1) \cdot 1.6$ ms where depth, d, is the distance of a task, in nodes, from the start node of a task graph. Each communication event requires 1.8 Mb of data to be transferred. The graphs are composed of the 21 task types within the EEMBC Networking and Telecom benchmarks. Communication wire delay factor, communication wire energy factor, and clock energy factor were calculated based on the 0.18 μm process parameters given in the literature [144], with a V_{DD} of 1.8 V. We used 32 bit wide busses. Wire delay and energy consumption per μm per transition are calculated based on the use of a buffer separation distance that optimizes delay per μm. This optimal buffer separation is internally computed. The maximum clock reference frequency is 500 MHz and the maximum interpolating clock synthesizer numerator is eight. Note that internal frequencies may, therefore, be higher than 500 MHz. For each example, the same parameters were given to TGFF and MOCSYN. Only the random seed given to TGFF is varied, to produce different task sets based on the same parameters. We use a maximum aspect ratio of

two for the generated floorplans. Based on the automated analysis of wiring density in a number of designs, e.g., PipeRench [145], using modified design rule checking software [1], we believe that 0.8 is reasonable bound on routing density, and used this bound for our bus topology generator. We rounded the prices of the solutions up to the nearest dollar.

The bottom two rows in Table 7.1 show the number of problems for which a version of MOCSYN, with floorplan block placement and bus topology generation, produced better and worse solutions than the limited version of MOCSYN associated with the column. The first column in Table 7.1 shows the example number. Many of these examples had tight deadlines; they were difficult to solve. Entries of n.a. indicate situations in which no solution was found. Note that there is no guarantee that a solution exists for each example. Therefore, when no solution was found for a given example running in any of the four modes of operation, we omitted the example from the table and moved on to the next example. Each of the examples in this section took less than 1.25 minutes to complete on a 900 MHz Intel Pentium III machine running Linux. Allowing longer run-times might have produced lower prices for these examples. However, in these experiments, we were interested in relative values: we wanted to determine the value of including a number of low-level algorithms within system-on-chip synthesis.

The second column shows the price of solutions produced by MOCSYN when carrying out block placement-based wire delay estimations. The third column shows the price of solutions under the assumption that the distance in the block placement between each pair of cores is equal to the maximum distance between any pair of cores. Although this estimate may appear conservative, it is not possible to derive a tight bound on the maximum separation between any pair of cores without carrying out block placement in the inner loop. Thus, in practice, this estimate would probably be even more conservative if an inner-loop block placement tool were not available. The fourth column

shows the prices of solutions produced under the assumption that communication events take almost no time. After the optimization run is complete, solutions that are invalid because their schedules do not meet their hard real-time deadlines are eliminated. The fifth column shows the price of solutions that result from allowing MOCSYN to carry out block placement in the inner loop to accurately estimate communication delay but allowing only a global bus to be used, instead of an arbitrary priority-based topology containing up to eight busses. Although there are a few examples for which MOCSYN was able to produce superior results by making worst-case or best-case assumptions about wire delay instead of carrying out floorplan block placement or limiting itself to a single system-wide bus, these cases are fairly uncommon. Such deviations from the overall trend are not surprising, given that MOCSYN used a probabilistic optimization algorithm. The arbitrary bus topology version of MOCSYN outperforms the single-bus version in 36 out of 50 cases. However, the single-bus version outperforms the arbitrary topology bus version in only nine out of 50 cases. This is understandable: if the best solution to a particular problem uses only a single system-wide bus, then the one-bus solver will concentrate its search on this area of the solution space. The full-featured version of MOCSYN produces better solutions than the limited versions 10.5 times more often than it produces inferior solutions. From this, we conclude that, for difficult problem instances in which the problem specifications contain some degree of parallelism and communication, it is important to include floorplan block placement and bus topology generation within synthesis of a system-on-chip composed of IP cores.

7.10.3 Multiobjective optimization for the E3S benchmarks

This section presents the result of using MOCSYN to conduct multiobjective optimization on the E3S benchmarks described in Section 6.8. When MOCSYN is run in

Table 7.2: Multiobjective optimization for the E3S benchmarks

Example	Price ($)	Average power (mW)	Soft DL viol. prop.	Area (mm^2)
Automotive-Industrial	91	120	0.59	2.0
Networking	61	72	0.94	38.4
	188	843	0.25	139.4
	190	1339	0.23	173.3
Telecom	223	246	2.43	2.0
	238	239	2.49	5.5
	238	243	2.51	5.2
	334	261	2.16	3.0
	354	402	1.96	80.8
	433	516	1.83	84.5
	433	667	1.41	84.5
Consumer	134	281	1.40	21.6
Office Automation	64	370	0.23	32.8
	66	55	0.00	7.2

the multiobjective optimization mode, it produces a set of solutions, each of which is superior, in some way, to at least one other solution. Table 7.2 shows the sets of solutions produced for the five task sets in the E3S benchmarks suite. We rounded the prices and power consumptions of the solutions up to the nearest dollar and milliwatt. MOCSYN took less than 30 CPU minutes when run on each of these benchmarks. For most of the benchmarks, MOCSYN found multiple solutions that trade off price, average power consumption, soft deadline violation proportion, and area. For certain problems, it is possible for the lowest-price solution to also have the lowest power consumption, soft deadline violation proportion and area. In such cases, a multiobjective optimizer will find only one solution. This may have been the case for the Automotive-Industrial and Consumer benchmarks. Note that MOCSYN produced dramatically different solutions

to some problems, e.g., the prices of solutions to the Telecom benchmark ranged from $223 to $433, the power consumptions ranged from 246 mW to 667 mW, the soft deadline violation violation proportions ranged from 1.41 to 2.43, and the areas ranged from 2.0 mm^2 to 84.5 mm^2. These benchmarks are available via the E3S link on the http://-www.ee.princeton.edu/˜cad/projects.html web page.

7.11 Conclusions

In this chapter, we presented a method for the synthesis of core-based, single-chip, low-price, low-power, real-time, multi-rate, heterogeneous embedded systems. A multi-objective PRSA algorithm that allows exploration of the Pareto-optimal set of architectures instead of providing a designer with a single solution, was applied to a number of examples. MOCSYN's use of automatic clock selection, block placement-based communication delay and power estimation, and arbitrary bus topology generation allows it to efficiently solve the core-based synthesis problem. We experimentally determined that it is important to carry our floorplanning block placement and bus topology generation within SOC synthesis.

Chapter 8

Wireless Low-Power Client-Server System Synthesis

In this chapter, we present COWLS, a hardware-software co-synthesis algorithm that targets embedded systems composed of servers and low-power clients that communicate with each other through a channel of limited bandwidth, e.g., a wireless link. A novel scheduling algorithm is used to pipeline the execution of tasks that serve multiple clients associated with a given server. COWLS simultaneously optimizes the price of the client-server system, the power consumption of the client, and the response times of tasks that have only soft deadlines, while meeting all the hard deadlines. It produces numerous solutions that trade off different architectural features, e.g., price, power consumption, and response time, of an embedded client-server system. As far as we know, this is the first synthesis algorithm of its kind. We present the experimental results for a low-power, client-server camera system as well as several randomized examples.

A bandwidth-constrained embedded client-server system is a special-purpose computer in which clients and servers communicate with each other via a channel of limited bandwidth. Clients are frequently consumer products, e.g., portable communication devices, for which price is often particularly important. Server price is also an important factor, although it is usually less important than client price because clients typically

outnumber servers. In this work, we assume that servers have access to high-capacity power supplies. In order to maintain mobility, clients may be small and battery-powered. Therefore, client power consumption must be minimized to reduce heat production and increase battery life. Clients or servers may initiate communication events.

The literature contains numerous case studies of embedded client-server system design and general descriptions of the client-server problem domain. Some researchers have discussed wireless and cellular systems [146], [147], some have focused on embedded systems in which the server is a satellite [148], [149], and others have studied telerobotics, systems in which a robot is partially or totally controlled via a limited-bandwidth communication channel [150], [151]. The majority of previous research on embedded client-server systems either surveys the problems typically faced by the designer of such systems or provides case studies detailing specific solutions to individual problems.

Despite the previous work dealing with embedded client-server systems, we know of no previous work that automatically synthesizes such systems. COWLS automatically synthesizes architectures for embedded client-server systems, taking into account price, power consumption, bandwidth requirements, as well as task deadlines. It uses a novel scheduling algorithm that pipelines the computation and communication associated with the multiple clients that may interact with a server.

In the next section, we formulate the problem solved by COWLS. Section 8.2 provides a motivational example, describing the different ways in which a designer, or hardware-software co-synthesis algorithm, might partition functionality between client and server. Section 8.3 describes the scheduling, and client-server pipelining, algorithms used by COWLS. In Section 8.4 we describe the method COWLS uses to calculate the costs, e.g., price, power consumption, and soft deadline violation, of each architecture. Section 8.5 shows the results of using COWLS to produce client-server architectures for

the E3S benchmarks. In addition, we give experimental results demonstrating the effect of using client-server pipelining during scheduling. In Section 8.6, we draw conclusions.

8.1 Problem formulation

In this section, we present the client-server synthesis problem formulation used for COWLS.

The independent synthesis of a client or server is similar to the distributed, heterogeneous embedded system co-synthesis problem. The optimization infrastructure used by COWLS is similar to that described in Chapters 5 and 6. However, COWLS targets the servers and clients simultaneously, and examines the consequences of allowing tasks to migrate between clients and servers. A designer may specify the behavior and timing constraints of a client-server system using a modified version of the model presented in Section 3.3. This version also allows some tasks to have their assignment constrained to processing elements (PEs) in the clients or PEs in the server, although many tasks will be free to migrate between client and server during synthesis. It is also necessary to describe the characteristics of the resources that may be used to meet these requirements. We model three main types of resources: PEs, communication resources, and memory.

As described in Section 3.6, PEs model general-purpose or special-purpose processors that are capable of executing tasks. However, COWLS models two classes of PEs: client PEs and server PEs. Client PEs may only exist within the client's PE allocation. Server PEs may only exist within the server's PE allocation. In general, server PEs will have better performance than client PEs but their power consumptions will be higher.

Task assignments are modified using the algorithm described in Section 6.4. This algorithm was originally designed to improve the performance of our optimization infrastructure when synthesizing client-server systems containing low-bandwidth communication resources. By considering the expected impact upon bandwidth caused by each potential change in task assignment, COWLS is able to avoid task assignments that increase communication time without compensating improvements in computation time or reduction in PE overloading.

COWLS synthesizes embedded systems containing arbitrary-topology busses and point-to-point communication links, as well as the primary communication resources that are used to connect clients and servers. There are a number of attributes associated with each type of communication resource. There may be multiple communication resources within the client, and within the server. Different primary communication resources may be available. However, only one primary communication resource may be present in a client-server pair, as multiple wireless transmitters and receivers will typically result in unreasonably expensive client-server systems. Each type of communication resource has a price per instance (to represent bus controller price), a price per contact (to represent bus bridge or interface circuit price), packet size (which can be very small to model communication that is not packet-based), energy consumption per packet, transmission time per packet, and maximum number of contacts. A communication resource's number of contacts is the number of different PEs that it may connect together, i.e., a communication resource with two contacts is a point-to-point link. Primary communication resources have four prices: the client and server each have a price per instance and a price per contact. For primary communication resources, each contact is associated with a PE, on the client or server, that needs to be connected to the primary communication resource.

COWLS uses task sets to specify client-server behaviors and timing constraints. These task sets are identical to those described in Section 3.3, with one addition: any task may have its assignment locked to a client PE, a server PE, or be permitted to execute on either a client or server PE. Given the client-server system requirements, in the form of a task set, the attributes of the PEs, memory, and communication resources available, as well as the number of clients and servers in the client-server system, COWLS attempts to synthesize client-server systems that meet the requirements with minimal price, client power consumption, and soft deadline violation. COWLS contains algorithms that carefully consider the impact of task assignment, defined in Section 3.1, on the wireless communication resource. In addition, the scheduler in COWLS uses a novel method of client-server pipelining.

An architecture's costs are derived from the manner in which resources are used in its construction. Therefore, by attempting to meet real-time constraints, one ensures that high-speed PEs, well-suited to tasks they execute, are used for tasks that lie along critical paths in the task graphs. By attempting to minimize price, one ensures the use of PEs that are capable of carrying out the required tasks with minimal price. By attempting to minimize client power consumption, one minimizes the number of power-intensive tasks run on power-hungry PEs located on the client. Of course, some of these goals conflict with each other. For this reason, a single run of COWLS generates multiple solutions that explore the tradeoffs among different costs.

8.2 Motivating example

A synthesis system for the client-server problem domain should simultaneously optimize multiple costs. It must also consider the differences in cost between executing a task on a client or a server. It is, therefore, necessary to allow tasks to migrate from one

side of the primary communication resource, the link connecting the client and server, to the other. Optimizing only one cost, or considering local improvements instead of system-level improvements, is likely to lead to a poor overall solution.

We will show how COWLS explores the design space in a manner that allows it to uncover a high-quality design similar to one proposed in the motivational example, and consider other options that trade off different system costs.

Consider a system specification requiring a battery-powered camera to transmit digital images to a base station via a limited-bandwidth wireless link. If the designer has decided that the video information should be compressed, but has not yet decided what sort of processor should be used to carry out this operation, or even whether it should be done by the client or the server, COWLS will simultaneously explore the different options.

Figure 8.1 shows the Consumer benchmark from the E3S benchmarks suite described in Section 6.8. In this example, images must initially be generated on the client camera. They are then filtered, on either the client or server, converted to another image format, and compressed. Images must be transferred to the *sink* task within 2.5 s and, ideally, within 0.1 s. Image capturing (represented by the *src* node in the left graph), and display must be carried out on the client. Data storage (represented by the *sink* node in the left graph) data retrieval (represented by the *src* node in the right graph) and image printing must be carried out on the server. Storage has a hard deadline of 2.5 s and a soft deadline of 0.1 s. Printing has a hard deadline of 15 s and a soft deadline of 5 s. Display has a hard deadline of 15 s and a soft deadline of 1 s.

In this motivational example, we will focus on the left task graph in Figure 8.1. This task graph carries out image acquisition (*src*), filtering (*filt-x*), conversion (*rgb-yiq*), compression (*cjpeg*), and storage (*sink*). In one possible client-server partitioning of this graph, shown in Figure 8.2, filtering, conversion, and data compression are executed on

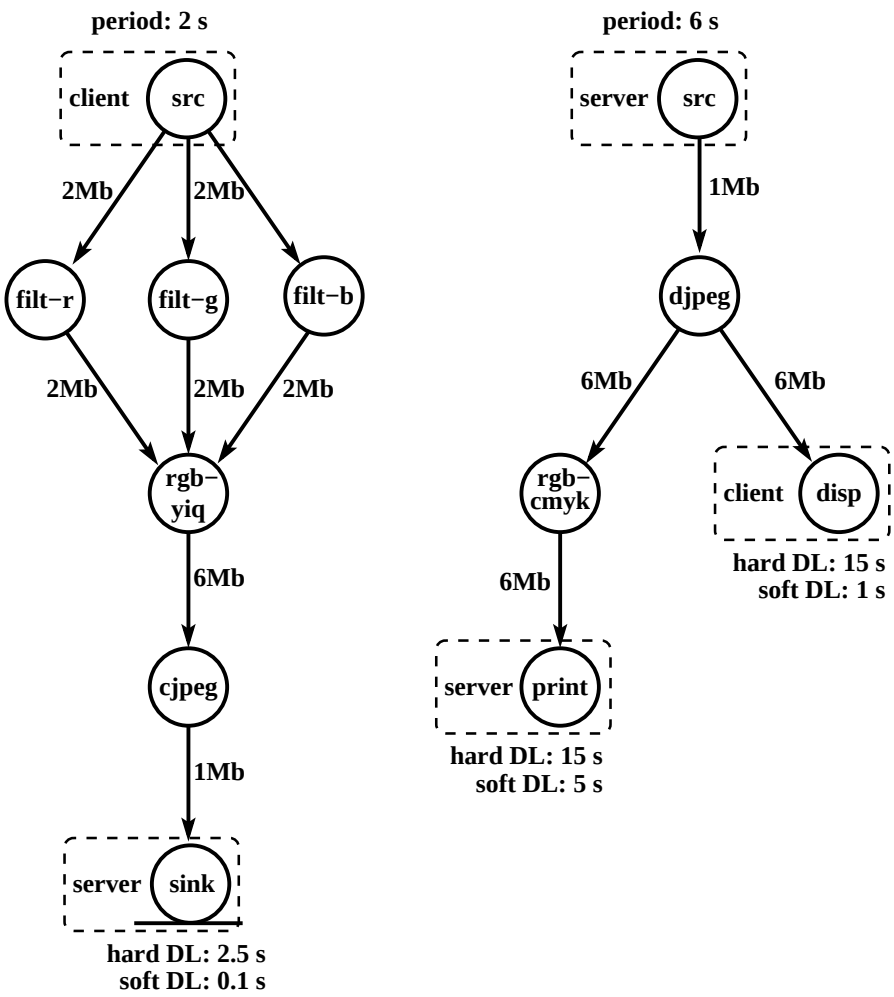

Figure 8.1: Camera specification, i.e., the Consumer E3S client-server benchmark.

the client and all other tasks are carried out on the server. This partitioning reduces the load on the wireless communication link to 1 Mb per task graph execution and allows an inexpensive primary communication resource to be used between the client and the server. However, carrying out data compression on the client requires increased client price and power consumption.

In another possible partitioning, shown in Figure 8.3, image acquisition (*src*) executes on the client and all other tasks execute on the server. In this partitioning, the

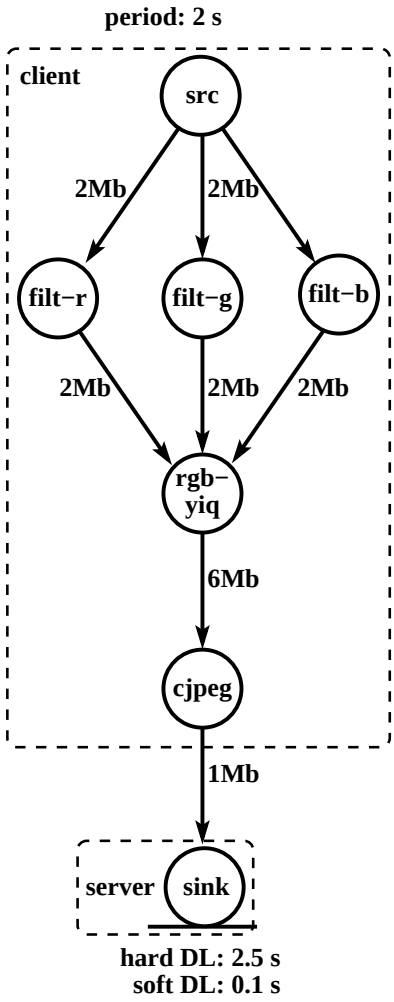

Figure 8.2: Camera architecture 1

client executes only essential functions, shifting all other computational burdens to the server. This decreases the client's price and power consumption. However, it increases the demands upon the communication link between the client and the server, increasing its price and power consumption. Although some of the tradeoffs facing the designer of client-server systems are apparent even from this simple example, COWLS is capable of solving problems that are significantly larger and more complicated.

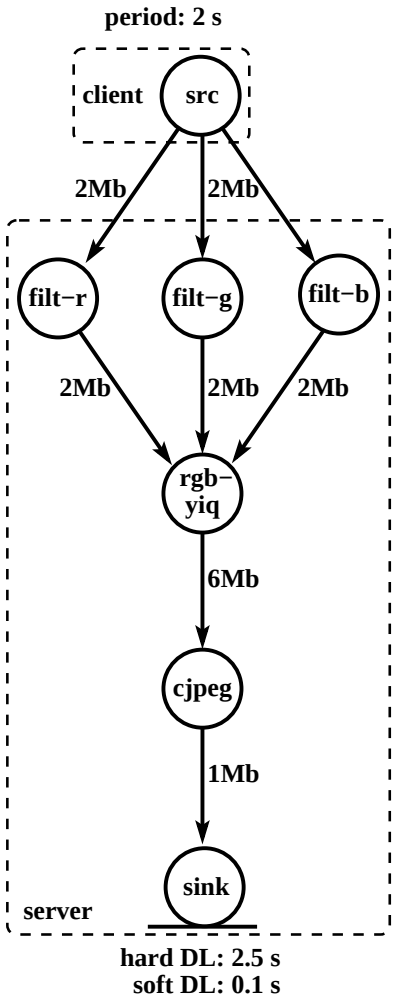

Figure 8.3: Camera architecture 2

8.3 Scheduling and client-server pipelining

In this section, we describe the scheduling algorithm used in COWLS.

In order to determine a solution's client power consumption, soft deadline violation, and hard deadline violation, it is necessary to generate its complete schedule. COWLS uses a rapid multi-rate list scheduler (see Section 6.6) that is capable of handling task graphs with periods that are greater than, equal to, or less than the deadlines in the task

graphs. The scheduler treats time as circular, i.e., an event that occurs at one point in time also occurs at every integer multiple of the hyperperiod from that point in time. This scheduler operates in two stages. During the first stage, the scheduler determines a priority for each task. During the second stage, communication events are assigned to communication resources, communication events are scheduled, and tasks are scheduled.

In order to prioritize tasks, the approximate earliest finish times (EFTs) and latest finish times (LFTs) of every task is determined by conducting a modified breadth-first search of each task graph. At this point, task assignments are fixed. Therefore, the execution time of each task is known. Communication event assignments are not fixed when EFT and LFT calculations are carried out. Therefore, it is not possible to know the exact amount of time required to carry out each communication event. A communication event's time is approximated by taking the maximum amount of time required by the event on any of the communication resources that connect the PEs to which the communication event's parent and child tasks are assigned. Raw times are used for EFT and LFT computation, i.e., these time values are not multiplied by the number of clients per server. A more detailed explanation of this decision requires knowledge of the method of pipelining used in COWLS. We explain this concept in later in this section. Slack is the difference between a task's LFT and its EFT. The scheduler uses negative slack in order to prioritize task scheduling, i.e., low-slack paths in the task graphs have high scheduling priorities. If the schedule produced in this manner fails to meet all hard real-time deadlines, COWLS retries scheduling using negative LFT and negative earliest start time (EST) for prioritization.

Once tasks are prioritized, the second scheduling stage is entered. During this stage, the contents of a continuously updated prioritized list of tasks, whose data dependencies have been satisfied, are iteratively scheduled. Recall that some task graphs will be

scheduled multiple times during the hyperperiod. Given that t is a task, c is the offset of a task's copy in the hyperperiod, m is the maximum copy number for a given task, then a task's proportional copy number, p, is defined as follows,

$$p = \frac{c}{m}$$

tasks are sorted in the following manner. If the slacks of the tasks are unequal, the task with the lower slack is scheduled. If slacks are equal, the task with the lower proportional copy number is scheduled.

When a task is selected for scheduling, each of its incoming communication events is first scheduled on one of the communication resources connecting the PEs to which the task and its parent are assigned. The communication resource that allows the communication event to finish at the earliest time is used. If the tasks are assigned to the same PE, communication is treated as instantaneous. If they are assigned to PEs separated into the client and server, the communication event is scheduled on the primary communication resource, i.e., the wireless link.

While scheduling, bus contention is explicitly simulated. The scheduler is deterministic, i.e., given a particular resource allocation and task assignment, it always produces the same complete, static schedule. Therefore, after scheduling, the worst-case completion times of each task and communication event are known. This allows straightforward calculation of soft and hard deadline violations. In addition, the scheduler determines the communication resources upon which each communication event occurs. This information allows the calculation of power consumption by the client's communication resources. The power consumption of each task that executes on the client, as well as the power consumed by the client PEs while idle and communicating, is added to the client communication resource power consumption to determine the total client power consumption.

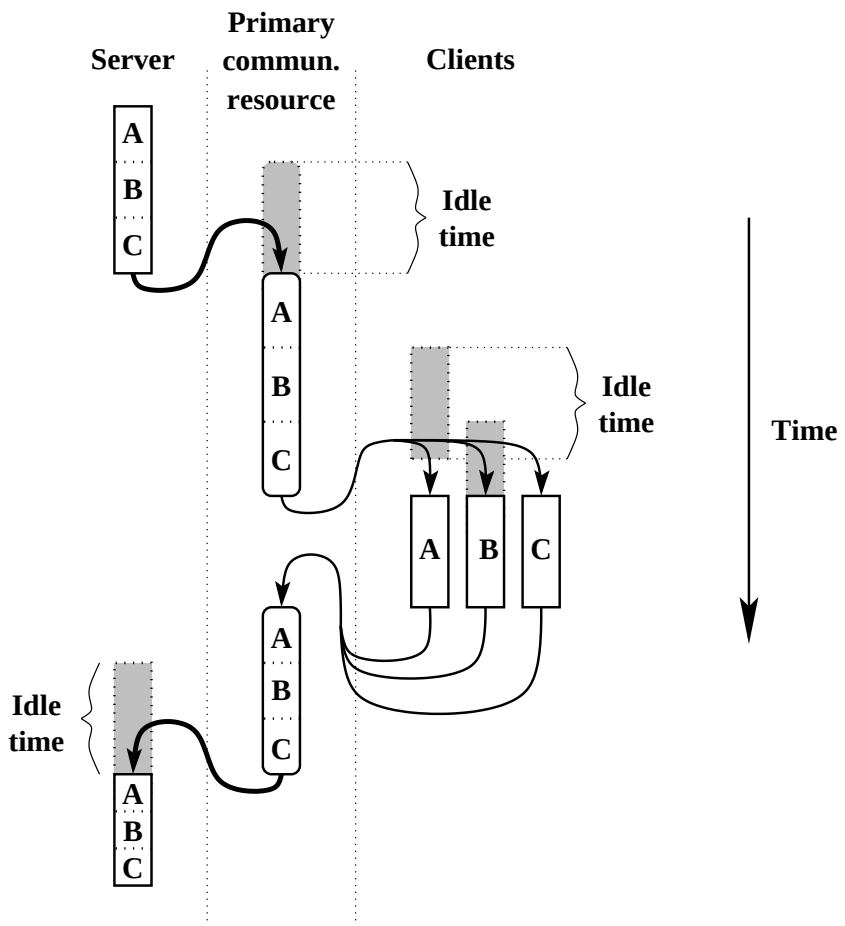

Figure 8.4: Part of a non-pipelined schedule

Recall that there may be multiple clients per server. It is necessary to ensure that a server is capable of executing the tasks associated with each client. The most straightforward way of accomplishing this is to multiply the execution times of the tasks and communication events on the server, and the communication events between the server and clients, by the *client-server ratio*, i.e., the number of clients per server. However, in order to ensure that this straightforward approach is correct, it is necessary to delay the execution of the corresponding tasks on each client until all the tasks have received the data upon which their execution depends, and provide buffers for the transmitted data.

Consider the schedule portion shown in Figure 8.4. Time increases from the top of the figure to the bottom. The left column depicts the schedule for the server. In the top rectangle of this column, each of the three portions (A, B, and C) corresponds to a task associated with one of three clients. In this figure, a straightforward, non-pipelined method of scheduling is used. The communication events that transmit data from the server to the client do not begin until the tasks associated with each client have completed execution. Similarly, none of the clients begins execution until data have been transmitted to each client. This results in the primary communication link and clients sitting idle when they might otherwise be carrying out work. There are a number of ways that this problem might be remedied.

One possible approach is to explicitly schedule each client separately, thereby allowing every task to execute as soon as its incoming data are ready. This approach has two disadvantages, one tolerable and one intolerable. Scheduling each client separately would increase the average run-time of the scheduler by a factor of the client-server ratio. However, this synthesis-time cost might be tolerable if the increased scheduling flexibility resulted in improved schedules. More importantly, this approach would result in each client having a different schedule. We considered the resulting increased complexity of manufacturing, debugging, and maintaining such a system sufficient to disqualify this approach. To give some idea of the problems associated with such a scheme, note that it would require the maintenance of a number of client designs equal to the client-server ratio.

The approach we selected gains a significant amount of scheduling flexibility without sacrificing synthesis-time efficiency or dramatically increasing the complexity of producing, debugging, and maintaining the embedded system. We pipeline the execution of tasks and communication events associated with different client copies. However, we constrain each client to the same schedule. Each client's schedule is offset,

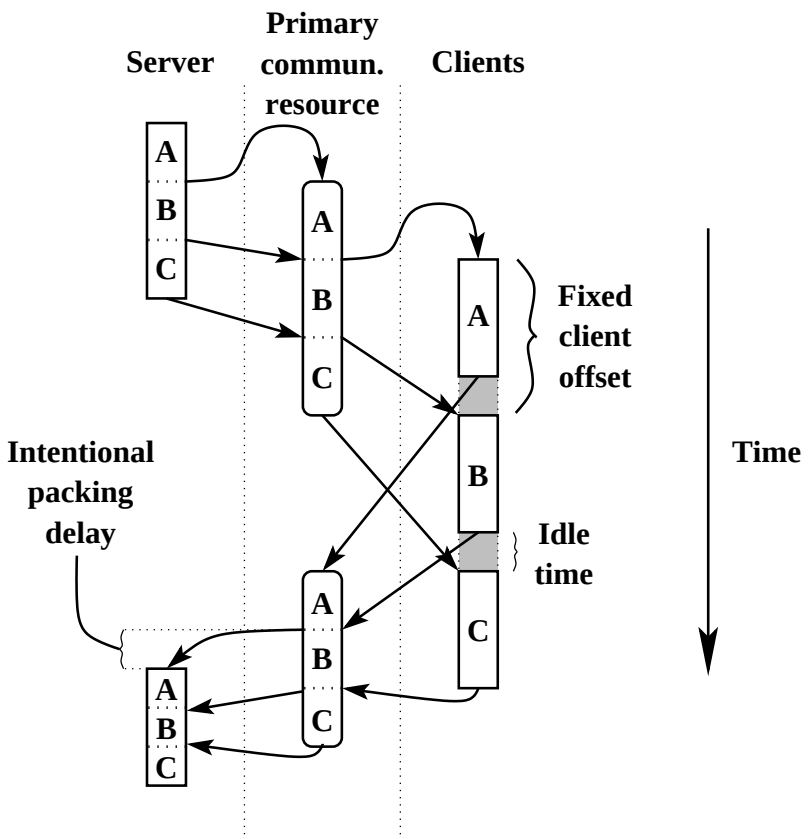

Figure 8.5: Part of a pipelined schedule with a large client offset

in time, by a fixed duration from every other client's schedule. The approach may be most directly illustrated with the aid of a diagram. Figure 8.5 is analogous to Figure 8.4. However, it shows a portion of a schedule produced using our pipelining approach. Note that the first series of communication events (shown at the top of the center column) may begin as soon as their parent tasks have completed. Similarly, the client task may begin execution as soon as their data have arrived, under the constraint that each client task must be separated from its corresponding task in other clients by a fixed amount of time, the client offset. As a result of adhering to a fixed client offset, it is only necessary to produce one client schedule explicitly. Each of the other client schedules is equivalent to the explicit schedule offset in time by an integer multiple of the client offset.

As described in Section 8.3, the raw execution time of tasks is used during EFT and LFT calculation. Pipelining frequently allows tasks to be scheduled as soon as the corresponding incoming communication event has completed. Therefore, using raw task and communication event durations allow more accurate EFT and LFT estimates than using task and communication event durations multiplied by the client-server ratio.

Consider the second set of server tasks in Figure 8.5. Note that the task associated with client copy A begins execution later than its incoming data are ready. This intentional packing delay is introduced to ensure that the tasks are scheduled as one contiguous event. We considered the alternative of allowing the tasks to be separated by an arbitrary amount of time. However, this leads to a dramatic increase in the time complexity of the scheduling algorithm with little gain in scheduling flexibility. By allowing arbitrary gaps between the scheduling events of the tasks and communication events associated with different clients, one introduces numerous (a number equal to the client-server ratio minus one, in general) gaps into the schedule every time an event is scheduled. Scheduling complexity is increased, not only by the necessity of checking each of these gaps when every new event is scheduled, but more importantly, by the necessity of finding a location for new events, each of which consists of a pattern of gaps and active periods. We avoid these problems by making a set of tasks, associated with different clients, contiguous.

Even if allowing non-contiguous scheduling of the events associated with different clients did not grossly increase computational complexity, it would be of dubious benefit. Figure 8.6 shows a portion of a pipelined schedule without packing. Consider the second server task set, to the lower left. By allowing arbitrary delays between the tasks associated with different clients, we have traded a moderate idle slot in a position where it can easily be filled or masked by other tasks in practice, for numerous (equal to the

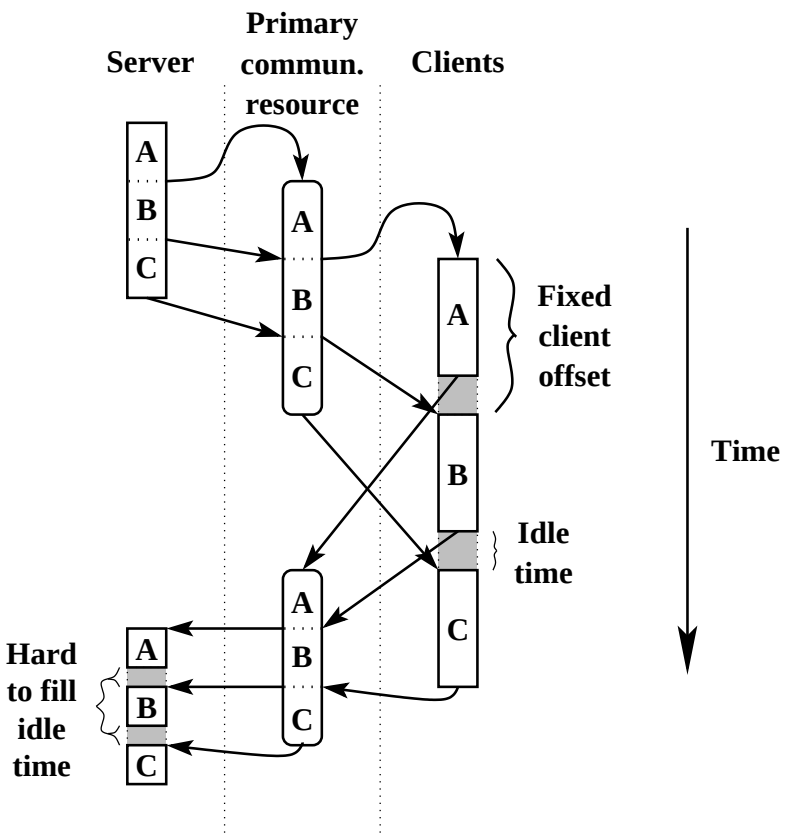

Figure 8.6: Part of a pipelined schedule without packing

client-server ratio minus one) small idle slots that increase the computational complexity of scheduling and are difficult to fill or mask. These observations led us to use the packing approach.

We initially considered the selection of the client offset to be an important problem. Compare the client schedules of Figure 8.5 and Figure 8.7. In the first case, idle time is introduced between client tasks by using a client offset that is larger than the ideal offset. In the second case, the execution of the first client's task is delayed in order to enforce the constraints imposed by a client offset that is smaller than the ideal offset. Unfortunately, it is necessary to use a single client offset for all tasks in order to ensure that all client schedules are identical. Therefore, the client offset is likely to be too large for some tasks

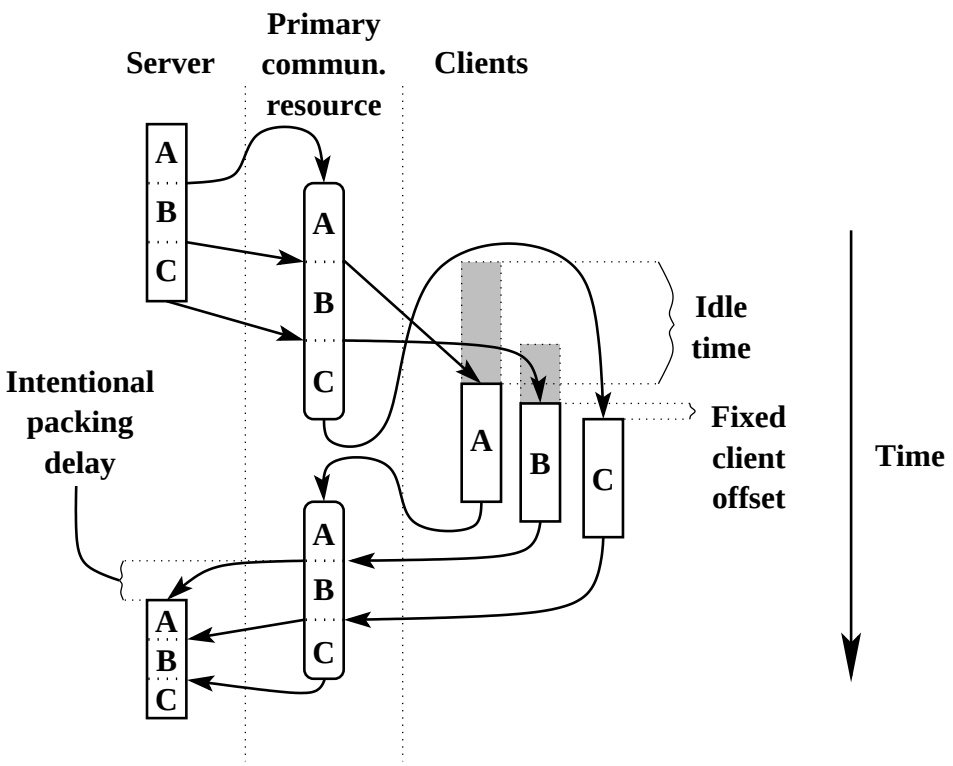

Figure 8.7: Part of a pipelined schedule with a small client offset

and too small for others in any problem of moderate complexity. One must select a client offset that provides a good tradeoff between these two alternatives. We set the client offset to be equal to the average time required by the communication events assigned to the primary communication resource. We experimentally determined that the qualities of the results produced by a synthesis run are not strongly dependent upon the client offset, as long as a few conditions hold. An explanation of this phenomenon and a comparison between non-pipelined and pipelined scheduling are presented in Section 8.5.

8.4 Cost calculation

In this section, we describe the process by which a solution's costs are calculated.

After making changes to solutions, it is necessary to determine whether or not those changes resulted in improved costs. Thus, after modifying a solution, COWLS carries out cost calculation to determine its aggregate price, the client's power consumption, and the degree to which soft deadlines are violated. In addition to these visible costs, there are a number of hidden costs that need never be displayed to the designer. Hard deadline violation is an example of such a cost. All solutions in which the hard deadline violation is non-zero are eliminated before results are presented to the designer. However, during optimization, solutions with hard real-time deadline violations are allowed to exist, for they have the capacity to evolve into high-quality, valid solutions during optimization. Soft deadline violation proportion is the sum of the soft deadline violation times in every copy of each task graph, divided by the hyperperiod.

Once a schedule is computed for a solution, that solution's client power consumption and soft deadline violation information is stored in a cache (see Section 6.7) and used for any equivalent solutions that subsequently arise during optimization. Aggregate price is computed by taking the sum of the prices of the PEs, task execution memory, communication buffer memory, communication resources, and the primary communication resource associated with the client, multiplying this by the expected number of clients, and adding to this the sum of the prices of the resources used in the server multiplied by the expected number of servers. This gives a total client-server system price.

8.5 Experimental results

In this section, we present experimental results and discuss their implications. These results provide a reference point for other researchers, suggest the superiority of particular synthesis tool design decisions, and allow a deeper understanding of the client-server synthesis problem. Note that we have already discussed the suitability of the optimization infrastructure used by COWLS by comparing its performance with past work in Section 6.9.

8.5.1 Multiobjective optimization for the E3S benchmarks

This section presents the result of using COWLS to conduct multiobjective optimization on the E3S benchmarks described in Section 6.8. We have modified these benchmarks to ensure that at least one task in each task graph has its assignment locked to the client and at least one has its assignment locked to the server. For example, see the Consumer benchmark discussed in Section 8.2. In addition, we relaxed some deadlines and periods. Some of the distributed system E3S benchmarks had deadlines that were too tight for timing constraints to be met when communication via a wireless channel was necessary. We used three Linux machines to produce these results: a Pentium III running at 900 MHz, an Athlon Thunderbird running at 1.4 GHz, and an Athlon running at 650 MHz.

Table 8.1 shows the sets of solutions produced for the five task sets in the E3S benchmarks suite. There are five clients for each server. For these benchmarks, COWLS was used to explore the tradeoffs among different system costs, instead of attempting to minimize a single cost. Given a similar amount of CPU run time, it would be possible to better optimize a single cost by ignoring all other costs. However, this approach would ignore the fundamentally multiobjective nature of embedded system design. COWLS

Table 8.1: Multiobjective optimization

Example	Price ($)	Average power (mW)	Soft DL viol. prop.
Automotive-Industrial	518	186	0.00
	563	98	0.00
Networking	372	136	0.86
	408	136	0.74
	423	134	0.86
	507	134	0.80
	543	134	0.75
	583	135	0.74
Telecom	659	143	1.62
	659	147	0.98
	659	152	0.64
	660	146	0.92
	673	143	1.52
	996	366	0.58
	1168	145	0.88
	1559	327	0.62
	1684	343	0.52
	2902	344	0.51
Consumer	610	126	0.98
	1038	167	0.61
	2890	165	0.65
Office Automation	344	159	1.14
	385	158	0.71
	418	157	0.75
	436	157	0.72
	476	172	0.69
	492	164	0.69
	651	162	0.68
	664	162	0.67
	893	158	0.68

took less than 80 CPU minutes when run on each of these benchmarks. We rounded the prices and power consumptions of the solutions up to the nearest dollar and milliwatt. For most of the benchmarks, COWLS found numerous solutions that trade off price, average power consumption, and soft deadline violation proportion. Note that COWLS produced multiple solutions for each benchmark. In particular, let us revisit the camera (E3S Consumer) example we described in Section 8.2. COWLS produced three wireless client-server architectures for this example. These architectures had prices ranging from $610 to $2,890, power consumptions ranging from 126 mW to 167 mW, and soft deadline violation proportions ranging from 0.61 to 0.98. The first solution to the camera example contains an IBM PowerPC 405GP running at 266 MHz on the client, an ST Microelectronics ST20C2 running at 50 MHz on the server, and an IEEE 802.11 Lucent Wavelan card. COWLS found that the requirements placed on the wireless communication resource could be significantly reduced by assigning all tasks, prior to compression, to the client in the left, data acquisition, task graph in Figure 8.1. This corresponds with the assignment decisions specified in Figure 8.2. The other two solutions have relatively more PEs and communication resources and use these resources to reduce soft deadline violation. These benchmarks are available via the E3S link on the http://www.-ee.princeton.edu/˜cad/projects.html web page.

In the interest of evaluating the performance of the client-server pipelining algorithm described in Section 8.3, we did a number of experiments in which we compared different versions of pipelining scheduler with each other, and with a straightforward non-pipelining scheduler. Multiobjective optimization significantly complicates presentation of, and comparison between, the results of different optimization runs because each run produces numerous examples. For these comparative examples, we forced

Table 8.2: Price-only pipelining comparison experiments with an offset factor of 1.0

Example	Price with pipelining	Price without pipelining	Example	Price with pipelining	Price without pipelining
1	525	526	2	885	448
3	547	653	4	671	671
5	686	849	6	845	861
7	542	1092	8	617	618
9	719	910	10	561	1035
11	513	590	12	583	408
13	3277	n.a.	14	954	954
15	740	622	16	695	1003
17	491	500	18	1455	n.a.
19	1149	754	20	829	773
21	726	809	22	n.a.	1017
23	663	663	24	874	n.a.
25	431	586	26	465	716
27	444	570	28	919	n.a.
29	1564	1564	30	1442	n.a.
31	430	430	32	557	515
33	1020	690	34	952	558
35	440	603	36	1016	1127
37	657	657	38	420	441
39	464	927	40	820	787
41	618	988	42	914	1017
43	843	1369	44	714	714
45	615	868	46	832	758
47	744	n.a.	48	897	825
49	754	n.a.	50	2085	n.a.
Improved: 31					
Degraded: 12					

COWLS to ignore soft deadline violation and power, concentrating only on price optimization. As a result, each run produces only one result. We rounded the prices of the solutions up to the nearest dollar.

Table 8.2 shows the result of running COWLS on 50 examples in which the processors come from the EEMBC benchmarks suite and the task sets are randomly generated. Solution quality improved 2.58 times as frequently as it degraded. For the examples in this table, we used the 17 processors from the E3S benchmarks suite derived from the EEMBC benchmarks as described in Section 6.8. For each processor, we generated a server version and a client version. The server version is identical to the EEMBC processor. The client version has one-fifth the power consumption of the EEMBC processor and five times the execution time, for each task, but is otherwise identical. Our task sets each contain 12 tasks. Each task type is randomly selected from the Networking and Telecom EEMBC benchmarks. Each communication event has a quantity of 1 kb. Approximately a third of the tasks must be assigned to a client, a third must be assigned to the server, and a third may be assigned to either client or server. There are five clients for each server. There is no guarantee that every example generated in this manner will have a valid solution. In this table, an entry of n.a. indicates that no solution was found for the problem and parameters associated with the entry. For cases in which no solutions were found by either the client communication pipelining nor client communication non-pipelining version of COWLS, we omitted the example from the table.

We found that pipelining schedules usually results in an improvement to solution quality. As shown in Table 8.2, solution quality improved approximately two and a half times as frequently as it degraded. Although there were some cases when using a non-pipelining scheduler allowed the production of a superior solution, one should not draw the conclusion that it would be wise to run the scheduler in pipelining and

non-pipelining mode for every cost evaluation and take the best cost. By doubling the amount of time required for each solution evaluation, one would halve the number of solutions that may be evaluated. One could, instead, use the scheduling method that is generally superior, i.e., the pipelining scheduler, and allow a more thorough exploration of the solution space, guided by the evolutionary algorithm, in the same amount of time.

As discussed in Section 8.3, we had initially considered the selection of a client offset ratio to be an important problem. However, in practice, solution quality is highly resistant to degradation. Varying this ratio from zero to two results in only small changes to the number of cases for which pipelining resulted in an improvement to solution quality. Solution quality remains independent of the client offset ratio until it approaches the ratio of primary link communication time to computation time. Examining the schedules with a simple graphing tool revealed that, up until this point, the task delays required due to dependency on data transmitted via the primary communication link mask the idle slots that result from having a large client offset factor.

8.6 Conclusions

COWLS automatically synthesizes embedded client-server systems. It uses a multi-objective evolutionary algorithm to simultaneously produce multiple solutions that trade off different costs. It optimizes price, client power consumption, and soft deadline violations under hard real-time constraints and constrained client-server communication bandwidth. COWLS incorporates a novel and tractable scheduling algorithm that pipelines the execution of tasks associated with different clients while maintaining identical client schedules. This form of pipelining has been found to improve solution quality in the majority of cases.

Chapter 9

Hardware-Software Co-Synthesis of Dynamically Reconfigurable Embedded Systems

In this chapter, we describe our co-synthesis algorithm for hardware-software systems containing dynamically reconfigurable hardware. Field programmable gate arrays (FPGAs) are commonly used in embedded systems. Although it is possible to reconfigure FPGAs while an embedded system is operational, this feature is seldom exploited. Recent improvements in the flexibility and reconfiguration speed of FPGAs have made it practical to reconfigure them dynamically, i.e., while the embedded system containing them is operating, thereby reducing the amount of hardware required in an embedded system. We have developed a synthesis algorithm, called CORDS, that produces multi-rate, real-time, periodic distributed embedded system architectures containing dynamically reconfigurable FPGAs. Executing different tasks on the same FPGA requires that potentially time-consuming reconfiguration be carried out between tasks. CORDS uses a novel dynamic priority, multi-rate scheduling algorithm to deal with this problem. Experimental results indicate that using dynamically reconfigured FPGAs in distributed

real-time embedded systems has the potential to reduce their price, and allow the synthesis of architectures that meet system specifications that would otherwise be infeasible.

9.1 Motivation

Until recently, dynamic reconfiguration of FPGAs in hard real-time embedded systems has been impractical. FPGA reconfiguration times have conventionally been on the order of 100 ms. However, recently a number of companies have released products that improve upon the reconfiguration times of existing FPGAs by an order of magnitude or more [152], [153]. In particular, the largest member of the Xilinx XC6200 family, the XC6264, can be completely reconfigured in under 200 μs. However, a price is paid for this speed. Rapid reconfiguration FPGAs can cost approximately ten times as much as FPGAs using conventional architectures. Rapid reconfiguration FPGAs are a new product and production has been limited. Therefore, their price is likely to decrease in the future. Already, many of the features that used to appear only in research parts, e.g., the Xilinx XC6200, have been incorporated into mainstream parts, e.g., the Xilinx Virtex series. Nonetheless, if price is a concern, it is important to consider more conventional FPGAs, which have large reconfiguration times. If one derives a schedule that locates different instances of the same task type adjacent to each other, the number of reconfigurations an FPGA needs to undergo will be reduced, resulting in significant time savings.

FPGAs fit naturally into the hardware-software co-synthesis design flow. The holy grail of configurable computing research is a system that will accept a problem description in a general-purpose programming language, automatically partition it between hardware (FPGAs) and software (general-purpose processors), synthesize the required hardware, and manage communication between the two domains. This problem closely

mirrors the co-synthesis problem. By using FPGAs in co-synthesis, designers can take advantage of research in the reconfigurable computing field. There are already systems that accept algorithm descriptions in general-purpose languages, like ANSI-C, and automatically produce FPGA configurations [154].

CORDS was the first co-synthesis system to handle dynamically reconfigurable FPGAs, although others have subsequently considered their use [69], [71], [155]. CORDS automatically selects an allocation from a set of FPGAs, general-purpose processors, and communication resources. It assigns tasks to FPGAs and general-purpose processors, and determines the connectivity of communication resources. Finally, it derives schedules for tasks and communication events. It optimizes the sequence of tasks on FPGAs to reduce the impact of reconfiguration time on system performance while considering the priorities of individual tasks.

CORDS uses an evolutionary optimization infrastructure that incorporates numerous problem-specific heuristics. It is similar to the algorithms described in Chapters 5 and 6, and differs primarily by modeling FPGAs and scheduling tasks on FPGAs in a way that takes advantage of dynamic reconfiguration.

The rest of this chapter is organized as follows. In Section 9.2, we describe our model for FPGAs. Section 9.3 describes the scheduling algorithm run within CORDS. In Sections 9.4 and 9.5, we give the results of running CORDS on a collection of hardware-software co-synthesis problems and present conclusions.

9.2 FPGA model

In CORDS, each FPGA type is described by a set of scalars and vectors. A list of the scalars follows:

- price

- number of devices, i.e., configurable logic blocks (CLBs), on the FPGA
- input/output (I/O) pins available
- bits required to configure entire FPGA
- reconfiguration clock frequency
- energy per device switching event
- energy per I/O switching event
- proportion of clock cycles in which I/O pins switch

An FPGA type's vectors describe characteristics of each task type that may be executed on the FPGA type. Each vector contains an entry for each task type. A list of entries in these vectors follows:

- a Boolean variable indicating whether the task type is executable on the FPGA type
- worst-case execution time of the task type
- devices, e.g., CLBs, required to implement the task type
- energy per device switching event
- energy per I/O switching event
- proportion of clock cycles I/O pins switch
- proportion of clock cycles during which devices, e.g., CLB, are active
- number of input pins required by task type

- number of output pins required by task type

The uses of some of these variables are self-explanatory, e.g., FPGA prices are used to calculate embedded system prices. However, the purposes of some variables merit further explanation. An FPGA is overloaded if it doesn't have enough devices, e.g., CLBs, to implement the tasks assigned to it. The number of devices on an FPGA and the devices required for different task types variables are used to ensure that FPGAs are not overloaded. If a task type requires more devices than are available on an FPGA type, the task type may not be executed. In the algorithm described in this dissertation, we allow only one task type to execute on an FPGA at a time. However, this approach was subsequently extended to allow concurrent execution of different tasks in different portions of the same FPGA [155]. The variables associated with the number of input and output pins available on an FPGA are used to ensure that tasks will be assigned only to FPGAs with a sufficient number of pins. The number of bits required to reconfigure the entire FPGA variable allows the determination of the number of bits required to program one of the FPGA's devices. This, in conjunction with the number of devices required for a given task type, and the reconfiguration clock frequency, allows the reconfiguration time required for each task type to be determined.

9.3 Scheduling

In this section, we describe the scheduling algorithm used in CORDS. When the scheduling algorithm is invoked, CORDS has already determined PE allocations, communication link allocations, task assignments, and communication link connectivities. Thus, it is only necessary to determine the time at which each task is executed, the communication resource to which each communication event is assigned, and the time at which each communication event occurs. This problem is NP-hard for distributed

systems [112], and is further complicated by consideration of reconfiguration, i.e., on FPGAs, the amount of time a task requires depends on the previous and next task in the FPGA's schedule. We, therefore, resort to a heuristic scheduling algorithm. CORDS uses a static critical path scheduling algorithm with dynamic task reordering based on FPGA reconfiguration time. Reordering is dynamic but the resulting schedule is static, i.e., CORDS computes the time at which each event is carried out in order to determine whether or not hard deadlines are met by the schedule. Such guarantees are not possible, in general, when priorities are allowed to vary during the operation of the synthesized architecture.

Earliest finish times are computed by conducting a topological search of a task graph, starting from the node with no incoming edges, and assuming worst-case reconfiguration times for all tasks that are assigned to FPGAs. Latest finish times are computed by conducting a backward topological search of the task graph, starting from the nodes that have deadlines, and assuming worst-case reconfiguration times for all tasks that are assigned to FPGAs.

Reconfiguration delay is the amount of reconfiguration time an FPGA would require to change from the configuration capable of executing the task most recently scheduled on the FPGA, to a configuration capable of executing another task. Suppose two tasks, f and g, are both assigned to the same FPGA. If f was the task most recently scheduled to the FPGA, then the FPGA is configured to execute a task of f's type. If g is the same type of task as f, then the FPGA need not be reconfigured between their execution, otherwise the FPGA needs to be reconfigured. Some FPGAs are capable of partial reconfiguration. For such FPGAs, the reconfiguration time for a pair of configurations depends on the number of CLBs used by each configuration, in addition to the similarity between the configurations.

There is a reconfiguration delay associated with every task that is assigned to an FPGA. The reconfiguration delay for a task of type h, assigned to an FPGA whose most recently scheduled task was also of type h, is zero. Reconfiguration delay is dynamically adjusted during the execution of the scheduling algorithm. Every time a task is removed from the pending list, a dynamic check is first made to determine whether or not executing another task first would be likely to reduce total FPGA reconfiguration time without causing deadlines to be missed. *Dynamic priority* is defined to be the sum of a task's negative slack and its weighted negative reconfiguration delay. If s is slack, w is reconfiguration weight, and d is reconfiguration delay, then the dynamic priority, p, is as follows:

$$p = -s - w \cdot d$$

Reconfiguration weight is a positive scalar that is used to manipulate the contribution of reconfiguration delay to dynamic priority. In practice, a value ranging from one to two produces good results. We used a value of one for the experiments in this chapter. It may seem counter-intuitive to increase the dynamic priority of tasks with low reconfiguration times. However, this encourages similar tasks to be scheduled on an FPGA consecutively, reducing the amount of reconfiguration necessary. If the task, u, that was just removed from the pending list is assigned to an FPGA, then the dynamic priorities of all the other tasks in the pending list that are assigned to the same FPGA as u are compared with u's dynamic priority. If another task has a higher dynamic priority than u, it is removed from the pending list and scheduled immediately, after which time u is again considered for scheduling. When two tasks have equal dynamic priorities, the task belonging to the earlier copy of a task graph is scheduled first.

Suppose there are two tasks, l and m, in the pending list and assigned to the same FPGA. Suppose l has a slack of 4 ms and a reconfiguration delay of 5 ms, and task m has a slack of 8 ms. The task most recently scheduled to m's FPGA was of the same

type as m. Therefore, m's reconfiguration delay is 0 ms. Assuming a reconfiguration weight of one, task l has a dynamic priority of

$$-4ms - 5ms = -9ms$$

Task m has a dynamic priority of

$$-8ms - 0ms = -8ms$$

Thus, although task l has less slack than task m, i.e., it lies along a more critical path, task m will be scheduled first. Scheduling m before another task is scheduled to its FPGA is likely to reduce the reconfiguration time required. Consider, next, a comparison between task m and task n that has a slack of 1 ms, a reconfiguration delay of 5 ms, and a resulting dynamic priority of

$$-1ms - 5ms = -6ms$$

Although scheduling m first has the potential to reduce the reconfiguration time of m's FPGA, n's extremely low slack makes it dangerous to take a chance on delaying n. Therefore, n will be scheduled before m.

The first step of scheduling an individual task, t, is to schedule all of its incoming edges, i.e., communication events. Each edge is scheduled to a communication resource connecting the PE to which t is assigned and the PE to which t's parent is assigned. When multiple communication resources are available, CORDS selects the communication resource upon which the communication event will complete at the earliest time. If either of the communicating PEs does not have communication buffers, CORDS schedules the communication event to the unbuffered PEs, as well. If there are no communication resources connecting the PEs involved, CORDS notes this in the architecture's cost set as a unschedulable communication event.

9.4 Experimental results

We use a set of task graphs, processors, and communication resources produced by TGFF [129] based on information found in trade journals [156], datasheets [153], and discussions with a representative of Xilinx Corporation. The same optimization parameters, e.g., solution pool size, are used by CORDS for all of the examples within each of the following tables. Each of our 35 examples contains five task graphs. Each task graph contains an average of 20 tasks. There are 15 types of tasks, five types of processors, ten types of FPGAs, and five types of communication resources. The tightness of the deadlines differs from example to example. The depth of a task is the number of tasks on the longest path between it and the start task. The tasks in Example A1 that have deadlines, have an average deadline of 70 ms multiplied by the depth of the task. In each subsequent example, the average task deadline increases by 450 ms, multiplied by the depth of the task. Thus, the average task deadline in Example A5 is 1.87 s times the depth of the task. The seed given to TGFF's random number generator for each example is equivalent to that example's number, e.g., TGFF is seeded with three for Example A3 and Example C3. The processors have an average price of \$20, with a variability of \$10, i.e., processor prices range from \$10 to \$30. Tasks have an average execution time of 300 ms, with a variability of 285 ms, on the processors. Preemption time has an average of 150 μs with a variability of 140 μs. Execution time and preemption time are both inversely correlated to processor cost. Tasks executed on processors require an average of 40 KiB of memory, with a variability of 28 KiB. 9.7% of processors lack communication buffers. Communication resources have an average price of \$20 with a variability of \$10. Communication time is 50 μs per KiB, with a variability of 40 μs per KiB. Communication events have an average size of 42 KiB with a variability of 40 KiB. Memory has a price \$3.17 per MiB, with a minimum unit size of 256 KiB.

For FPGAs, average task execution time is 20 ms with a variability of 19 ms. The average task execution time on FPGAs, relative to the average task execution time on processors, is approximately one twelfth as high, a conservative estimate based upon the literature, in which speedups of 20-100 times are frequently reported. The average memory load of a task executed on an FPGA is 42 KiB with a variability of 28 KiB, in addition to the memory required to hold the CLB contents for the task. XC6200 family parts have price ranging from $200 to $400. The XC6200 family is a low-volume and high-cost part used primarily for research. Xilinx Corporation is, however, integrating many of the features present in the XC6200 family into high-volume Virtex parts. The prices given here are rounded to the nearest $100 at the request of a representative of Xilinx Corporation.. The average number of CLBs required by a task implemented on a 6200 family FPGA is 2000, with a variability of 1970. Task reconfiguration time for the 6200 family is 5 ns per CLB. The XC6216 provides 4096 CLBs. The XC6264 provides 16386 CLBs.

Eight XC4000 series parts are used in the examples. Their price ranges from approximately $30 to $400. Their CLB counts range from 100 to 1024. XC4000 series members do not support partial reconfiguration, i.e., each reconfiguration requires the entire FPGA to be programmed. Therefore, task CLB counts only affect the total memory requirements of the tasks, not their XC4000 series FPGA's reconfiguration time.

Note that although our algorithm and model support it, for these examples, we do not optimize power consumption. We are currently in the process of integrating our optimization infrastructure with scheduling algorithms [155] and a power model [157], [158] developed by a colleague.

For each example, CORDS required less than 15 CPU minutes on a 200 MHz Pentium Pro processor. *Deadline violation* is the amount by which an architecture overran its deadlines, as a percentage of the sum of the maximum deadlines in each copy of

Table 9.1: Resource modification experiments

Example	Price ($) or ⟨deadline viol. (%)⟩ w. processors only	Price ($) w. processors and XC4000s	Price ($) w. processors and XC6200
A1	⟨unsched.⟩	162	360
A2	⟨65.32⟩	32	175
A3	⟨1.47⟩	45	226
A4	⟨3.48⟩	66	346
A5	⟨0.15⟩	61	503
A6	89	39	65
A7	108	43	91
A8	60	23	32
A9	116	20	117
A10	38	29	38
A11	54	54	62
A12	16	16	16
A13	63	54	70
A14	34	36	34
A15	52	31	52

the task graph. When forced to use processors only, CORDS was unable to produce a solution for Example A1 in which all tasks were scheduled within the hyperperiod, even when deadline violations were allowed. The second column in Table 9.1 shows the best architectures produced by CORDS when it uses only processors. For high example numbers, in which deadlines are loose, processors alone are sufficient to produce valid architectures. For the examples with tighter deadlines, CORDS is able to synthesize valid architectures by using a combination of processors and FPGAs. The third column shows the best architectures produced by CORDS when using processors and XC4000 series FPGAs. The fourth column shows the best architectures produced when using processors and XC6200 family FPGAs.

Table 9.2: Conventional vs. rapid reconfiguration FPGAs

Example	Price ($) or ⟨deadline viol. (%)⟩ w. processors and XC4000s	Price ($) w. processors and XC6200s
B1	72	589
B2	⟨1.05⟩	178
B3	27	228
B4	⟨6.80⟩	647
B5	62	504

In general, by using XC4000 series and XC6200 family parts, CORDS was able to produce valid architectures for a number of examples that could not be solved using only processors. Using XC4000 series FPGAs typically resulted in a reduction of price, when compared to architectures using only processors. As a result of the high price of 6200 family parts, architectures containing processors and 6200 family parts are generally more expensive than architectures containing processors and 4000 series parts. However, in some cases the more rapid reconfiguration of 6200 family parts allows the satisfaction of specifications that are not met using only processors and 4000 series parts. This is especially true for examples in which reconfiguration time is similar to computation time. The examples shown in Table 9.2 differ from those in Table 9.1 in three ways: the amount of time spent executing tasks and communicating data are reduced such that reconfiguration time and execution time for tasks associated with a 4000 series part are similar, there are five task types instead of fifteen, and tasks with deadlines have an average deadline of 32 ms times the depth of the task. In general, when CORDS produces a valid architecture using either processors and 4000 series parts, or processors and 6200 family parts, the architecture composed of processors and 4000 series parts is less expensive. However, a design using processors and 6200 family parts

Table 9.3: Dynamic priority experiments for XC4000 series

Example	Price ($) or ⟨deadline viol. (%)⟩ w.o. dynamic priority	Price ($) w. dynamic priority	Price decrease (%)
C1	48	49	-2.08
C2	78	64	17.95
C3	56	25	55.36
C4	⟨0.02⟩	133	n.a.
C5	90	56	37.78
C6	32	33	-3.12
C7	81	77	4.94
C8	27	10	62.96
C9	90	51	43.33
C10	61	55	9.84
C11	62	67	-8.06
C12	25	10	60.00
C13	70	47	32.86
C14	72	34	52.78
C15	69	24	65.22

are sometimes capable of meeting specifications that are not met using processors and 4000 series parts.

The examples shown in Table 9.3 are different from those shown in Table 9.1 in one way: the tasks in examples in Table 9.3, which have deadlines, have an average deadline of 310 ms times the depth of the task. Table 9.3 compares the quality of the architectures produced by CORDS running in two different modes. The second column shows architectures produced when CORDS only considers static task slack during scheduling. The third column shows the architectures produced when CORDS reorders tasks based on their dynamic priorities. In example C4, reordering based on dynamic task priorities

allowed CORDS to produce a valid architecture when scheduling based on static priorities alone produced no architectures that met their deadlines. Reordering based on dynamic priority improved architecture price in 11 of the examples. For three examples, reordering resulted in a slight increase in price. However, for the 14 examples for which reordering resulted in a change in price, the average price reduction was approximately 30%.

9.5 Conclusion

CORDS is the first co-synthesis system to consider the effects of dynamically reconfiguring FPGAs during the operation of an embedded system, and reduce the amount of FPGA reconfiguration time. Experimental results indicate that time multiplexing tasks on dynamically reconfigurable FPGAs has the potential to decrease system price and allow otherwise infeasible specifications to be met.

Chapter 10

Analysis of Energy Consumption in Embedded Operating Systems

The increasing complexity and software content of embedded systems has led to the frequent use of system software to help applications access hardware resources easily and efficiently. In this chapter, we present a method for detailed analysis of real-time operating system (RTOS) power consumption. RTOSs form an important component of the system software layer. Despite the widespread use of, and significant role played by, RTOSs in mobile and low-power embedded systems, little is known about their power consumption characteristics. This work presents a method of producing a hierarchical energy consumption profile for applications as they interact with an RTOS. As a proof-of-concept, we use our infrastructure to produce the power profiles for a commercial RTOS, μC/OS [159], running several applications on an embedded system based on the Fujitsu SPARClite processor [160]. These examples demonstrate that an RTOS can consume a significant fraction of system power and, in addition, impact the power consumed by other software components. We discuss ways in which application software can be designed to use an RTOS in a power-efficient manner. We believe that this work is a first step towards establishing a systematic approach to power optimization of embedded systems containing RTOSs.

10.1 Introduction

Embedded systems often contain programmable processors and peripherals in addition to application-specific hardware. The complexity of applications and underlying hardware, tight performance and power budgets, as well as aggressive development schedules, require application developers to use run-time support software. This support usually takes the form of an RTOS, run-time libraries, and device drivers [161]–[167]. RTOSs are used in embedded systems with soft real-time constraints, as well as formal real-time systems with hard real-time constraints. In the interest of brevity, we will use the term RTOS to refer to all operating systems (OSs) targeting time-constrained embedded systems.

An RTOS provides a number of services to an embedded system designer. In Figure 10.1, the boxes at the upper-left corner depict different applications that may be run on an embedded system. The ovals depict the tasks composing a personal communication

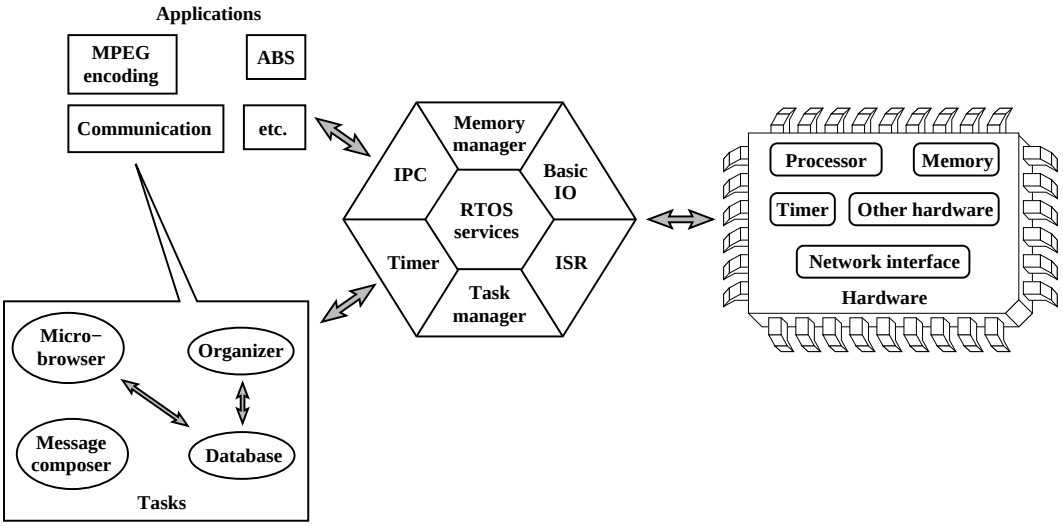

Figure 10.1: Overview of RTOS services.

device application. The arrows between these ovals represent communication or synchronization between tasks. The RTOS is depicted in the center of the figure. Hardware resources are shown to the right. An RTOS's services provide an interface between applications and an embedded system's hardware, thereby simplifying the work of application designers. For example, the RTOS provides the designer with timer management routines that may be used without detailed knowledge about the timer hardware in the embedded system.

In addition to simplifying the use of hardware, the interrupt service routines (ISRs) provided by an RTOS allow hardware to signal an application. The device driver and memory management portions of an RTOS simplify embedded system design by providing the designer with routines to ease the management of hardware resources. In addition, an RTOS manages the execution of, and interaction between, tasks in an application. It schedules the tasks in an application, ensuring that the highest-priority task has access to an embedded system's hardware resources at any given time. It also provides for communication and synchronization among tasks. In short, it manages the details of task interaction and provides a simplified interface to hardware resources.

Unlike general-purpose operating systems, RTOSs often sacrifice some functionality for the sake of compactness, predictability, and speed. A number of services typically provided by general-purpose operating systems are not useful in most embedded applications, e.g., support for multiple users or complex file-systems. By omitting such features, the size of an RTOS may be reduced, decreasing memory requirements and, therefore, embedded system cost. General-purpose operating systems usually try to complete their duties quickly. However, they typically do not provide a hard guarantee that a task will complete by a certain time. True RTOSs differ from general-purpose operating systems by making hard real-time guarantees about the time requirements of the critical services they provide. Note that some people refer to all embedded operating

systems as RTOSs, even if that do not provide hard guarantees on OS service execution times.

Typical applications involve significant use of RTOS primitives, the complex interactions among which are hidden from the application software developer. Although abstracting away the detailed behavior of RTOS services allows embedded system designers to more easily manage complexity, tight performance and power constraints sometimes demand more detailed analysis. An RTOS accounts for a significant fraction of the computational effort expended by an embedded system. Therefore, designers need to be aware of the potential performance and power impact of RTOS use. Commercial RTOS manuals and datasheets typically include estimates of the execution times for various parts of the RTOS running on specific hardware configurations. However, vendors do not provide information about RTOS power consumption characteristics. In addition, state-of-the-art techniques in embedded software power analysis do not clearly separate and analyze power consumed in RTOS components. We propose and demonstrate a method of conducting a detailed hierarchical analysis of the power consumption and execution time of embedded system applications running on a multi-tasking RTOS. In addition, our work is a first step towards analyzing and characterizing power consumptions of different RTOS components.

The rest of this chapter is organized as follows. Section 10.2 introduces related research and summarizes our contributions. Section 10.3 demonstrates the impact of the RTOS on embedded software energy consumption, using various illustrative examples. It also describes how insights into RTOS effects on energy can be used to optimize software to reduce energy consumption. Section 10.4 describes our energy analysis infrastructure, and presents an overview of the μC/OS RTOS. Section 10.5 presents quantitative experimental results on several example embedded software systems, on

which we base our analysis of RTOS energy effects. Section 10.6 concludes and makes recommendations to designers of low-power embedded systems that use RTOSs.

10.2 Related work and contributions

The importance of reducing power consumption in embedded systems has now been widely recognized, and a large body of work has focused on estimating, managing, and reducing power consumption in various system components. For hardware design, techniques have been developed to estimate and optimize power consumption starting from the algorithm and architectural design phases, down to the circuit design and technology optimization steps [168]–[172]. Application, semiconductor technology, cost, and time-to-market trends are causing a shift toward increased software content in embedded systems and systems-on-chip. As a result, designers and users of embedded software must be increasingly aware of power issues. While power dissipation is inherently a property of the underlying system hardware, a knowledge of the embedded software that runs on the hardware is useful in order to analyze and improve the system's power consumption characteristics.

Recognizing the important role played by embedded software in determining system power consumption, researchers have started to investigate techniques for software power analysis and power-efficient software design. Power analysis techniques have been proposed for embedded software based on instruction-level characterization [173] and simulation of the underlying hardware [174]. Techniques to improve the efficiency of software power analysis through statistical profiling have been proposed [175]. The system-on-chip design paradigm, which enables integration of processors, peripherals, busses, and complex user-defined logic blocks, has fueled research in

hardware and software power consumption estimation [176]–[180]. Reducing embedded software power consumption through compiler optimizations [181], source-level transformations [178], [182], customized memory management schemes [183], power management schemes [168], [184], device driver and operating system policies [185], and variable-voltage processors [186]–[190] has been investigated. Researchers have also investigated sophisticated methods of using operating systems to dynamically disable peripherals in order to save power [191]–[193]. Others have advocated re-designing page allocation and communication policies to decrease energy consumption [194].

Our work focuses on understanding and characterizing the power effects of RTOS and application software. Our goal is to provide designers with a method of determining the system-specific changes to the interaction between application software and RTOS that will most effectively reduce system power consumption. The steps required to reduce system power consumption are necessarily dependent on the specific RTOS and processor being used. We applied this method to the μC/OS RTOS [163] and applications running on the Fujitsu SPARClite processor. However, our method of hierarchically analyzing RTOS and application software power consumption [195] can be applied to different processors and RTOSs, e.g., an ARM processor running Linux [196]. Others have subsequently used a simulation-based approach to analyze RTOS power consumption [197], [198]. We modeled the SPARClite processor's sleep mode. It was observed that the RTOS, itself, can consume a significant amount of power. We present quantitative results for energy and time consumed by different operating system tasks, such as context switching, scheduling, inter-process communication, and timer management. In addition, we present concrete examples of the ways in which information derived from RTOS power analysis can be used to optimize embedded software power consumption. Our method of RTOS power analysis can be used for research on high-level

power-modeling of different RTOS components. These models can be incorporated into power-aware system-level design tools.

10.3 Motivation for RTOS energy analysis

In this section, we illustrate, with examples, the impact of an RTOS usage on system energy and time consumption. The RTOS energy analysis infrastructure described in Section 10.4 is used to provide a quantitative breakup of the energy and time consumed by different parts of the application and RTOS. Our analysis identifies the key sources of energy consumption in the system. Significant savings in energy consumption are obtained by re-writing the application software to use the RTOS in a more energy-efficient manner.

Energy consumption information is generally more useful, when optimizing an embedded systems's battery lifespan, than power consumption information. Even in situations requiring the optimization of power consumption, e.g., building an embedded system with limited short-term heat dissipation, one may frequently convert an energy-reduced system to a power-reduced system by reducing the system's clock rate, putting it in a reduced power consumption sleep mode part of the time, or reducing the voltage at which some of its components operate. Therefore, we focus on the energy consumption of a number of simulated embedded systems in this chapter. In addition, we give time consumption profiles for these examples. Note that the power consumption profile follows directly from the energy and time consumption profiles.

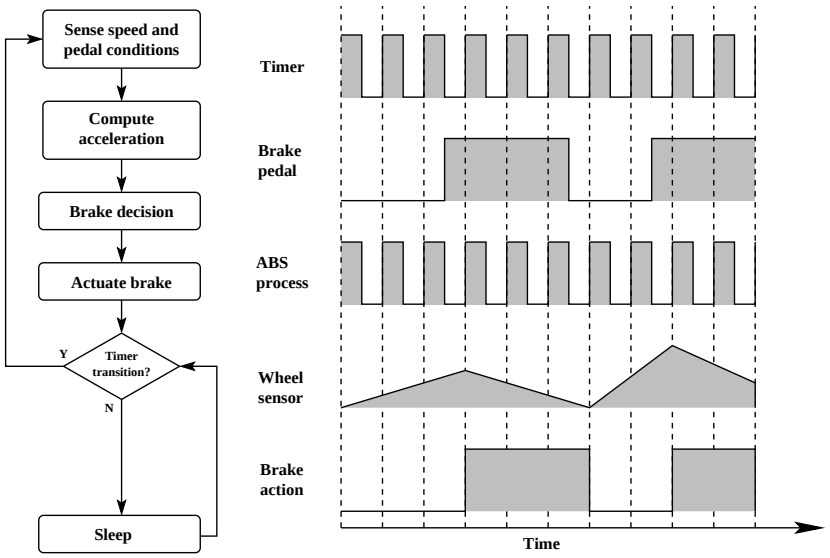

Figure 10.2: A straightforward implementation of the ABS example.

10.3.1 Anti-lock braking example

Our first example is based on embedded software used in an automotive anti-lock braking system (ABS) [199]. The system uses a timer wake-up signal to trigger execution of the ABS process. The flow chart shown in Figure 10.2 depicts part of an ABS. The ABS process calls the *Sense brake pedal* and *Sense speed* functions that sense the brake pedal and the current angular velocity of the wheel, respectively. It then computes the current speed and acceleration of the automobile, and uses the speed, acceleration, and brake pedal status to decide whether to apply the brakes, pump the brakes, release the brakes, or do nothing. This braking decision is conveyed to the *Actuate brake* function, which clamps the brake calipers, if appropriate. The simulated vehicle was subjected to an input trace during which its speed and brake pedal conditions change multiple times. The energy consumption profile is shown in the *non-gate* bar of Figure 10.4a.

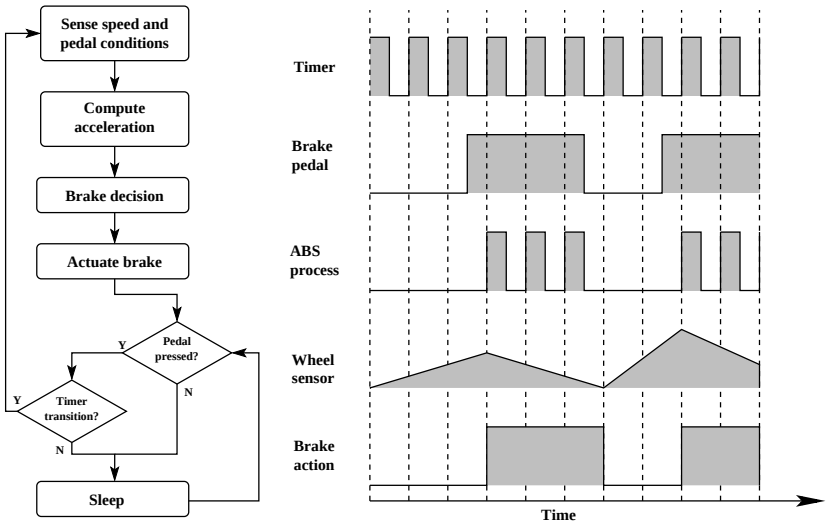

Figure 10.3: An energy-optimized implementation of the ABS example.

In the straightforward implementation of the ABS example, illustrated in Figure 10.2, the processor is awakened and the ABS process executes with every timer tick. Note that even this straightforward implementation is power-aware: it uses the processor's sleep mode between sensor sampling events instead of continuously leaving the processor in its high-power active mode. However, it frequently executes without changing the condition of the brake calipers. This unnecessary execution requires energy that might otherwise be conserved. By changing the algorithm slightly, such that it only wakes up the processor on a timer tick if the brake pedal is depressed (as shown in Figure 10.3), the embedded system's energy consumption is reduced. As shown in the *gated* energy bar of Figure 10.4a, the energy-optimized implementation of the ABS example consumes 65.0% less energy than the straightforward implementation. Most of the energy savings result from allowing the SPARClite processor to remain in the sleep mode, and the DRAM to remain in a low-power self-refresh mode, through timer ticks during which it is certain that the brake calipers need not be clamped. As the execution time in each case was 14 seconds (see Figure 10.4b), power consumption also

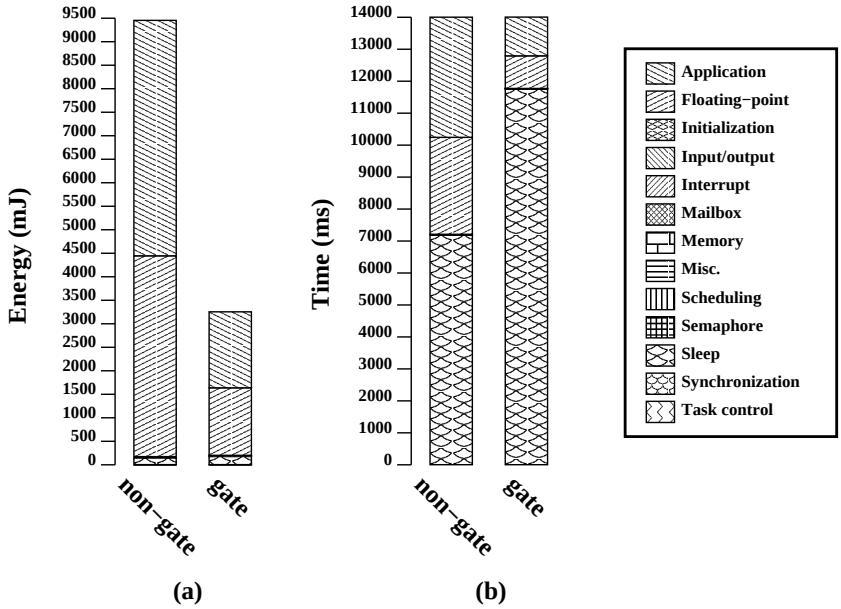

Figure 10.4: ABS example: (a) energy, and (b) execution time consumption by RTOS service category.

reduced by 65.0% in the energy-optimized version. In both versions of this example, operating system and board support services accounted for approximately half of the system's energy consumption. In this example, floating point service routines account for the majority of RTOS energy consumption. Although some of the functions listed in the bar chart's key account for little energy, we have listed all categories to keep the keys of different figures consistent.

10.3.2 Commodity trading agent example

In our second example, we consider a market composed of commodity trading agents. As shown in Figure 10.5, each agent has money, and four different types of commodities. The starting quantity of each commodity is randomly initialized. Randomly selected agents broadcast, to all other agents, their desire to sell a particular commodity. Agents receiving the broadcast respond with an offer price computed from

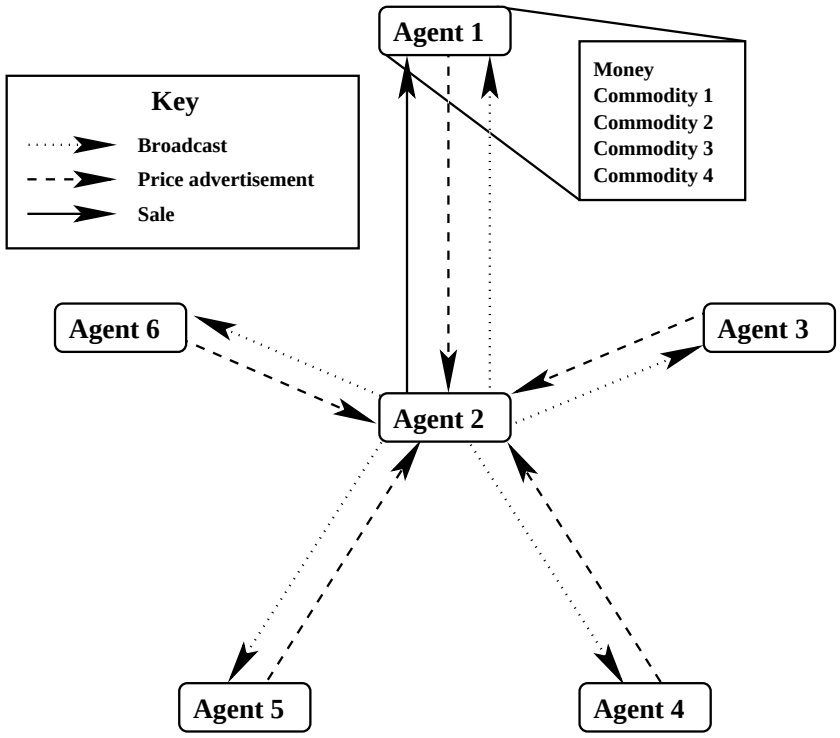

Figure 10.5: An overview of the commodity trading agent example.

the agent's supply-price curve for the commodity under consideration. The seller agent uses its supply-price curve to determine whether the highest received offer is higher than its internal valuation of the commodity under consideration at the quantity it currently owns. If so, it sells one unit of the commodity to the agent making the highest offer.

The *mail* bar of Figure 10.6a shows the energy consumption profile for an embedded system running the commodity trading example, when implemented using RTOS mailboxes to transmit messages between agents. In addition, the *mail* version relies on the RTOS scheduler to manage the activity of different agents. The *tuned* bar shows the energy consumption for code that is carefully hand-tuned to use shared memory for message communication, and avoid the use of RTOS mailboxes or scheduler. In the *mail*

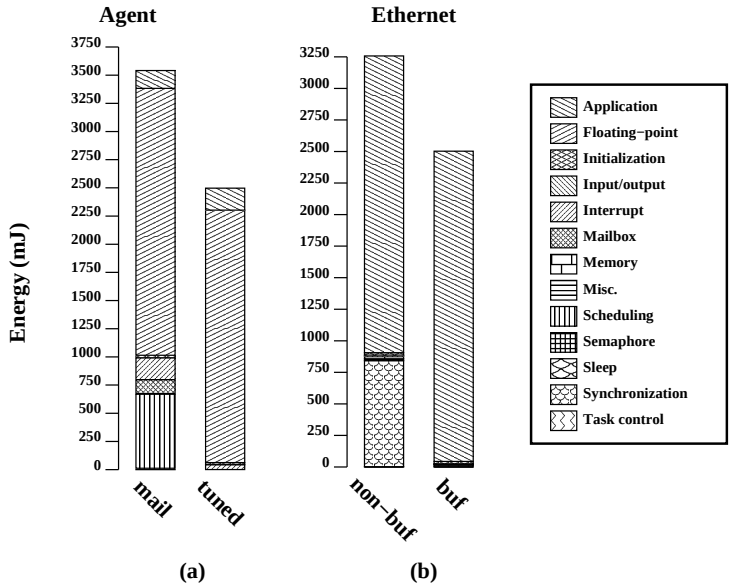

Figure 10.6: (a) Commodity trading agent example energy, and (b) Ethernet interface example energy by RTOS service category.

version, the RTOS is responsible for 95.5% of the embedded system's energy consumption. Interrupt handling, mailbox services, and scheduling, alone, account for 27.6% of the energy consumption. In the *tuned* version, the RTOS is responsible for 92.2% of the energy consumption. Interrupt handling, mailbox services, and scheduling account for 2.0% of the energy consumption.

As shown in Figure 10.6a, there is an energy cost associated with using the RTOS scheduler and mailboxes to allow a more versatile and maintainable implementation. The *tuned* version required only 70% of the energy of the *mail* version. However, adding new prioritized tasks to the *mail* version is simple, while changing the behavior of the *tuned* version is more difficult. In this case, a designer may trade off flexibility and maintainability for energy savings.

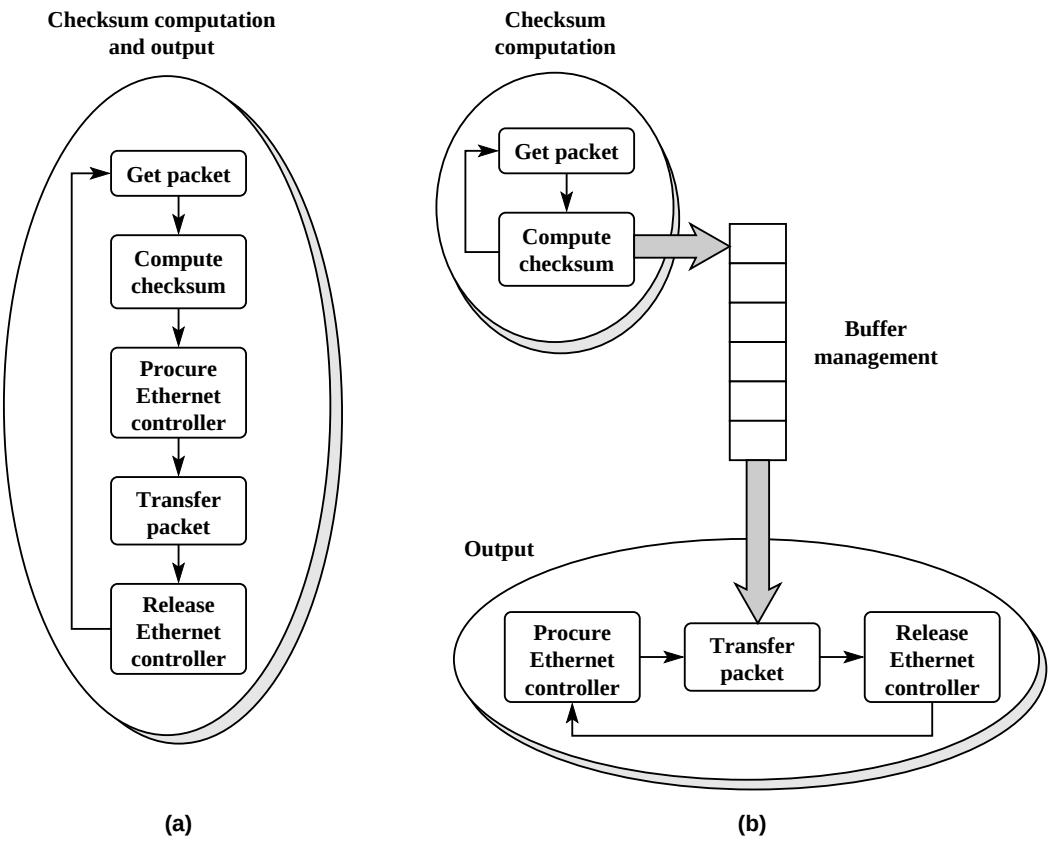

Figure 10.7: (a) A straightforward implementation, and (b) a multi-process implementation of the Ethernet interface example.

10.3.3 Ethernet interface example

In our third example, we consider checksum computation and interfacing with an Ethernet controller that has high per-access overhead. This action occurs at the lowest level of a TCP/IP protocol stack. Incoming packets are processed to derive their checksums. The packets are subsequently transmitted to the output device.

The most straightforward implementation of this algorithm, shown in Figure 10.7a, processes each packet as soon as it is available. However, in this example, preparing the Ethernet controller to receive a packet, represented by the *procure Ethernet controller* operation in Figure 10.7a, is costly. The *non-buf* bar in Figure 10.6b shows the energy

consumed by this straightforward implementation, broken down by RTOS service and application categories.

It is possible to amortize the cost of *procure Ethernet controller* over the transmission of multiple packets by decoupling packet generation from transmission to the Ethernet controller. In this energy-optimized implementation, the application is broken into three tasks, as shown in Figure 10.7b. The *checksum computation* task communicates packets to the *buffer management* task via shared memory. When the *buffer management* task has enqueued a number of packets, it transfers them to the *output* task that procures the Ethernet controller and transmits all the packets in its queue.

The *buf* energy bar in Figure 10.6b shows the energy consumed by the energy-optimized version of the Ethernet interface example. Although some energy or time is consumed by functions in each of the classifications listed in the key, some of these classifications account for very little energy or time consumption, and are barely visible in the bar charts.

Energy optimization of the Ethernet interface example results in a 23.1% overall decrease in energy consumption, with most of the savings resulting from reduced reliance on hardware access synchronization and initialization services. Power consumption reduced by 0.1%, i.e., the energy savings resulted from a reduction in execution time, not average power consumption. The energy saved in the hardware access synchronization and initialization services was sufficient to more than offset a 2.9% increase in energy resulting from the increased complexity of the multiple-task implementation. One could easily convert some of these energy savings into power savings by putting the processor and memory into sleep mode for the amount of time saved in the buffered version. In this example, the RTOS consumed only 1.2% of the overall energy in the version that

was not energy-optimized, and a similar percentage of overall energy in the energy-optimized version. However, in a number of other examples shown in Section 10.5, the RTOS consumes a substantial fraction of the embedded system's energy.

The examples presented in this section demonstrate the manner in which an RTOS power analysis infrastructure may be used to determine promising areas for power optimization and evaluate the tradeoffs between power and other costs. Understanding the effects of an RTOS on time and energy enables a designer to optimize the energy consumption of an embedded system.

10.4 Energy analysis infrastructure

In this section, we present our RTOS energy analysis framework. We first describe the inputs and outputs of our framework. Next, we present a high-level view of its building blocks, and the manner in which they interact to analyze the system energy consumption. We then present some details of individual building blocks.

10.4.1 Inputs and outputs

Our framework can be used to analyze the energy consumption of an application, consisting of multiple tasks, executing under a multi-tasking operating system. These tasks interact with each other, as well as with peripheral devices such as universal asynchronous receivers and transmitters (UARTs), brake sensors, and other hardware components. The embedded system is simulated to obtain a detailed report of the energy consumed by different application and RTOS functions.

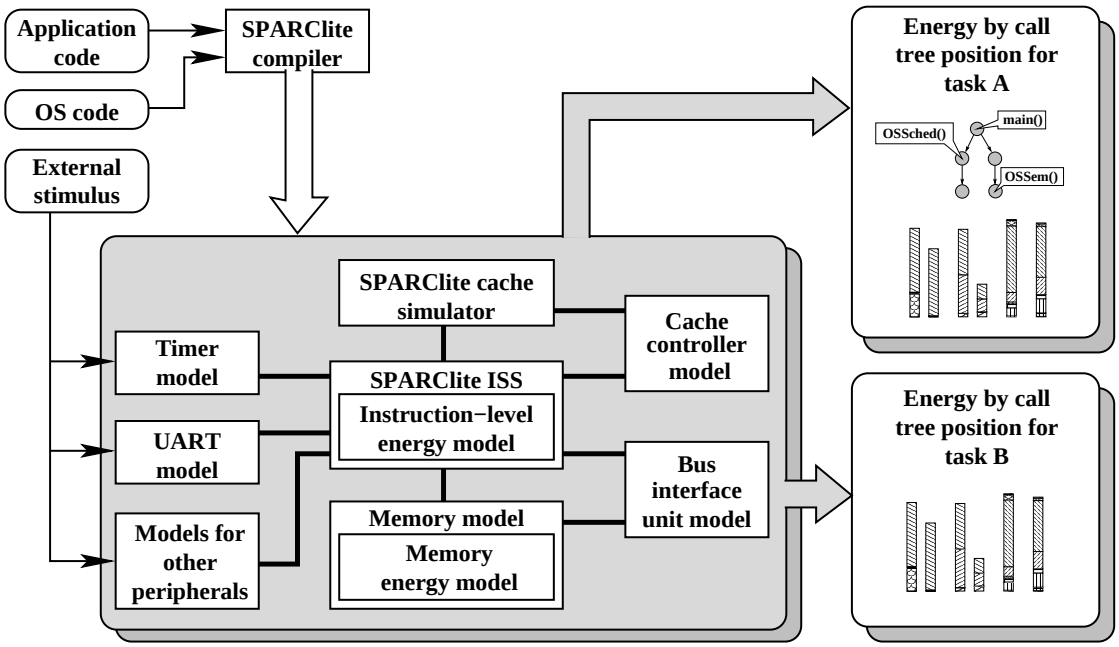

Figure 10.8: Energy analysis framework.

Figure 10.8 depicts our energy analysis framework. The application, which consists of multiple processes, is compiled and linked together with the μC/OS RTOS and Fujitsu's SPARClite run-time libraries. In addition, a model of the system's environment or external stimuli is provided to our framework.

The outputs of our software, shown at the right of Figure 10.8, include call-trees for each task and the RTOS. Each call-tree node corresponds to a function call, and has a child node for each function call instance that occurs within it. Recursive functions are supported. The time and energy resulting from recursive function invocations are recorded in that function's call tree node. An edge from function *foo* to function *bar* indicates that *foo* calls *bar*. The nodes of the call-tree are annotated with the functions they represent, and the energy and time consumed by each invocation of the function. The contributing sources of energy consumption within the function, e.g., instruction execution, stalls, dynamic random access memory (DRAM) refreshing, are recorded.

Table 10.1: Hierarchical call-tree for the semaphore example

		Function	$\frac{\text{Energy}(\mu J)}{\text{invocation}}$	Energy (%)	Time (ms)	Calls
`realstart` 25.40 mJ total 2.43 %	`init_tvecs`		1.31	0.00	0.00	1
	`init_timer` 18.01 mJ total 1.72 %	`liteled`	4.26	0.00	0.00	1
	`startup` 7.39 mJ total 0.71 %	`do_main`	7363.11	0.70	5.57	1
		`save_data`	5.08	0.00	0.00	1
		`init_data`	4.23	0.00	0.00	1
		`init_bss`	2.86	0.00	0.00	1
		`cache_on`	8.82	0.00	0.01	1
`Task1` 508.88 mJ total 48.69 %	`win_unf_trap`		6.09	1.16	9.43	1999
	`OSDisableInt`		0.98	0.09	0.82	1000
	`OSEnableInt`		1.07	0.10	0.92	1000
	`OSSemPend` 104.59 mJ total 10.01 %	`win_unf_trap`	6.00	0.57	4.56	999
		`OSDisableInt`	0.94	0.18	1.56	1999
		`OSEnableInt`	0.94	0.18	1.56	1999
		`OSEventTaskWait`	13.07	1.25	9.89	999
		`OSSched`	66.44	6.35	51.95	999
	`OSSemPost` 9.82 mJ total 0.94 %	`OSDisableInt`	0.96	0.09	0.78	1000
		`OSEnableInt`	0.98	0.09	0.81	1000
	`OSTimeGet` 4.62 mJ total 0.44 %	`OSDisableInt`	0.84	0.08	0.66	1000
		`OSEnableInt`	0.98	0.09	0.81	1000
	`CPUInit` 0.29 mJ total 0.03 %	`BSPInit`	3.52	0.00	0.00	1
		`exceptionHandler`	15.51	0.02	0.17	15
	`printf` 368.07 mJ total 35.22 %	`win_unf_trap`	6.18	0.59	4.87	1000
		`vfprintf`	355.04	33.97	257.55	1000

Note that if a function h is called from two functions f and g, we create separate nodes in the call-tree corresponding to these two scenarios. This ensures that the energy consumption statistics of a function are separated by caller. Each call instance's energy information can be examined separately or the call-instances may be combined in order to find the total energy consumed by all of the instances of a function located at a given position in the call-tree. At each position in the call-tree, detailed information is reported about the sources of energy consumption within the function. In addition, a total hierarchical energy consumption, equal to the sum of the total energy consumptions of a node's children, is given.

Table 10.1 shows a portion of the automatically formatted output of the system when analyzing a semaphore example. In this example, concurrent tasks are synchronized through the use of RTOS services. We present this table in order to give the reader a concrete idea of the sort of output the embedded system power analysis tool produces. Note that each context, e.g., *realstart* and *Task1*, is a separate start node in the call-tree hierarchy. The same function may appear more than once in the call-tree, if it is called from different locations, e.g., the window underflow trap service routine *win_unf_trap* in *Task1*. Although only energy per invocation, percentage of total energy, total time, and number of calls are displayed in this table, the analyzer also produces more detailed reports on embedded system attributes, e.g., it can separate energy consumption into sleep energy, stall energy, cache stall energy, memory access energy, memory idle energy, and instruction processing energy.

For the sake of brevity, the call-tree has been pruned to limit its depth and breadth. We have truncated the call-tree at a depth of three and omitted the *Task2* context. For example, the table shows information about the *realstart* and *Task1* contexts. *Task1* calls *OSSemPend* that, in turn, calls a number of other functions, including *OSSched*. Although *OSSched* calls other functions, they are omitted from the table for brevity. *OSSemPend* consumed 104.59 mJ, including the energy consumed by all of the other functions it calls. *OSSched* consumed 66.44 mJ per invocation and it is invoked 999 times at this position in the call-tree. Including the energy of the other functions it calls, it consumes 6.35% of the total system energy and executes for a total of 51.95 ms. Note that the figure produced by multiplying the energy consumption of each child function called by *OSSemPend* by the number of times the child function is called is slightly lower than *OSSemPend*'s total energy consumption. The difference between these figures is the amount of energy consumed by local instructions in *OSSemPend*, i.e., computations that do not involve function calls.

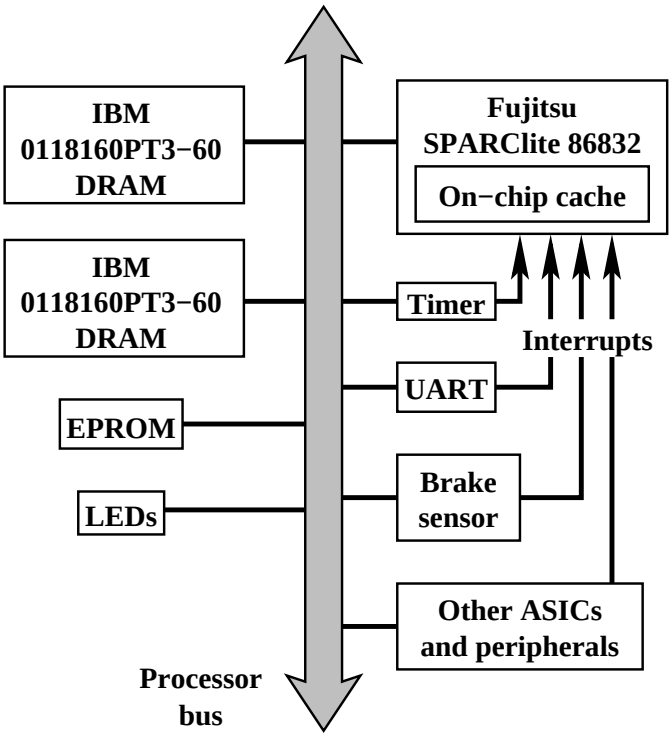

Figure 10.9: Modeled architecture.

10.4.2 System overview

We now describe the operation of our energy analysis framework. The simulated embedded system consists of a processor interacting with a set of application-specific integrated circuits (ASICs) and other peripherals. As shown in Figure 10.9, our energy analysis infrastructure models a Fujitsu SPARClite processor, connected to two fast page-mode DRAMs, a timer, a UART, and a number of other peripherals. Cycle-accurate simulators have a reputation for being slow. However, this approach is sufficiently fast to handle substantial applications; a similar simulation infrastructure subsequently built by colleagues booted Linux in less than five minutes on a Pentium III processor running at 667 MHz [196].

In order to analyze the energy consumption of the system, we need detailed functional models and energy models of its constituent parts. Instruction-level power models for the Fujitsu SPARClite processor and internal cache can be found in the literature [173]. The internal operation of the SPARClite processor is simulated using a power-aware version of an instruction set simulator (ISS) built by Li and Henkel [178] that was, in turn, built upon work by Ye et al. [200]. We modified this ISS to handle interaction with other components in the modeled embedded system. We have implemented an easy-to-use, object-oriented, inheritance-based method of adding new hardware to the simulated system, e.g., the brake sensors used in the ABS example. Application-specific devices may interrupt the operation of the processor. We use interrupt routines based on those found in the Fujitsu MB86832 evaluation kit, and μC/OS. Applications run under μC/OS. The addition of hardware interrupts to the embedded system simulator required significant changes to maintain correct simulation. In particular, it is not possible to use off-line hardware models in the presence of co-processor generated interrupts.

The ISS simulates the cycle-by-cycle execution of the processor, i.e., it accounts for effects such as branch delays, pipeline flushes, control-flow mispredictions, etc. We have enhanced this ISS in a number of ways. In order to account for the effects of cache misses, we added an on-line cache simulator designed specifically to model the SPARClite processor's cache. It is necessary to use an on-line cache simulator in order to know, during execution, whether or not a cache miss has occurred. An off-line cache simulator would not allow the correct simulation of an embedded system because, due to races with interrupts generated by other peripherals, the presence or absence of a miss penalty may change the flow of execution. The cache simulator accounts for the cache and memory behavior. We model a number of SPARClite-specific features. Among these, low-power sleep mode is particularly important. In addition, we model

external memory. Specifically, we simulate the cache and on-board bus interface unit of a Fujitsu MB86832 [160], [201], as well as the operation of two IBM0118160PT3-60 low-power fast page-mode DRAMs [202]. Memory energy consumption is derived from the manufacturer's data-sheet, and depends on the DRAM's mode of operation. We consider the energy required to drive the processor-memory bus. Our power model is built from datasheets [202] and published current measurements [173]. If the hardware implementation of an additional device a designer wants to integrate into the system is known, its energy consumption can be computed using known energy analysis techniques [168], [169], [172].

As mentioned earlier, our energy analysis framework organizes energy consumption data by function. Therefore, in addition to evaluating the energy consumed by the system in a cycle, our energy analyzer needs to keep track of the function and process that are currently being executed. In general, the manner in which the context is determined is specific to the operating system, and the processor being considered. μC/OS performs scheduling and context switch occurs through the function *OSSched*. Our framework uses this information to keep track of context switches. Function calls are performed using the *jmpl* instruction from the SPARC assembly language. The name of the function to which control flow is transferred is determined from the symbol table. The symbol table associates an address with each function and global variable. The problem of tracking returns from function calls is complex and requires information specific to the instruction set architecture of the processor being used, the manner in which the compiler translates different control-flow constructs in the high-level programming language into assembly code, and information specific to the RTOS code that performs context switching.

Our energy analysis technique is non-intrusive. This differs with many well-known software debugging and performance analysis techniques that augment the program to

be analyzed with monitoring code in order to enhance observability of the program state and internals. While the addition of monitoring code eases analysis, it results in a loss of accuracy because the monitoring code modifies the parameters that needs to be measured: execution time and energy. Additionally, this extra code may change the order in which tasks execute in an embedded system containing multiple hardware devices. The need to perform cycle-accurate performance analysis is heightened in the presence of external devices that communicate with the processor. Inaccuracies in timing can cause a change in the functionality of the system being implemented, leading to inaccurate control-flow and energy results. Since we use cycle-accurate processor and cache energy models, our framework does not suffer from this problem. When run on a 336 MHz UltraSPARC-II with four gigabytes of memory, the simulator takes approximately 40 minutes to simulate the 14-second original version (i.e., non-gate) of the ABS example and approximately 12 minutes to simulate the 2.5-second original version (i.e., non-buf) of the Ethernet interface example.

There is one caveat regarding the power model used for the SPARClite processor. We selected the Fujitsu SPARClite MB86832 for simulation because an evaluation kit for this processor is currently available from Fujitsu, allowing us to use their development tool's electrically programmable read-only memory (EPROM) code to facilitate the simulation of a concrete embedded system core. However, we do not currently have a power model for the MB86832. We used the instruction-level power model for the Fujitsu SPARClite MB86934 that is available [173]. The core clock frequency for the modeled processor is 80 MHz, while the core clock frequency used to build the power model is 20 MHz. The I/O clock frequency for the modeled processor is 26.7 MHz, while the core clock frequency used to build the power model is 10 MHz. It was necessary to scale the current values in the power model in order to account for the increased core clock frequency. According to the MB86832 data-sheet, current scales linearly with

clock frequency [201]. This behavior is to be expected for conventional, low-leakage CMOS processes. The instruction-level power model does not separate the power consumed in the processor core from the power consumed in the I/O circuits. We relied on the relative contributions given in the MB86934 data-sheet in order to scale the separate components of the overall current correctly [203].

10.4.3 System details

In this section, we describe the operation of two key components of our target system architecture: the processor and the operating system. We first present an overview of the processor, and then briefly describe the μC/OS RTOS.

Our system is built around a Fujitsu SPARClite MB86832, a 32-bit RISC processor, operating at 80 MHz, with an external bus speed of 26.7 MHz. It implements a superset of the SPARC v8 instruction set architecture. Its integer unit has a five-stage pipeline that can handle data interlocks, and a branch handler to perform control-flow transfers efficiently. The bus interface unit is capable of providing single-cycle access to the on-chip cache. The processor has 136 registers, organized into eight overlapping register windows, and 8 KiB instruction and data caches. Multiply and divide operations are supported by dedicated, on-chip hardware that can complete 32-bit multiplications in five cycles. The processor also has a power-down mode that can be employed to reduce energy consumption.

We have taken care to simulate the context-dependent IBM0118160PT3-60 memory and MB86832 bus interface unit timing in sufficient detail to ensure that memory accesses require the number of cycles implied by the timing diagrams in the specifications. In addition, we simulate stalls resulting from periodic distributed DRAM refreshes.

μC/OS is Jean Labrosse's portable real-time kernel for microprocessors and microcontrollers. We use the version Brad Denniston ported to the MB86832 processor.

μC/OS has been used in many commercial applications, and its performance is comparable to that of other commercial RTOSs. μC/OS supports multitasking, and can handle up to 63 concurrent processes. The kernel is fully preemptive. The RTOS is designed to be scalable, i.e., designers who do not require some of its features may save memory by easily building a light-weight version. The RTOS provides a number of services such as scheduling, task management, inter-process communication, memory management, interrupt handling, and timer-related services. We chose μC/OS for our experiments because it is modular, well-designed, and well-documented; its source code is readily available. Further information on μC/OS can be found on the Internet at http://www.uCOS-II.com, or in Labrosse's book [163].

10.4.4 Extending our approach to other embedded systems

Our approach for analyzing RTOS and application software power consumption can be extended to other processors and operating systems. However, there are system-dependent components in this approach.

It is necessary to have ISSs for the processors used in the target embedded system. There must be a method for tracing the status of the simulated processor cycle by cycle, in order to record energy consumption, detect context switches, and simulate interaction with other hardware in the embedded system. Although it is conceivable for an ISS to provide a run-time interface meeting these requirements, it is our belief that, in practice, the ISS source code will be required. ISSs are available for a number of popular architectures. Vendors sometimes provide simulators for more exotic processors. A designer who wants to use our power analysis method on complex processors for which ISSs are not available will face a substantial burden. Fortunately, getting access to simulation modules for system-specific ASICs is likely to be straightforward, as the in-house simulators used to design and debug the ASICs are likely to be available.

Ideally, the source code of the RTOS, including low-level system support software, will be available. Our approach is useful even if the RTOS source code is not available. However, in this case it will be more difficult to apply. It is necessary for the embedded system simulator to detect context switches. However, the way in which a context switches is RTOS-dependent. If the RTOS source code is not available, it is necessary to learn how the RTOS handles context switches based on disassembled binaries and EPROM images. In addition, it is important for the designer to understand how different components of an RTOS interact in order to best optimize its usage. Unless the documentation of the RTOS is detailed, a designer interested in making the best possible use of the RTOS without access to its source code will be forced to learn about its operation by tracing its execution at the instruction level, or by disassembling it. This sort of reverse engineering can be time-consuming and costly. However, even in the absence of the RTOS source code, our approach remains useful. By indicating which RTOS services have high energy consumption, it allows the designer to focus attention on understanding, i.e., reverse engineering, those services.

Unless power consumption was a primary consideration in RTOS design, minor changes to an RTOS can significantly improve its power consumption characteristics. A feature of μC/OS provides support for this observation. When no user-defined processes are running, an idle task executes. Normally, this task repeatedly increments a variable. By comparing the actual number of increments in a given time-span with the maximum number of increments possible in that time-span, μC/OS keeps track of the percentage of time spent idle. This behavior is beneficial, as long as one is not trying to minimize power consumption. There are sophisticated approaches one could use to dramatically reduce idle power consumption. However, even the straightforward expedient of preventing the variable from being incremented eliminates numerous writes to the processor's write-through cache, thereby reducing memory power consumption.

The ability to make changes to the source code of an RTOS increases the designer's flexibility in optimizing embedded system power consumption. However, even if the source code is not available, our approach allows a designer to modify the use of RTOS services in order to reduce power consumption.

Finally, it is necessary to have power models for the embedded system devices that consume a significant amount of power. It is our hope that, in the future, hardware vendors will see the competitive advantage of providing customers with detailed power information about their products. Until this practice becomes common, designers who want to apply our approach will be forced to rely on power models and analysis techniques found in the literature [168], [169], [172], [173], internally developed power models, or the limited power information found in conventional datasheets. Note that, for some processors, this power information is sufficient to allow a reasonable estimate of power consumption.

10.5 Results and case studies

We analyzed the energy consumption of μC/OS RTOS when running several embedded applications. In all cases, we targeted the Fujitsu SPARClite processor based embedded system presented in Section 10.4.2. Some applications were abstracted from real embedded system application software, while others were designed to exercise specific RTOS functions and services. Overall, care was taken to ensure that key RTOS functions and services were used by the chosen applications.

For each example, we categorized energy consumption by RTOS and application service type, as explained in the following list.

- **Application**: Non-RTOS functions.
- **Floating-point**: Integer operations to simulate floating point math.

- **Initialization**: Embedded system initialization functions. This is typically executed only once during an application's run.
- **Input/output**: Input and output formatting and communication with the system's UART channels.
- **Interrupt**: Interrupt service routines.
- **Mailbox**: Code to handle task communication with mailboxes.
- **Memory**: Memory initialization, allocation, and copying functions.
- **Misc.**: Functions not in other categories.
- **Scheduling**: Task scheduling.
- **Semaphore**: Semaphore-based task synchronization code.
- **Sleep**: Sleep mode.
- **Synchronization**: Non-semaphore-based task synchronization code.
- **Task control**: Task management, e.g., task creation.

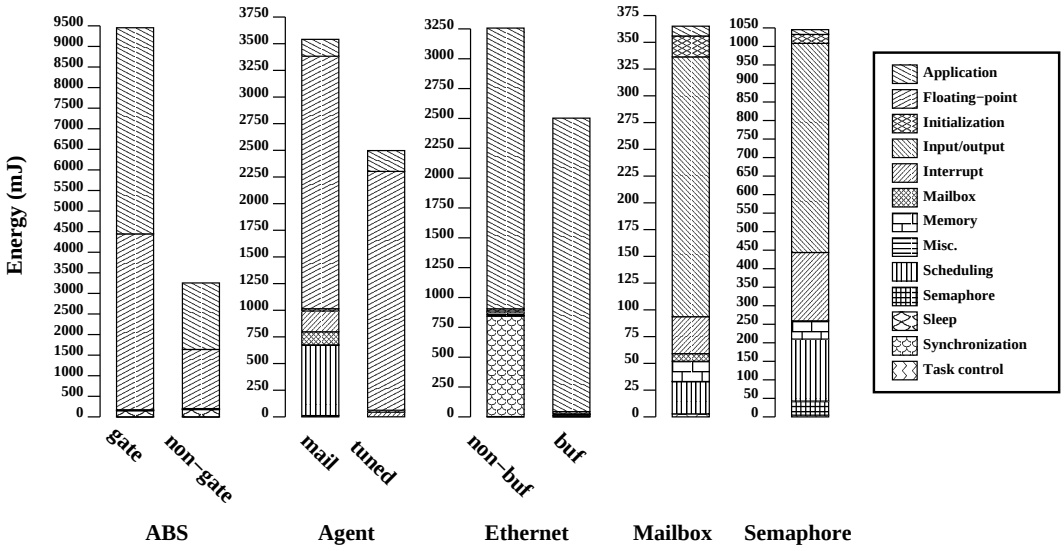

Figure 10.10: Energy consumption profiles.

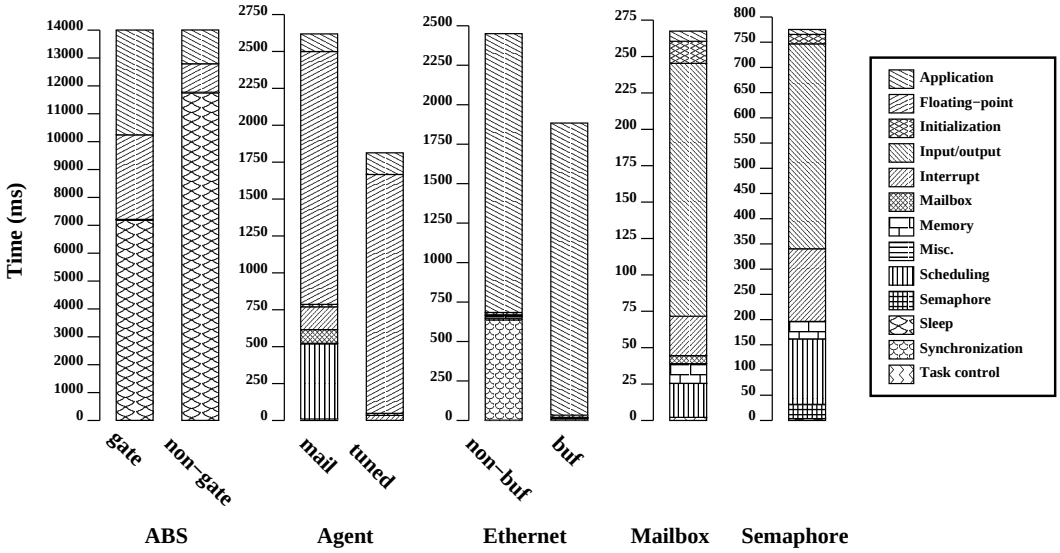

Figure 10.11: Time consumption profiles.

Figure 10.10 shows the energy consumed by different RTOS services and the applications, themselves. Each vertical bar represents a distinct example. Vertical bars are divided to indicate functions. For instance, in the mailbox example, I/O primitives used by the RTOS account for a larger portion of the energy consumption than any other function category. Figure 10.11 presents a similarly formatted breakdown of time consumption by RTOS service and function category.

The Ethernet and ABS examples are described in Section 10.3. The ratio of processor energy consumption to DRAM energy consumption varied from 2.71 (in the energy-optimized version of the Ethernet interface example) to 2.94 (in the energy-optimized version of the ABS example). The results in this section, and in Section 10.3, indicate that an embedded system's RTOS may be directly responsible for a significant portion of the embedded system's energy consumption. The percentage of system energy directly consumed by the RTOS may vary dramatically from approximately 1% (in the energy-optimized version of the Ethernet interface example) to 99% (in the mailbox example), depending on the degree to which the application code relies on RTOS services. Even when the RTOS does not directly consume a significant percentage of the system's energy, one can significantly reduce overall energy consumption by more wisely using RTOS services, as demonstrated by the different versions of the ABS example.

The mailbox example illustrates the use of mailboxes for inter-process communication. It consists of three application tasks that communicate via the shared memory mailbox communication service provided by μC/OS. The tasks also perform writes to the UART. Figure 10.10 shows that, in this example, the main sources of energy consumption are input/output primitives, interrupt service routines, task scheduling, as well as RTOS and processor initialization code. Mailbox management services also consume a small but significant fraction of the system's energy. Formatting and transmitting data to the UART can be energy-intensive, and should be sparingly used in an

energy-constrained implementation. The application code relies heavily on RTOS and processor support routines. As a result, the application code only consumes 1.0% of the total system energy, with RTOS and processor support services consuming the other 99.0%.

In the semaphore example, concurrent tasks are synchronized through the use of RTOS services. RTOS primitives that post and release semaphores account for a small but significant portion of the system's energy consumption. The application code consumed 1.2% of the total system energy, with RTOS and processor support services consuming the other 98.8%.

From the results presented above, one can observe that the embedded system consumed significantly less power during sleep mode (14.2–18.0 mW depending on example) than when running in other modes. As described in Section 10.4, a call-tree node holds the total time and energy of all function calls located at a given position in the call-tree. The average power consumption of call-tree nodes, i.e., context-dependent function execution, varied from 769 mW (*OSEnableInt*) to 1,047 mW (*uart_delay*). However, the differences among the power consumption of RTOS service classes were smaller. Average RTOS service class power consumption varied from 842 mW (for interrupt service routines) to 976 mW (for floating-point routines). While there was a strong correlation between execution time and energy consumption for the examples in which sleep mode was not used, it would be unwise to generalize this observation to all embedded systems. In embedded systems containing peripheral processors that consume a substantial amount of energy, and whose control is relegated to a subset of the RTOS service classes, there would be substantial differences between the power consumptions of different RTOS service and function categories.

Table 10.2: RTOS service energy per invocation

Service	Minimum energy (μJ)	Maximum energy (μJ)
AgentTask	3.13	4727.88
fpdiv_parts	4.23	261.22
BSPInit	3.52	3.55
fpmul	21.57	40.66
CPUInit	286.98	291.39
fpmul_parts	4.73	43.83
GetPsr	0.38	0.55
fptodp	17.46	49.72
GetTbr	0.38	0.67
fstat	4.61	16.34
InitTimer	2.53	2.56
fstat_r	7.83	31.42
OSCtxSw	46.63	65.65
init_bss	2.86	3.07
OSDisableInt	0.84	1.31
init_data	4.23	4.51
OSEnableInt	0.84	1.31
init_timer	18012.10	20347.00
OSEventTaskRdy	26.45	30.54
init_tvecs	1.31	1.31
OSEventTaskWait	11.62	13.75
isatty	1.77	1.77
OSEventWaitListInit	30.35	31.48
liteled	4.26	4.26
OSInit	7036.20	7057.59
litodp	10.22	225.33
OSMboxCreate	41.04	43.25
localeconv	1.74	2.35
OSMboxPend	10.11	130.59
localeconv_r	0.42	0.83
OSMboxPost	7.78	129.06
lshrdi3	2.63	3.37
OSMemCreate	31.37	31.61
make_dp	9.87	40.44
OSMemGet	10.00	12.34
malloc_r	71.09	71.50

Continued on next page.

Table 10.2: RTOS service energy per invocation (continued)

Service	Minimum energy (μJ)	Maximum energy (μJ)
OSMemInit	4432.06	4432.59
mbtowc	3.21	4.07
OSMemPut	9.71	11.89
memchr	1.95	15.19
OSQInit	60.02	62.72
memmove	3.91	20.67
OSSched	10.24	80.73
morecore_r	57.07	57.27
OSSemCreate	41.60	43.40
pack_d	6.01	24.65
OSSemPend	9.83	112.72
pack_f	3.49	7.66
OSSemPost	9.24	115.69
printf	367.52	890.27
OSStartHighRdy	20.53	20.82
putCharPort1	19.22	32.51
OSTCBInit	42.31	45.68
putchar	6.78	7.40
OSTaskCreate	84.28	87.98
putchar_r	5.56	6.07
OSTaskCreateExt	2123.10	2145.03
putstr	64.01	66.09
OSTaskCreateHook	1.92	1.94
rand	2.35	3.15
OSTaskStkInit	16.54	31.76
rand_range	912.52	1003.22
OSTaskSwHook	0.53	1.13
rdtbr	0.38	0.88
OSTimeGet	4.62	5.29
rint	3.70	435.11
Roulette	957.48	5684.69
save_data	5.08	5.22
agent_broadcast	990.70	4714.15
sbrk	4.86	19.06
agent_buy	7.22	8.94
sbrk_r	7.14	33.56

Continued on next page.

Table 10.2: RTOS service energy per invocation (continued)

Service	Minimum energy (μJ)	Maximum energy (μJ)
agent_init	71.19	211.09
sfvwrite	44.19	648.40
agent_offer	239.13	1279.00
sinit	35.45	36.31
agent_price	227.02	830.43
sitofp	7.67	86.79
agent_sell	6.26	933.14
smakebuf	94.37	118.08
cache_off	3.18	3.18
sprint	47.51	651.70
cache_on	8.68	8.82
std	8.95	9.36
do_global_ctors	3.26	3.26
swbuf	152.65	152.65
dpadd	31.31	139.92
swrite	149.93	607.27
dpcmp	18.68	23.27
swsetup	101.59	125.31
dpdiv	28.58	291.14
uart_delay	14.25	14.68
dpmul	29.08	74.04
unpack_d	5.24	8.59
dpsub	26.54	286.74
unpack_f	3.60	6.10
dptoli	8.44	16.75
vfprintf	354.51	872.99
exceptionHandler	15.26	18.86
vfprintf_r	346.54	859.51
fflush	159.41	625.94
win_ovf_trap	11.25	12.09
fpadd_parts	3.59	255.83
win_unf_trap	6.00	11.84
fpcmp_parts	3.47	5.76
write	143.41	577.06
fpdiv	21.17	72.81
write_r	146.30	591.68

Table 10.2 shows the minimum and maximum energy per invocation for each RTOS service, board support package routine, and standard library routine used in our examples. These routines might consume less energy than the minimum in the table, or more energy in the maximum in the table, if they are used in a manner not encountered in any of our examples. However, for applications similar to our examples, these values provide a reasonable range for the energy costs of RTOS services and other support routines.

10.6 Conclusions and recommendations

In this chapter, we have described the design and implementation of an RTOS power analysis infrastructure. Examples were presented to illustrate the application of this infrastructure. By analyzing a commercial RTOS, μC/OS, running several applications, we have demonstrated that the manner in which the RTOS is used has a significant impact on an embedded system's power consumption. Insights derived from such RTOS power analysis may be used to optimize embedded software power consumption and drive research on high-level power modeling of different RTOS components. Furthermore, this work enables power-efficient RTOS and application design, and may be incorporated into power-aware system-level design tools.

Based upon our observations, we have found a few general guidelines that designers should follow in order to use an RTOS in a power-efficient way. However, before presenting these guidelines, we must first mention a few caveats. The most power-efficient implementation of embedded system software is processor-dependent and RTOS-dependent. We strongly suggest implementing a prototype without expending heroic efforts on low-level power optimization. One should start trading off code flexibility and maintainability for power efficiency only after it is clear, e.g., via energy

profiling, which portion of the RTOS or application code is unnecessarily consuming power. The guidelines we present, here, are no substitute for using a detailed power analysis infrastructure, of the sort presented in this chapter, during the design of an embedded system.

A number of energy reduction options are available to an embedded system designer with access to an RTOS, as follows.

- Rewrite high energy consumption portions of an application to avoid unnecessary use of the RTOS scheduler.

- When synchronization between tasks is implicitly carried out, do not use RTOS services to carry out (redundant) synchronization. This may be easier said than done because redundant synchronization can make code more robust.

- Take advantage of RTOS primitives, e.g., process support, to allow easy implementation of multi-process schemes that amortize the costs of high-overhead operations.

- If power analysis indicates that memory management consumes a substantial proportion of embedded system power, consider custom, e.g., uniform block, memory management for commonly allocated and deallocated data types.

- Concentrate on special modes available in the processor. Most designers already pay some attention to code execution time and, in the absence of special processor modes, there is a strong correlation between execution time and energy for general-purpose processors. However, using special processor modes, e.g., sleep mode, can dramatically reduce power consumption. One can leverage an RTOS to easily retrofit an existing application for power reduction, e.g., one may use a low-priority task that puts a processor into sleep mode.

We emphasize that the above recommendations are not exhaustive; they will not be beneficial for every embedded system. Our strongest suggestion is to examine an embedded system's RTOS and application energy profile before attempting to power-optimize code.

Chapter 11

Comparisons with Related Work

In this chapter, we describe three algorithms that are closely related to our work and point out differences with our work.

Axelsson compared the performance of a tabu search algorithm, a simulated annealing algorithm, and a genetic algorithm on the real-time partitioning problem [65]. His tabu search algorithm and simulated annealing algorithm had better performance than his genetic algorithm. However, it would be dangerous to consider his three algorithm instances to be representatives of the three algorithm classes. In addition, it would be dangerous to generalize results obtained for the real-time partitioning problem to the hardware-software co-synthesis problem.

An early version of MOGAC, running in simulated annealing mode, was capable of generating high-quality solutions for problem instances in which link synthesis was not necessary. Even in these instances, adding crossover resulted in faster convergence to results with the same quality. The main problem with Axelsson's results is that they compare naive versions of the three algorithm types: simulated annealing, genetic, and tabu search. Although it is easier to get simulated annealing and tabu search functioning at a basic level, than it is to get genetic algorithms working well, simulated annealing and tabu search lack the ability to share information between different solutions. In addition, simulated annealing and tabu search are poorly suited to multiobjective optimization,

when compared with genetic algorithms. In short, Axelsson's comparison is not relevant to the problems we are dealing with for the following reasons:

- His work solves a significantly simpler problem than that tackled by our co-synthesis and system synthesis software.
- The genetic algorithm he implemented does not attempt to preserve locality.
- His conclusion, "... genetic algorithms are less suitable, due to the difficulty in defining a reasonable crossover operator," is nearly correct. However, he has succeeded in finding a fault only with his implementation, not with genetic algorithms, in general.

In our evolutionary algorithms, care is taken to preserve locality in string encodings and crossover. We use a clustering method to prevent the production of structurally invalid solutions. Although the resulting genetic algorithm is complicated, it is effective. In summary, Axelsson's results are interesting and valuable. However, they should not be generalized beyond their proper scopes.

Teich et al. applied a multiobjective genetic algorithm to the heterogeneous distributed system co-synthesis problem [79]. Their approach does not target systems with hard real-time constraints. Power consumption is not considered. Multi-rate systems, and systems containing task graphs with periods less than their deadlines, are not handled. They use a method of crossover that randomly selects bits to swap and does not attempt to preserve sequences of bits describing related attributes, i.e., it does not attempt to preserve locality. As described in Section 4.3, if n is the solution pool size, this may result in up to an $\mathcal{O}\left(n^2\right)$ slowdown in the rate at which solutions are implicitly evaluated, when compared to a genetic algorithm using a locality preserving crossover method [120]. In this work, solutions that are not valid, and that cannot be made valid

by the application of a repair operator, are immediately terminated. Multiobjective optimization is not performed. Their experimental results consist of one small example and no comparisons are made with other co-synthesis systems.

Oh and Ha applied a heuristic to the heterogeneous distributed system co-synthesis problem [75]. They compare the results produced by their algorithm with those produced by MOGAC. Their algorithm is able to find lower-price solutions than MOGAC using less CPU time for some problem. However, we have subsequently improved our optimization infrastructure, allowing our algorithm to produce superior solutions to two of the task sets for which they reported results, as shown in Section 6.9.

Chapter 12

Contributions and Conclusions

We have presented algorithms for hardware-software co-synthesis and embedded system synthesis. The optimization framework, upon which they are built, produces solutions to the conventional co-synthesis problem that match or surpass those produced by prior work. The CPU times required by these algorithms are often orders of magnitude better than many prior algorithms. Their CPU time requirements increase slowly with increasing problem complexity. In addition, we have presented a framework for the energy analysis of real-time operating systems (RTOSs).

Although we carefully compared the results produced by our optimization framework with those produced by prior art, our primary goal was solving new problems. Each algorithm tackles a different class of embedded systems and considers essential details that have typically been ignored in past work, for the sake of simplicity. Comparisons between these algorithms and alternative approaches that do not model physical details with as much accuracy indicate that it is important for an embedded system synthesize algorithm to consider some detailed physical realities, even during high-level design.

We believe that embedded system synthesis is an inherently multiobjective problem and, to the best of our knowledge, we were the first to formulate hardware-software co-synthesis as a multi-objective optimization problem.

We were the first to formulate and solve the heterogeneous system-on-chip synthesis problem. We consider intellectual property (IP) core clock selection. Our algorithms take physical realities, e.g., routing congestion, wire delay, and bus topology, into consideration. We developed a novel and efficient bus topology generation algorithm that optimizes communication contention under routability constraints. We proposed a new system-on-chip clock selection method. Our experiments demonstrate that it is important to consider a number of low-level details during system-on-chip synthesis [204].

We were the first to synthesize heterogeneous distributed systems containing dynamically reconfigurable hardware [205], although others had started work on this problem at our time of publication [69]. Our scheduler considers, and minimizes, inter-task reconfiguration delay. We demonstrate that considering reconfiguration delay allows the synthesis of superior embedded system architectures.

We were also the first to solve the limited bandwidth client-server system synthesis problem. We took care to pipeline the execution of tasks associated with different clients while maintaining identical client designs. During synthesis, tasks automatically migrate across wireless communication resources in order to improve system price, power consumption, and speed [206].

All of our work considers the power consumption of the synthesized real-time embedded system. Our work was among the first to consider power during heterogeneous distributed system synthesis [207]. To our knowledge only Dave and Jha [128] as well as Kirovski and Potkonjak [208] preceded us.

In addition to power consumption, the following costs may be considered, depending on the problem targeted: price, soft deadline violation, and area. Instead of collapsing these costs into a scalar with a weighting sum, these costs are simultaneously optimized by allowing multiple solutions that trade off the different costs to evolve in

parallel. Ours was the first system synthesis work to conduct multiobjective optimization in this manner. In addition, to the best of our knowledge, we were the first to devise an arbitrary-dimension dynamic locality preserving crossover selection method. This method is used within our evolutionary algorithm based system synthesis framework.

We were the first to build and describe an infrastructure that analyzes the contribution of an application and real-time operating system to an embedded system's power consumption [195]. This simulator provides detailed information about the power consumption impact of each portion of an embedded system's software. It allows one to find bounds on the energy consumptions of operating system service routines.

We initially set out with the goal of automating the design of a broad class of embedded systems. In the process of working toward this goal, we have explored, and solved, a number of specific problems within this research area. We attempted to select problems that are likely to become more important in the next few years and tested our ideas by complete implementation in software, comparison against past work, and execution on high-quality benchmarks. It is our hope that others can benefit from our work. If you found this work interesting, and would like to discuss it, please contact me.

Robert P. Dick

dickrp@ee.princeton.edu

Appendix A

Task Graphs for Free

In this appendix, we present a user-controllable, general-purpose, pseudorandom task graph generator called Task Graphs For Free (TGFF). TGFF creates problem instances for use in allocation and scheduling research. It has the ability to generate independent tasks as well as task sets that are composed of partially ordered task-graphs. A complete description of a scheduling problem instance is created, including attributes for processors, communication resources, tasks, and inter-task communication. The user may parametrically control the correlations between attributes. Sharing TGFF 's parameter settings allows researchers to reproduce the examples used by others, regardless of the platform on which TGFF is run. This work was done in collaboration with David L. Rhodes and Wayne Wolf.

A.1 Introduction

Research in embedded real-time systems and operating systems, as well as in more general allocation and scheduling fields, is hampered by the lack of a common base of examples. In general, an example used in allocation and scheduling research consists of a task set and a database of processors and communication resources. A *task set* is a

collection of *task graphs*, each of which is a directed acyclic graph (*DAG*) of communicating tasks. Generation of sample task sets is often a requirement when comparing allocation or scheduling methods with each other [209], [210]. There are generally no standard task sets available, making comparison of different methods all but impossible. Moreover, since task set generation is only a secondary aspect of scheduling research, the details necessary to enable exact recreation of another researcher's task sets are usually lacking. At best, re-implementation of another researcher's random task set generation algorithm is tedious. At worst, the new implementation subtly differs from the algorithm used in the work with which a comparison is made, resulting in misleading experimental results. These problems conspire to make it difficult to compare one's new allocator or scheduler with existing algorithms.

This situation would be improved by the existence of a standard, shareable base of task sets that are sufficiently general to enable applicability to a wide range of areas (e.g., embedded systems and parallel computing) and that can be tuned to particular problem domains. Shareable examples have been critical to progress in other areas such as computer-aided design and computer science, e.g., the standard ISCAS digital circuits used to compare digital circuit simulators [211] or the DIMACS Boolean formula sets used for satisfiability solvers [212]. However, a survey in the area of task sets reveals that researchers are 'on their own'; this is true among both the industrial and academic research communities. Allocation and scheduling research is a sufficiently broad area that any static set of examples meeting the needs of the majority of researchers would be gigantic. TGFF gives researchers the flexibility to dynamically tailor examples to their work while making it easy for others to regenerate these examples, given knowledge of the parameters used. It has been used by numerous scheduling and allocation researchers in published work.

Some allocation and scheduling research for very high-level system design assumes that there are no data dependencies between different tasks in a task set, while at the other extreme, directed, *cyclic* task-graphs usually arise in low-level or small-grain arenas, for example, in instruction-level code analysis. TGFF's task graph format, the DAG, is commonly used in medium-level and high-level allocation and scheduling research in academia and industry [67], [76], [81]. TGFF is nonetheless capable of generating sets of independent tasks as a special case of the sets of DAGs for which it is primarily intended.

TGFF includes a pseudorandom number generator [213]. This generator behaves identically on any machine that represents mantissas with 24 or more bits. Given the same command line options, TGFF will generate the same task set, processors, and communication resources when run on nearly any architecture that supports floating point computation.

A.2 Task set generation

Task graphs may be roughly categorized by their structural properties. DAGs generated to solve some numeric or algorithmic method, for example an FFT computation or a Quicksort, exhibit a particularized (and predictable) structure. Although there also appears to be a lack of shareable task graphs in this 'structured-graph' regime, these types of graphs are more easily documented and re-created than more randomly structured graphs. Thus, the TGFF effort focuses on random task graph generation subject to the limitations and parameters provided by the user.

TGFF generates a given number of random task graphs, where the graph *nodes* are tasks and the graph *arcs* represent communication between tasks. Arcs are associated

with parametrically controlled data volume scalars; they represent inter-process communication and impose a partial order on nodes. TGFF accepts a random number generator seed parameter, among others. The value of the seed affects both the structure as well as other aspects of the task set. Task set *families* containing an arbitrary number of task sets may be generated by varying the seed while holding all other parameters constant.

Terse documentation of each commend-line parameter is provided with the software. Therefore, only a high-level description is given here. One of the most challenging aspects of generating task graphs is developing an algorithm for defining their structure. For TGFF, there are a number of parameters relevant to the task graph structure: the average, n, and multiplier, m, for the lower bound on the number of nodes in a graph, and the maximum in-degree, id, and out-degree, od, of graph nodes. While id and od are fixed for every task graph generated in the task set, a value for the lower bound is selected at random from the uniform range $[n - m,\ n + m]$.

Let x be a lower bound on the number of nodes in a task graph, as randomly selected from the uniform range $[n - m,\ n + m]$. The task graph is constructed by first creating a single-node graph and then iteratively augmenting it until the number of nodes in the graph is greater than or equal to x.

The augmentation operates as follows. First randomly select either a *fan-out* step or a *fan-in* step (with equal probability). If it is a fan-out step, find the set of nodes that have the largest amount of 'available' out-degree, i.e., those with the maximum difference between od and the actual number of out-arcs, and call this maximum difference r. Assuming that $r > 0$, randomly pick a node, p, from the set, and then add y nodes and arcs to the graph from p to each of these new y nodes where y is a random number ranging from 0 to r.

If it is a fan-in step, find a set of existing nodes that are not over their od limit and call the cardinality of this set q. Assuming that $q > 0$, randomly select a value z in the

range $[0,\ \max(q, id)]$. Add a single node to the graph and z arcs from z nodes from the set to this new node.

This procedure generates DAGs that honor the in-degree and out-degree limits, contain at least x nodes, have a single start node, and do not have duplicated arcs (e.g., those between the same pair of nodes). The actual number of nodes in the generated task graph ranges from x to $x + od - 1$.

TGFF associates a deadline with every *terminal node* (a node that has no outgoing arcs) in the task graphs it produces. A heuristic is used to generate deadlines that are likely to be challenging but tractable. If depth p is the length of the maximum-length path from a task graph's start node to a given node, e is the user-specified average amount of time taken to execute a task, and laxity l is an arbitrary scalar, then the deadline d for that node is set in the following manner:

$$d = p \cdot e \cdot l$$

Task sets containing task graphs with differing periods are termed multi-rate task sets. TGFF is capable of parametrically generating the periods of task graphs in multi-rate task sets. The user specifies an array of period multipliers that is used to determine the relative periods of different task graphs in the task set. Selecting only small integer multipliers allows one to generate a task set that can feasibly be scheduled with the least common multiple scheduling method [105]. However, a user is free to specify multipliers that are vastly different or for which the least common multiple is large, relative to the individual multipliers. Given mul_ar (an array of user-provided period multipliers), p_laxity (a user-provided scalar), and tg_ar (an array containing all the task graphs in the task set), TGFF uses the algorithm in Figure A.1 to assign a period to each task graph. This algorithm generates periods that are based on the period multiplier array provided by the user and are loosely related to the deadlines of individual task graphs.

```
mul_ar is a user-specified array of multipliers
tg_ar is an array of task graphs
mul_ls is an empty list
p_laxity is a user-specified scalar

while mul_ls→elements < tg_ar→elements:
        select mul randomly from mul_ar
        append mul to mul_ls

sort mul_ls in increasing order
sort tg_ar in order of increasing deadlines

gr = tg_ar[last]→deadline / mul_ls[last]

for each i in all task graph indexes:
        tg_ar[i]→period = gr · mul_ls[i] · p_laxity
```

Figure A.1: Period computation algorithm

An important characteristic of task sets is the relation between the deadlines and the periods of their task graphs. While some schedulers allow periods that are less than deadlines (e.g., [67], [214]), many do not. If requested, TGFF prevents the period of any task graph from being greater than any of the deadlines within it.

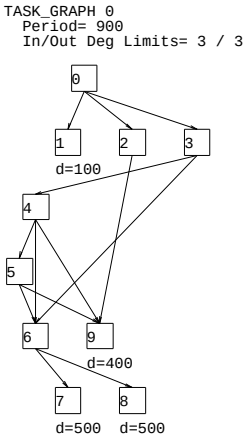

Figure A.2: Result for *tgff -n1 -e3:3 -g10:2 -r5*

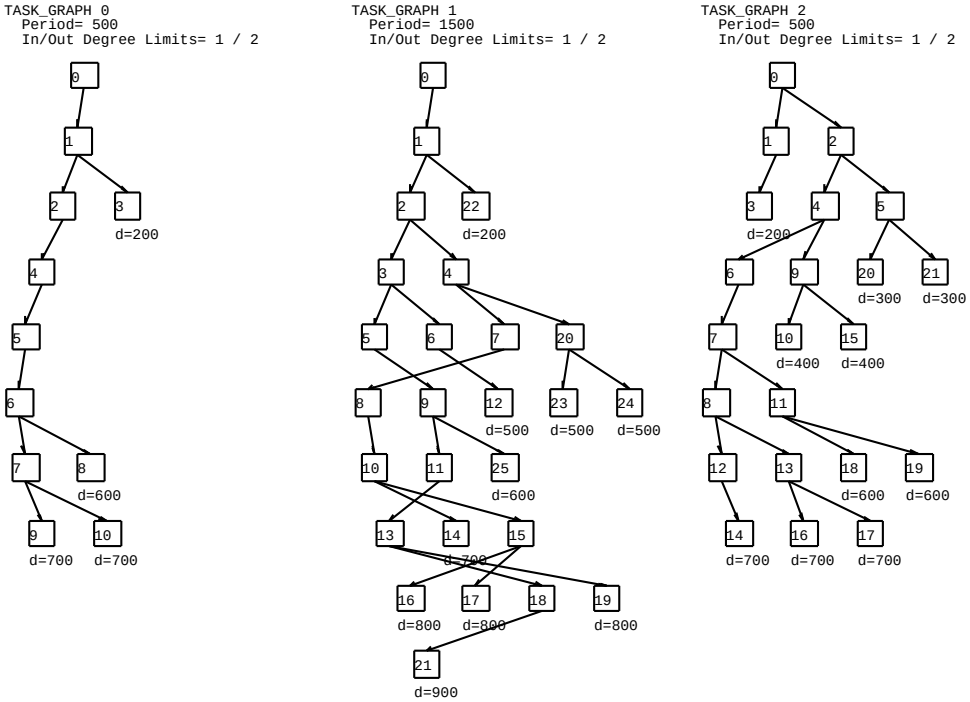

Figure A.3: Result for *tgff -e1:2 -g15:14*

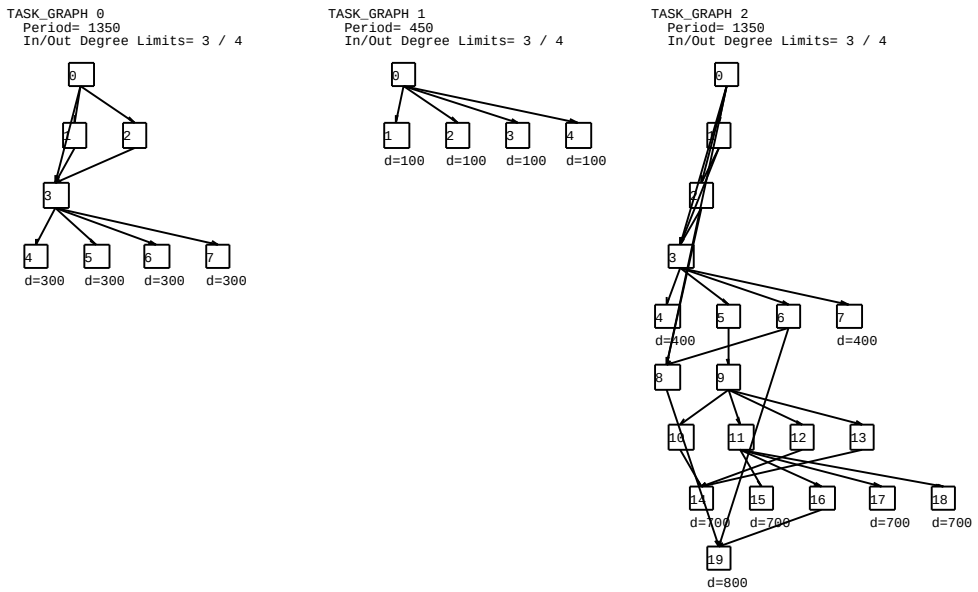

Figure A.4: Result for *tgff -e3:4 -g20:18 -r3*

In addition to the primary output file, a PostScript file depicting the task set is generated. Figure A.2 shows an example task graph output by TGFF's PostScript facility. This is a problem instance with a single task graph (*-n1*), a maximum in-degree and out-degree of two (*-e3:3*), a number of nodes ranging from eight to twelve per task graph (*-g10:2*), and a random seed of five (*-r5*). In this illustration, each task is represented by a square and is labeled with its number. In addition to its task number, each terminal node is labeled with its deadline. A task graph family of 50 single task graphs can be generated by running TGFF with the following flags, '*-n 1 -s*x,' where x is given integer values in the set $\{0, 1, 2, \cdots, 49\}$. This statement is sufficient documentation to enable other researchers to reproduce the same family. Figure A.3 shows the task set produced when TGFF is run with its in-degree restricted to one and its out-degree restricted to two (*-e1:2*), forcing TGFF to generate out-trees rather than more general

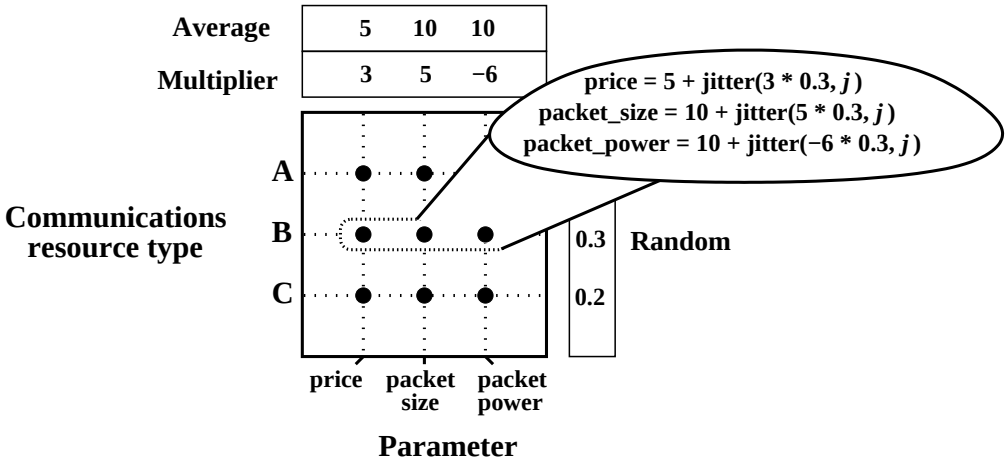

Figure A.5: Setting communication resource attributes

DAGs. As another example, Figure A.4 shows the generation of three task graphs with widely varying numbers of tasks.

A.3 Database generation

Some work in allocation and scheduling optimizes multiple attributes, e.g., execution time, power consumption, testability, and cost. TGFF supports this by allowing an arbitrary number of attributes that may be correlated or uncorrelated, to be associated with each processor and communication resource.

Although attribute generation for processors and communication resources is similar, communication resource attribute generation is more straightforward. This process is most easily illustrated with an example. Figure A.5 depicts attribute generation for communication resources. TGFF generates a random scalar ($com{\rightarrow}rand$), ranging from -1 to 1, for each communication resource. The user specifies an average, a, and a multiplier (m) value for each communication resource attribute, as well as a jitter, j, for the task set. Given a scalar, x, and the task set jitter, j, the function $\mathrm{jitter}\,(x, j)$ returns a

randomly selected number, c_r, from the uniform range $[x \cdot (1-j)\,,\ x \cdot (1+j)]$. With this function, and the parameters specified by the user, TGFF generates the attributes, q, for each communication resource, i.e.,

$$q = a + \text{jitter}\,(m \cdot c_r,\ jit)$$

A processor has attributes that are independent of tasks, as well as attributes that indicate the behavior of each task on that processor. Independent attribute generation is analogous to communication resource attribute generation. Task-processor intersection attributes that provide information about a task's execution on a particular processor, are generated with procedure similar to the one illustrated in Figure A.5. However, for task-processor intersections, the procedure operates in three dimensions instead of two. In addition to an array of random numbers associated with processors, there is a similar array associated with tasks. Each attribute depends on the processor and task for which the attribute is being generated.

TGFF has a number of default attributes: *cost* for processors, *cost* and *transmitrate* for communication resources, and *exec_time* for tasks. These attributes can be augmented or altered. As an example demonstrating TGFF's generality, consider the following scenario: one wants to add an attribute that defines a *setup* time for communication resources. This attribute is, in general, to be inversely related to cost. By giving TGFF the following command-line flag, *-C '10:5:t:cost 100:-80: f:setup'*, one declares that *cost* has an average value of 10, a multiplier of 5, and is an integer. Similarly, *setup* has a average value of 100, a multiplier of -80, and is a real number. Setting *cost*'s multiplier to a positive value and *setup*'s multiplier to a negative value causes these variables, in general, to be inversely related to each other. A portion of the resulting output appears in Figure A.6.

```
@COMMUN 0 {
#   cost      setup
      12   68.5145
}

@COMMUN 1 {
#   cost     setup
       9   119.64
}

@COMMUN 2 {
#   cost      setup
      10   92.5214
}
```

Figure A.6: Communication resource attributes

A.4 Conclusions

TGFF provides a standard method for generating random allocation and scheduling problem instances involving periodic or non-periodic task sets. Users have parametric control over an arbitrary number of attributes for tasks, processors, and communication resources. TGFF is capable of generating problem instances that are tuned to particular domains in allocation and scheduling research. However, the ease with which its parameters can be changed allows it to be applied to many allocation and

scheduling domains. Although TGFF simplifies the rapid production of large families of examples, this work's primary goal is to encourage comparison of allocation and scheduling algorithms by making it practical to reproduce the examples used by other researchers. The source code for TGFF is available via the "projects" link on the http://www.ee.princeton.edu/˜cad web page.

Appendix B

Implementation

This appendix is included as a reference to researchers who are interested in using our implementations as a starting point for their own research. We implemented the hardware-software co-synthesis and embedded system synthesis algorithms described in this dissertation in the C++ programming language, with heavy use of the standard template library (STL). We wrote a foundation library that is used extensively in our system synthesis software. It is 10,000 lines long and contains code for the following data structures and algorithms:

- 2-D associative matrices
- arbitrary-dimension geometric hypercubes
- arbitrary-dimension ragged and hypercube dynamically resizable arrays
- binary trees
- dereferencing containers and iterators
- fast static-dimension arrays
- floating-point epsilon comparison

- function objects
- generic object management interfaces for cloning, printing, and debugging
- high-quality lagged Fibonacci random number generator
- highly efficient and type-safe graphs
- highly efficient bidirectional maps
- highly efficient interval sets
- memory tracking
- minimal spanning trees
- numerous deterministic and probabilistic mathematical operations
- numerous numerical search algorithms
- numerous other data structures and debugging support routines.
- parsing operations
- smart pointers

This foundation library is portable to machines with up-to-date C^{++} compilers.

The unified (MOCSYN, CORDS, and COWLS) system synthesis algorithms consist of approximately 25,000 lines of code, in addition to the 18,000 lines used for the separate MOGAC. These include a floorplanner, interconnect performance estimation model, bus topology generator, etc. Initially, the synthesis algorithms were implemented separately. However, in the current implementation, code is shared between the different algorithms, when practical. As a result, it is possible to change multiple algorithms by

changing the code in one place. The data structures and algorithms within this implementation provide clean interfaces. They have been used as a starting point by a number of researchers. This system synthesis software is portable with one exception: we use a Linux-specific method of determining the memory usage of the current process in order to control the size of the architecture cache described in Section 6.7.

Bibliography

[1] Z. Luo, M. Martonosi, and P. Ashar, "Edge-endpoint-based configurable hardware architecture for VLSI layout design rule checking," *VLSI Design*, vol. 10, no. 3, pp. 249–263, 2000.

[2] J. Cong and Z. Pan, "Interconnect performance estimation models for design planning," *IEEE Trans. on Computer-Aided Design*, pp. 739–752, June 2001.

[3] P. Thoma, "Automotive electronics - A challenge for systems engineering," in *Proc. of Design, Automation and Test in Europe Conf.*, p. 4, Mar. 1999.

[4] R. Hersch, "Embedded processor and microcontroller primer and frequently asked questions." Posted monthly to the comp.arch.embedded USENET group. These figures initially came from World Semiconductor Trade Statistics.

[5] J. Turley, "Embedded processors by the numbers," *Embedded Systems Programming*, vol. 12, May 1999.

[6] J. Turley, "ARM the big winner for 1998; Motorola's 68K still on top," *Microdesign Resources*, vol. 31, Jan. 1999. Cahners Electronics Group.

[7] J. Child, "Survey finds embedded efforts lagging, lacking," *EE Times*, Apr. 2001.

[8] S. Napper, "Embedded system design plays catch-up," *Computer*, pp. 118–120, Aug. 1998.

[9] N. Sherwani, *Algorithms for VLSI Physical Design Automation: Second Edition*. Kluwer Academic Publishers, Boston, 1995.

[10] A. Raghunathan, N. K. Jha, and S. Dey, *High-level Power Analysis and Optimization*. Kluwer Academic Publishers, Boston, 1997.

[11] W. H. Wolf, *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers, CA, 2001.

[12] J. K. Adams and D. E. Thomas, "The design of mixed hardware/software systems," in *Proc. of Design Automation Conf.*, pp. 515–520, June 1996.

[13] G. De Micheli and R. K. Gupta, "Hardware/software co-design," *Proc. of IEEE*, vol. 85, pp. 349–365, Mar. 1997.

[14] R. Ernst, "Codesign of embedded systems: Status and trends," *IEEE Design and Test of Computers*, vol. 12, pp. 45–54, Apr. 1998.

[15] L. Garber and D. Sims, "In pursuit of hardware-software codesign," *Computer*, vol. 31, pp. 12–14, June 1998.

[16] G. Goossens, J. V. Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P. G. Paulin, "Embedded software in real-time signal processing systems: Design technologies," *Proc. of IEEE*, vol. 85, pp. 436–454, Mar. 1997.

[17] K. G. Shin and P. Ramanathan, "Real-time computing: A new discipline of computer science and engineering," *Proc. of IEEE*, vol. 82, pp. 6–23, Jan. 1994.

[18] W. H. Wolf, "Hardware-software co-design of embedded systems," *Proc. of IEEE*, vol. 82, pp. 967–989, July 1994.

[19] B. Dasarathy, "Timing constraints of real-time embedded systems: Constructs for expressing them, methods of validating them," *IEEE Trans. on Software Engineering*, vol. 11, pp. 80–86, Jan. 1985.

[20] J. Gong, D. D. Gajski, and S. Bakshi, "Model refinement for hardware-software codesign," *ACM Trans. on Design Automation of Electronic Systems*, vol. 2, pp. 22–41, Jan. 1997.

[21] J. P. Calvez and O. Pasquier, "Performance analysis of embedded HW/SW systems," in *Proc. of Int. Conf. on Computer Design*, pp. 5–22, Jan. 1998.

[22] T. Benner, R. Ernst, I. Konenkamp, P. Schuler, and H.-C. Schaub, "A prototype system for verification and emulation in hardware-software cosynthesis," in *Proc. of Int. Workshop on Rapid System Prototyping*, pp. 54–59, June 1995.

[23] S. L. Coumeri and D. E. Thomas, "A simulation environment for hardware-software codesign," in *Proc. of Int. Conf. on Computer Design*, pp. 58–63, Oct. 1995.

[24] K. Hines and G. Borriello, "Optimizing communication in embedded system co-simulation," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 121–125, Mar. 1997.

[25] C. Kuttner, "Hardware-software codesign using processor synthesis," *IEEE Design and Test of Computers*, vol. 13, no. 3, pp. 43–53, 1996.

[26] J. A. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design," in *Proc. of Design Automation Conf.*, pp. 178–183, June 1997.

[27] D. E. Thomas, J. K. Adams, and H. Schmit, “A model and methodology for hardware-software codesign,” *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 6–15, 1993.

[28] C. Castelluccia, W. Dabbous, and S. O’Malley, “Generating efficient protocol code from an abstract specification,” *IEEE Trans. on Networking*, vol. 5, pp. 514–524, Aug. 1997.

[29] L. Freund, D. Dupont, M. Israël, and F. Rousseau, “Interface optimization during hardware-software partitioning,” in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 75–79, Mar. 1997.

[30] A. Jirachiefpattana and R. Lai, “A rapid prototyping development system,” in *Proc. of Int. Workshop on Rapid System Prototyping*, pp. 118–124, June 1995.

[31] J. Smith and G. De Micheli, “Automated composition of hardware components,” in *Proc. of Design Automation Conf.*, pp. 14–19, June 1998.

[32] M. Adé, R. Lauwereins, and J. A. Peperstraete, “Buffer memory requirements in DSP applications,” in *Proc. of Int. Workshop on Rapid System Prototyping*, pp. 108–123, June 1994.

[33] C. N. Coelho Jr., C.-Y. J. Yang, and V. Mooney, “Redesigning hardware-software systems,” in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 116–123, Sept. 1994.

[34] G. Gogniat, M. Auguin, and C. Belleudy, “A generic multi-unit architecture for codesign methodologies,” in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 23–27, Mar. 1997.

[35] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," in *Proc. of Design Automation Conf.*, pp. 299–302, June 1997.

[36] P. Asar, "Towards a multi-formalism framework for architectural synthesis: The ASAR project," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 25–32, Sept. 1994.

[37] I. Bolsens, H. J. De Man, B. Lin, K. V. Rompaey, S. Vercauteren, and D. Verkest, "Hardware/software co-design of digital telecommunication systems," *Proc. of IEEE*, vol. 85, pp. 391–418, Mar. 1997.

[38] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. of Computer Simulation*, vol. 4, pp. 155–182, Apr. 1994.

[39] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "A formal methodology for hardware/software co-design of embedded systems," *Micro*, vol. 14, pp. 26–36, Aug. 1994.

[40] P. H. Chou and G. Borriello, "Modal processes: Toward enhanced retargetability through control composition of distributed embedded systems," in *Proc. of Design Automation Conf.*, pp. 88–93, June 1998.

[41] X. Hu, J. G. D'Ambrosio, B. T. Murray, and D.-L. Tang, "Codesign of architectures for automotive powertrain modules," *Micro*, pp. 17–24, Aug. 1994.

[42] T. B. Ismail, M. Abid, K. O'Brien, and A. Jerraya, "An approach for hardware-software codesign," in *Proc. of Int. Workshop on Rapid System Prototyping*, pp. 73–80, June 1994.

[43] A. Kalavade and E. A. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design and Test of Computers*, vol. 3, pp. 16–28, Sept. 1993.

[44] C. Passerone, L. Lavagno, M. Chiodo, and A. Sangiovanni-Vincentelli, "Fast hardware/software co-simulation for virtual prototyping and trade-off analysis," in *Proc. of Design Automation Conf.*, pp. 389–394, June 1997.

[45] J. D'Ambrosio and X. Hu, "Configuration-level hardware/software partitioning for real-time systems," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 34–41, Aug. 1994.

[46] K. S. Chatha and R. Vemuri, "An iterative algorithm for hardware-software partitioning, hardware design space exploration and scheduling," *Design Automation for Embedded Systems*, vol. 5, pp. 281–293, Aug. 2000.

[47] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search," *Design Automation for Embedded Systems*, vol. 2, pp. 5–32, Jan. 1997.

[48] R. Ernst, J. Henkel, and T. Benner, "Hardware/software cosynthesis for microcontrollers," *IEEE Design and Test of Computers*, vol. 12, pp. 64–75, Dec. 1993.

[49] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, "System-level exploration with SpecSyn," in *Proc. of Design Automation Conf.*, pp. 812–817, June 1998.

[50] R. K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design and Test of Computers*, vol. 10, pp. 29–41, Sept. 1993.

[51] J. Henkel and R. Ernst, "A hardware/software partitioner using a dynamically determined granularity," in *Proc. of Design Automation Conf.*, pp. 691–696, June 1997.

[52] A. Kalavade and E. A. Lee, "The extended partitioning problem: Hardware/software mapping and implementation-bin selection," in *Proc. of Int. Workshop on Rapid System Prototyping*, pp. 12–18, June 1995.

[53] Z. Karakehayov, "A fine-grained approach to distributed embedded system design," in *Proc. of Int. Conf. on Parallel and Distributed Computing and Systems*, pp. 376–380, Oct. 1995.

[54] P. V. Knudsen and J. Madsen, "PACE: A dynamic programming algorithm for hardware/software partitioning," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 82–95, Mar. 1996.

[55] B. Koroušić-Seljak and J. E. Cooling, "Optimization of multiprocessor real-time embedded system structures," in *Proc. of Mediterranean Electrotechnical Conf.*, pp. 313–316, Apr. 1994.

[56] C.-H. Lee and K. G. Shin, "Optimal task assignment in homogeneous networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, pp. 119–129, Feb. 1997.

[57] H. Liu and D. F. Wong, "Integrated partitioning and scheduling for hardware/software co-design," in *Proc. of Int. Conf. on Computer Design*, pp. 609–614, Oct. 1998.

[58] M. Potkonjak and J. Rabaey, "Algorithm selection: A quantitative computation-intensive optimization approach," in *Proc. of Int. Conf. on Computer-Aided Design*, pp. 90–95, Oct. 1994.

[59] D. Saha, R. S. Mitra, and A. Basu, "Hardware software partitioning using genetic algorithm," in *Proc. of Int. Conf. on VLSI Design*, pp. 155–159, Oct. 1998.

[60] D. Towlsey, “Allocating programs containing branches and loops within a multiple processor system,” *IEEE Trans. on Software Engineering*, vol. 12, pp. 1018–1024, Oct. 1986.

[61] F. Kaudel, “Comments on ‘Allocating programs containing branches and loops within a multiple processor system’,” *IEEE Trans. on Software Engineering*, vol. 16, p. 471, Apr. 1990.

[62] D. Towlsey, “Corrections to ‘Allocating programs containing branches and loops within a multiple processor system’,” *IEEE Trans. on Software Engineering*, vol. 16, p. 472, Apr. 1990.

[63] F. Vahid, T.-D. Le, and Y.-C. Hsu, “A comparison of functional and structural partitioning,” in *Proc. of Int. Symp. on System Synthesis*, pp. 121–126, Nov. 1996.

[64] F. Vahid and T.-D. Le, “Towards a model for hardware and software functional partitioning,” in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 116–123, Mar. 1996.

[65] J. Axelsson, “Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search strategies,” in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 161–165, Mar. 1997.

[66] A. Bender, “Design of an optimal loosely coupled heterogeneous multiprocessor system,” in *Proc. of European Design and Test Conf.*, pp. 275–281, Mar. 1996.

[67] B. P. Dave, G. Lakshminarayana, and N. K. Jha, “COSYN: Hardware-software co-synthesis of heterogeneous distributed embedded systems,” *IEEE Trans. on VLSI Systems*, vol. 7, pp. 92–104, Mar. 1999.

[68] B. P. Dave and N. K. Jha, "COHRA: Hardware-software cosynthesis of hierarchical heterogeneous distributed embedded systems," *IEEE Trans. on Computer-Aided Design*, vol. 17, pp. 900–919, Oct. 1998.

[69] B. Dave, "CRUSADE: Hardware/software co-synthesis of dynamically reconfigurable heterogeneous real-time distributed embedded systems," in *Proc. of Design, Automation and Test in Europe Conf.*, pp. 97–104, Mar. 1999.

[70] P.-A. Hsiung, "CMAPS: A cosynthesis methodology for application-oriented parallel systems," *ACM Trans. on Design Automation of Electronic Systems*, vol. 5, pp. 51–81, Jan. 2000.

[71] B. Jeong, S. Yoo, S. Lee, and K. Choi, "Hardware-software cosynthesis for run-time incrementally reconfigurable FPGAs," in *Proc. of Asia and South Pacific Design Automation Conf.*, pp. 169–174, Jan. 2000.

[72] I. Karkowski and H. Corporaal, "Design space exploration algorithm for heterogeneous multi-processor embedded system design," in *Proc. of Design Automation Conf.*, pp. 82–87, June 1998.

[73] K. Kuchcinski, "Embedded system synthesis by timing constraints solving," in *Proc. of Int. Symp. on System Synthesis*, pp. 50–57, Sept. 1997.

[74] C. Lee, M. Potkonjak, and W. Wolf, "Synthesis of hard real-time application specific systems," *Design Automation for Embedded Systems*, vol. 4, no. 4, pp. 215–242, 1999.

[75] H. Oh and S. Ha, "Hardware-software cosynthesis technique based on heterogeneous multiprocessor scheduling," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 183–1878, May 1999.

[76] S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. of Parallel & Distributed Computing*, vol. 16, pp. 338–351, Dec. 1992.

[77] M. Schwiegershausen and P. Pirsch, "Formal approach for the optimization of heterogeneous multiprocessors for complex image processing schemes," in *Proc. of European Design Automation Conf.*, pp. 8–13, Sept. 1995.

[78] S. Srinivasan and N. K. Jha, "Hardware-software co-synthesis of fault-tolerant real-time distributed embedded systems," in *Proc. of European Design Automation Conf.*, pp. 334–339, Sept. 1995.

[79] J. Teich, T. Blickle, and L. Thiele, "An evolutionary approach to system-level synthesis," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 167–171, Mar. 1997.

[80] W. H. Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems," *IEEE Trans. on VLSI Systems*, vol. 5, pp. 218–229, June 1997.

[81] T.-Y. Yen and W. H. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. of Int. Conf. on Computer-Aided Design*, pp. 288–294, Nov. 1995.

[82] Y. Xie and W. Wolf, "Allocation and scheduling of conditional task graph in hardware/software co-synthesis," in *Proc. of Design, Automation and Test in Europe Conf.*, pp. 620–625, Mar. 2001.

[83] F. Kordon and W. E. Kaim, "H-COSTAM: A hierarchical communicating state-machine model for generic prototyping," in *Proc. of Int. Workshop on Rapid System Prototyping*, pp. 131–138, June 1995.

[84] J. Hou and W. Wolf, "Process partitioning for distributed embedded systems," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 70–76, Mar. 1996.

[85] P. V. Knudsen and J. Madsen, "Graph based communication analysis for hardware/software codesign," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 131–135, May 1999.

[86] A. Dasdan, D. Ramanathan, and R. K. Gupta, "Rate derivation and its applications to reactive, real-time embedded systems," in *Proc. of Design Automation Conf.*, pp. 263–268, June 1998.

[87] R. K. Gupta, "Framework for interactive analysis of timing constraints in embedded systems," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 44–51, Mar. 1996.

[88] X. Hu and R. S. Sambandam, "Multi-valued performance metrics for real-time embedded systems," *Design Automation for Embedded Systems*, vol. 5, pp. 5–28, Feb. 2000.

[89] B.-D. Rhee, S. L. Min, S.-S. Lim, H. Shin, C. S. Kim, and C. Y. Park, "Issues of advanced architectural features in the design of a timing tool," in *Proc. of Workshop on Real-Time Operating Systems and Software*, pp. 59–62, May 1994.

[90] T. Benner and R. Ernst, "An approach to mixed systems co-synthesis," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 9–14, Mar. 1997.

[91] P. H. Chou, R. B. Ortega, and G. Borriello, "The Chinook hardware/software co-synthesis system," in *Proc. of Int. Symp. on System Synthesis*, pp. 22–27, Sept. 1995.

[92] D. L. Rhodes and W. Wolf, "Co-synthesis of heterogeneous multiprocessor systems using arbitrated communication," in *Proc. of Int. Conf. on Computer-Aided Design*, pp. 339–342, Nov. 1999.

[93] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Trans. on Computer-Aided Design*, vol. 16, Dec. 1997.

[94] J. M. Rabaey and L. M. Guerra, "Exploring the architecture and algorithmic space for signal processing applications," in *Proc. of Int. Conf. on VLSI and CAD*, pp. 315–319, Nov. 1993.

[95] Y. Xie and W. Wolf, "Co-synthesis with custom ASICs," in *Proc. of Asia and South Pacific Design Automation Conf.*, pp. 129–133, Jan. 2000.

[96] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "A formal specification model for hardware/software codesign," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, Oct. 1993.

[97] B. Lee and E. A. Lee, "Hierarchical concurrent finite state machines in ptolemy," in *Proc. of International Conf. on Applications of Concurrency to System Design*, pp. 34–40, Mar. 1998.

[98] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[99] L. A. Cortéz, P. Eles, and Z. Peng, "A survey on hardware/software codesign representation models," tech. rep., Dept. Computer and Information Science, Linköping University, June 1999.

[100] L. L. E. A. Yakovlev, L. Gomes, *Hardware Design and Petri Nets*. Kluwer Academic Publishers, Boston, 2000.

[101] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proc. of IEEE*, pp. 366–390, Mar. 1997.

[102] "The hardware-software co-synthesis benchmarks mailing list." http://www.ee.-princeton.edu/˜cad/cosynth-benchmarks.

[103] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop, "Scheduling of conditional process graphs for the synthesis of embedded systems," in *Proc. of Design, Automation and Test in Europe Conf.*, pp. 132–139, Feb. 1998.

[104] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, and L. Thiele, "Representation of process mode correlation for scheduling," in *Proc. of Int. Conf. on Computer-Aided Design*, pp. 54–61, Nov. 1998.

[105] E. L. Lawler and C. U. Martel, "Scheduling periodically occurring tasks on multiple processors," *Information Processing Letters*, vol. 7, pp. 9–12, Feb. 1981.

[106] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proc. of Design Automation Conf.*, pp. 456–461, June 1995.

[107] Z. Chen and K. Roy, "A power macromodeling technique based on power sensitivity," in *Proc. of Design Automation Conf.*, pp. 678–683, June 1998.

[108] M. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and minimization techniques for embedded DSP software," *IEEE Trans. on VLSI Systems*, vol. 5, pp. 123–135, Mar. 1997.

[109] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee, "Instruction level power analysis and optimization of software," *J. VLSI Signal Processing*, vol. 13, no. 2–3, pp. 223–238, 1996.

[110] "A simple method of estimating power in XC4000XL/EX/E FPGAs," June 1997.

[111] R. Y. Chen, R. M. Owens, M. J. Irwin, and R. S. Bajwa, "Validation of an architectural level power analysis technique," in *Proc. of Design Automation Conf.*, pp. 242–245, June 1998.

[112] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, NY, 1979.

[113] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA, 1984.

[114] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. McGraw-Hill Book Company, NY, 1990.

[115] W. Green, *Introduction to Operations Engineering*. Righard D. Irwin, Inc., IL, 1971.

[116] E. L. Lawler and D. E. Wood, "Branch-and-bound methods: A survey," *Operations Research*, pp. 699–719, July 1966.

[117] C. H. Papadimitriou and K. Stiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[118] F. Glover and M. Laguna, *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

[119] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines*. John Wiley & Sons, Chichester, England, 1989.

[120] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[121] A. Neubauer, "The circular schema theorem for genetic algorithms and two-point crossover," in *Proc. of Genetic Algorithms in Engineering Systems: Innovations and Applications*, pp. 209–214, Sept. 1997.

[122] S. W. Mahfoud and D. E. Goldberg, "Parallel recombinative simulated annealing: A genetic algorithm," *Parallel Computing*, vol. 21, pp. 1–28, Jan. 1995.

[123] C. M. Fonseca and P. J. Fleming, "Multiobjective genetic algorithms made easy: Selection, sharing and mating restrictions," in *Proc. of Genetic Algorithms in Engineering Systems: Innovations and Applications*, pp. 45–52, Sept. 1995.

[124] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proc. of IEEE*, vol. 82, pp. 55–67, Jan. 1994.

[125] S. Kim and J. Browne, "A general approach to mapping of parallel computations upon multiprocessor architectures," in *Proc. of Int. Conf. on Parallel Processing*, vol. 2, pp. 1–8, Aug. 1988.

[126] A. E. Smith and D. M. Tate, "Genetic optimization using a penalty function," in *Proc. of Int. Conf. on Genetic Algorithms*, pp. 499–503, July 1993.

[127] T.-Y. Yen, *Hardware-Software Co-Synthesis of Distributed Embedded Systems*. PhD thesis, Dept. of Electrical Engg., Princeton University, June 1996.

[128] B. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of embedded systems," in *Proc. of Design Automation Conf.*, pp. 703–708, June 1997.

[129] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. of Int. Workshop on Hardware/Software Co-Design*, pp. 97–101, Mar. 1998.

[130] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz, "BOA: The Bayesian optimization algorithm," in *Proc. of the Genetic and Evolutionary Computation Conf.*, July 1999.

[131] "Embedded microprocessor benchmark consortium." http://www.eembc.org.

[132] R. Weiss, "32-bit cores drive systems-on-a-chip," *Computer Design*, pp. 82–89, Sept. 1996.

[133] "Design and reuse." http://www.design-reuse.com/.

[134] W. Wolf, "Floorplanning: The art of chip-level design," *Electronics J.*, pp. 8–13, Oct. 1998.

[135] M. Kishinevsky, J. Cortadella, and A. Kondratyev, "Asynchronous interface specification, analysis and synthesis," in *Proc. of Design Automation Conf.*, pp. 2–7, June 1998.

[136] L. F. G. Sarmenta, G. A. Pratt, and S. A. Ward, "Rational clocking," in *Proc. of Int. Conf. on Computer Design*, pp. 217–278, Oct. 95.

[137] S. Moore, G. Taylor, R. Mullins, and P. Robinson, "Point to point GALS interconnect," in *Proc. of Int. Symp. on Asynchronous Circuits and Systems*, pp. 769–775, Apr. 2002.

[138] T. Chelcea and S. M. Nowick, "A low-latency FIFO for mixed-clock systems," in *Proc. of IEEE Computer Society Annual Workshop on VLSI*, pp. 21–28, Apr. 2000.

[139] M. Bazes, R. Ashuri, and E. Knoll, "An interpolating clock synthesizer," *J. of Solid-State Circuits*, vol. 31, pp. 1295–1300, Sept. 1996.

[140] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. of Design Automation Conf.*, pp. 173–181, June 1982.

[141] L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Information and Control*, vol. 57, pp. 91–101, May/June 1983.

[142] M. Wang and M. Sarrafzadeh, "Modeling and minimization of routing congestion," in *Proc. of Asia and South Pacific Design Automation Conf.*, pp. 185–190, Jan. 2000.

[143] F. K. Hwang, "On steiner minimal trees with rectilinear distance," *SIAM J. on Applied Mathematics*, pp. 104–114, Jan. 1976.

[144] J. Cong, Z. Pan, L. He, C.-K. Koh, and K.-Y. Khoo, "Interconnect design for deep submicron ICs," in *Proc. of Int. Conf. on Computer-Aided Design*, pp. 478–485, Nov. 1997.

[145] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "Piperench: A coprocessor for streaming multimedia acceleration," in *Proc. of Int. Symp. on Computer Architecture*, pp. 28–39, June 1999.

[146] D. Halchin and M. Golio, "Trends for portable wireless applications," *Microwave J.*, vol. 40, pp. 62–78, Jan. 1997.

[147] S. Komaki and E. Ogawa, "Trends of fiber-optic microcellular radio communication networks," *IEICE Trans. Electronics*, vol. E79-C, pp. 98–103, Jan. 1996.

[148] G. Comparetto and R. Ramirez, "Trends in mobile satellite technology," *Computer*, vol. 30, pp. 44–52, Feb. 1997.

[149] F. Ananasso and F. D. Priscoli, "Issues on the evolution towards satellite personal communication networks," in *Proc. of Global Telecommunications Conf.*, pp. 541–545, Nov. 1995.

[150] R. E. Barry and J. P. Jones, "Rapid world modeling from a mobile platform," in *Proc. of Int. Conf. on Robotics and Automation*, pp. 72–78, Apr. 1997.

[151] D. W. Gage, "Telerobotic requirements for sensing, navigation, and communications," in *Proc. of National Telesystems Conf.*, pp. 145–148, May 1994.

[152] "Altera ARC-PCI reconfigurable computing platform." http://www.altera.com/-html/new/pressrel/pr_arc-pci.html.

[153] "Xilinx part information." http://www.xilinx.com/partinfo/.

[154] D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs," in *Proc. Symp. on FPGAs for Custom Computing Machines*, pp. 136–144, Apr. 1995.

[155] L. Shang and N. K. Jha, "Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable fpgas," in *Proc. of Int. Conf. on VLSI Design*, pp. 345–352, Jan. 2002.

[156] "Computer design." Product trends sections of vol. 35: n. 2, 6, 8, 9, vol. 36: n. 1, 9, and vol. 37: n. 1–3.

[157] L. Shang and N. K. Jha, "High-level power modeling of CPLDs and FPGAs," in *Proc. of Int. Conf. on Computer Design*, pp. 46–51, Sept. 2001.

[158] L. Shang, A. S. Kaviani, and K. Bathala, "Dynamic power consumption in virtex^{TM}-II FPGA," in *Proc. of Int. Symp on Field Programmable Gate Arrays*, pp. 157–164, Feb. 2002.

[159] J. J. Labrosse, *MicroC/OS-II*. R & D Books, Lawrence, KS, 1998.

[160] Fujitsu Microelectronics, Inc., "MB8683x user's guide."

[161] S. Heath, *Embedded Systems Design*. Butterworth-Heinemann, Boston, MA, 1997.

[162] J. J. Labrosse, *Embedded Systems Building Blocks*. R & D Books, Lawrence, KS, 1997.

[163] J. J. Labrosse, *MicroC/OS-II*. R & D Books, Lawrence, KS, 1998.

[164] P. A. Laplante, *Real-Time Systems Design and Analysis: An Engineers Handbook*. IEEE Press, Piscataway, NJ, 1993.

[165] R. Sharma, "Distributed application development with Inferno," in *Proc. Design Automation Conf.*, pp. 146–150, June 1999.

[166] D. Stepner, N. Rajan, and D. Hui, "Embedded application design using a real-time OS," in *Proc. Design Automation Conf.*, pp. 151–156, June 1999.

[167] W. Warner, "Non-pre-emptive multithreading performs embedded software's juggling act," *Electronic Design News*, vol. 44, pp. 117–126, July 1999.

[168] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Academic Publishers, Norwell, MA, 1997.

[169] A. R. Chandrakasan and R. W. Brodersen, *Low Power Digital CMOS Design*. Kluwer Academic Publishers, Norwell, MA, 1995.

[170] G. Yeap, *Practical Low Power Digital VLSI Design*. Kluwer Academic Publishers, Norwell, MA, 1998.

[171] J. Monteiro and S. Devadas, *Computer-Aided Design Techniques for Low Power Sequential Logic Circuits*. Kluwer Academic Publishers, Norwell, MA, 1996.

[172] J. Rabaey and M. P. (Editors), *Low Power Design Methodologies*. Kluwer Academic Publishers, Norwell, MA, 1996.

[173] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *IEEE Trans. VLSI Systems*, vol. 2, pp. 437–445, Dec. 1994.

[174] T. Sato, Y. Ootaguro, M. Nagamatsu, and H. Tago, "Evaluation of architecture-level power estimation for CMOS RISC processors," in *Proc. Symp. Low Power Electronics*, pp. 44–45, Oct. 1995.

[175] C. T. Hsieh, M. Pedram, G. Mehta, and F. Rastgar, "Profile-driven program synthesis for evaluation of system power dissipation," in *Proc. Design Automation Conf.*, pp. 576–581, June 1997.

[176] L. Benini and G. De Micheli, "System-level power optimization: Techniques and tools," in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 288–293, Aug. 1999.

[177] B. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of embedded systems," in *Proc. Design Automation Conf.*, pp. 703–708, June 1997.

[178] Y. Li and J. Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in *Proc. Design Automation Conf.*, pp. 188–193, June 1998.

[179] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, and A. Sangiovanni-Vincentelli, "Efficient power estimation techniques for HW/SW systems," in *Proc. VOLTA'99 Int. Wkshp. on Low Power Design*, Mar. 1999.

[180] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John, "Using complete machine simulation for software power estimation: The SoftWatt approach," in *Proc. Int. Symp. High Performance Computer Architecture*, pp. 141–150, Feb. 2002.

[181] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: An overview," in *Proc. Symp. Low Power Electronics*, pp. 38–39, Oct. 1994.

[182] T. Simunic, G. De Micheli, and L. Benini, "Energy-efficient design of battery-powered embedded systems," in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 212–217, Aug. 1999.

[183] J. L. da Silva, F. Catthoor, D. Verkest, and H. De Man, "Power exploration for dynamic data types through virtual memory management refinement," in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 311–316, Aug. 1998.

[184] Q. Qiu, Q. Wu, and M. Pedram, "Stochastic modeling of a power-managed system: Construction and optimization," in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 194–199, Aug. 1999.

[185] L. Benini, A. Bogliolo, S. Cavallucci, and B. Ricco, "Monitoring system activity for OS-directed dynamic power management," in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 185–190, Aug. 1998.

[186] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power optimization of variable voltage core-based systems," in *Proc. Design Automation Conf.*, pp. 176–181, June 1998.

[187] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 197–202, Aug. 1998.

[188] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. Int. Symp. Low Power Electronics & Design*, pp. 76–81, Aug. 1998.

[189] N. K. Jha, "Low power system scheduling and synthesis," in *Proc. Int. Conf. Computer-Aided Design*, pp. 259–263, Nov. 2001.

[190] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proc. ACM Symposium on Operating Systems Principles*, pp. 89–102, Dec. 2001.

[191] L. Benini, M. Kandemir, and J. Ramanujam, eds., *Proc. Wkshp. Compilers & Operating Systems for Low Power*. Kluwer Academic Publishers, to appear in 2002.

[192] Y.-H. Lu, L. Benini, and G. De Micheli, "Power-aware operating systems for interactive systems," *IEEE Trans. on VLSI Systems*, vol. 10, Apr. 2002.

[193] T. Simunic, L. Benini, P. W. Glynn, and G. D. Micheli, "Dynamic power management for portable systems," in *Proc. MOBICOM*, pp. 11–19, Aug. 2000.

[194] A. Vahdat, A. R. Lebeck, and C. S. Ellis, "Every joule is precious: The case for revisiting operating system design for energy efficiency," in *Proc. ACM SIGOPS European Workshop*, Sept. 2000.

[195] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Power analysis of embedded operating systems," in *Proc. Design Automation Conf.*, pp. 312–315, June 2000.

[196] T. K. Tan, A. Raghunathan, and N. K. Jha, "EMSIM: An energy simulation framework for an embedded operating system," in *Proc. Int. Symp. Circuits & Systems*, May 2002.

[197] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob, "The performance and energy consumption of three embedded real-time operating systems," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 203–210, Nov. 2001.

[198] T. Simunic, L. Benini, and G. De Micheli, "Cycle-accurate simulation of energy consumption in embedded systems," in *Proc. of Design Automation Conf.*, pp. 867–872, June 1999.

[199] CoWare *N2C Training Manual*, 1999.

[200] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," in *Proc. of Int. Conf. on Computer Design*, pp. 452–457, Oct. 1993.

[201] Fujitsu Microelectronics, Inc., "SPARClite series 32-bit RISC embedded processor MB86832 databook," 1998.

[202] IBM, "1995 DRAM databook," 1994.

[203] Fujitsu Microelectronics, Inc., "MB86934: 930 series 32-bit RISC embedded processor datasheet," 1996.

[204] R. P. Dick and N. K. Jha, "MOCSYN: Multiobjective core-based single-chip system synthesis," in *Proc. of Design, Automation and Test in Europe Conf.*, pp. 263–270, Mar. 1999.

[205] R. P. Dick and N. K. Jha, "CORDS: Hardware-software co-synthesis of reconfigurable real-time distributed embedded systems," in *Proc. of Int. Conf. on Computer-Aided Design*, pp. 62–68, Nov. 1998.

[206] R. P. Dick and N. K. Jha, "COWLS: Hardware-software co-synthesis of distributed wireless low-power embedded client-server systems," in *Proc. of Int. Conf. on VLSI Design*, pp. 114–120, Jan. 2000.

[207] R. P. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems," in *Proc. of Int. Conf. on Computer-Aided Design*, pp. 522–529, Nov. 1997.

[208] D. Kirovski and M. Potkonjak, "System-level synthesis of low-power hard real-time systems," in *Proc. of Design Automation Conf.*, pp. 697–702, June 1997.

[209] T. Yang and A. Gerasoulis, "DSC: Scheduling parallel tasks on an unbounded number of processors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 30, pp. 951–967, Sept. 1994.

[210] W. Zhao, K. Ramamritham, and J. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Trans. on Computers*, vol. 36, pp. 949–960, Aug. 1987.

[211] M. Sengupta, "ISCAS '89 benchmark information," Mar. 89. http://www.cbl.-ncsu.edu/CBL_Docs/iscas89.html.

[212] D. Du, J. Gu, and P. M. Pardalos, "Satisfiability problems: Theory and applications," in *DIMACS: Series in Descrete and Applied Mathematics and Computer Science*, vol. 35, American Mathematical Society, Providence, RI, 1997.

[213] G. Marsaglia and A. Zaman, "Toward a universal random number generator," *Statistics and Probability Letters*, vol. 9, pp. 35–39, Jan. 1990.

[214] R. P. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for hardware-software co-synthesis of distributed embedded systems," *IEEE Trans. on Computer-Aided Design*, vol. 17, pp. 920–935, Oct. 1998.