INSTRUCTION-SET CUSTOMIZATION FOR MULTI-TASKING EMBEDDED SYSTEMS

HUYNH PHUNG HUYNH

NATIONAL UNIVERSITY OF SINGAPORE

October 2009

INSTRUCTION-SET CUSTOMIZATION FOR MULTI-TASKING EMBEDDED SYSTEMS

HUYNH PHUNG HUYNH

(B.Eng., Ho Chi Minh University of Technology)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

SINGAPORE

October 2009

List of Publications

- *Instruction-Set Customization for Real-Time Embedded Systems*. Huynh Phung Huynh and Tulika Mitra. Design Automation and Test in Europe (**DATE**), April 2007.

- *An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization*. Huynh Phung Huynh, Edward Sim and Tulika Mitra. 7th ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (**CASES**), October 2007.

- *Processor Customization for Wearable Bio- monitoring Platforms*. Huynh Phung Huynh and Tulika Mitra. IEEE International Conference on Field Programmable Technology (**FPT**), December 2008.

- *An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization*. Huynh Phung Huynh, Edward Sim and Tulika Mitra. Springer Journal of Design Automation for Embedded Systems, 2009.

- *Runtime Reconfiguration of Custom Instructions for Real-Time Embedded Systems*. Huynh Phung Huynh and Tulika Mitra. Design Automation and Test in Europe (**DATE**), April 2009.

- *Evaluating Tradeoffs in Customizable Processors*. Unmesh Dutta Bordoloi, Huynh Phung Huynh, Samarjit Chakraborty and Tulika Mitra. Design Automation Conference (**DAC**), July 2009.

- *Runtime Adaptive Extensible Embedded Processors - A Survey*. Huynh Phung Huynh and Tulika Mitra. The 9th International Workshop on Systems, Architectures, Modeling, and Simulation (**SAMOS**), July 2009.

- *System Level Design Methodologies for Instruction-set Extensible Processors*. Huynh Phung Huynh. 12th Annual ACM SIGDA Ph.D. Forum at Design Automation Conference (**DAC**), July 2009.

Acknowledgements

I deeply appreciate my advisor professor Tulika Mitra for her guidance. Without her, it hardly for me to finish this thesis. She guided me not only with the knowledge of a passionate scientist but also with her kindness and patience. I am sincerely grateful to her. I wish all the best to her and her family. I would like to thank the members of my thesis committee, professor Wong Weng Fai, professor P.S. Thiagarajan and professor Samarjit Chakraborty for their valuable feedback and suggestions that helped me to determine the story line of this thesis. Moreover, I would like to thank professor Jürgen Teich as my external examiner and professor Abhik Roychoudhury as my oral panel member. The valuable feedback from the professors will help me very much along my future research career.

I would like to thank Edward Sim Joon, Unmesh Dutta Bordoloi and Liang Yun as my collaborators in the works of chapter 6, 4 and 5 respectively. I would like to thank my fellow colleagues in the embedded system research lab. They are Pan Yu, Vivy Suhendra, Ju Lei, Ramkumar Jayaseelan, Ge Zhiguo, Nguyen Dang Kathy, Phan Thi Xuan Linh, Raman Balaji, Ankit Goel, Sun Zhenxin, Ioana Cutcutache, Andrei Hagiescu, Deepak Gangadharan, Huynh Bach Khoa, Liu Shanshan, Achudhan Sivakumar, Dang Thi Thanh Nga, Wang Chundong, Qi Dawei, Liu Haibin. The research discussions and entertainment events with them made my Ph.D. candidate life more meaningful. Moreover, I would like to thank my Vietnamese friends, Dau Van Huan, Huynh Kim Tho, Huynh Le Ngoc Thanh, Tran Anh Dung, Do Hien, Nguyen Chi Hieu, Hoang Khac Chi, Nguyen Tan Trong, who gave me strong encouragements.

My parents and my grand parents always support me that gave me ultimate power to finish this thesis. I hope that they are very happy and proud of my achievements. My wife, Phan Hoang Yen, always stays by my side and strongly supports me during the tough periods of my Ph.D. candidate. There is no word to express my love, respect and gratitude to them.

Contents

List of Publications iii

Acknowledgements iv

Abstract x

List of Tables xii

List of Figures xiii

1 Introduction 1

1.1 Instruction-Set Extensible Processor 4

1.2 Instruction-Set Customization for Multi-tasking Embedded Systems 6

1.3 Contributions of The Thesis 9

1.4 Organization of The Thesis 11

2 Background and Related Works 13

2.1 Architecture of Instruction-Set Extensible Processor 13

2.2 Instruction-Set Customization Compilation Flow 17

2.3 Custom Instructions Generation for an Application 18

2.3.1 Custom Instructions Identification 19
2.3.2 Custom Instructions Selection 20
2.3.3 Integrated Custom Instructions Generation 22
2.4 Customization for MPSoC 22
2.5 Reconfigurable Computing 23

3 Customization for multi-tasking real-time embedded systems 26

3.1 Customization for Real-Time Systems 27
3.1.1 Problem Formulation 27
3.1.2 Motivating Example 29
3.1.3 Customization for EDF Scheduling 30
3.1.4 Customization for RMS 32
3.2 Experimental Evaluation 35
3.2.1 Performance 37
3.2.2 Energy 39
3.3 Summary 40

4 Evaluating design trade-offs for custom instructions 41

4.1 Problem Statement 44
4.1.1 Task Model 44
4.1.2 Intra-Task Custom Instructions Selection 45
4.1.3 Inter-Task Custom Instructions Selection 46
4.2 Evaluating Design Trade-offs 48
4.2.1 Intra-Task Trade-offs 48
4.2.1.1 The GAP Problem 50
4.2.2 Inter-Task Trade-offs 53

4.3 Experimental Evaluation 55
4.4 Summary 59

5 Iterative custom instruction generation **60**
5.1 Iterative Approach 63
5.2 Custom Instruction Generation 65
5.2.1 Definitions 66
5.2.2 Region Selection 67
5.2.3 MLGP Algorithm 68
5.3 Experimental Evaluation 74
5.3.1 Experimental Setup 74
5.3.2 System-Level Design 75
5.3.3 Efficiency of MLGP Algorithm 78
5.4 Summary 84

6 Runtime reconfiguration of custom instructions **85**
6.1 System Design Flow 90
6.2 Partitioning Problem 92
6.3 Partitioning Algorithm 95
6.3.1 Overview 97
6.3.2 Spatial Partitioning 100
6.3.3 Temporal Partitioning 101
6.4 Experimental Evaluation 105
6.4.1 Efficiency and Scalability of Algorithms 107
6.4.2 Case Study of JPEG Application 110
6.5 Summary 115

7 Runtime reconfiguration of custom instructions for multi-tasking embedded systems **116**
7.1 Problem Formulation 118
7.2 Algorithm 121
7.2.1 A Simple Solution 122
7.2.2 Deadline Constraints 124
7.2.3 Runtime Reconfiguration 125
7.2.4 Putting It All Together 128
7.3 Experimental Evaluation 130
7.3.1 ILP Formulation 130
7.3.1.1 Uniqueness Constraint 130
7.3.1.2 Resource Constraint 131
7.3.1.3 Scheduling Constraint 131
7.3.1.4 Objective Function 132
7.3.2 Experimental Setup 133
7.3.3 Experimental Results 135
7.4 Summary 137

8 A case study of processor customization **138**
8.1 Wearable Bio-monitoring Applications 141
8.1.1 Continuous Monitoring of Vital Signs 141
8.1.2 Fall Detection 143
8.2 Processor Customization 144
8.2.1 Conversion to Fixed Point Arithmetic 145
8.3 Experimental Results 146
8.4 Summary 148

9 Conclusions and Future Work **149**

Bibliography **151**

Abstract

Generating a set of custom instructions for an application is crucial to the efficiency of instruction-set extensible processor. Over the past decade, most research works focused on automated generation of custom instructions. The state-of-the-art techniques are fairly effective at generating a set of custom instructions with high performance potential for an application. However, while multi-tasking applications have become popular in embedded systems, instruction-set customization for multi-tasking embedded systems has largely remained unexplored.

Envisioning the crucial need of design methodologies for instruction-set customization for multi-tasking embedded systems, we first explore custom instructions generation in the context of multiple real-time tasks executing under a real-time scheduling policy. As custom instructions may reduce the processor utilization for a task set through performance speedup of the individual tasks, customization may enable a previously unschedulable task set to satisfy all the timing requirements.

We extend our study in instruction-set customization for real-time embedded systems to consider the conflicting tradeoffs among multiple objectives (e.g., performance versus area). As we expose multiple solutions with different tradeoffs, designers have more flexibility to select an appropriate implementation for the system requirements. In particular, we propose an efficient polynomial time algorithm to compute an approximate Pareto front in the design space.

Our design flow so far takes a bottom-up approach where a large amount of time is spent in identifying all possible custom instructions for all constituent tasks while only a small subset of these custom instructions are finally selected. Based on this observation, we investigate an iterative custom instruction generation scheme that takes a top-down approach and directly zooms into the task creating the performance bottleneck. This way,

we avoid the expensive custom instruction generation process for all the tasks.

The second part of the thesis focuses on further improving the application speedup of customization through runtime reconfiguration. The total area available for the implementation of the custom instructions in an embedded processor is limited. Therefore, we may not be able to exploit the full potential of all the custom instructions in an application. In this context, runtime reconfiguration of custom instructions appears quite promising. To support designers in instruction-set customization with runtime reconfiguration capability, we first develop an efficient framework that starts with a sequential application specified in ANSI-C and can automatically select appropriate custom instructions as well as club them into one or more configurations.

Finally, we extend runtime reconfiguration of custom instructions to multi-tasking applications with real-time constraints. We propose a pseudo-polynomial time algorithm that performs near-optimal spatial and temporal partitioning of custom instructions to minimize processor utilization while satisfying all the real-time constraints.

List of Tables

3.1 Composition of Task sets . . . 36

4.1 Composition of the task sets. . . . 56

4.2 Speedup obtained from our approximation scheme for the task sets 1 – 5. . . 57

5.1 Benchmark Characteristics. The maximum and average size of basic block (BB) are given in term of primitive instructions. . . . 76

5.2 Task Sets. . . . 77

6.1 Running time of the algorithms for synthetic input. . . . 108

6.2 CIS versions for JPEG application. . . . 112

7.1 CIS Versions of the tasks. . . . 134

7.2 Running Time of Optimal and DP in seconds. . . . 137

List of Figures

1.1 Instruction-Set Extensible Processor 4

1.2 Instruction-Set Extensible Processor Design Flow 5

1.3 Design flow of instruction-set customization for multi-tasking systems . . . 7

1.4 Motivating example for dynamic reconfiguration of CFU (AU: arithmetic/logic unit, MU: multiplier unit). 9

1.5 Roadmap of thesis 12

2.1 Instruction-Set Extensible Processor 14

2.2 Four types of instruction-set extensible processors. 15

3.1 Application performance versus hardware area for different processor configurations corresponding to g721 decoding task. 28

3.2 Shortcomings of Customization for Individual Tasks Using Heuristics: a) Equal Hardware Area Division among Tasks. b) Smallest Deadline First. c) Highest Utilization Reduction First. d) Highest Ratio of Reduction of Utilization to Hardware Area. e) Optimal Solution 29

3.3 Utilization versus Area for different task sets under EDF and RMS scheduling policies. 38

3.4 Area versus Energy for Task Set 3 under EDF and RMS scheduling policies. 39

4.1 Motivating Example. 45
4.2 Solving the GAP problem for the corner point *A* will either return a dominating solution or declare that there is no solution in the shaded area. 50
4.3 The overall two-stage approximation scheme. 55
4.4 The exact and approximate Pareto curves for $\varepsilon = 0.69$, 3. (a) *workload-area* Pareto curve for *g721decode*. (b) *utilization-area* Pareto curve for task set 1 58

5.1 Regions and Custom Instructions. 66
5.2 Illustration of Multi-Level Graph Partitioning. The dashed lines show the projection of a vertex from a coarser graph to a finer graph. 68
5.3 Reduction in processor utilization with increasing number of iterations . . . 78
5.4 (a) Analysis time of our approach with varying input utilization for all 5 task sets; and (b) Hardware area required by custom instructions with varying input utilization for all 5 task sets 79
5.5 Speedup versus Analysis Time 81
5.6 Design tradeoffs in processor customization. 83

6.1 Stretch S6000 datapath [38]. 86
6.2 Spatial and temporal partitioning of the custom instructions of an application and the state of the CFU fabric during execution. 88
6.3 System design flow 90
6.4 Motivating Example. 94
6.5 Three phases of iterative partitioning algorithm for number of configurations = 2 98
6.6 Reconfiguration cost graph from loop trace 102

6.7 Modeling the temporal partitioning problem as k-way graph partitioning problem. 104
6.8 Comparison of the quality of the solutions returned by the algorithms for synthetic input. Exhaustive search fails to return any solution with more than 12 hot loops. 109
6.9 An example of custom instruction for Stretch processor. 111
6.10 Comparison of the quality of solutions for the case study of JPEG application. 114

7.1 A set of periodic task graphs and its schedule 118
7.2 Running Example 123
7.3 Task Graphs 133
7.4 Comparison of DP, Optimal, and Static 136

8.1 Wearable bio-monitoring. 139
8.2 Pulse Transmit Time [35]. 141
8.3 Bio-monitoring Applications. 142
8.4 Performance Speedup with Customization. 147

Chapter 1

Introduction

Over the past decade, electronic products (such as consumer electronics, multimedia and communication devices) have dramatically increased in terms of both quantity and quality. Each such product is typically powered by a computer system that is constrained by small size, high performance with low power consumption or low temperature. This kind of computer system is called an *embedded system* because it is typically *embedded* inside the electronic device. As silicon density doubles every 18 months according to Gordon E. Moore's observation, the more functionalities can be integrated into an electronic product which leads to more complexity of the corresponding embedded system. Moreover, embedded systems design is also constrained by short time-to-market window due to the short life cycle of electronic products as well as the competitive market. Therefore, there is a necessity of an efficient design methodology for current generation embedded systems.

The traditional solution of increasing the clock frequency of the processor core to improve the performance is not feasible because the corresponding power dissipation will outweigh the performance benefits. In fact, power dissipation is roughly proportional to the square of the operating voltage and the maximum operating frequency is roughly linear in the operating voltage [73]. Moreover, the increase in power dissipation results in an

increase heat dissipation, which requires cooling system for embedded System-On-Chip (SoC) devices. Moreover, hot chips increase the size of the required power supplies, increases noise and decreases system reliability. Consequently, clock rates for typical embedded processor cores have increased slowly over the past two decades to only few hundred MHz.

In order to maximize the performance as well as minimize power consumption and area overhead, designing ”hand-crafted” Application Specific Integrated Circuit (ASIC) for embedded system appears quite promising. However, ASIC has a long time-to-market from specification to final product that requires (at least): Register Transfer Level (RTL) code development, functional verification, logic synthesis, timing verification, place and route, prototype build and test, and system integration with software test. For any small changes to system specification or errors in the design, most of ASIC development stages must be redone. Moreover, software development has access to ASIC devices only at the system integration stage. Therefore, ASIC is inflexible in the changes (i.e, functionality) of current generation embedded systems. In addition, due to the increasing complexity of hardware designs, implementing the whole application onto ASIC may be infeasible and too expensive.

In contrast to ASIC, a general-purpose processor is completely flexible to accommodate a wide range of applications with arbitrary complexity because of its generic Instruction Set Architecture (ISA). The functionalities of general purpose processor are determined by the programs running on it. These programs are composed of sequences of instructions in the processor’s ISA. In order to change the functionality of general purpose processor, we simply change the corresponding program (also called software) and we do not modify anything in hardware. However, due to the generic nature of the ISA and the sequential execution, a simple computation in hardware is decomposed into multiple instructions that

results in large code size and high number of instructions fetching and decode. Therefore, execution time as well as power consumption of the same simple computation on general-purpose processor are very high.

Combining the efficiency of ASIC and the flexibility of general purpose processor, reconfigurable hardware, such as Field Programmable Gate Array (FPGA), was expected to be a promising solution for embedded software design. With the ability of runtime reconfiguration, different computations can be reconfigured onto FPGA at runtime. However, runtime reconfiguration comes at a price of reconfiguration delay. Typically, FPGAs not only achieve high performance through parallel computation and hardware virtualization but also offer the flexibility of easily changing the functionalities of the application or design after devices deployment. However, FPGAs are not as performance efficient as ASIC and the unit cost is very high. Moreover, FPGAs consume more power than ASIC because programmability requires more transistors than a customized circuit. Finally, compared to general purpose procesor, parallel programming in hardware description language requires much more effort than code development for general purpose procesor.

Recently, there is a trend to customize an existing processor core to target a specific application [48]. Instead of building a brand new processor from scratch by going through long hardware/software co-design flow (from specification to system integration and test), an existing processor core is typically customized by removing functional units that are unused for a specific application to reduce die size, power consumption and cost. Moreover, processor customization can be done through changing the micro-architectural parameters such as the cache sizes, memory or register files sizes, etc. More importantly, a customizable processor may support application-specific extensions of the core instruction set. This kind of customizable processor is also called *instruction-set extensible processor*.

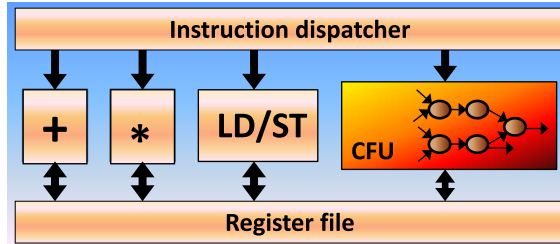


Figure 1.1: Instruction-Set Extensible Processor

1.1 Instruction-Set Extensible Processor

Custom instructions encapsulate the frequently occurring computation patterns in an application. They are implemented as custom functional units (CFU) in the datapath of the existing processor core (Figure 1.1). Because CFU is closely coupled with the existing processor core, instruction-set extensible processors overcome the limited bandwidth of off-chip bus interface in the typical coupling between processor core and FPGA or co-processor. Instruction-set extensible processor achieves performance speedup through chaining and parallelization of a sequence of primitive instructions, which are sequentially executed in general purpose processor. Moreover, packing multiple primitive instructions into a single custom instruction results in smaller number of instructions in the executable file, which leads to smaller numbers of instruction fetching, decoding as well as temporary registers. As a result, instruction-set extensible processor (extensible processor for short) not only achieves high performance but also low power consumption.

Tailoring an instruction-set extensible processor to a specific application demands a considerable amount of manual effort. Therefore, it is necessary to automate the process to create an extensible processor from high-level description of an application. This automated process can generate both hardware implementation of extensible processor core and relevant software tools such as instruction set simulator, compiler, debugger, assembler and related tools to create applications for extensible processors. Generating custom

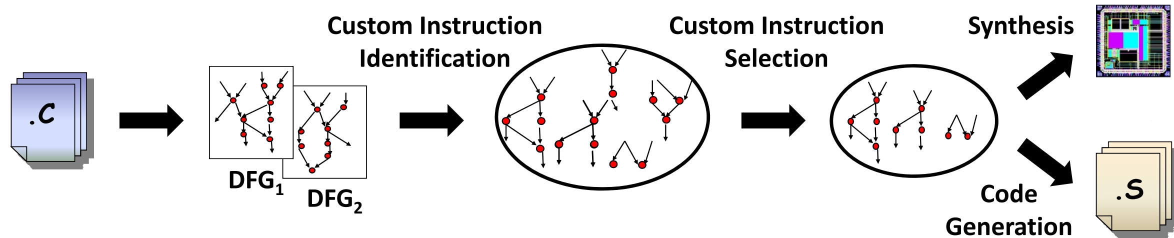


Figure 1.2: Instruction-Set Extensible Processor Design Flow

instruction specifications is crucial to the efficiency of extensible processor. To generate the best custom instructions for an application, designers need to be expert in hardware design as well as understand the nature of the application clearly. Consequently, custom instructions generation for a complicated application may require substantial effort for the designers. Therefore, recent research has focused on automated generation of custom instructions [8, 81, 22, 15, 21, 103, 9, 5, 17, 23, 24, 90, 7, 95].

Typically, automated custom instructions generation for an application consists of two basic steps: custom instructions identification and custom instructions selection. Custom instructions identification enumerates a large set of valid custom instruction candidates from the application's dataflow graph and their frequency via profiling (Figure 1.2). A valid custom instruction must satisfy micro-architecture constraints such as maximum number of input/output and convexity constraints. Input/output constraint specifies the maximum number of input and output operands allowed for a custom instruction, respectively. This constraint arises due to the limited number of register file read/write ports available on a processor. Moreover, under convexity constraint a non-convex custom instruction which has inter-dependency with operations outside the custom instruction is infeasible because the custom instruction cannot be executed atomically. Given this library of custom instruction candidates, the second step selects a subset of custom instructions to maximize the

performance under different design constraints such as hardware area. The state-of-the-art techniques are fairly effective at identifying a set of custom instructions with high performance potential for a single task application.

1.2 Instruction-Set Customization for Multi-tasking Embedded Systems

In multi-tasking embedded systems, multiple tasks share the embedded processor at runtime. Most of these tasks are compute-intensive kernels. Moreover, timing constraints (deadlines) are often imposed on multi-tasking applications such as flight control systems. If a multi-tasking system fails to meet its deadline, the computation of each individual task should be speeded-up so that the deadlines can be satisfied. Extensible processor cores appear to be quite helpful in this scenario. Because custom instructions may reduce the processor utilization for a task set through performance speedup of the individual tasks. This improvement may enable an unschedulable task set to satisfy all the timing requirements. In addition, lower processor utilization due to customization opens up the possibility to execute non-real-time tasks alongside real-time tasks. Finally, a lower utilization can exploit voltage scaling to lower the operating frequency/voltage of the processor which helps to reduce energy consumption.

Given a multi-tasking real-time embedded system, instruction-set customization for individual tasks may lead to local optima. We have to take into account the complex interplay among the tasks enabled by the real-time scheduling policy and the traditional design flow is changed as Figure 1.3. First, custom instructions are identified for each individual task (from T_1 to T_N). Then, custom instructions are selected among constituent tasks under area constraint as well as real-time constraint through design space exploration. The objective

of the selection is to maximize performance, minimize processor utilization or minimize energy consumption. Selected custom instructions will be synthesized and included in the customized processor. Finally, code generation is performed to use the newly defined custom instructions.

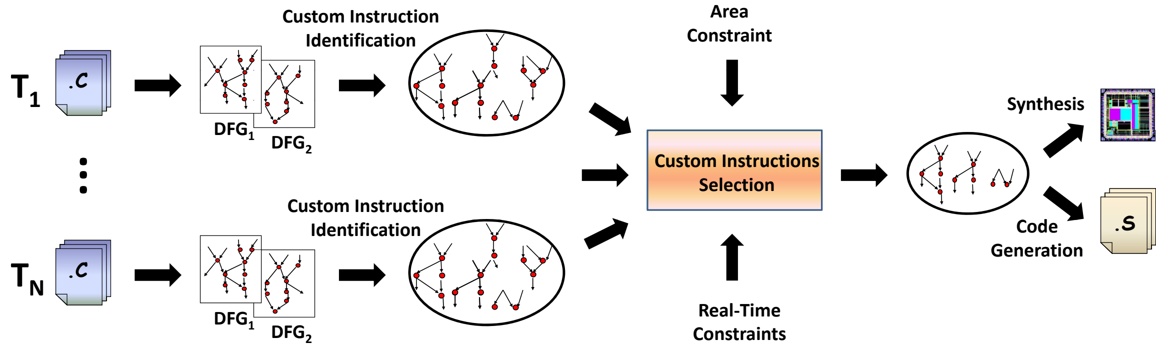


Figure 1.3: Design flow of instruction-set customization for multi-tasking systems

In order to tackle the complex design space exploration of instruction-set customization for multi-tasking real-time embedded systems, we propose efficient algorithms to minimize the processor utilization through the *optimal* custom instructions selection among constituent tasks while satisfying the task deadlines under an area constraint. We extend our study to consider the conflicting tradeoffs among multiple objectives (e.g., performance versus area). As we expose multiple solutions with different tradeoffs, designers have more flexibility to select an appropriate implementation for the system requirements. In particular, we propose an efficient polynomial time algorithm to compute an approximate Pareto front in the design space.

One drawback of the design flow in Figure 1.3 is that it is a bottom-up approach. That is a large amount of time is invested to identify all the custom instructions for all the constituent tasks while only a small subset of custom instructions are finally selected. Based

on this observation, we investigate an iterative custom instruction generation scheme that is highly efficient for customization of multi-tasking systems. In our iterative scheme, we focus on custom instructions generation of the critical tasks and the critical paths within such tasks. As a result, our iterative approach can quickly return a first-cut solution for the critical region in the critical paths. If the first-cut solution satisfies the design requirements, the customization process can be stopped and a large amount of redundant design space exploration is avoided. On the other hand, if the design requirements are not satisfied, the iterative process continues to select the next critical region to generate custom instructions.

Instruction-set customization significantly improves the performance for embedded systems. However, the total area available for the implementation of the CFUs in a processor is limited. In multi-tasking embedded system, each task typically requires unique custom instructions. Therefore, we may not be able to exploit the full potential of all the custom instructions in these high-performance embedded systems. Furthermore, it may not be possible to increase the area allocated to the CFUs due to the linear increase in the cost of the associated system. Fortunately, instruction-set extensible processors can support runtime reconfiguration of custom instructions. Basically, custom instructions can share the CFUs in time-multiplexed fashion at runtime. For multi-tasking systems, runtime reconfiguration is especially attractive, as the fabric can be tailored to implement only the custom instructions required by the active task(s) at any point of time. Of course, this virtualization of the CFU fabric comes at the cost of reconfiguration delay. Therefore, we propose efficient methodologies to strike the right balance between the number of configurations and the reconfiguration cost so that performance is maximized.

Figure 1.4 illustrates a scenario where runtime reconfiguration of custom instructions may improve the performance of the application. Set A represents a set of custom instructions that are selected from a particular application. Set B and set C are disjoint subsets

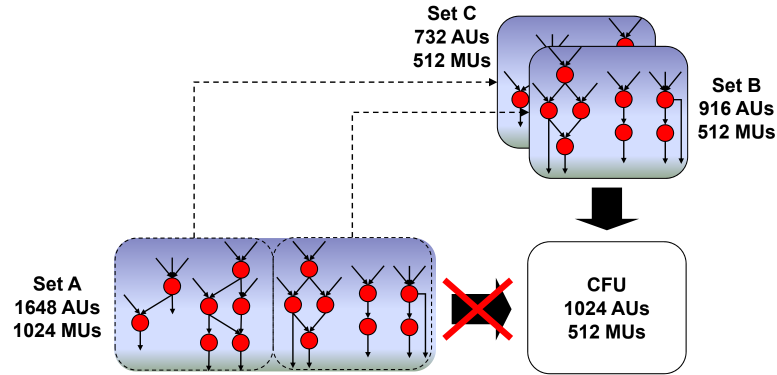


Figure 1.4: Motivating example for dynamic reconfiguration of CFU (AU: arithmetic/logic unit, MU: multiplier unit).

of set A. The available resources in the CFU are insufficient to implement all the custom instructions in Set A. If run-time reconfiguration is not supported, the designer is forced to implement some subset of A into the CFU; thus limiting the potential performance enhancement. On the other hand, both set B and set C are small enough to fit into the CFU. With runtime reconfiguration ability we can exploit all the custom instructions in set A by loading set B or set C into the CFU at different phases of execution of the application. Therefore, the performance benefit of all the custom instructions in set A can be obtained after subtracting reconfiguration cost, even though the available hardware is insufficient to support set A in one configuration.

1.3 Contributions of The Thesis

Envisioning the crucial need of design methodologies for instruction-set customization for multi-tasking embedded systems, this thesis explores customization in the context of multi-tasking real-time systems. The later part of the thesis exploits runtime reconfiguration of custom instructions to further improve the performance speedup of the application.

1. **Customization for multi-tasking real-time embedded systems**: Custom instructions can help to reduce the processor utilization for a task set through performance speedup of the individual tasks. This improvement may enable a task set that was originally unschedulable to satisfy all the timing requirements. Therefore, we propose optimal algorithms to select the optimal set of custom instructions for a task set to minimize the processor utilization while all the timing requirements are satisfied. Moreover, our study also shows that energy consumption can be reduced with the enhancement of custom instructions.

2. **Evaluating design trade-offs for custom instructions**: Our first solution to processor customization for multi-tasking embedded system optimizes for a single objective such as optimizing performance under pre-defined hardware area constraint. We extend our solution to consider multiple objectives, e.g. performance versus area and processor utilization versus area. In particular, we develop a polynomial-time approximation algorithm to systematically evaluate the design tradeoffs in instruction-set customization.

3. **Iterative custom instruction generation**: We investigate an iterative custom instruction generation scheme that is highly efficient for customization of multi-tasking systems. We adopt a top-down approach where the system level performance requirements guide the customization process to zoom into the critical tasks and the critical paths within such tasks. Moreover, an efficient custom instruction generation algorithm is proposed to enhance our iterative approach.

4. **Runtime reconfiguration of custom instructions**: The efficiency of runtime reconfiguration of custom instructions depends on the right number of configurations and partitioning custom instructions into each configuration. We develop a framework

that starts with a sequential application specified in ANSI-C and can automatically select appropriate custom instructions as well as club them into one or more configurations so that the performance is maximized.

5. **Runtime reconfiguration of custom instructions for multi-tasking embedded systems**: We extend our study of runtime reconfiguration of custom instructions to multi-tasking applications with real-time constraints. We propose a pseudo-polynomial time algorithm that performs near-optimal spatial and temporal partitioning of custom instructions to minimize processor utilization while satisfying all the real-time constraints

6. **A case study of processor customization**: To demonstrate the efficiency of instruction set customization, wearable bio-monitoring applications are selected as a case study for processor customization.

1.4 Organization of The Thesis

The roadmap of the thesis is shown in Figure 1.5. We discuss background and related work to our study in Chapter 2. Custom instructions for real-time embedded systems is studied in Chapter 3. In Chapter 4, we develop a polynomial-time approximation algorithm to systematically evaluate the design tradeoffs of custom instructions. We present an iterative custom instruction generation scheme in Chapter 5. In Chapter 6, we present runtime reconfiguration of custom instructions for a sequential application. We consider runtime reconfiguration of custom instructions for multi-tasking applications in Chapter 7. Chapter 8 presents a case study of processor customization. Finally, Chapter 9 concludes this thesis and enumerates the directions to extend our study.

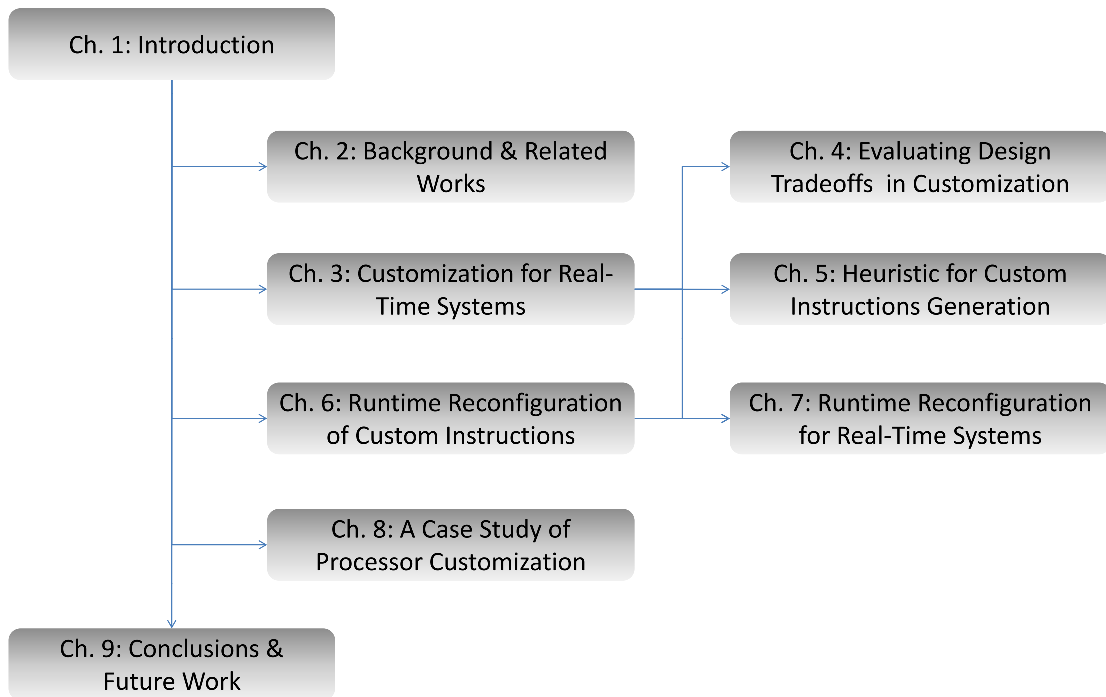


Figure 1.5: Roadmap of thesis

Chapter 2

Background and Related Works

We start this chapter with the key architectural features of an instruction-set extensible processor. Next, we describe the compiler design flow to support instruction-set extensible processors. This is followed by different automated custom instructions generation methods. In the next section, we present the study in the customization for Multi-Processor System on Chip (MPSoC). Finally, we summarize related works in the reconfigurable computing community.

2.1 Architecture of Instruction-Set Extensible Processor

Instruction-set extensible processor (extensible processor for short) significantly reduces the design and verification effort by using software programmable Custom Functional Units (CFUs) instead of hardwired control logic. Most of the control flow is managed by software running on the processor core and instruction decoder generates the appropriate control signals for the execution of CFU. This software based approach makes the design more resilient against any later changes in system specification.

As mentioned earlier, a CFU is integrated into the datapath of the existing proces-

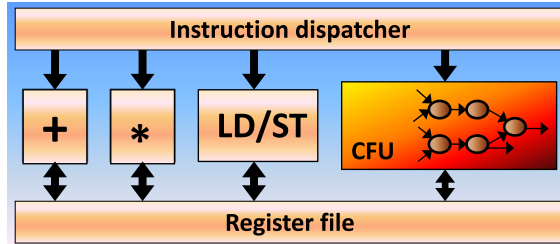


Figure 2.1: Instruction-Set Extensible Processor

sor core. CFU shares register file ports, operand buses, forwarding and interlock logic with traditional functional units. CFU can access memory system through load/store units (LD/ST). However, integration of CFU into the datapath has certain constraints. First, the silicon area of the CFU is limited and custom instructions must fit into the available area. Second, the available register file ports and dedicated data transfer channels constrain the data bandwidth between CFU and the existing datapath. Finally, a fixed length instruction word can encode a limited number of input and output operands of a custom instruction.

With the typical architecture of the instruction-set extensible processor in Figure 2.1, after selected custom instructions are synthesized as CFUs and fabricated, we can not change the custom instructions anymore (Figure 2.2.a). Therefore, this type of architecture is called static configuration. Xtensa [37], ARC 700 family [4], MIPS32 74K [1] are some examples of well-known commercial static extensible processors. Therefore, we need to design and fabricate different customized processors for different application domains. A processor customized for one application domain may fail to provide any tangible performance benefit for a different domain. Soft core processor with extensibility features that are synthesize in FPGAs (e.g., Altera Nios [3], Xilinx MicroBlaze [98]) may resolve this problem as the customization can be performed post-fabrication. However, customizable soft cores suffer from lower frequency and higher energy consumption issues because the entire processor is implemented in FPGAs (and not just the CFUs). Besides cross-domain performance

problems, extensible processors are also limited by the amount of silicon available for implementation of the CFUs. As embedded systems progress towards highly complex and dynamic applications (e.g., MPEG-4 video encoder/decoder, software-defined radio), the silicon area constraint becomes a primary concern. Moreover, for highly dynamic applications that can switch between different modes (e.g., runtime selection of encryption standard) with unique custom instructions requirements, a customized processor catering to all the scenarios will clearly be a sub-optimal design. In this context, extensible processor with the ability of runtime reconfiguration offers a potential solution to all these problems.

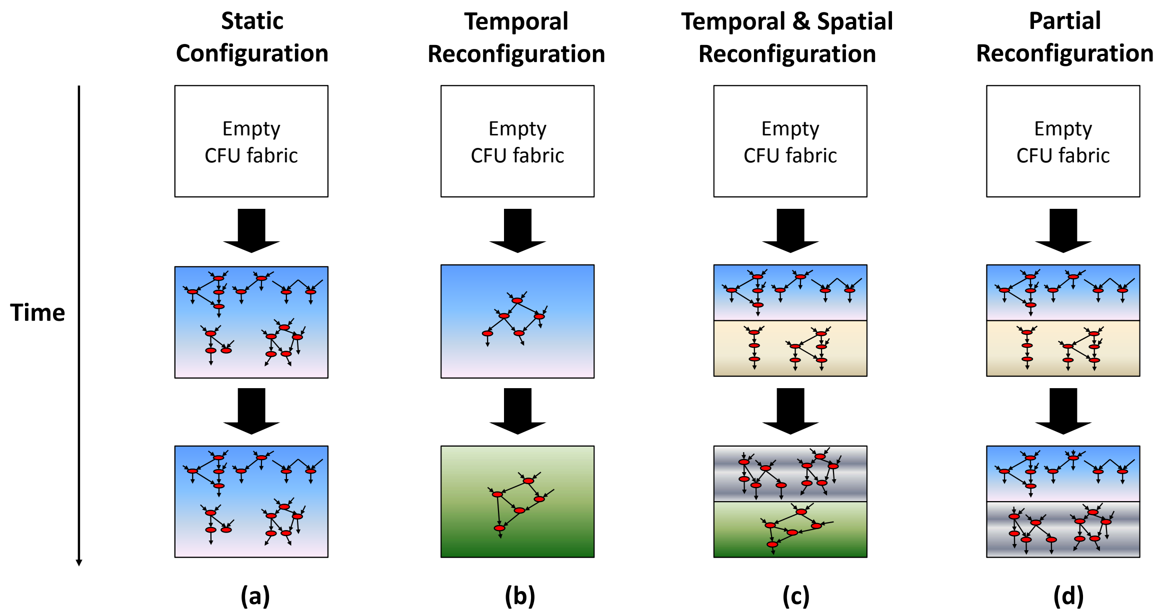


Figure 2.2: Four types of instruction-set extensible processors.

Runtime reconfigurable extensible processors can be configured at runtime to change its custom instructions and the corresponding CFUs. Clearly, to achieve runtime reconfiguration, the CFUs have to be implemented in some form of reconfigurable logic. But the processor core is implemented in ASIC to provide high clock frequency and better energy efficiency. As CFUs are implemented in reconfigurable logic, these extensible pro-

cessors offer full flexibility to adapt (post-fabricate) the custom instructions according to the requirement of the application running on the system and even midway through the execution of an application. Runtime reconfiguration consists of temporal reconfiguration, temporal and spatial reconfiguration and partial reconfiguration.

Temporal reconfiguration: This architecture allows only one custom instruction to exist at any point of time. Some examples of temporal reconfigurable processors are Programmable Instruction Computer [84] and OneChip [49]. That is, there is no spatial sharing of the reconfigurable logic among custom instructions (Figure 2.2.b). Moreover, temporal reconfiguration can result in high reconfiguration cost specially if two custom instructions in the same code segment are executed frequently, for example, inside a loop body.

Temporal and spatial reconfiguration: This architecture enables spatial reconfiguration, that is, the reconfigurable hardware can be shared among multiple custom instructions. Some examples of temporal and spatial reconfigurable processors are Chimaera [100] and Stretch [38]. The combination of spatial and temporal reconfiguration is a powerful feature that partitions custom instructions into multiple configurations, each of which contains one or more custom instructions (Figure 2.2.c). This clustering of multiple custom instructions into a single configuration can significantly reduce the reconfiguration overhead.

Partial reconfiguration: This architecture provides the ability to reconfigure only part of the reconfigurable fabric. Some examples of partial reconfigurable processors are Dynamic Instruction Set Computer [96], XiRisc [71] and Rotating Instruction Set Processing Platform [11]. With partial reconfiguration, idle custom instructions can be removed to make space for the new instructions. Moreover, as only a part of the fabric is reconfigured, it further saves reconfiguration cost (Figure 2.2.d).

2.2 Instruction-Set Customization Compilation Flow

Automated custom instructions generation for a given application to meet the design goals is the main challenge of customizing processors. Automated custom instructions generation is performed by augmenting the conventional compilation flow with a few steps supporting custom instructions generation.

Given the application code , conventional compiler front-end performs lexical, syntax and semantic analysis to transform high-level language statements into machine-independent Intermediate Representation (IR). Then, IR optimizer performs constant propagation, dead code elimination, common subexpression elimination, etc. Next, back-end of the compiler generates binary executable codes for the target processor from the optimized IR. During back-end phase, instruction binding allocates IR objects to actual architectural resources as well as operations to instructions. Register allocation binds operands to registers or memory locations. Instruction scheduling takes cares of concurrencies and dependencies among instructions by allocating them to different time slots. Moreover, the back-end phases also perform machine dependent optimizations.

For custom instructions generation, the IR is formed into Control Flow Graph (CFG). The nodes of CFG are the basic blocks of the application. A basic block has only one entry statement and only one exit statement. An edge between two basic blocks in CFG represents the control flow between them (if-else, loops or function calls).

Control dependencies do no exist in a basic block but data dependencies do. Each basic block is represented in the form of Data Flow Graph (DFG). For each basic block, DFG has operations as nodes and edges between nodes show data dependencies. Each node of DFG is typically bound to one machine instruction through instruction binding. A cluster of operations inside DFG can form a custom instruction, which is represented as a subgraph

of DFG.

Custom instructions generation starts with compiling the application written in high-level language such as C/C++. Then, the application is profiled by executing with standard input data sets on the base processor. Typically, hot basic blocks take up a significant portion of the application's total execution time. Therefore, hot basic blocks should be considered for custom instructions identification, which results in a set of high potential custom instruction candidates for hardware implementation. If these custom instructions are implemented in hardware, execution time of the application, originally in pure software, can be significantly reduced. Custom instruction candidates must first satisfy architectural constraints such as input, output and convexity constraints. After the custom instructions identification, a subset of custom instruction candidates are selected to maximize the performance of the application under different design constraints such as hardware area constraint. Finally, subgraphs corresponding to selected custom instructions are identified in the DFG of each basic block and replaced by custom instructions. Custom instructions generation is performed after IR optimizer and before register allocation.

2.3 Custom Instructions Generation for an Application

Custom instructions are typically generated for an application through two phases: custom instructions identification and custom instructions selection. First, frequently occurring computation patterns are extracted from the DFG of the program. Then, a subset of the extracted patterns are selected to maximize a design criteria (e.g., performance gain) under some design constraints (e.g., hardware area).

2.3.1 Custom Instructions Identification

Custom instructions are identified in the scope of a basic block. For crossing basic blocks code motion [34], predicated execution [42] and control localization [67] techniques are applied before identifying custom instructions. A custom instruction candidate is an induced subgraph of the DFG. Therefore, custom instructions identification problem is to identify subgraph candidates for custom instructions in a DFG. The number of custom instruction candidates of a DFG is exponential in terms of the number of nodes of the DFG. However, number of feasible subgraphs is limited by architectural constraints such as convexity and input/output constraints.

A greedy algorithm [82] is developed to identify the maximal Multiple Inputs Single Output (MISO) patterns. The algorithm starts from the sink node of the data flow graph (DFG) and tries to add its parents as long as the number of inputs is not greater than the maximum allowed inputs and there is only one output. Therefore, the complexity of the algorithm is linear in the number of nodes in the DFG. On the other hand, identifying Multiple Inputs Multiple Outputs (MIMO) patterns is difficult as there can potentially be exponential number of them in terms of the number of nodes in the DFG. [8, 81, 22, 15, 21, 103] enumerate all possible custom instruction candidates. Atasu *et al.* [8] use Integer Linear Programming solution while Pozzi *et al.* [81] and Cheung *et al.* [22] use exhaustive search with pruning heuristics. Bonzini *et al.* [15] prove the number of valid convex custom instructions is $O(n^{N_{in}+N_{out}})$ for a DFG which has n nodes and N_{in}, N_{out} are input/output constraints. However, the complexity grows dramatically when input/output constraints are relaxed and the size of DFG is quite large. Yu *et al.* [103] propose a scalable three phases algorithm, which cuts down a large amount of computation to enumerate all custom instructions. Later, Chen *et al.* [21] propose another algorithm having similar runtime to Yu's algorithm.

The worst case complexity for enumerating all possible custom instructions is exponential. Therefore, heuristic algorithms are proposed to improve the analysis time. Different clustering techniques are used in [9, 5, 17, 23, 24, 90] for fast enumeration of good custom instruction candidates. Arnold *et al.* [5] use an iterative technique that replaces the occurrences of previously identified smaller patterns with single nodes to avoid the exponential blow-up. Baleani *et al.* [9] add nodes to the current pattern in topological order till input or output constraint is violated. The algorithm then starts a new pattern only with the node that caused the violation. Sun *et al.* [90] prune less potential custom instructions through guide functions while Clark *et al.* [24] expand the custom instruction from a seed node only in the directions that can possibly lead to good pattern. Choi *et al.* put constraint on the number of operations which can be included in a subgraph. Brisk *et al.* [17] use All-Pairs Common Slack Graph to evaluate the feasibility that two operations may be paired (grouped) together. The top ranked pairs are merged as single nodes and can be used in the later steps. Recently, [7, 95] relax the constraints on the number of input and output operands to generate custom instructions.

In order to increase the potential of instruction parallelism to provide better performance if the base architecture does not support instruction-level parallelism, a subgraph candidate may contain one or more disconnected subgraphs. [81, 23, 36] consider disconnected subgraph as well as connected subgraph with a custom instruction candidate.

2.3.2 Custom Instructions Selection

The benefit of a custom instruction candidate is computed as the product of its speedup (if implemented in CFU compared to software) and its execution frequency via profiling. Each custom instruction also comes with a cost value in terms of silicon area. Given the library of custom instruction candidates, custom instructions selection step selects a subset

of custom instructions to maximize the performance under different design constraints such as silicon area. The first reason for this objective function is that the silicon area is limited for CFUs. Selecting many custom instructions for the application not only costs more silicon area, but also makes the circuit design more complicated such as decoding and/or bypass network. Therefore, only the most efficient custom instructions will be selected. Second, only a subset of custom instructions will cover the application code during code generation. Typically, a base operation is covered by at most one custom instruction. Otherwise, the same computations are unnecessarily duplicated for these custom instructions and unschedulable code may be generated.

Arnold *et al.* [5] propose a dynamic programming solution to select optimal subset of custom instructions. However, dynamic programming solution does not take into consideration subgraph isomorphism and therefore does not minimize the number of custom instructions. Sun *et al.* [89] develop a branch and bound algorithm for custom instructions selection. Cong *et al.* [25] formulate custom instructions selection as an 0-1 Knapsack problem while Lee *et al.* [66] formulate custom instruction selection as an Integer Linear Programming problem. Recently, Wolinski *et al.* [97] consider the integration of custom instruction selection, binding and scheduling using constraint programming. Greedy heuristics are also proposed based on different priority functions for custom instruction candidates [24, 22, 64]. To overcome local optima, genetic algorithm (GA) is employed in [86] based on the idea of chromosome evolution. In [80], GA is also used to optimize performance using runtime reconfigurable functional units. Simulated annealing (SA) is applied in [43] to overcome the local optima. These heuristics trade-off the optimal results with the analysis time complexities. Typically, they return pretty good results compared to the optimal results with much faster analysis times (in term of seconds). Most studies consider single objective, e.g performance gain, hardware area, etc. Some other methods consider

the multi-objective solutions such as performance gain and area. A Multi-objective GA based method is described in [18] to discover the Pareto front with performance and area as multiple objectives.

2.3.3 Integrated Custom Instructions Generation

There are few works [6, 68, 81, 13] that combine the two steps (custom instructions identification and selection) and generate custom instructions in an integrated task. Two methods in [6, 68] use Integer Linear Programming (ILP) solutions to generate a single best custom instruction for each iteration. In each iteration, ILP solver evaluates and returns the best custom instruction. Similarly, both *Iterative* selection algorithm [81] and *ISEGEN* algorithm [13] generate the best custom instruction for each iteration. The only difference is that *Iterative* algorithm applies the optimal single-cut (single custom instruction) identification algorithm [81] to generate a quality custom instruction while *ISEGEN* algorithm [13] uses the basic principles of Kernighan-Lin min-cut heuristic [59]. Once the best custom instruction is generated, its constituent nodes are removed from consideration in following iterations. Thus, the current custom instruction may affect the quality of its neighborhood custom instructions in the following iterations and the process is likely to reach local minima.

2.4 Customization for MPSoC

The state-of-the-art techniques are fairly effective at identifying a set of custom instructions with high performance potential for an application. However all of these techniques focus on sequential application. Instruction-set customization for multi-tasking applications has largely remained unexplored except for [91]. Fei *et al.* [91] study custom instructions gen-

eration for a task graph of an application for a MPSoC platform. Constituent tasks of a task graph have dependencies. The objective of their study is to minimize the execution time of the task graph after it is mapped into multiple processors. Recently, Javaid *et al.* [53] present a design flow to customize streaming application on heterogeneous pipelined multiprocessor systems. However, they do not really consider custom instructions generation for multi-tasking applications under timing constraints. Our study will focus on custom instructions generation for multi-tasking applications under real-time scheduling policy.

2.5 Reconfigurable Computing

Our works on runtime reconfiguration focus on temporal and spatial reconfiguration of extensible processors. We first investigate the efficiency of runtime reconfiguration of custom instructions for a sequential application. Then, we extend runtime reconfiguration of custom instructions to multi-tasking applications with real-time constraints. The major part of the research on runtime reconfiguration comes from the reconfigurable computing community.

Usually, the temporal and spatial partitioning are done at coarse-grained level (such as task graph representation of an application) [10, 20, 58, 92], though there exist some exceptions. Li *et al*. [69] partition at the loop level while Purna and Bhatia [83] perform partitions on the data flow graph. With a task graph as input, computing reconfiguration costs becomes simple because the underlying directed acyclic graph representation ensures at most one reconfiguration between any two nodes. It should be noted that while Purna and Bhatias work [83] partitions at the finer granularity of functions and operators, their work uses directed acyclic data flow graph as input as well. However, for fine-grained (loop level) customization, reconfiguration cost model is complex as the number of reconfigurations for

one loop depends on temporal partitioning of all the other loops. Li's work [69] does not consider reconfiguration cost during partitioning process but it deducts reconfiguration cost when computing performance gain.

In a different direction, Bondalapati and Prasanna [14] focus on mapping the statements within a loop into configurations to obtain a configuration sequence that gives the least execution time. While dynamic reconfiguration is used as well, their work focuses on intra-loop selection of configurations, i.e., their work operates on one loop only. Hardnett *et al.* [40] form a framework in which the dynamically reconfigurable architectural design space may be explored for specific applications. However, custom instructions do not share the same functional unit, i.e., no spatial partitioning is required. Secondly, the problem of reconfiguration cost is not addressed directly. Rather, custom instructions are de-selected to relieve resource pressure rather than optimizing overall performance. In general, temporal and spatial partitioning at loop level while considering reconfiguration cost is still a challenge for our study.

Related works to instruction-set customization for multi-tasking systems with runtime reconfiguration support also mainly come from reconfigurable computing. Co-synthesis of multiple periodic task graphs with real-time constraint onto heterogeneous distributed embedded systems is addressed in [26, 62]. [41] partitions a task graph with timing constraints into a set of hardware units. Enforcing schedulability of real-time tasks with hardware implementation appears in [85]. None of these techniques takes into account the reconfiguration overhead or possibility of both spatial and temporal partitioning. [30, 72] co-synthesize real-time task graphs onto distributed systems containing dynamically reconfigurable FPGAs. These works assume a single hardware implementation of a task in FPGA and do not explore the hardware design space to evaluate tradeoffs between different implementations of the same task. Moreover, they do not consider any hardware area con-

straint, an important constraint of instruction-set customization. Therefore, we investigate an efficient algorithm which takes into account most of the key design issues of instruction-set customization such as hardware area constraints, multiple implementations of the same task, temporal and spatial partitioning and real-time constraints.

Chapter 3

Customization for multi-tasking real-time embedded systems

One of the major challenges in the effective deployment of customizable processors is the development of the design-automation tool chain. In particular, a major research focus in the recent past has been automated generation of suitable instruction-set extensions for an application [48]. Given a single sequential application, the goal here is to select a set of custom instructions that optimizes certain design criteria (such as power or performance) under pre-defined design constraints (such as silicon area).

Multi-tasking real-time embedded systems add substantial complexity to this design space exploration process. The optimization problem in this context is to minimize the processor utilization (through custom instructions) while satisfying the task deadlines under an area constraint. Clearly, a naïve approach of optimizing the execution time of each task in isolation will miss certain opportunities. We have to take into account the complex interplay among the tasks enabled by the real-time scheduling policy.

In this chapter, we explore customization in the context of multi-tasking real-time embedded systems. We propose efficient algorithms to select the *optimal* set of custom in-

structions for a multi-tasking real-time workload in Section 3.1. We consider two popular real-time scheduling policies: a static priority based Rate-Monotonic Scheduler (RMS) and a dynamic priority based Earliest Deadline First (EDF) scheduler. For EDF scheduling policy, we employ a dynamic programming solution whereas for RMS scheduling, we resort to an efficient branch-and-bound based search algorithm. In Section 3.2, our experimental evaluation with a large number of workloads confirms the benefit of processor customization in real-time systems.

3.1 Customization for Real-Time Systems

3.1.1 Problem Formulation

In the classic model of a real-time system, a set of tasks are executed periodically. Each task T_i is associated with a period P_i and a worst-case execution time C_i. An instance of the task T_i is released periodically once every P_i time units. The task instance should complete execution by its deadline, which is typically defined as the end of the period. The goal of real-time scheduling is to meet the deadline of every task. Schedulability analysis determines whether a specific set of tasks can be successfully scheduled using a specific scheduler. Given a set of N independent, preemptable, and periodic tasks on a uniprocessor, let U be the total utilization of this task set. U quantifies the fraction of processor cycles used by a task set. Therefore, a *necessary* condition for feasible scheduling of a task set [70] is

$$U = \sum_{i=1}^{N} U_i = \sum_{i=1}^{N} \frac{C_i}{P_i} \leq 1 \tag{3.1}$$

We would like to explore the opportunities opened up by instruction-set customization in this context. Each task T_i has a set of custom instructions enhanced configurations with different performance/silicon area tradeoff. The higher is the area cost of a custom instruc-

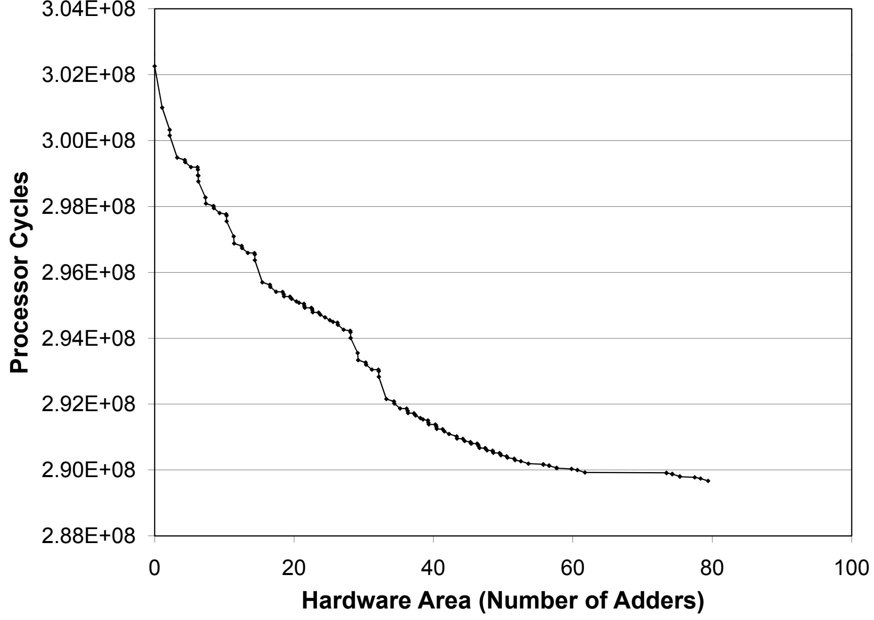


Figure 3.1: Application performance versus hardware area for different processor configurations corresponding to g721 decoding task.

tion configuration, the better is its performance. Let $config_{i,j}$ (for $j = 1 \ldots n_i$) be the jth configuration corresponding to task T_i and n_i is the number of configurations for task T_i. In addition, let $cycle_{i,j}$ and $area_{i,j}$ denote the application performance in processor cycles and gate count of $config_{i,j}$. We assume that $config_{i,1}$ corresponds to the configuration without any custom instruction, i.e., $area_{i,1} = 0$ and $cycle_{i,1} = C_i$ (the task performance without any enhancement). For example, Figure 3.1 shows the set of configurations corresponding to g721 decoding task.

Given (1) a set of independent, preemptable, and periodic tasks, (2) a specific scheduling policy (RMS or EDF), and (3) a total area budget *AREA* for the custom instructions, our goal is to select an appropriate configuration for each task such that the task set is schedulable and the total utilization U is minimized.

Task (T_i)	Deadline (Period)	Execution Time	$config_{i,1}$ ($area_{i,1}, cycle_{i,1}$)	$config_{i,2}$ ($area_{i,2}, cycle_{i,2}$)	Number of Instances
T_1	6	2	0,2	7,1	4
T_2	8	3	0,3	6,2	3
T_3	12	6	0,6	4,5	2

$$U = U_1 + U_2 + U_3 = \frac{2}{6} + \frac{3}{8} + \frac{6}{12}$$

$$= \frac{2}{6} + \frac{3}{8} + \frac{6}{12} = \frac{2*4}{24} + \frac{3*3}{24} + \frac{6*2}{24} = \frac{29}{24} > 1$$

(a) $U' = \frac{2*4}{24} + \frac{3*3}{24} + \frac{6*2}{24} = \frac{29}{24} > 1$

(b) $U' = \frac{1*4}{24} + \frac{3*3}{24} + \frac{6*2}{24} = \frac{25}{24} > 1$

(c) $U' = \frac{1*4}{24} + \frac{3*3}{24} + \frac{6*2}{24} = \frac{25}{24} > 1$

(d) $U' = \frac{1*4}{24} + \frac{3*3}{24} + \frac{6*2}{24} = \frac{25}{24} > 1$

√ (e) $U' = \frac{2*4}{24} + \frac{2*3}{24} + \frac{5*2}{24} = \frac{24}{24} = 1$

Figure 3.2: Shortcomings of Customization for Individual Tasks Using Heuristics: a) Equal Hardware Area Division among Tasks. b) Smallest Deadline First. c) Highest Utilization Reduction First. d) Highest Ratio of Reduction of Utilization to Hardware Area. e) Optimal Solution

3.1.2 Motivating Example

Figure 3.2 shows an example of a system consisting of three periodic tasks. Each task is specified by its deadline which is equal to its period and the worst-case execution time. Let us assume EDF scheduling policy is considered. Obviously, the task set in Figure 3.2 is not schedulable because the total utilization U greater than 1. Therefore we need to select appropriate custom instructions for the task set to make the task set schedulable. We further assume that the available hardware area is equal to 10.

The easiest solution is that we divide the available hardware resource equally among the three tasks and select custom instructions separately under the individual hardware area constraint for each task to $\lfloor \frac{10}{3} \rfloor = 3$. In this case, we can not select any custom instruction because available hardware area is less than the required hardware area for any custom

instruction and processor utilization is still greater than 1 (Figure 3.2.a). The next solution is that we prioritize the tasks and select the highest priority task to consider customization first and so on. Priorities can be computed based on:

- Task priorities in EDF policy that the ask with smaller deadline has higher priority.
- With the enhancement of custom instructions, the task with higher reduction of processor utilization has higher priority.
- With the enhancement of custom instructions, the task with higher ratio of reduction of processor utilization to consumed hardware area has higher priority.

All three solutions, in which T_1 is the only task using custom instructions, can not reduce total processor utilization to less than or equal to 1 (Figures 3.2.b, 3.2.c, 3.2.d respectively). Therefore, it is non-trivial to apply instruction set customization for a single task to multi-tasking embedded system with real-time constraints. We need efficient instruction-set customization methodologies to come up with a feasible solution in which T_2 and T_3 are implemented with their corresponding custom instructions while T_1 is implemented in software (Figure 3.2.e).

3.1.3 Customization for EDF Scheduling

Earliest Deadline First (EDF) is an optimal dynamic priority scheduling policy. It executes at any instant, the ready task with the closest deadline. If more than one ready tasks have the same deadline, EDF randomly selects one for execution. A task set is schedulable under EDF policy if the total utilization (U) is less than or equal to 1 [70] (Equation 3.1).

We develop an algorithm to select the appropriate configuration for each task such that the total utilization of the task set is minimized. As the value of total utilization determines

the feasibility of an EDF schedule, the algorithm, by definition, works towards meeting task deadlines. If the minimum utilization returned by the algorithm is greater than 1, then the task set cannot be scheduled even with custom instruction enhancements.

We propose a pseudo-polynomial time dynamic programming algorithm that returns the *optimal* solution. Let $U_i(A)$ be the *minimum* total utilization of tasks $T_1 \dots T_i$ under an area budget A. Then $U_i(A)$ can be defined recursively.

$$U_i(A) = \min_{\substack{j=1 \dots n_i \\ area_{i,j} \le A}} \left(\frac{cycle_{i,j}}{P_i} + U_{i-1}(A - area_{i,j}) \right) \tag{3.2}$$

That is, given an area A, we explore all possible configurations for T_i and choose the one that results in minimum utilization for tasks $T_1 \dots T_i$. The base case for task T_1 is

$$U_1(A) = \min_{\substack{j=1 \dots n_1 \\ area_{1,j} \le A}} \left(\frac{cycle_{1,j}}{P_1} \right) \tag{3.3}$$

The minimum utilization for tasks $T_1 \dots T_N$ under area budget $AREA$ then corresponds to $U_N(AREA)$.

Algorithm 1: **Custom Instructions selection under EDF**

Input: Task Set $\mathrm{T_1, \dots, T_N}$ with configurations; Area constraint: AREA

Result: Minimum utilization

for A = 0 to AREA in steps of Δ **do**

$\quad \mathrm{U_1(A) \leftarrow \min_{\substack{j=1 \dots n_1 \\ area_{1,j} \le A}} \left(\frac{cycle_{1,j}}{P_1} \right)}$

end

for A = 0 to AREA in steps of Δ **do**

 for i=2 to N **do**

$\quad \mathrm{U_i(A) \leftarrow \min_{\substack{j=1 \dots n_i \\ area_{i,j} \le A}} \left(\frac{cycle_{i,j}}{P_i} + U_{i-1}(\lfloor \frac{A - area_{i,j}}{\Delta} \rfloor \times \Delta) \right)}$

end

return $\mathrm{U_N(AREA)}$;

Algorithm 1 encodes this recursion as a bottom-up dynamical programming algorithm.

The step value Δ determines the granularity of area. Δ is the greatest common divisor of all configurations' area of all tasks and AREA. The time complexity of this algorithm is $O(N \times \frac{Area}{\Delta} \times x)$ where $x = max_{i=1...N}(n_i)$.

3.1.4 Customization for RMS

Rate Monotonic Scheduler (RMS) is an optimal static-(fixed-) priority scheduling policy using the task's period as the task's priority. RMS executes at any instant the ready task with the shortest period, i.e., the task with the shortest period has the highest priority. If more than one ready tasks have the same period, RMS randomly selects one for execution. Unlike EDF, however, there exist task sets with $U \leq 1$ that are not schedulable under RMS. There are no known polynomial time exact schedulability tests for RMS. Therefore, we use a recently proposed exact schedulability test [12] that is more efficient than the previously proposed tests.

Theorem 1 *Given a periodic task set $T_1, \ldots, T_N$ in increasing order of periods*

1. *T_i can be scheduled using RMS if and only if:*

$$L_i = \min_{t \in S_{i-1}(P_i)} \frac{\sum_{j=1}^{i} \left\lceil \frac{t}{P_j} \right\rceil C_j}{t} \leq 1$$

   where $S_i(t)$ is defined by the following recurrent expression:

$$\begin{cases} S_0(t) = \{t\} \\ S_i(t) = S_{i-1}\left(\left\lfloor \frac{t}{P_i} \right\rfloor P_i\right) \cup S_{i-1}(t) \end{cases}$$

2. *The entire task set is schedulable using RMS if and only if:*

$$\max_{i=1...N} L_i \leq 1$$

L_i is the utilization of $T_1 \dots T_i$ in the time interval [0,t]. Due to the double recurrence form of the definition, the worst-case cardinality of a generic $S_i(t)$ set is 2^i. In case two sets, $S_{i-1}\left(\left\lfloor \frac{t}{P_i} \right\rfloor P_i\right)$ and $S_{i-1}(t)$, overlap, the cardinality reduces.

The complexity of the schedulability test renders the design space exploration under the RMS policy more difficult compared to the EDF policy. Given a task set scheduled with RMS, it is possible to have two customized configurations p and p' such that $U(p) < U(p')$ but p' meets all the task deadlines whereas p does not. That is, we can no longer minimize only the total utilization without checking the feasibility of the schedule.

We propose a Branch and Bound search algorithm to select appropriate configuration for each task such that the entire task set is schedulable under Theorem 1 and the total utilization of the task set is minimized. Branch-and-bound deals with optimization problems over a search space that can be presented as the leaves of a search tree. The search is guaranteed to find the optimal solution, but its complexity in the worst case is as high as that of exhaustive search. The pseudo code is given as Algorithm 2.

Each level i in the branch-and-bound search tree corresponds to the choice of configuration for the task T_i. Thus, each node at level i corresponds to a partial solution with the configurations about the tasks T_1 up to T_i. Whenever we reach a leaf node of the search tree, we have a complete solution. The power of branch-and-bound algorithm comes from the effective pruning of the design space. We prune the design space under the following conditions.

First, during the traversal of the search tree, the minimum utilization achieved so far at any leaf node is kept as a bound $MinU$. At any non-leaf node m in the search tree, we compute a lower bound, $bound(m)$, on the minimum possible utilization at any leaf node in the subtree rooted at m. The lower bound is computed by summing up the utilization of the tasks that have been enhanced with custom instructions and the minimum utilization of

Algorithm 2: **Custom Instructions selection under RMS**

Input: Task Set T_1 **to** T_N with configurations; Area constraint: AREA

Output: Minimum utilization

```
begin
    U ← 0; optimalSoln ← ∅; A ← AREA; MinU ← Σ_{i=1}^{N} C_i/P_i;
    /* T_1 is highest priority task */;
    search (T_1, U, A, ∅);
    return MinU;
end

Function search (T_i, U, A, Soln)
for config_{i,j} (j ∈ 1 to n_i) in increasing order of execution time do
    if (area_{i,j} ≤ A) and T_i is schedulable with cycle_{i,j} then
        partialSoln ← Soln ∪ config_{i,j} ; A ← A − area_{i,j}; U ← U + cycle_{i,j}/P_i;
        if T_i = T_N then
            if U < MinU then
                MinU ← U;  optimalSoln ← partialSoln;
            end
            return;
        end
        if bound(partialSoln) < MinU then
            search (T_{i+1}, U, A, partialSoln);
        end
    end
end
```

the remaining tasks (which is the utilization when enhanced with the best possible custom instruction configuration). If $bound(m) \geq MinU$, then the search space corresponding to the subtree rooted at m can be pruned. Moreover, at any level, the configuration with the minimum execution time is considered first. This ensures greater possibility of obtaining a low utilization value $MinU$ quickly and thereby achieve effective pruning during the subsequent traversal.

Second, we select the appropriate configuration for each task in the order of decreasing priority, i.e., the highest priority task is considered first. Recall that RMS is a static priority preemptive schedule. A higher priority task can preempt a lower priority task but not the other way round. Suppose we have a partial solution where the configurations corresponding to the first $i-1$ high priority tasks (T_1 to T_{i-1}) have been chosen. Suppose further that the tasks T_1 to T_{i-1} all meet their respective deadlines with the chosen configurations. Any lower priority task, such as T_i, cannot preempt the higher priority tasks and hence the higher priority tasks will not miss their deadlines due to the introduction of T_i. Thus, the task traversal order ensures that at level i of the search tree we only need to check the schedulability of task T_i (i.e., whether $L_i \leq 1$ in Theorem 1). Moreover, if T_i fails to meet its deadline, we can prune the subtree rooted at the corresponding node.

Finally, if the area constraint is violated at any node, then the subtree rooted at the corresponding node is pruned.

3.2 Experimental Evaluation

We use 8 benchmarks from MiBench and one benchmark from Mediabench (`g721_encoder`) for our experiments. We create six task sets using these benchmarks; each task set consists of four benchmarks as shown in Table 3.1.

Task set	Benchmarks
1	crc32, sha, jpeg_decoder, blowfish
2	blowfish, adpcm_decoder, crc32, jpeg_encoder
3	adpcm_encoder, blowfish, jpeg_decoder, crc32
4	sha, susan, crc32, g721_encoder
5	adpcm_decoder, jpeg_decoder, crc32, blowfish
6	crc32, sha, blowfish, susan

Table 3.1: Composition of Task sets

We choose the Xtensa [37] processor platform from Tensilica for our experiments. Xtensa is a configurable processor core allowing application-specific instruction-set extensions. We use the XPRES compiler provided by Tensilica to generate the custom instructions from the C code corresponding to a task. Multiple custom instruction configurations are generated for each task based on the trade off between area and performance (see Figure 3.1). The maximum performance gain for the individual tasks vary from 3.5% to 27% with area budget ranging from 1K to 23K logic gates.

To set the periods for the tasks, we choose a total utilization for the task set (without any custom instructions) and then select the periods to achieve the corresponding utilization. Let C_i be the execution time of task T_i without using custom instructions. Then we set the period P_i for each task T_i as $P_i = \alpha_i \times C_i$ such that $\sum_{i=1}^{N} \frac{C_i}{P_i} = U$. We choose five different utilization factors U = 0.80, 1.00, 1.05, 1.08 and 1.10. A task set is EDF-schedulable if U = 0.8 or 1.0; but may or may not be RMS-schedulable. In this case, we are interested in finding out how much we can reduce the utilization by using custom instructions. For $U > 1.0$, a task set is not schedulable originally. It may become schedulable by using custom instructions. The greater the original utilization factor, the more difficult it is to

schedule the tasks using custom instructions.

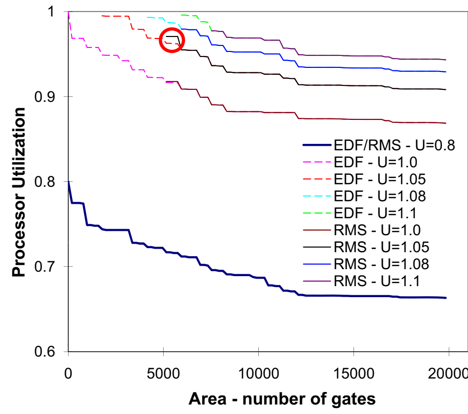
For each task set, we vary the hardware area constraint from 0 to Max_Area at an interval of $0.01 \times Max_Area$. The Max_Area for each task set is simply the summation of the maximum area requirements of the constituent tasks. A task set enhanced with custom instructions at Max_Area explores the limit of speedup achievable. The stricter the area constraint, the more difficult it is to schedule a task set and/or achieve lower utilization.

3.2.1 Performance

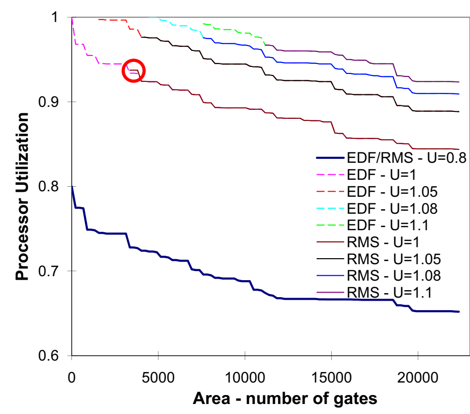
Figure 3.3 shows the utilization versus hardware area trade-off for the different task sets. For each task set and an original utilization factor, we apply both RMS and EDF scheduling policies. Our algorithms take less than 0.1 sec to return the solution for any task set and scheduling policy. The Y-axis shows the reduced utilizations. The utilization of a task set decreases with increasing hardware area because we can accommodate more custom instructions. Overall, we get up to 19% reduction in utilization. On an average, we get about 14% (13%) reduction in utilization at roughly 75% (50%) of Max_Area .

The reduced utilization values for a given area constraint are mostly identical for RMS and EDF. At original utilization $U = 0.8$, all our task sets are RMS-schedulable (they are, by definition, EDF schedulable). As adding custom instructions strictly improves the execution time of each task, the task sets remain scheduable for all possible configurations. We choose configurations for each task such that the total utilization is minimized and the task set is schedulable. Therefore, RMS and EDF scheduling policies select identical custom instruction configurations at a given area.

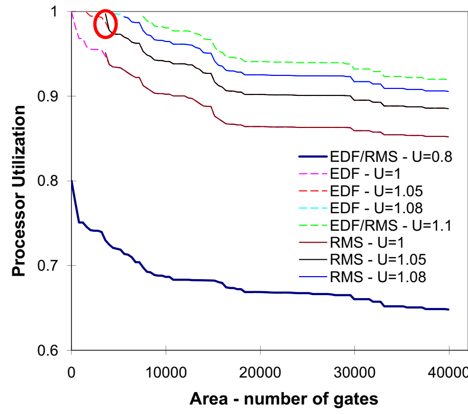
However, at original utilization $U = 1.0$, a task set without custom instructions enhancements may not be RMS-schedulable. Indeed, all our task sets are not schedulable under RMS policy at $U = 1.0$. Therefore, given a strict area budget, we fail to schedule the



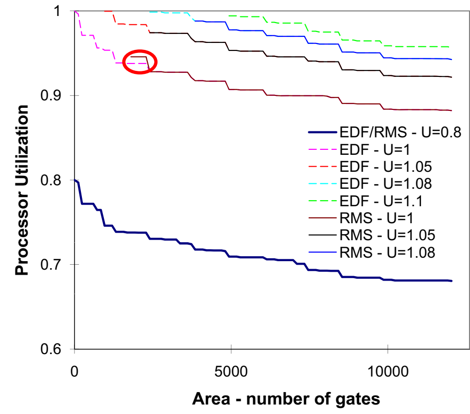
(a) Task Set 1



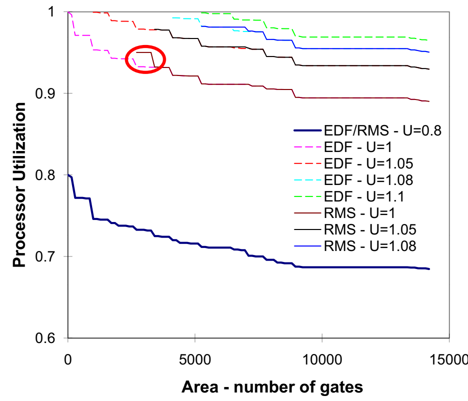
(b) Task Set 2



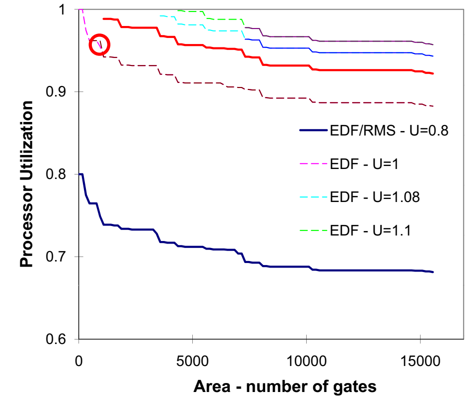
(c) Task Set 3



(d) Task Set 4

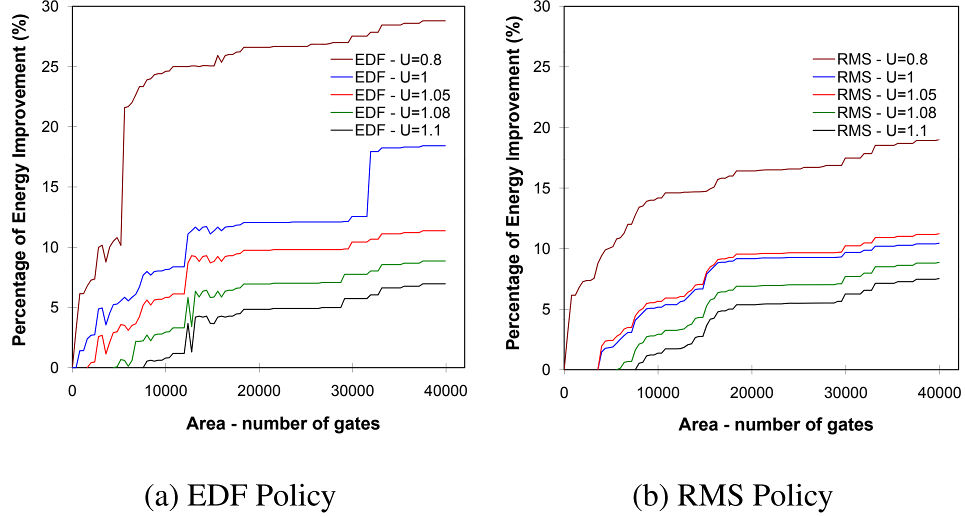


(e) Task Set 5



(f) Task Set 6

Figure 3.3: Utilization versus Area for different task sets under EDF and RMS scheduling policies.



(a) EDF Policy

(b) RMS Policy

Figure 3.4: Area versus Energy for Task Set 3 under EDF and RMS scheduling policies.

task sets under RMS policy even using custom instructions. As the area budget increases, a task set becomes RMS-schedulable and produces identical reduced utilization for both policies. In general, at any original utilization value greater than 1.0, a task set under the EDF policy becomes schedulable earlier compared to the RMS policy. The highlighted portions in the figure shows the design points where a task set is schedulable under both EDF and RMS policy; but produces different reduced utilization values.

3.2.2 Energy

A lower processor utilization opens up the opportunity to lower the operating frequency/voltage of the processor through voltage scaling technology. This may result in substantial energy savings. We employ the static voltage scaling algorithms for read-time systems proposed in [79]. Given a scheduling policy (RMS or EDF), the voltage scaling algorithm chooses the lowest operating voltage, frequency pair such that the task set still remains schedulable.

We first select the optimal customization for the task set under an area constraint. We apply static voltage scaling to obtain the lowest operating voltage/frequency corresponding to the original (no custom instructions) and the optimal configuration. We compare the

energy consumptions corresponding to these two configurations over the hyper-period (the least common multiple of the task periods) of the task set. At some original utilizations, the task set is not schedulable without customization. In these cases, we perform the comparison w.r.t the first schedulable solution. As Xtensa does not support voltage scaling, we use Transmeta TM5400 [94] processor to explore the savings in energy due to the reduction in processor utilization. We scale the frequency values from 300MHz (1.2 Volt) to 633MHz (1.6 Volt).

Figure 3.4 shows the relation between the hardware area and energy consumption under RMS and EDF scheduling policies. We can obtain up to 30% energy reduction. On an average, the energy reduction is 10% for RMS and 14% for EDF at 75% of Max_Area. Better energy savings for EDF is an artifact of the voltage scaling algorithm [79]. It can use aggressive voltage scaling for EDF policy due to its simpler schedulability test ($U \leq 1.0$). But for RMS, it uses a conservative schedulability condition that is sufficient but not necessary. In other words, it misses out certain opportunities and may select a higher operating frequency.

3.3 Summary

We explore instruction-set customization for multi-tasking real-time embedded systems. We propose algorithms to select inter-task optimal customizations under EDF and RMS scheduling policies. We achieve significant reduction in processor utilization and overall energy consumption through custom instructions.

Chapter 4

Evaluating design trade-offs for custom instructions

The past decade has seen a flurry of research activity on automated identification and selection of custom instructions for an application or an application domain. However, most of these design efforts have focused on single-objective optimization, for example, choosing an optimal set of custom instructions either in terms of performance or energy (such as our work in Chapter 3). As the performance/energy improvement offered by custom instructions come at the cost of silicon area, this optimization is typically constrained by a pre-defined silicon area. The designer, on other other hand, can benefit significantly if the automation tools expose all the conflicting trade-offs (e.g., performance versus area) instead of offering a point solution. It is then up to the designer to choose an appropriate trade-off point. More formally, we are interested in generating the *Pareto-optimal curve* in a multi-objective design space (e.g., performance and area) where (a) no point is better than any other point on the curve with respect to both objectives and (b) no improvement can be made in any objective without trading-off or worsening the other.

Unfortunately, it turns out that computing the Pareto-optimal curve for our design prob-

lem is computationally intractable. Therefore, state-of-the-art customization tool-chains (such as Tensilica's XPRES compiler [93]) adopt ad-hoc methods that simply compute the best performing design choices at arbitrary silicon area constraints. In this chapter, we propose a systematic methodology to explore the performance-area trade-offs in designing customizable processors. We present a polynomial-time approximation algorithm to compute this trade-off. Moreover, as the Pareto curve may potentially contain exponential number of design points, it is impossible to compute this entire set in polynomial time. Hence, our polynomial-time approximation algorithm, by default, has to approximate the (potentially exponential size) set of points on the Pareto curve with only a polynomial number of points. In a typical design cycle of customizable processors, the system designer inspects all the points in the Pareto curve and then selects one, or at most a few implementations. Hence, from a practical perspective, we feel it is more meaningful if the designer is presented with a reasonably few well-distinguishable trade-offs rather than an exponentially large number of solutions, many of which are very similar to each other. Our approximation algorithm is therefore not only attractive in terms of time-complexity, but also returns more meaningful solutions, in terms of the size of the solution set (including the spread/distribution of solutions in this set).

We explore this approximation solution to Pareto curve generation in the context of multi-tasking real-time embedded applications running on customizable processors as presented in Chapter 3. Typically, given a multi-tasking application to be implemented on a customizable processor, there are a large number of implementation possibilities with different subsets of custom instructions leading to different processor utilization versus area trade-offs. We would like to identify *all* schedulable implementations that expose the different possible performance trade-offs.

Formally, for any schedulable implementation, let (U,A) denote the corresponding utilization of the base processor and the hardware cost arising from the use of custom instructions. We are then interested in identifying all possible Pareto-optimal solutions $\{(U_1,A_1),$ $\ldots,$ $(U_n,A_n)\}$ that capture the different performance trade-offs [28]. Each (U_i,A_i) in this set has the property that there does not exist any schedulable implementation with a performance vector (U,A) such that $U \leq U_i$ and $A \leq A_i$, with at least one of the inequalities being strict. Further, let S be the set of performance vectors (i.e., (utilization, area) tuples) corresponding to all schedulable implementations. Let P be the set of performance vectors $(U_1,A_1),\ldots,(U_n,A_n)$ corresponding to all the Pareto-optimal solutions. Then for any $(U,A) \in S-P$ there exists a $(U_i,A_i) \in P$ such that $U_i \leq U$ and $A_i \leq A$, with at least one of these inequalities being strict (i.e., the set P contains *all* performance trade-offs). The vectors $(U,A) \in S-P$ are referred to as *dominated solutions*, since they are dominated by one or more Pareto-optimal solutions.

We present a polynomial-time approximation scheme for computing the *utilization-area* Pareto curve. Our proposed solution for computing this Pareto curve involves two distinct stages. First, in the *intra*-task stage, *each* individual task is analyzed. Given the library of possible custom instructions for the task, all possible custom instruction configurations are generated exposing the *workload-area* Pareto curve. In the *inter*-task stage, we consider *all* the tasks in the task-set and their *workload-area* configurations, to generate the processor *utilization-area* Pareto curve P for the overall task set. Our framework for approximately computing the trade-offs extends to both of the above stages.

The algorithmic techniques presented in this chapter have been motivated by [75]. The result that for *any* Pareto curve and *any* ε, there *exists* a polynomial-size ε-approximate Pareto curve is shown in [75]. However, for many problems, efficiently (i.e., in polynomial time) *computing* such approximate Pareto curves might not be possible. Our main technical

contribution is to show that such ε-approximate Pareto curves can be efficiently computed in the domain of custom instruction selection. An important consequence of this result is a formal basis for custom instruction selection. It shows that in the quest for efficiency, there is no need to resort to ad-hoc techniques – as currently adopted in state-of-the-art customization tool chains – for identifying best-performing custom instruction choices.

Here, it should be noted that in this chapter we have taken a classical approximation algorithms standpoint, where the goal is to provide *formal* guarantees on the quality of the results obtained. Our work differs from the existing large body of work on multiobjective optimization [28] that relies on heuristics and randomized search techniques such as evolutionary algorithms.

The organization of this chapter is as follows. We formally define approximation scheme in Section 4.1 with detailed algorithms in Section 4.2. Our framework is evaluated in Section 4.3.

4.1 Problem Statement

4.1.1 Task Model

In this chapter, we use a periodic, preemptive and independent task model (as in Chapter 3). Similarly, we are interested in the custom instruction selection for a task set $\tau = \{T_1, T_2, \ldots, T_m\}$ consisting of m hard real-time tasks, with the constraint that the task set is schedulable. Any task T_i can get triggered independently of other tasks in τ. Each task T_i generates a sequence of jobs; each job is characterized by the three parameters – the period (P_i) which is the time interval that must elapse before the successive job of the task T_i is triggered, the *deadline* (D_i) by which each job generated by T_i must complete since its release time, and *workload* (E_i) or the worst case execution requirement of any job generated

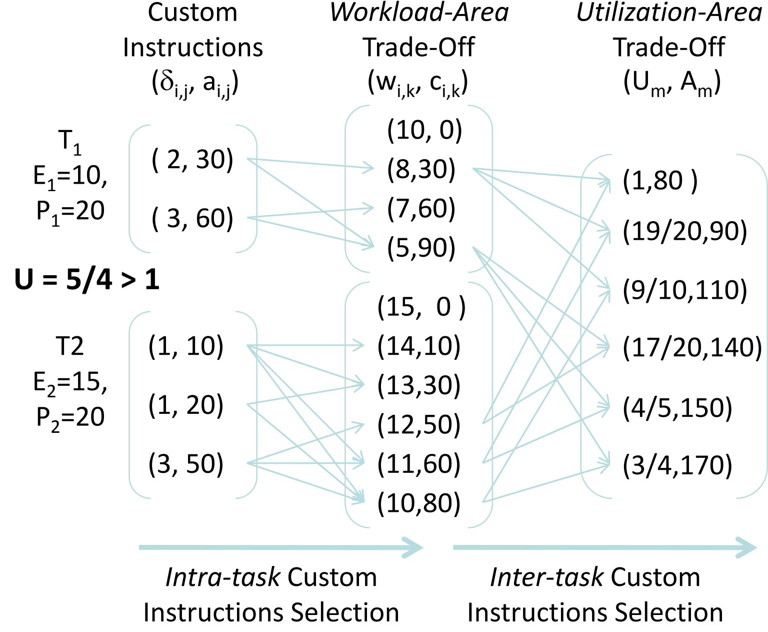


Figure 4.1: Motivating Example.

by T_i.

Throughout this chapter, we assume the underlying scheduling policy to be the earliest deadline first (EDF). Assuming that for all tasks T_i, $D_i \geq P_i$, the schedulability of the task set τ can be given by the following condition.

Theorem 1 *A set of periodic tasks* τ *is schedulable under EDF if and only if* $(U = \sum_{i=1}^{m} \frac{E_i}{P_i}) \leq 1$ *where* U *is the processor utilization due to* τ *[70].*

4.1.2 Intra-Task Custom Instructions Selection

We now state the intra-task custom instruction selection problem for a task T_i. For the task T_i, let there be n_i custom instruction candidates. Each of these n_i custom instruction is associated with a certain hardware area. Choosing the the jth custom instruction will lower the workload of the task T_i by $\delta_{i,j}$. Equivalently, the new workload will be $E_i - \delta_{i,j}$. Hence,

for each task T_i we have a set of choices $S_i = \{(\delta_{i,1}, a_{i,1}), \ldots, (\delta_{i,n_i}, a_{i,n_i})\}$, where $a_{i,j}$ is the hardware cost associated with the jth custom instruction. Our objective is to select a set of custom instructions that would minimize the workload on the base processor, as well as use the minimum amount of hardware area for custom instructions. In other words, our goal is to compute all *workload-area* trade-offs in the form of a Pareto curve $\{(w_{i,1}, c_{i,1}), \ldots, (w_{i,N_i}, c_{i,N_i})\}$, where $w_{i,j}$ is the workload of task T_i accelerated with a particular set of custom instructions and $c_{i,j}$ is the corresponding cost in terms of silicon area.

In Figure 4.1, we illustrate this problem for two different tasks T_1 and T_2. The task characteristics for T_1 are $\{E_1 = 10, P_1 = 20\}$. Note that the task T_1 has two entries in its library of custom instructions, and thus, $n_1 = 2$. Following our notation, $\delta_{1,1} = 2$ and $\delta_{1,2} = 3$, and the corresponding hardware areas are $a_{1,1} = 30$ and $a_{1,2} = 60$. The goal is to identify the *workload-area* Pareto curve for the task T_1. For example, in this case the Pareto curve consists of four solutions $\{(10,0), (8,30), (7,60), (5,90)\}$. Note that solution which has area equal to 0 does not use any custom instruction. Therefore, the application workload is not reduced and is the highest amongst all the solutions. On the other hand, the solution $(5,90)$ contains both the custom instructions and has the smallest workload with the largest area. The task characteristics for T_2 and the custom instruction candidates may be read in the same way as described above for T_1.

4.1.3 Inter-Task Custom Instructions Selection

Above, we discussed the intra-task custom instruction selection problem. For each task T_i, let there be N_i hardware implementation choices in the *workload-area* Pareto curve $\{(w_{i,1}, c_{i,1}), \ldots, (w_{i,N_i}, c_{i,N_i})\}$, that is computed by the intra-task custom instruction selection phase. Each of these N_i choices represent a custom instruction configuration for the task. A task may be chosen to run in one these configurations where it will incur certain a

hardware cost $c_{i,j}$ and would lower the execution time of the task on the processor from E_i to $w_{i,j}$.

However, in a typical real-time embedded system there is not one task but a set of tasks running on a processor clocked with a certain frequency. Thus, a designer is interested not in the performance or workload of a single task, but rater the *utilization* of the processor by the entire task-set. The goal in the inter-task custom instruction selection phase is to identify one custom instruction configuration for each task, which would minimize the overall base processor utilization and minimize the total hardware cost. Therefore, similar to intra-task customization, we generate a *Pareto curve* containing the *Pareto-optimal* set of *utilization-area* vectors $\{(U_1,A_1),\ldots,(U_n,A_n)\}$ with the trade-off between processor utilization U and hardware area A.

In Figure 4.1, we showed the intra-task custom instruction selection for two different tasks. For T_1 and T_2 we have 4 and 6 elements in the custom instruction configurations respectively. For example, for the task T_1, we have $\{(w_{1,1} = 10, c_{1,1} = 0), (w_{1,2} = 8, c_{1,2} = 30), (w_{1,3} = 7, c_{1,3} = 60), (w_{1,4} = 5, c_{1,4} = 90)\}$. The goal is to identify the *utilization-area* Pareto curve for the task set $\{T_1, T_2\}$. As shown in Figure 4.1, in this case the Pareto curve consists of six solutions $\{(1,80), (\frac{19}{20},90), \ldots, (\frac{3}{5},170)\}$. Thus, without using any custom instructions the task set $\{T_1, T_2\}$ is not schedulable, with $U = \frac{5}{4} > 1$. But the use of custom instructions speed up task executions, thereby lowering the load/utilization on the base processor. This yields six schedulable solutions with conflicting *utilization-area* trade-offs.

4.2 Evaluating Design Trade-offs

We shall now present our algorithms for efficiently computing the Pareto curves in the *intra-task* and the *inter-task* custom instruction selection phases that we described above. It may be noted that computing the exact Pareto curves in both these cases is computationally intractable. First, such Pareto curves would typically contain an exponential number of trade-off points (which obviously cannot be computed in polynomial time). Second, it may be shown using a reduction from the classical Knapsack problem, that the problem of computing even one point on the Pareto curve is NP-hard. Hence, our algorithms approximate both – the number of points on the Pareto curve, as well as the "coordinates" of these points on the curve.

4.2.1 Intra-Task Trade-offs

In what follows, we first present a pseudo-polynomial time dynamic programming algorithm (called *DP*) to compute the *exact* Pareto curve, which is then used to devise an approximation scheme. Below, we introduce the necessary notations and then present the *DP* recursion. Let the maximum cost associated with any custom instruction be C, i.e., $C = max_{(j=1,2,\ldots,n_i)} a_{i,j}$, where n_i is the number of custom instruction candidates for the task T_i and $a_{i,j}$ is the hardware cost associated with the jth custom instruction. Let $\omega_{k,j}$ be the minimum workload that might be achieved by considering only a subset of custom instructions of task T_i from $\{1,2,\ldots,k\}$ when the cost is exactly j. The algorithm *DP* first initializes the values for $k = 0$ as $\omega_{0,0} = E_i$, and $\omega_{0,j} = 0$ for $j = \{1,\ldots,n_iC\}$. Note that n_iC is an upper bound on the total hardware cost that might be incurred. After initialization, the DP computes the values of $\omega_{k,j}$ (for $k = 1$ to $k = n_i$) by the recursion defined below:

$$\omega_{k,j} \leftarrow \min\{\omega_{k-1,j}, \omega_{k-1,j-a_{i,k}} - \delta_{i,k}\} \tag{4.1}$$

That is, given an area j, we explore all possible configurations for T_i and choose the one that results in minimum workload for custom instructions $1, \ldots, k$.

After running *DP* to completion, we retain the undominated solutions from amongst the solutions in the final iteration ($k = n_i$) to obtain the exact *workload-area* Pareto curve, $\{(w_{i,1}, c_{i,1}), \ldots, (w_{i,N_i}, c_{i,N_i})\}$, where N_i is the size of the Pareto curve. This algorithm runs in pseudo-polynomial time $O(n_i^2 C)$, and will suffer from long running times. Hence, our goal is to *approximately* compute this curve in *polynomial* time.

Our approximation scheme takes an error parameter ε as input and returns an ε-*approximate Pareto curve* that we denote as ε-Pareto curve (or $\mathcal{P}_\varepsilon$). Given a Pareto curve $\mathcal{P} = \{(x_1, y_1), \ldots, (x_p, y_p)\}$, an ε-approximate Pareto curve $\mathcal{P}_\varepsilon$ is defined as *any* set $\mathcal{P}_\varepsilon = \{(x'_1, y'_1), \ldots, (x'_q, y'_q)\}$ such that for any $(x_i, y_i) \in \mathcal{P}$, there exists a $(x'_j, y'_j) \in \mathcal{P}_\varepsilon$ for which $x'_j \leq (1+\varepsilon)x_i$ and $y'_j \leq (1+\varepsilon)y_i$. In other words, corresponding to any point on the Pareto curve $\mathcal{P}$, there exists a point on $\mathcal{P}_\varepsilon$, each of whose coordinates are at most ε distance away from the corresponding coordinates of the point on $\mathcal{P}$.

Papadimitriou and Yannakakis [75] has shown that for any multi-objective optimization problem and any ε, there *exists* a polynomial-sized ε-approximate Pareto curve $\mathcal{P}_\varepsilon$. Further, [75] showed that a necessary and sufficient condition for computing such a $\mathcal{P}_\varepsilon$ in polynomial time is the existence of a polynomial-time algorithm for solving, what was referred to as the *GAP problem*. In what follows, we state the version of the GAP problem that arises in our setting and show that it can be solved in polynomial time. Finally, we outline our scheme to compute the approximate Pareto curves using the polynomial time *GAP* subroutine.

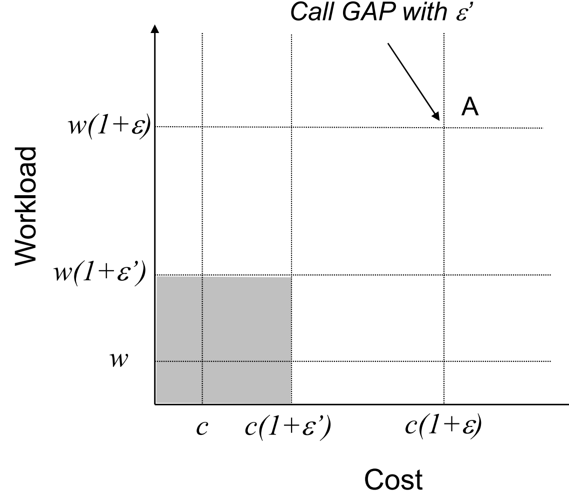


Figure 4.2: Solving the GAP problem for the corner point A will either return a dominating solution or declare that there is no solution in the shaded area.

4.2.1.1 The GAP Problem

For a two-dimensional multiobjective optimization problem, the GAP problem can be stated as follows: Given a vector $b = (b_1, b_2)$, either return a solution vector which dominates b, or report that there is no solution better than b by at least a factor of $1+\varepsilon$ in both dimensions. In our setting, the objective is to minimize the workload of a task T_i, $W(S) = E_i - \sum_{j=1}^{n_i} x_{i,j}\delta_{i,j}$ and the cost $C(S) = \sum_{j=1}^{n_i} x_{i,j}a_{i,j}$, where $x_{i,j}$ is a boolean variable which is true if the jth custom instruction is chosen for the solution S. Hence, the corresponding GAP problem can be stated as follows.

Problem Statement: Given a cost a, workload w and an $\varepsilon \geq 0$, either return a solution S such that $C(S) \leq c$ and $W(S) \leq w$, or else declare that there is no solution S such that $C(S) \leq \frac{a}{1+\varepsilon}$ and $W(S) \leq \frac{w}{1+\varepsilon}$.

Solving the GAP Problem: We now present a polynomial-time algorithm to solve this

GAP problem. It involves the following two steps:

Step 1: *Transforming Costs*

Let $r = \lceil \frac{n_i}{\varepsilon} \rceil$. Modify each cost $a_{i,j}$ of task T_i to $a'_{i,j}$ such that $a'_{i,j} = \lceil \frac{a_{i,j} r}{a} \rceil$. Now, consider the problem of determining whether there exists a solution with the modified costs such that $A'(S) \leq r$. Let us call this problem GAP$'$. We shall show that solving GAP is equivalent to solving GAP$'$. Towards this, we enumerate two properties below.

(a) If a solution with the transformed costs satisfies $A'(S) \leq r$, then $A(S) \leq a$.

Proof of property (a):

$$\sum a'_{i,j} x_{i,j} = \sum \left\lceil \frac{a_{i,j} x_{i,j} r}{a} \right\rceil \geq \frac{r}{a} \sum a_{i,j} x_{i,j}$$

Hence,

$$A'(S) \leq r \Rightarrow \frac{r}{a} \sum a_{i,j} x_{i,j} \leq r \Rightarrow A(S) \leq a$$

(b) If a solution satisfies $A(S) \leq \frac{a}{1+\varepsilon}$, then $A'(S) \leq r$.

Proof of property (b):

$$A(S) \leq \frac{a}{1+\varepsilon} \Rightarrow \sum a_{i,j} x_{i,j} \leq \frac{a}{1+\varepsilon} \Rightarrow \sum \frac{a_{i,j} x_{i,j}}{a} \frac{n_i}{\varepsilon} \leq \frac{n_i}{\varepsilon(1+\varepsilon)}$$

$$\Rightarrow \sum \left\lceil \frac{a_{i,j} x_{i,j} r}{a} \right\rceil \leq \left\lceil \frac{n_i}{\varepsilon(1+\varepsilon)} \right\rceil \Rightarrow A'(S) \leq \left\lceil \frac{n_i}{\varepsilon} \right\rceil = r \Rightarrow A'(S) \leq r$$

From property (a), we know that if this problem returns an affirmative answer then the GAP problem would also return a dominating solution. On the other hand, if GAP$'$ returns a negative answer then property (b) leads to the conclusion that there is no solution with cost $\leq a/(1+\varepsilon)$. Hence, from the above properties we can infer that solving GAP$'$ is equivalent to solving the original GAP problem.

Step 2: *Solving GAP′*

We present a dynamic programming algorithm to solve the GAP′ problem. This algorithm can be constructed with the following adjustments to Algorithm *DP*.

1. Run Algorithm *DP* with the modified costs $a'_{i,j}$.

2. Instead of iterating over all the cost values up to $n_i C$, iterate only up to a cost value of r, where $r = \lceil \frac{n_i}{\varepsilon} \rceil$.

3. Finally, if the minimum value in the final array computed by Algorithm DP is such that it is $\leq w$, then return the solution otherwise declare that there is no solution.

Computing each row of the table built by this dynamic programming algorithm requires $O(r)$ running time. Hence, this algorithm runs in time $O(n_i^2/\varepsilon)$.

The above polynomial time subroutine for solving the GAP problem proves the existence of an *fully polynomial-time approximation scheme* (FPTAS) for computing the approximate workload-area Pareto curve $\mathcal{P}_\varepsilon$ which is polynomial in the input size and in $1/\varepsilon$. This is because the following FPTAS can be devised using the algorithm for solving GAP. First, geometrically partition the objective space along all dimensions with a ratio $1+\varepsilon'$, where $\varepsilon' = (1+\varepsilon)^{1/2} - 1$. For each corner point of this grid, call the GAP routine (i.e. the algorithm for solving GAP) with the parameter ε', and keep all the undominated solutions (see Figure 4.2 for an illustration of this procedure). This implies that for each rectangle which contains a solution in the exact Pareto curve, there will also be a solution within the same rectangle which belongs to $\mathcal{P}_\varepsilon$. The *distance* between these two solutions can be bounded using the dimensions of the rectangle. Hence, for every solution s in the Pareto curve, there exists a solution q in $\mathcal{P}_\varepsilon$ such that $\frac{q}{(1+\varepsilon)} \leq s$. Moreover, because the number of rectangles is polynomially bounded, it follows that the number of points in $\mathcal{P}_\varepsilon$ will also be a polynomial.

Algorithm 3: Approximating the Pareto curve

1. Partition the range of costs from 1 to $n_i C$ geometrically with a ratio $1+\varepsilon' = (1+\varepsilon)^{1/2}$, thus dividing the cost space into $O(log_{1+\varepsilon} n_i C)$ coordinates.

2. For each coordinate b, call *Algorithm DP* with transformed costs $a'_{i,j} = \lceil \frac{a_{i,j} r}{b} \rceil$, where $r = \lceil \frac{n_i}{\varepsilon'} \rceil$.

3. For each run of Step 2, find the solution with the minimum utilization.

4. Retain all the undominated solutions from the solutions found in Step 3. This will represent a ε-Pareto curve.

Algorithm 3 summarizes the above steps to compute the ε-approximate *workload-area* Pareto curve in some more detail. Note, that in step 1 of Algorithm 3 we partition only the area space (and not both workload and area space). This is because if a point (w, c) dominates the corner (w_1, c_1) and $w_1 < w_2$, then (w, c) definitely dominates (w_2, c_2) . In steps 2 and 3, we scale the costs, run Algorithm *DP* for every co-ordinate in the partitioned cost space and retain the minimum workload at each co-ordinate. The runtime complexity of this algorithm is $O(\frac{n_i^2}{\varepsilon} log_{1+\varepsilon} n_i C)$.

4.2.2 Inter-Task Trade-offs

The existence of an inter-task approximation scheme to compute the *utilization-area* Pareto curve may be argued in the same fashion as for the intra-task approximation scheme described above. This scheme takes the set of pareto-optimal solutions $\mathcal{P}_i$ for each task T_i as input (as shown in the previous section), and generates the set of global design trade-offs $\bar{\mathcal{P}}$ for the entire task set. Each global design configuration $S \in \bar{\mathcal{P}}$ contains contains exactly one solution from each $\mathcal{P}_i$ (for each task T_i). If a brute-force approach is used to examine all possible global design configurations, then $|\mathcal{P}_1| \times \ldots \times |\mathcal{P}_m|$ solutions will have to be

examined, where $|\mathcal{P}_i|$ denotes the number of solutions in the set $\mathcal{P}_i$. Hence, the number of solutions grow exponentially with the number of tasks m in the task set, and moreover, not all of these solutions would be optimal (i.e., they would be *dominated* by some Pareto-optimal solution). Our goal is to instead efficiently (but approximately) compute just the Pareto-optimal global design configurations.

Broadly, the procedure follows the same steps as described in Algorithm 3. However, note that the main difference is the core dynamic programming recursion that is invoked in Step 2 (Algorithm 3), which we present below. Let $U_{i,j}$ be the minimum utilization that might be achieved by considering only a subset of tasks from $\{1,2,\ldots,i\}$ when the cost is exactly j. Then, $U_{i,j}$ is defined recursively as below, where $\{w_{i,k}, c_{i,k}\} \in \mathcal{P}_i$, (workload-area Pareto curve), and E_i and P_i denote the *execution* time and the *minimum separation* of the task T_i.

$$U_{i,j} \leftarrow \min \left\{ \begin{array}{l} U_{i-1,j}, U_{i-1,j-c_{i,1}} - (E_i - w_{i,1})/P_i \\ \vdots \\ U_{i-1,j}, U_{i-1,j-c_{i,N_i}} - (E_i - w_{i,N_i})/P_i \end{array} \right\} \tag{4.2}$$

We summarize the overall two-stage approximation scheme in Figure 4.3. There are two distinct stages: (i) the intra-task stage to compute the *workload-area* Pareto curve for each task, and (ii) the *inter-task* stage which generates the *utilization-area* Pareto curve for the entire task set. The scheme for approximating the Pareto curve follows the three main steps shown on the right hand of Figure 4.3. Note that at each stage, the approximation scheme takes as an input an error parameter ε (chosen by the designer) and returns an ε-*approximate* Pareto curve. These parameters might be different for the two stages.

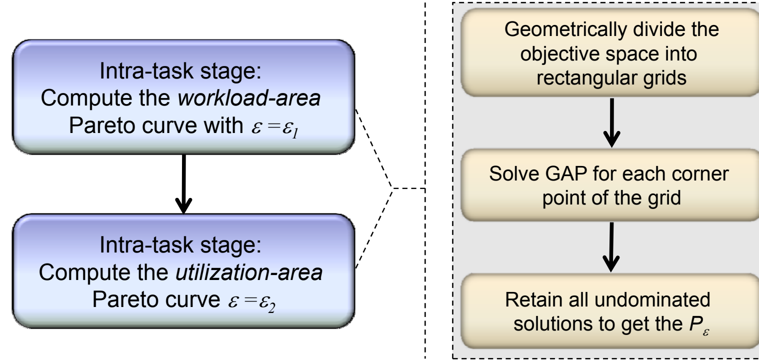


Figure 4.3: The overall two-stage approximation scheme.

4.3 Experimental Evaluation

In this section we report some of the experimental results obtained by running our approximation algorithm on a set of well-known benchmarks. We compare the running times of the optimal algorithm against our approximation scheme, and also illustrate the difference in the sizes of P_ε (the approximate Pareto curves) and the exact Pareto curve.

Experimental setup: We used five WCET benchmarks [87] *(compress, jfdctint, ndes, edn, adpcm)*, two benchmarks *(aes, sha)* from MiBench [39], three benchmarks *(g721 encoder, djpeg, cjpeg)* from MediaBench [65] and one *(ispell)* Trimaran benchmark [19] for our experiments.

Given the C code of an application, we used the Trimaran compilation framework to generate optimized intermediate code, as well as profiling information such as instruction execution frequencies. We then constructed the data flow graph for each basic block and enumerated all possible custom instructions using well-known algorithms from the literature (e.g., those proposed in [81, 101]). The workload was measured in terms of Multiply-Accumulate (MAC) operation cycles and hardware area in terms of the number of adders.

Task Set	Benchmarks
1	cjpeg, adpcm, aes, compress, rijndael ispell
2	djpeg, g721decode, cjpeg, ispell, adpcm jfdctint, aes
3	cjpeg, ispell, edn, sha, g721decode, djpeg compress, ndes
4	adpcm, rijndael, cjpeg, ispell, sha ndes, djpeg, compress, edn
5	aes, djpeg, g721decode, rijndael, jfdctint cjpeg, edn, ispell, sha, ndes

Table 4.1: Composition of the task sets.

Further, we assumed a single-issue in-order base processor core with a perfect cache.

We created five task sets (see Table 4.1) with the number of tasks in each set varying from 6 - 10, using the benchmarks described above. To set the minimum inter-triggering period P_i for the tasks, we chose a total utilization for the task set (without any custom instructions) and then selected P_i to achieve the corresponding utilization. We also chose five different utilization factors U = 0.80, 1.00, 1.05, 1.08 and 1.10. When U = 0.8 or 1.0, a task set is schedulable without using any custom instructions. In these cases, we were interested in finding out by how much can we reduce the utilization through custom instructions, and what are the hardware trade-offs. For $U > 1.0$, a task set is not schedulable on its own. Here, the goal was to find schedulable solutions by using custom instructions, as well as expose the performance trade-offs (i.e., the corresponding silicon area required). All CPU times reported below were measured on a machine with Windows XP, running on a 3.0 GHz Pentium 4 CPU with 1 GB RAM. All the implementations were done in C++.

Task Sets:	1	2	3	4	5
$\varepsilon = 0.21$	643	1075	1037	990	729
$\varepsilon = 0.44$	3248	5918	5712	5457	3933
$\varepsilon = 0.69$	7106	14587	14389	13922	10208
$\varepsilon = 3.0$	29615	72525	89285	69054	77508

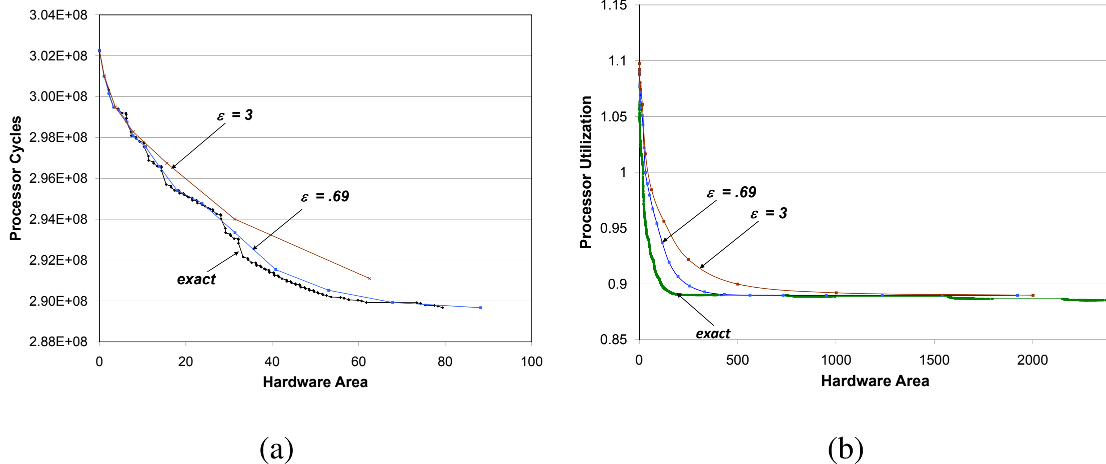
Table 4.2: Speedup obtained from our approximation scheme for the task sets 1 – 5.

Running times: Table 4.2 shows the running time speedups resulting from our approximation scheme, compared to computing the exact Pareto curve for three different values of ε, for each of the five task sets. Computing the exact Pareto curve for task sets $1 - 5$, required 139.78 sec, 514.20 sec, 622.32 sec, 747.17 sec, and 711.52 sec respectively. It may be noted that even for small values of ε (e.g., ε = 0.44) our approximation algorithm runs about three orders of magnitude faster than the exact algorithm. For larger values of ε (e.g., ε = 3), the speedups are even more significant (note that ε need not be ≤ 1). The reason behind choosing the values 0.21, 0.44, and 0.69 for ε is as follows. Our approximation algorithm involves the computation of $(1+\varepsilon)^{1/2}$. This value might turn out to be an irrational number if ε is not carefully chosen. Hence, to avoid any possible rounding-off errors in our implementation, the above values were chosen for ε.

Pareto curve size:

The *workload-area* Pareto curve (the output of intra-task stage) and the *utilization-area* Pareto curve (the output of intra-task stage) typically contain an exponential number of points. The approximation algorithm generates a polynomial-sized approximate Pareto curve $\mathcal{P}_\varepsilon$. We now compare the sizes of the exact Pareto curve and $\mathcal{P}_\varepsilon$.

For the intra-task results, Figure 4.4(a) shows the exact Pareto curve and the P_ε gen-



(a) (b)

Figure 4.4: The exact and approximate Pareto curves for $\varepsilon = 0.69$, 3. (a) *workload-area* Pareto curve for *g721decode*. (b) *utilization-area* Pareto curve for task set 1

erated by our algorithm for the *g721decode* benchmark. For the inter-task case, we show the results for task set 1 in Figure 4.4(b). For clarity, we have only plotted $\mathcal{P}_\varepsilon$ for $\varepsilon = 0.69$ and 3. Note that (i) the number of points in P_ε decrease as ε increases, and (ii) the gap between the exact and approximate curves widen with larger values of ε, implying that the relative error indeed increases. We would like to report that even for small values of ε (e.g., $\varepsilon = 0.21$), P_ε contains almost 97% fewer points compared to the exact Pareto curve. Similar trends were seen for all the other benchmarks and task sets.

Benefits of approximation: Although the running times associated with constructing the exact Pareto curve might seem to be small (i.e., 10-12 mins), in an interactive design process where the designer repeatedly makes design changes and generates new Pareto curves, this might hamper designer productivity. A tool which generates these curves faster (e.g., using our proposed approximation algorithms) would be more usable. Secondly, exact Pareto curves return too many (similar) design trade-offs, all of which might not be usable. Approximate Pareto curves return less, well spread out trade-offs, which might be more

manageable for the designer.

4.4 Summary

In this chapter, we proposed a framework for evaluating trade-offs in custom instruction selection for instruction set customizable processors. This framework consists of two stages – in the first custom instruction configurations representing different trade-offs are chosen for each task, and in the second, different configurations from each task are chosen to derive system-level trade-offs.

Chapter 5

Iterative custom instruction generation

The customization process so far has largely remained decoupled from the system-level design flow (such as our work in Chapter 3 and 4). Let us consider a canonical embedded application consisting of a set of tasks mapped to a single customizable processor. A typical design flow to accelerate this application with customization takes a bottom-up approach. The designer first generates a set of custom instructions for each individual task with the help of automated tool chains. This is followed by a design space exploration to select a subset of custom instructions for each task such that the overall performance and/or energy objectives of the system are satisfied [91, 44]. Given the complexity of custom instruction generation process, it is obvious that this bottom-up approach cannot scale beyond a small number of tasks in the system.

In this chapter, we introduce *iterative* custom instruction generation scheme that provides a close coupling between the customization process and the system-level design flow to get around the scalability problem. Our method is based on the observation that the bottom-up approach spends enormous effort in generating custom instructions for *all the tasks*. However, many of these tasks do not contribute to the system performance bottleneck and indeed the custom instructions generated for such tasks are effectively ignored in

the global selection phase. Instead, we advocate a top-down approach where the system level performance requirements guide the customization process to zoom into the critical tasks and the critical paths within such tasks. Our approach is iterative in the sense that we generate custom instructions in an on-demand basis. In other words, the iterative approach can quickly come up with a first-cut solution that can be iteratively refined (through inclusion of additional custom instructions) at the request of the designer.

It is relatively easy to identify critical tasks and the critical paths within such tasks in an embedded system. The main challenge for our iterative approach is quick generation of a set of quality custom instructions for the critical region. As customization process has traditionally been used in an off-line fashion, most techniques available in the literature are not suitable for our purpose. Custom instruction generation algorithms typically expose computational patterns at all possible granularity levels. In particular, these algorithms are computationally expensive as they generate many small patterns (consisting of few native operations) with the hope that such patterns will recur multiple times within the scope of the application. Instead, our goal is to quickly identify large patterns that can give us the required performance boost. Therefore, we design an algorithm that can efficiently partition the dataflow graph corresponding to the critical region into few large custom instructions. Our algorithm is named MLGP as it is inspired by multilevel graph partitioning algorithms [56] and satisfies the constraint of generating high-quality custom instructions with minimal effort.

Iterative custom instruction generation is quite a general concept and can be employed in different system-level design problems. In this study, we have selected customization for multi-tasking embedded systems with real-time constraints as a concrete problem to illustrate our approach. This is the same set up as in Chapter 3 and 4. Given a set of independent, preemptive, and periodic tasks, the goal is to come up with a set of custom

instructions such that all the tasks in the system can meet their deadlines. Experimental evaluation with realistic benchmarks show that our iterative scheme can meet this goal (if feasible) within few seconds.

Interestingly, it turns out that the MLGP algorithm, in isolation, is also quite competitive compared to perviously proposed custom instruction generation approaches. As mentioned earlier, most previous techniques are computationally expensive as they identify an optimal set of custom instructions for the entire application in one go. We compare our algorithm against state-of-the-art heuristic proposed in the literature for faster custom instruction generation, called *Iterative Selection* (IS) algorithm [81]. IS algorithm identifies one coarse-grained custom instruction per iteration and progressively improves the solution by adding more custom instructions. A comparison with IS for individual tasks reveals that (a) *MLGP*, in general, produces superior quality solutions much faster, and (b) *MLGP* exposes a wider range of design tradeoff in terms of hardware area and application speedup.

Our Contributions In summary, this work advances the state-of-the art in automated processor customization with the following concrete contributions.

- We design a fast and efficient custom instruction generation algorithm inspired by multi-level graph partitioning (MLGP) algorithm. Our algorithm is capable of generating high-quality custom instructions with substantially reduced analysis time.

- We develop an iterative customization framework that exploits the MLGP algorithm and creates an end-to-end solution that closely couples custom instruction generation with system-level design process. In other words, our framework has a global view of the system-level performance bottlenecks and hence can zoom into the critical regions to quickly alleviate the performance problem with processor customization.

We present our iterative custom instruction generation approach with a concrete design problem in Section 5.1. Detailed custom instruction generation algorithm is described in Section 5.2. In Section 5.3, experimental results are shown.

5.1 Iterative Approach

We illustrate the iterative custom instruction generation approach with the following concrete design problem: Implementation of a multi-tasking embedded application with real-time constraints on a customizable processor. In particular, we consider system consisting of a set of N independent, periodic and preemptive tasks. This task model is similar to the ones in Chapter 3 and 4. Recall that each task T_i has a period P_i and a worst case execution time (also called workload) C_i. Each task instance will be released periodically at the beginning of every P_i time units and must complete its execution by the end of the period. Therefore, the deadline of task T_i is equal to P_i. In this work, we choose earliest deadline first (EDF) as our scheduling policy; but our methodology is equally applicable to other scheduling policies. A task set is schedulable under EDF policy if the total processor utilization (U) is less than or equal to 1. That is, $U = \sum_{i=1}^{N} U_i = \sum_{i=1}^{N} \frac{C_i}{P_i} \leq 1$

Without loss of generality, let us assume that the task set is not schedulable, i.e., $U > 1$. Under this scenario, processor customization can provide the requisite performance boost to help the tasks meet their deadlines. The objective of our iterative approach in this context is to quickly come up with a set of custom instructions $\mathcal{CI}$ so as to lower the processor utilization below 1. The set $\mathcal{CI}$ is returned to the designer as the first working solution. If the designer so desires, our scheme will successively introduce additional custom instructions to $\mathcal{CI}$ so as to further lower the utilization. In case it is infeasible to reduce utilization below 1, the iterative approach brings down the utilization as much as possible.

Algorithm 4: **Iterative Approach**

Input: Task set $\mathcal{T}$ with N periodic Tasks

Result: Set of custom instructions $\mathcal{CI}$ and utilization U

1 compute WCET C_i for each task T_i in the task set $\mathcal{T}$;

2 $U = \sum_{i=1}^{N} \frac{C_i}{P_i}$; $U_{target} = 1$;

3 **while** *true* **do**

4   **if** $U \leq U_{target}$ **then** designer inputs new U_{target} or **return** $\{\mathcal{CI}, U\}$;

5   select the task T_i in $\mathcal{T}$ with the maximum utilization;

6     $\Delta = (U - U_{target}) \times P_i$;

7   select a subsequence of basic blocks S on the critical (WCET) path of T_i;

8     $gain$ = custom_instruction_generation($S, \Delta, \mathcal{CI}$);

9   **if** $gain > 0$ **then**

10   compute new WCET C_i' for Task T_i;

11         $U = U - \frac{C_i - C_i'}{P_i}$;

    else

12         $\mathcal{T} = \mathcal{T} - \{T_i\}$;

13   **if** $|\mathcal{T}| = 0$ **then return** $\{\mathcal{CI}, U\}$;

Algorithm 4 shows our iterative scheme applied for improving the schedulability of a set of real-time tasks with custom instructions. If the current utilization U has already satisfied the target utilization U_{target} (i.e., $U \leq U_{target}$), then the designer is given the option of either suggesting a new target utilization or accept the current set of custom instructions $\mathcal{CI}$ (line 4). The default target utilization is 1 (line 2), which is required to schedule the task set under EDF policy. We now select the task T_i with maximum utilization to enhance it with custom instructions (line 5). T_i has the maximum potential to reduce U through customization. The WCET of T_i has to be reduced at least by $\Delta = (U - U_{target}) \times P_i$ to meet the utilization target.

For the selected task, we first identify the WCET path through its program code by employing Timing Schema approach [76]. We sort the basic blocks along the WCET path in the descending order based on their weight (the execution time of a basic block over the program WCET). Then, we only select a subsequence of basic blocks S with high weight for custom instruction generation (line 7). Typically, S has a total weight that execeeds 90% of the program WCET. With S, we invoke custom instruction generation to enhance the current task (line 8). The goal of the custom instruction generation routine is to reduce the execution time of the basic blocks sequence S by amount Δ. If we could achieve speed up by customization for task T_i, its WCET and the system utilization are updated (lines 10-11).

If further performance gain is not achievable from the current task T_i, it is excluded from the task set. If we fail to meet the utilization target even after exploring all the tasks, then we simply return the set of custom instructions selected so far.

5.2 Custom Instruction Generation

Let us now proceed to describe our custom instruction generation algorithm — the key component of our overall iterative scheme. The input to this algorithm are: (i) a subsequence of basic blocks S along the critical (WCET) path of the program corresponding to the critical task T_i as described in Algorithm 4 (line 8), (ii) the amount Δ by which we need to reduce the execution time of S through customization, and (iii) the set the custom instructions already created $\mathcal{CI}$. The last input is required to identify isomorphic custom instructions generated during different iterations and take advantage of hardware area sharing.

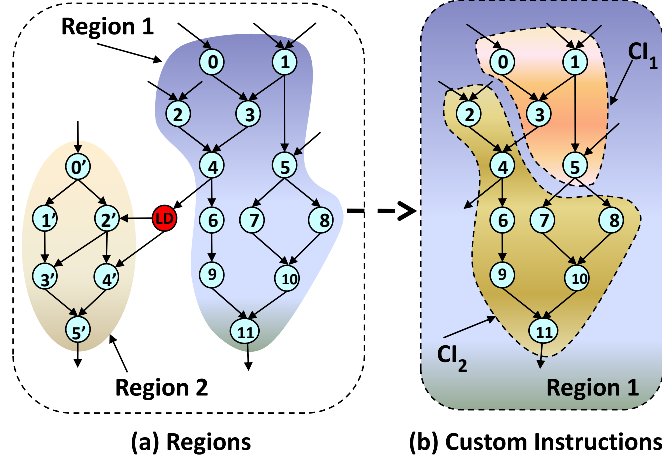


Figure 5.1: Regions and Custom Instructions.

5.2.1 Definitions

A Data Flow Graph (DFG) $G(V,E)$ represents the computation flow of data within a basic block. The nodes V represent the operations and the edges E represent the dependencies among the operations. $G(V,E)$ is always a directed acyclic graph (DAG). The architectural constraints may not allow some types of operations (e.g., memory access and control transfer operations) to be included as part of a custom instruction. These operations are considered as invalid nodes and the rest of operations are valid ones.

We let the invalid nodes partition the DFG into multiple regions. Given a DFG $G(V,E)$, we define a region $R(V',E')$ as a maximal subgraph of G such that (1) V' contains only valid nodes, (2) there exists an undirected path between any pair of nodes in V', and (3) there does not exist any edge between a node in V' and a valid node in $(V-V')$. Invalid nodes do not belong to any region. Figure 5.1(a) shows a DFG divided into two regions by a memory load operation (assuming memory load is an invalid operation).

A custom instruction CI is a subgraph that belongs to a region within a DFG. Let $IN(CI)$ and $OUT(CI)$ be the number of input and output operands of CI, respectively. Also, for any custom instruction, let N_{in} and N_{out} be the maximum number of input and

output operands allowed, respectively. This constraint arises due to the limited number of register file ports available on a processor. Any legal custom instruction CI must satisfy the constraints $IN(CI) \leq N_{in}$ and $OUT(CI) \leq N_{out}$. Moreover, a custom instruction must be a convex subgraph as non-convex subgraphs cannot be executed atomically. CI is convex if there exists no path in the DFG from a node $m \in CI$ to another node $n \in CI$, which contains a node $p \notin CI$. For example, $\{5,7,8,10\}$ is a convex subgraph but $\{5,7,10\}$ is a non-convex subgraph in Figure 5.1(a).

5.2.2 Region Selection

Given a subsequence of basic blocks S along the critical path of a task, we explore the basic blocks in S in descending order of weight. That is, the basic block with the highest weight is selected for custom instruction generation first. Recall that the weight of a basic block is defined by its contribution (in terms of execution time) to the critical path. We partition the selected basic block into multiple regions. These regions are again sorted in descending order based on their weights. The weight of a region is defined by the number of operations contained within that region. Then, we select the region with highest weight for generating custom instructions. Intuitively, we are selecting the region that has the maximum potential to reduce the WCET by Δ amount. Our problem now boils down to generating a set of custom instructions from the selected region so as to reduce the execution time as much as possible. We describe a solution to this problem in the next subsection.

If the custom instructions generated for the selected region can reduce the execution time by at least Δ, then we can simply return those custom instructions along with the *gain* to the higher-level routine (line 8 in Algorithm 4). Otherwise, we continue custom instruction generation for the next highest weight region of the current basic block or the next highest weight basic block if current basic block has been fully explored.

5.2.3 MLGP Algorithm

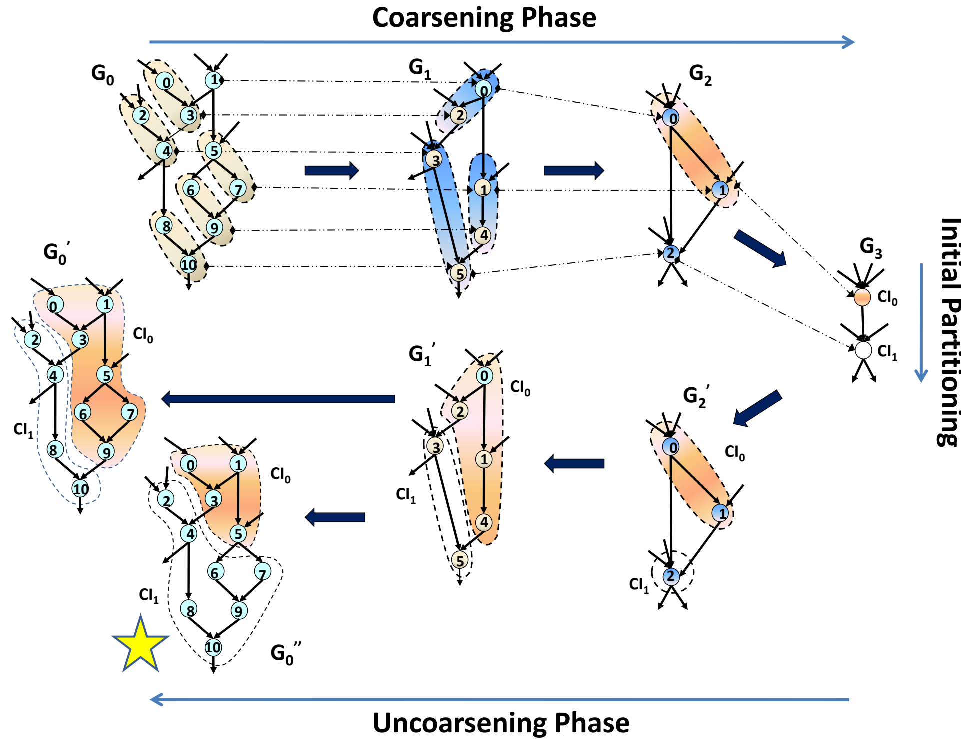


Figure 5.2: Illustration of Multi-Level Graph Partitioning. The dashed lines show the projection of a vertex from a coarser graph to a finer graph.

Overview Given a critical region, the goal of our custom instruction generation algorithm is to quickly reduce the execution time of the region as much as possible. As analysis time is a major concern for our iterative scheme, we cannot spend substantial effort required in exploring all possible custom instructions corresponding to the region and then selecting the optimal ones. Thus, our objective is to generate a set of coarse-grained but legal custom instructions from the region. We observe that our goal can be achieved by *partitioning* the data flow graph (DFG) corresponding to the region into one or more partitions. Each partition should satisfy the number of input and output operands constraints as well as the convexity constraint. That is, each partition can be treated as a custom instruction.

Figure 5.1(b) shows an example where a region has been partitioned into two legal custom instructions.

Graph partitioning is a well studied problem in the algorithm research community. In particular, the closest problem to ours is the k-way graph partitioning problem where the vertices of a graph are partitioned into k roughly equal partitions such that the number of edges connecting vertices in different partitions is minimized. However, our problem differs from k-way graph partitioning problem in several important aspects. First of all, we do not have any basis to choose a particular value of k — any value of k is fine with us as long the corresponding partition maximizes the performance gain. Second, the partitions in k-way graph partitioning problem are *not* constrained by input, output, convexity constraints. Third, instead of generating equal-sized partitions and minimizing edge-cut, our objective is to maximize the performance speedup. Finally, we are dealing with a directed graph and not an undirected graph as expected by k-way partitioning problem.

Nevertheless, it turns out that the basic structure used by multilevel recursive bisection algorithms employed to solve k-way graph partitioning problem can be quite effective in our context. Specifically, our custom instruction generation algorithm is inspired by a recently proposed multi-level algorithm due to Karypis and Kumar [56]. The basic structure of the algorithm is as follows. The graph G is first coarsened down to a small number of vertices (coarsening phase), the coarsest graph is partitioned into k parts (partitioning phase), and then this partitioning is projected back towards the original graph by periodically refining the k-partitioning (un-coarsening phase). The k-partitioning is refined on finer graphs as finer graphs have more degrees of freedom and hence provide more opportunity to improve the partitioning.

We adapt this multi-level paradigm to partition a directed graph into a small number of legal partitions so as to maximize the performance gain. We call our algorithm Multi-

Level Graph Partitioning (MLGP). To avoid artificially binding k to a particular value, we eliminate the k-partitioning phase from the MLGP algorithm. Instead, we simply set the number of partitions as the number of vertices in the coarsest graph. Figure 5.2 shows an illustration of the MLGP algorithm applied on the DFG of a region. The original graph has 11 vertices, which are coarsened into 2 vertices or 2 partitions. These partitions are successively refined in the uncoarsening phase to generate the final custom instructions.

Coarsening phase During the coarsening phase, a sequence of smaller graphs $G_i = (V_i, E_i)$ are constructed from the original directed graph $G_0 = G = (V, E)$ such that $|V_{i+1}| < |V_i|$. A vertex $v' \in V_{i+1}$ in a coarse graph G_{i+1} is formed by either combining two vertices $v, u \in V_i$ of finer graph G_i or by simply setting it to vertex $v \in V_i$ of G_i. In addition, a directed edge is built between two coarse vertices v and u in coarse graph G_{i+1} if there exists directed edge(s) between their constituent vertices in graph G_i.

Each vertex v' in a coarse graph is a subgraph of G_0 when projected from the constituent vertices of v' in the finer graph. A coarse vertex can potentially become a candidate for custom instruction. Therefore, when combining two vertices v and u to form v', we have to ensure that the subgraph corresponding to v' projected into the original graph G_0 satisfies input, output, and convexity constraints. Let $IN(v')$ and $OUT(v')$ be the number of input and output edges, respectively of the projected subgraph of v' in G_0. Note that $IN(v')$ and $OUT(v')$ are not the sum of input and output edges of coarse vertices v and u.

Our matching heuristic visits the vertices of G_i in random order. If a vertex $u \in V_i$ has not been matched yet, then we select it for matching to form a vertex v' in the coarser graph G_{i+1}. First, we identify the adjacent unmatched vertices of u that when combined with u will satisfy all the three constraints. Then we match u with the adjacent vertex v such that the ratio of performance gain to hardware area (*gain-area ratio* in short) of v' is

maximum. We define performance gain: $gain = sw_ltc(v') - hw_ltc(v')$, where $sw_lts(v')$ is the software latency of v' by summing up the software latency of all the vertices in the subgraph of v'; $hw_ltc(v')$ is the hardware latency of v' estimated from the critical path of the subgraph of v'. Hardware area is the sum of hardware area of all vertices in the subgraph of v'. On the other hand, if u cannot find a feasible matching, $v' = u$. In Figure 5.2, vertices 0 and 2 of G_1 are matched to form vertex 0 of G_2.

The coarsening phase ends when $G_{i+1} = G_i$. Let $G_m = (V_m, E_m)$ be the coarsest graph. Initial partitioning simply selects each vertex $v \in V_m$ as a custom instruction. These initial custom instructions will be refined as we go through un-coarsening phase to project back to G_0. In Figure 5.2, coarsening phase creates a sequence of coarse graphs $\{G_0, G_1, G_2, G_3\}$ and the initial partitioning partitions G_3 into two custom instructions CI_0 and CI_1.

Uncoarsening Phase During the uncoarsening phase, the partitioning of the coarsest graph G_m is projected back to the original graph by going through a sequence of finer graphs $G_{m-1}, \ldots, G_0$. In Figure 5.2, we label the graph during uncoarsening phase with G'_m for easy explanation. But in reality, $G'_m = G_m$. Consider a graph $G_i = (V_i, E_i)$. Its partitioning is represented by a partitioning vector P_i of length $|V_i|$ where for each vertex $v \in V_i$, $P_i[v]$ is an integer between 1 and $|V_m|$ (the number of partitions defined by the number of vertices in the coarsest graph). $P_i[v]$ indicates the partition to which vertex v belongs in graph G_i. In the coarsest graph G_m, each vertex belongs to its own partition.

Let V_i^v be the set of vertices of G_i (in our case one or two vertices) that have been combined to form a single vertex v in the next level coarser graph G_{i+1}. Then during uncoarsening, P_i is initially obtained from P_{i+1} by simply assigning the partitioning of v in the coarser graph ($P_{i+1}[v]$) to the partitioning of each vertex in V_i^v. That is, $P_i[u] = P_{i+1}[v], \forall u \in V_i^v$.

However, at every level of un-coarsening, we have the option of improving the projected partitioning P_i by moving some vertices from one partition to another. It is possible to improve the partitioning P_i compared to P_{i+1}. This is because, G_i is a finer graph and it allows more degrees of freedom to move the vertices. Local refinement based on Kernighan-Lin (KL) [59] or Fiduccia-Mattheyses (FM) [33] partitioning algorithms tend to produce good results for bi-partition. However, using KL or FM for refining multiple partitions is significantly more complicated because vertices can move from a partition to many others. Therefore, we propose a simple and efficient refinement algorithm (similar in spirit to the greedy refinement proposed in [56]) to target the objective and constraints of our problem.

Given $G_i = (V_i, E_i)$ with partitioning solution P_i, a vertex $v \in V_i$ is a boundary vertex of partition $P_i[v]$ if it has at least one adjacent vertex $u \in V_i$ such that v and u belong to different partitions, i.e., $P_i[v] \neq P_i[u]$. Otherwise, v is an internal vertex. For G'_1 in Figure 5.2, vertices $\{2,4\}$ are the boundary vertices of partition CI_0 while $\{0,1\}$ are the internal ones. Note that G'_1 is at the same coarse level of G_1. Our refinement algorithm visits boundary vertices in random order. If v is selected, let p_v is the subgraph of G_i w.r.t. current partition containing v and $NP[v]$ be the set of subgraphs of G_i w.r.t. neighborhood partitions to which vertices adjacent to v belong. Algorithm 5 tries to move v to neighborhood partitions if it is possible.

Let p' be the resulting partition after moving v to a neighborhood partition p. p' may violate constraints. If input constraint is violated by adding v (line 6), we try to reduce the number of inputs (line 7) by continuously adding vertices (in breadth first traversal order) of the *backward* subgraph rooted at v to p'. At each level (in breadth first traversal), the vertices are ordered w.r.t. the number of edges connecting the vertex to the partition p'. If a vertex is connected with p' via multiple edges, it has highest potential to reduce the number of inputs of p'. We define *permanent* inputs as the inputs of the original graph G_0. We stop adding vertices to p' if either (1) input constraint is surely violated because

Algorithm 5: **Moving vertex** v

Input: $G_i = (V_i, E_i)$, P_i and $NP[v]$

Result: Update P_i

```
1  best_ratio_improv = 0;
2  p'_v ← p_v \ {v};
3  if p'_v satifies all constraints then
4      for p ∈ NP[v] do
5          p' ← p ∪ {v};
6          if IN(p') > N_in then
7              Reduce_number_inputs(p');
8          if OUT(p') > N_out then
9              Reduce_number_outputs(p');
10         if p' satisfies all constraints then
11             ratio_improv ← gain(p')/area(p') − gain(p)/area(p) + gain(p'_v)/area(p'_v) − gain(p_v)/area(p_v);
12             if ratio_improv > best_ratio_improv then
13                 best_ratio_improv ← ratio_improv;
14                 Update best_solution;
15 if best_ratio_improv then
16     Update P_i;
```

number of permanent inputs of p' is more than N_{in}, or (2) input constraint is satisfied, or (3) either convexity or output constraint is violated. In G'_1 of Figure 5.2, if we move vertex 2 to CI_1, input constraint is violated (i.e., IN(CI_1) is 5 ¿ 4). Then, we continue adding vertex 0 and number of permanent inputs is greater than 4, we stop adding vertices. G'_0 is the finer graph which is projected from G'_1. In G'_0, after moving vertex 9 to CI_1, we continue adding vertices 6,7 which results in valid subgraph CI_1 in G''_0. Because G''_0 has higher gain-area ratio improvement than G'_0, G''_0 is the result of multi-level partitioning instead of G'_0.

Similarly, if output constraint is violated by adding v (line 8), we try to reduce the num-

ber of outputs (line 9) by continuously adding vertices of the *forward* subgraph rooted at v in order of breadth first traversal. Then, if p' is a valid subgraph, we compute its ratio improvement, *ratio_improv* (lines 10-11). Note that performance gain of p'_v is equal to 0 if p'_v is invalid custom instruction. If ratio improvement is better than best ratio improvement (*best_ratio_improv*) so far, we update best ratio improvement and the corresponding solution (lines 12-14). If there exists best ratio improvement, we update P_i (lines 15-16). Intuitively, we move vertex v to the neighborhood partition which has the best ratio improvement.

5.3 Experimental Evaluation

The evaluation of our approach consists of two separate set of experiments. First, we establish how our iterative approach can substantially expedite the system-level design process. The second set of experiments show that the custom instructions generated by MLGP algorithm are as good quality as the ones generated by the current state-of-the-art algorithms. We compare MLGP with *Iterative Selection* (IS) algorithm [81] which is one of the state-of-the-art algorithms generating good quality custom instructions.

5.3.1 Experimental Setup

For our experiments, we use Trimaran 4.0 [19] as front-end and in the back-end we assume a single-issue in-order processor core with perfect cache and branch prediction. Given an application, we first invoke Trimaran 4.0 [19] to compile the application and generate the intermediate machine code. Then, we build the program control flow graph and corresponding syntax free from the intermediate machine code. Subsequently, both the WCET computation and custom instruction generation are done based on the application's control

flow graph and syntax tree. We perform all the experiments on a 3GHz Pentium 4 CPU with 2GB memory.

We use Synopsys design tools with 0.18 micron CMOS cell libraries to synthesize primitive operations, e.g. addition, multiply, etc to get hardware area as well as execution time in hardware of each primitive operation. Based on these values of each operation, we can estimate latency and area of custom instructions through *hw_ltc()* function and sum of hardware area of all vertices in custom instructions respectively. Each custom instruction can have at most 4 input operands and 2 output operands. Execution cycles of a custom instruction is its latency normalized against a MAC, which has 1 cycle latency in the processor running at 120MHz.

5.3.2 System-Level Design

We evaluate our work using various benchmark programs which are shown in Table 5.1. Five task sets are created by random composing a subset of these benchmarks as shown in Table 5.2. For each task set, we choose a total utilization for the task set (without any custom instructions) and then select the periods of the constituent tasks to achieve the corresponding utilization. Recall that C_i be the WCET of task T_i without customization. Then we set the period P_i for each task T_i as $P_i = \alpha_i \times C_i$ such that $\sum_{i=1}^{N} \frac{C_i}{P_i} = U$. We would like to exploit customization to make an unschedulable task set become schedulable. Therefore, we vary total utilization factor U for a task set from 1.1 to 1.5 with interval of 0.1. The greater the original utilization factor, the more difficult it is to schedule the tasks using custom instructions.

Figure 5.3 plots the reduction in utilization through each iteration of Algorithm 4 for 5 task sets with different input utilization factors. The X axis and Y axis show number of iterations and the utilization factor for the whole task set. The utilization drops dramatically

Benchmark	**Source**	**WCET cycles**	**Max** BB size	**Average** BB size
adpcm	WCET Benchmarks	127,407	331	15
sha	Mibench	9,163,779	487	38
jfdctint	WCET Benchmarks	2,217	107	19
g721decode	Mediabench	113,295,478	80	9
lms	WCET Benchmarks	65,051	29	8
ndes	WCET Benchmarks	21,232	56	9
rijndael	Mibench	13,878,360	239	24
3des	Trimaran	106,062,791	2745	59
aes	Trimaran	30,638	227	16
blowfish	Mibench	435,418,994	457	22

Table 5.1: Benchmark Characteristics. The maximum and average size of basic block (BB) are given in term of primitive instructions.

after the first iteration and gradually reduces in the following iterations. It takes 4 or 5 iterations on average to bring the processor utilization below 1.0 (i.e., the task set becomes schedulable). The smaller the input utilization, the smaller is the number of iterations. This result shows that our iterative scheme efficiently achieves the necessary reduction of utilization (or execution time in general).

Figure 5.4(a) shows the analysis time (in seconds) of our methodology for different task sets with different input utilization factors. The X axis and Y axis show the input utilization and the analysis time of our methodology. For the task sets we experimented with, we can generate custom instructions to make an unschedulable task set become schedulable within 10 to 65 seconds. However, with higher input utilization, it may not be possible to

Task set	Benchmarks
1	3des, rijndael, sha, g721decode
2	sha, jfdctint, rijndael, ndes
3	ndes, g721decode, rijndael, sha
4	aes, 3des, adpcm, jfdctint
5	adpcm, jfdctint, rijndael, sha

Table 5.2: Task Sets.

obtain a feasible solution. For example, for task set 3 with input $U = 1.4, 1.5$, processor customization fails to bring the processor utilization below 1.0. Therefore, we only show partial results in these cases (see the red highlighted circle).

The faster analysis time of our approach confirms that the iterative methodology is very efficient for system-level design space exploration. If there is either feasible or infeasible solution, our approach returns results in seconds. As mentioned earlier, without the iterative approach, the designer has to first generate a set of custom instructions with varying tradeoffs (in area versus performance) for each task in the task set. Subsequently, he/she will select appropriate custom instructions from each task set so as to meet the system performance demand. However, it turns out that generating all possible custom instructions for four tasks in task set 1 using the state-of-the-art algorithm [81] takes more than half a day and even then the process does not terminate. This means that complete design space exploration using custom instructions for task set 1 may take at least half a day to finish or may even be infeasible. In constrast, our iterative approach returns the first-cut solution within 3 seconds to make task set 1 schedulable even when the input processor utilization is equal to 1.5.

Figure 5.4(b) shows the correlation between hardware area and input utilization U. The

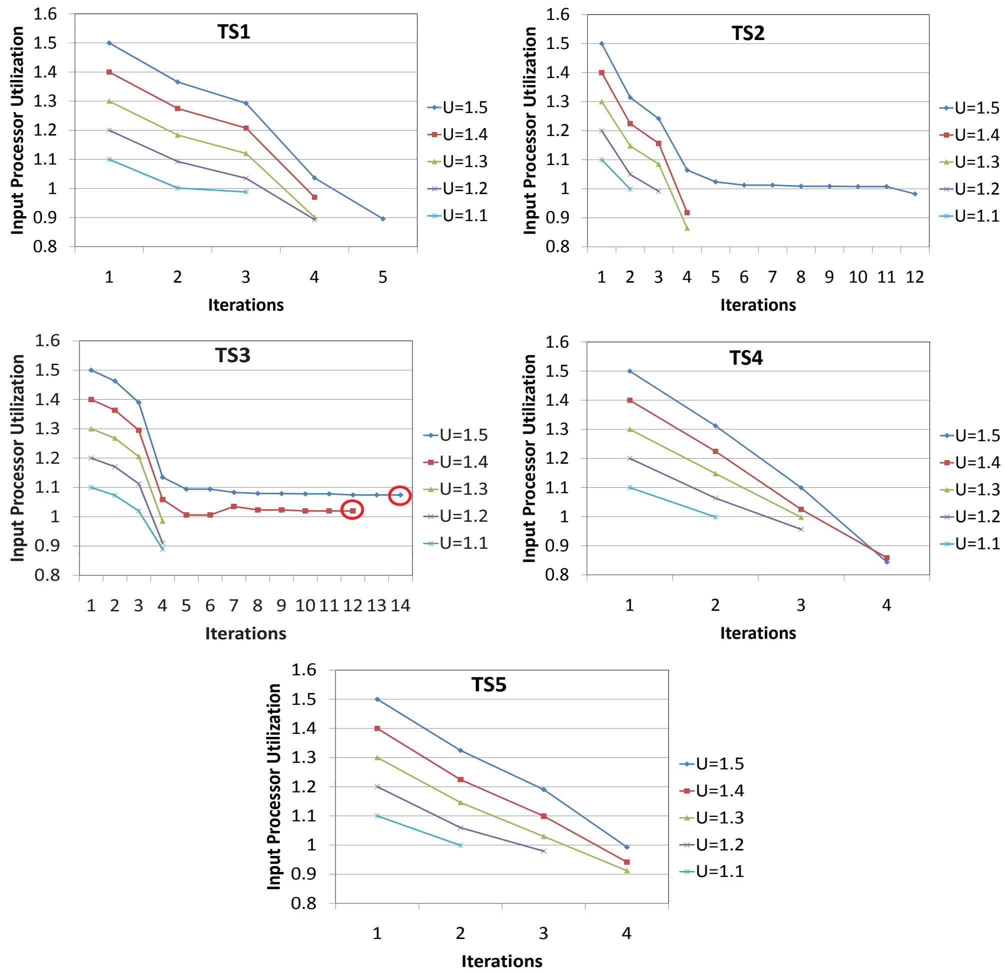
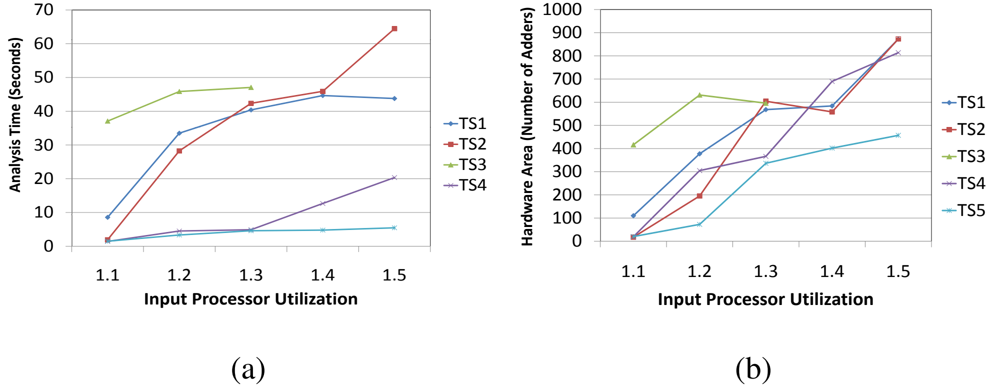


Figure 5.3: Reduction in processor utilization with increasing number of iterations

Y axis shows the hardware area in term of adders (hardware unit used by conventional custom instruction generation techniques) required by our solution and X axis again shows input utilization factor. As the input utilization increases, the hardware area correspondingly increases because more custom instructions are required to make the task set schedulable.

5.3.3 Efficiency of MLGP Algorithm

The heart of our system-level processor customization approach is the multi-level graph partitioning (MLGP) algorithm that on-the-fly generates high-quality custom instructions. In this section, we show that MLGP can generate high quality custom instructions which



(a) (b)

Figure 5.4: (a) Analysis time of our approach with varying input utilization for all 5 task sets; and (b) Hardware area required by custom instructions with varying input utilization for all 5 task sets

achieve high execution time reduction under a hardware area budget. To substantiate this claim, we compare MLGP with a state-of-the-art custom instruction generation algorithms, IS algorithm [81], with the objective of generating high quality custom instructions. It has been shown [81] that IS generates almost the optimal set of high quality custom instructions in practice without paying for the exponential computational complexity of the optimal algorithm.

We have implemented both the algorithms (MLGP, IS) in the Trimaran infrastructure as discussed in Section 5.3.1. The same synthesis tool and cell libraries have been used for all the algorithms. Moreover, all the algorithms have been restricted to generate custom instructions with at most 4 input register ports and 2 output register ports. Custom instructions do not include memory references or conditional branches. In this comparison, we implement the connected version of IS to generate connected custom instructions. Note that connected custom instructions generated from IS are almost the optimal set of connected custom instructions by simply not considering disjoint components in the core of IS.

Note that in these set of experiments we are concerned with average-case performance

improvement for each individual benchmark for the purpose of a clear comparison rather than WCET reduction in system-level design context. Therefore, we profile each benchmark separately with representative inputs within Trimaran infrastructure and annotate each basic block with its execution frequency. MLGP is suitability modified to work on the hot basic blocks in terms of execution frequency rather than the worst-case path. The critical basic blocks are sorted in decreasing order of their execution time before IS is executed. This setting helps IS to return better quality custom instructions early on.

Let $\mathcal{B}$ be the set of basic block in an application and let x_i and s_i be the execution frequency and software execution time of the basic block B_i, respectively. Then the software execution time of the application is given by $SW = \sum_{\mathcal{B}} x_i \times s_i$. Let h_i be the execution of the basic block B_i after applying processor customization. Then the reduced execution time of the application is $HW = \sum_{\mathcal{B}} x_i \times h_i$. *The speedup of the complete application due to processor customization is then* $speedup = \frac{SW}{HW}$.

We run the two algorithms on a variety of benchmarks. Some of the benchmarks contain only small basic blocks (e.g., `jfdctint, g721decode`) while others contain very large basic blocks (e.g., `3des`, and unrolled `sha`). The benchmark `3des`, for example, has 2745 nodes in the largest basic block.

Figure 5.5 plots the progress of MLGP and IS as they attempt to generate quality custom instructions for the entire benchmark. X-axes show the *analysis time* of the algorithms each time they generate new custom instructions. For IS, a custom instruction is generated after each iteration, while MLGP generates a set of custom instructions after processing a region in the basic block. This partially explains the faster analysis time of MLGP compared to IS. The Y-axes show the speedup for each benchmark.

The first observation from Figure 5.5 is that MLGP returns a set of quality custom instructions within one second and more custom instructions are quickly added as analysis

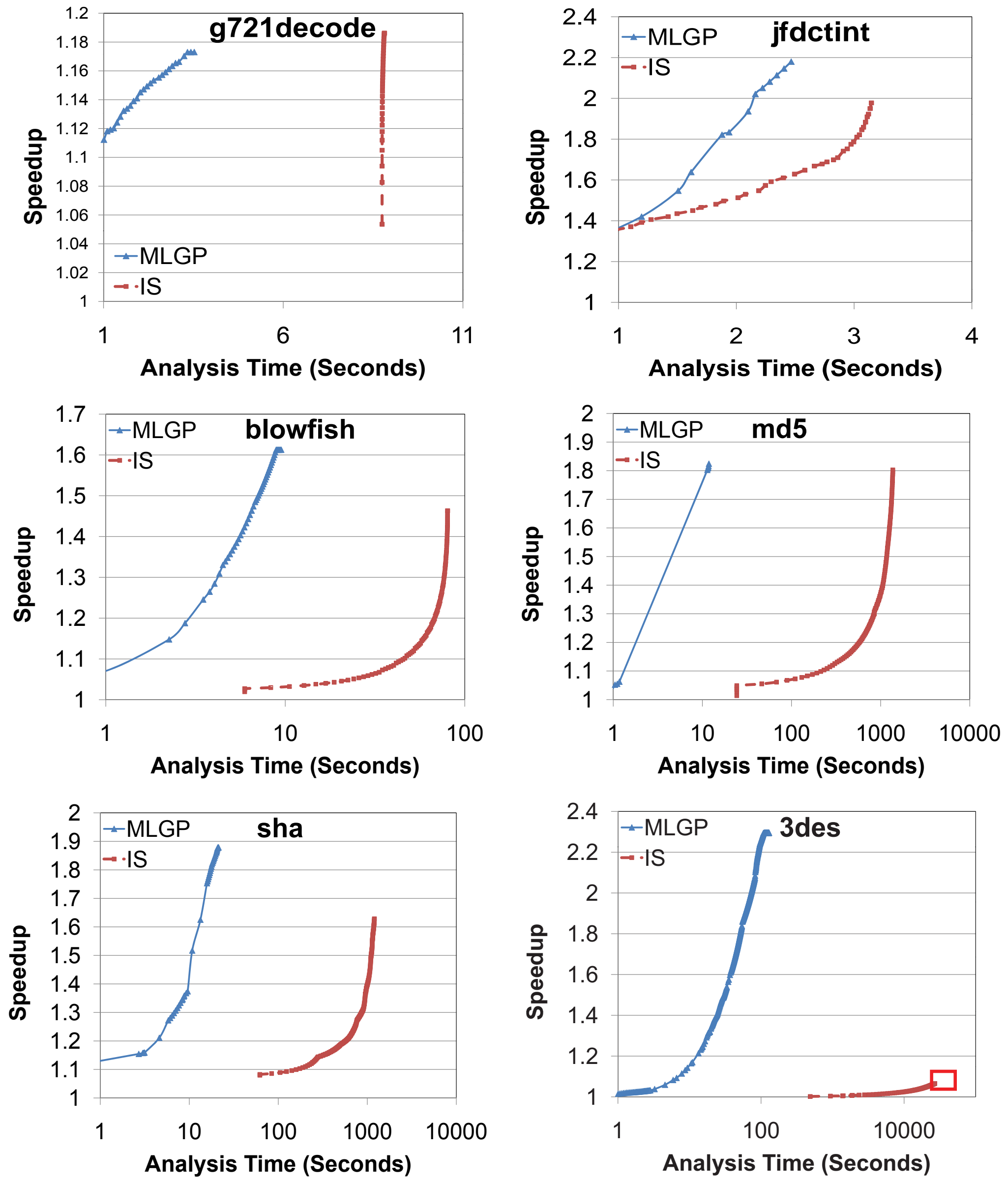


Figure 5.5: Speedup versus Analysis Time

time progresses. For most benchmarks, MLGP returns the complete set of custom instructions within 10 seconds of analysis time. On the other hand, IS takes much longer analysis time to return the first custom instruction for complex benchmarks (in the order of 1,000 seconds). Subsequent custom instructions are generated slowly. The second observation is that MLGP even performs better then IS for several benchmarks, e.g., `sha`, `blowfish`, `jfdctint`.

It should be noted though that IS, in general, can return high-quality custom instructions very quickly for small benchmarks (in the order of few seconds). However, it takes thousands of seconds for IS to return solution when the benchmark contains large basic blocks. Indeed, for `3des` with 2,745 instructions in a basic block, IS fails to generate the full set of custom instructions even after running for half a day. Therefore, we show only partial results (see the red highlighted rectangles).

Figure 5.6 shows the design tradeoffs (hardware area versus speedup) exposed by the two algorithms for processor customization. X and Y axes represent hardware area and speedup, respectively for each generated solution. The results suggest that MLGP solutions have, in general, better speedup under the same hardware area constraint compared to IS. This is because IS generates only one custom instructions per iteration and the nodes of this custom instructions are eliminated from further consideration. This strategy has the risk of getting stuck in a local optima as the generated custom instruction (though optimal at this point) can disable many choices in the future. In contrast, MLGP returns a set of custom instructions per iteration leading to better design space exploration. For `3des`, we only show partial results for IS (also see the red highlighted rectangle) as it fails to return complete set of custom instructions even after running for half a day.

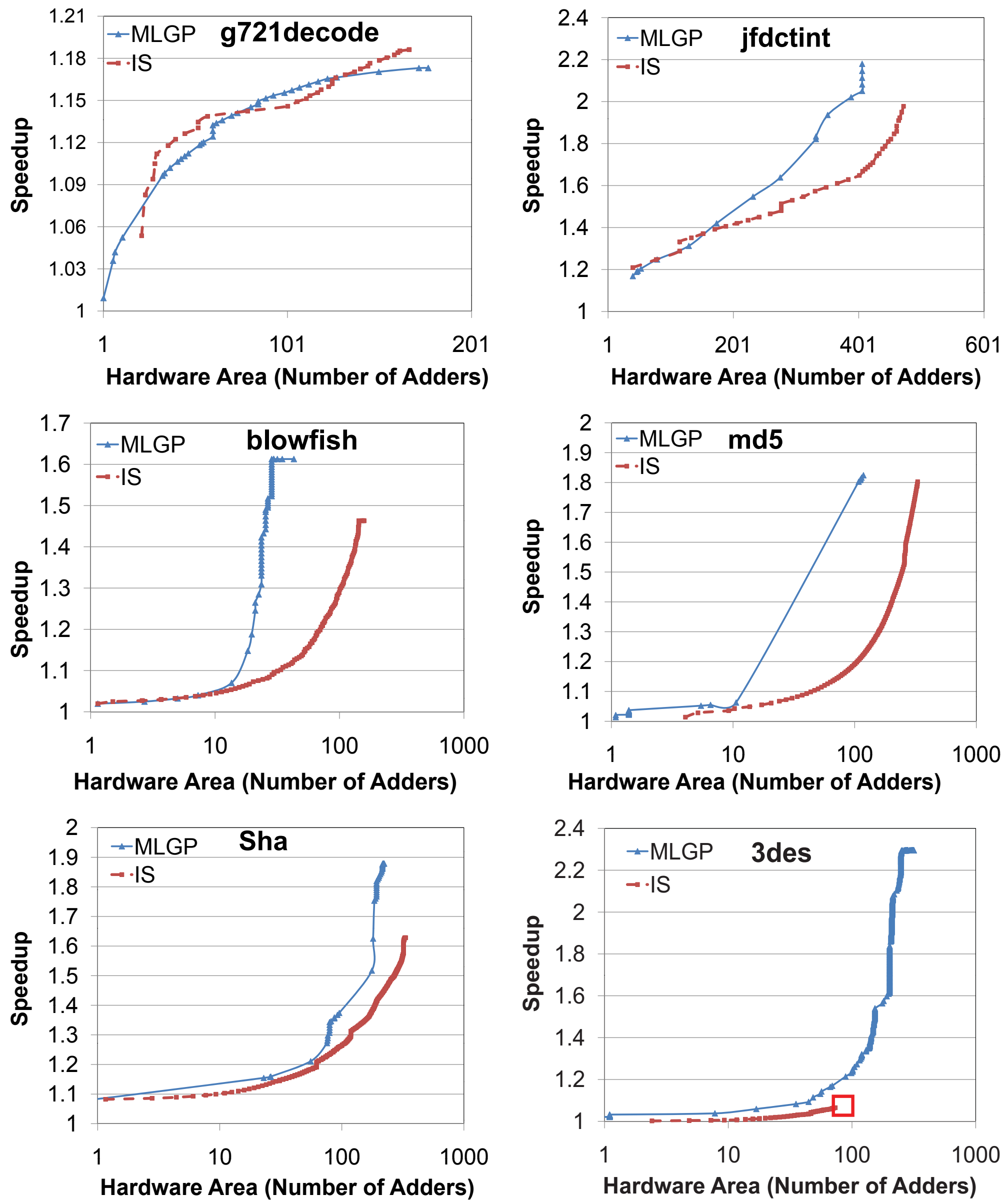


Figure 5.6: Design tradeoffs in processor customization.

5.4 Summary

We propose an iterative scheme to generate custom instructions in an on-demand basis guided by the system-level performance requirements. Our approach zooms into the critical region that is causing the performance bottleneck and starts the customization process from that region. We provide a close coupling between the system-level design and the customization algorithm. The critical component of our framework is an efficient algorithm based on multi-level graph partitioning that generates the custom instructions on-the-fly. Experimental results validate that our iterative scheme is quite effective in quickly producing good quality solutions.

Chapter 6

Runtime reconfiguration of custom instructions

Instruction-set customization can help to improve significant performance for embedded systems (such as our work in Chapter 3, 4 and 5). However, the total area available for the implementation of the CFUs in a processor is limited. Therefore, we may not be able to exploit the full potential of all the custom instructions in an application. This under-utilization is particularly true if the application consists of a large number of kernels and each kernel requires unique custom instructions — a scenario that is quite common in high-performance embedded systems. Furthermore, it may not be possible to increase the area allocated to the CFUs due to the linear increase in the cost of the associated system. In this context, runtime reconfiguration of the CFU fabric appears quite promising. Here the set of custom instructions implemented in the fabric can change over the lifetime of the application. For multi-kernel applications, runtime reconfiguration is especially attractive, as the fabric can be tailored to implement *only* the custom instructions required by the active kernel(s) at any point of time. Of course, this virtualization of the CFU fabric comes at the cost of reconfiguration delay. The designer has to strike the right balance between the

number of configurations and the reconfiguration cost.

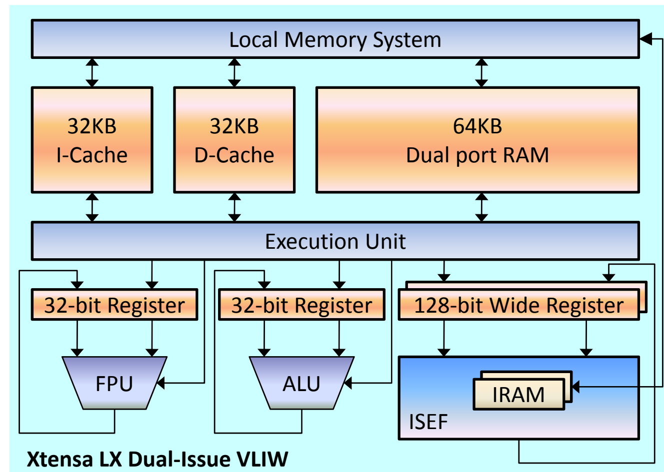


Figure 6.1: Stretch S6000 datapath [38].

To exploit this performance potential, commercial customizable processors supporting dynamic reconfiguration have been proposed. For example, Figure 6.1 shows the Stretch S6000 engine that incorporates Tensilica Xtensa LX dual-issue VLIW processor [37] and the Stretch Instruction Set Extension Fabric (ISEF). The ISEF is software-configurable datapath based on programmable logic. It consists of a plane of arithmetic/logic elements (AU) and a plane of multiplier elements (MU) embedded and interlinked in a programmable, hierarchical routing fabric. This configurable fabric acts as a custom functional unit to the processor. It is built into the processor's datapath, and resides alongside other traditional functional units such as the ALU and the floating point unit. The programmer defined application specific instructions (Extension Instructions) are implemented in this fabric. When a custom instruction is issued, the processor checks to make sure the corresponding configuration (containing the extension instruction) is loaded into the ISEF. If the required configuration is not present in the ISEF, it is automatically loaded prior to the execution

of the user-defined instruction. ISEF provides high data bandwidth to the core processor through 128-bit wide registers. In addition, 64KB embedded RAM is included inside ISEF to store temporary results of computation. With all these features, a single custom instruction can potentially implement a complete inner loop of the application. The Stretch compiler fully unrolls any loop with constant iteration counts.

The distinguishing aspect of ISEF is that it is run-time configurable and reloadable. If the computation resource requirement of the custom instructions exceeds the capacity of ISEF, the instructions can be partitioned into different *configurations*. When a user-defined instruction is issued, the S5 hardware checks to make sure that the corresponding configuration is loaded into the ISEF. If the required configuration is not present in the ISEF, it is automatically loaded prior to the execution of the user-defined instruction. In summary, the ISEF allows the system designers to define new instructions at runtime and thus extend the processor's instruction set.

Currently, it is the programmer's responsibility to manually choose and define the custom instructions and the configurations for architectures such as Stretch. Choosing an appropriate set of custom instructions for an application itself is a difficult problem. Significant research effort has been invested in developing automated selection techniques for custom instructions [5, 81, 24, 25, 57, 22, 66, 101, 102, 103]. Runtime reconfiguration has the additional complication of both *temporal and spatial partitioning* of the set of custom instructions in the reconfigurable fabric. Figure 6.2 shows how a C code accelerated with different Custom Instruction Sets (CIS) configures the CFU fabric during run-time. A CIS is a set of custom instructions corresponding to a program fragment. When the CFU fabric can accommodate more than one CIS, it is spatially partitioned among them. In Figure 6.2, configuring the CFU fabric with both `CIS-1` and `CIS-2` at the same time constitutes an example of spatial partitioning. The CFU fabric is temporally partitioned when it is loaded

with different configurations during run-time. In our example, the CFU fabric is configured with `CIS-3` after exiting the `for` loop. Therefore, the custom instructions of the entire application are partitioned into two temporal configurations: {`CIS-1, CIS-2`} and {`CIS-3`}.

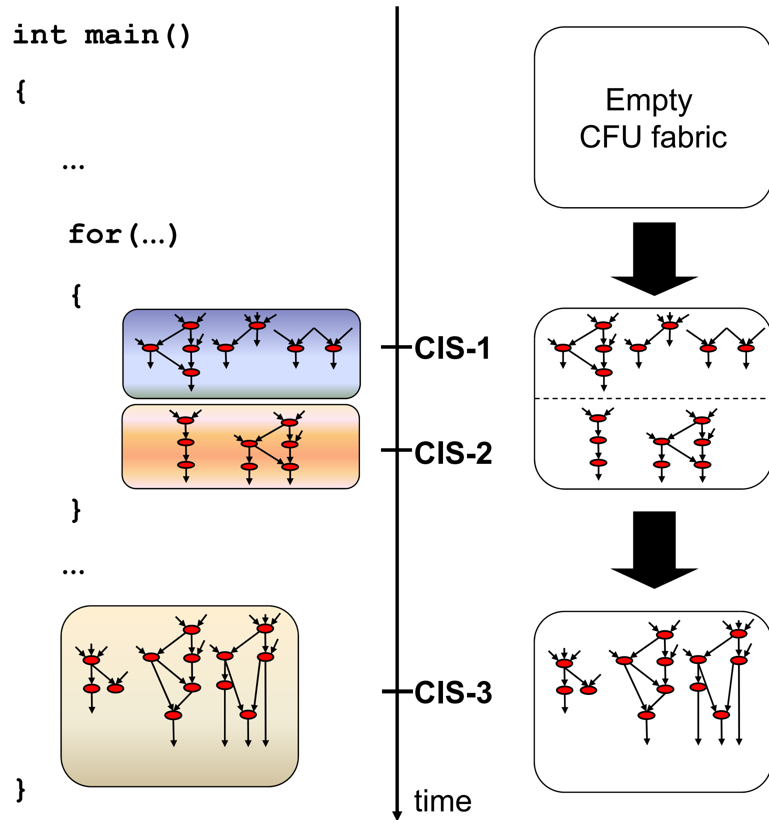


Figure 6.2: Spatial and temporal partitioning of the custom instructions of an application and the state of the CFU fabric during execution.

In this chapter, we develop a framework that starts with an application specified in ANSI-C and automatically selects appropriate custom instructions as well as clubs them into one or more configurations. We first extract a set of compute-intensive candidate loop kernels from the application through profiling. For each candidate loop, we generate one or more custom instruction-set versions differing in performance gain and area tradeoffs in

addition to the purely software version. The key component of our framework is an iterative partitioning algorithm. The partitioning algorithm selects appropriate custom instruction-set versions for the loops implemented in fabric and clubs them into suitable configurations to achieve the highest performance gain. We model the temporal partitioning of the custom instructions into different configurations as a k-way graph partitioning problem. We develop a dynamic programming based pseudo-polynomial time algorithm for the spatial partitioning of the custom instructions within a configuration. To the best of our knowledge, this is the first work that attempts automated custom instructions selection in the context of instruction-set extensible processor platforms with dynamic reconfiguration.

Most hardware-software partitioning solutions for FPGAs work at a coarse-grained level (such as task level). However, as we would like to accelerate complete applications specified in high-level programming languages such as ANSI-C, we focus on hot loop kernels instead. Note that the reconfiguration cost model at task level [20, 58] and data flow graph level [83] are simple because the underlying directed acyclic graph representation ensures at most one reconfiguration between any two nodes. In contrast, our dynamic reconfiguration cost model is complex as the number of reconfigurations for one loop depends on temporal partitioning of all the other loops. Furthermore, our methodology allows custom instruction sets corresponding to more than one loop to be placed within a single configuration. Thus spatial partitioning also plays a role in determining the performance gain of the application. The only other loop-level temporal partitioning work that we are aware of [69] considers only one loop per configuration.

The remainder of this chapter is structured as follows. Section 6.1 describes the system design flow. In Section 6.2, we present the problem formulation and a motivating example. Section 6.3 details our partitioning algorithm. Experimental setup and evaluation are described in Section 6.4.

6.1 System Design Flow

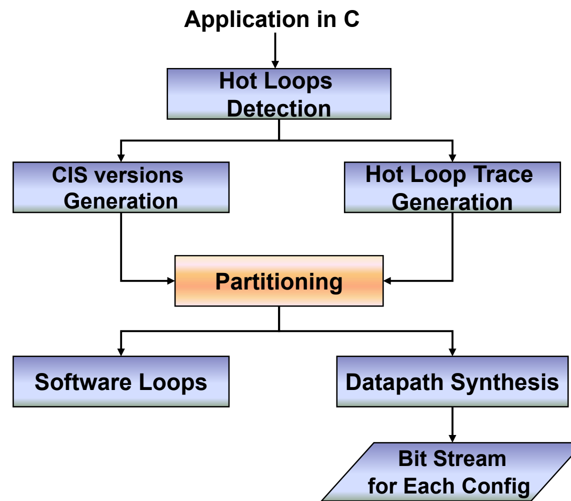


Figure 6.3: System design flow

Figure 6.3 shows the system design flow. The input to the design flow is the C source code of the application we want to accelerate. The output is the application accelerated with custom instructions and the synthesized datapath for each configuration. In the following, we describe each component of this design flow.

Hot loops detection Taking our cue from Amdahl's law, we focus on the loops that take up a significant portion of the application's total execution time. In particular, we define a loop with execution time greater than a certain percentage (typically $\geq 1\%$) of the application's overall execution time to be a *hot* loop. The hot loop detector identifies such loops through profiling. Although the total number of loops in an application may be large, we consider only the hot loops to reduce the computation cost of the partitioning algorithm significantly. At the same time, the performance gain we obtain is still comparable to the case where all the loops of the application are considered. This result is because the min-

imal performance gain of the cold loops are more than offset by the high reconfiguration overhead.

Custom instruction-set versions generation We generate multiple custom instruction-set (CIS) versions for each hot loop with a trade-off between hardware area and performance gain. A CIS version consists of a set of custom instructions extracted from the corresponding loop under an area constraint. Each CIS version is characterized by its area and performance gain. In general, the performance gain of a CIS version increases with larger area. To generate the CIS versions for a loop, we first identify [5, 81, 24, 25, 57, 102, 103] a large set of candidate patterns from the loop. Given this library of patterns, in the second step, we select a subset to maximize performance gain under hardware area constraint [5, 22, 24, 25, 66, 101]. As the area increases, a CIS version with higher performance gain will be generated by selecting a larger subset. Moreover, different CIS versions can be generated by loop transformations such as loop unrolling, software pipelining, loop fusion, and others.

Loop Trace The control flow among the hot loops is captured in the form of a loop trace (execution sequence of the loops) obtained through profiling. For typical embedded applications we have profiled, the number of hot loops and the loop trace size are quite small. For longer loop trace, we can use lossless compression techniques (such as SEQUITUR algorithm [74]) to compactly maintain the loop trace.

The hot loops with CIS versions and the loop trace are fed to the partitioning algorithm that decides the appropriate CIS version and configuration for each loop. The selected CIS versions to be implemented in hardware are then input into the datapath synthesis tool. It generates the bit stream corresponding to each configuration (based on the result of temporal partitioning). These bitstreams are used to configure the fabric at runtime. The

remaining loops are implemented in software on the core processor. Finally, the source code is modified to exploit the new custom instructions.

6.2 Partitioning Problem

We now formally define the partitioning problem for dynamic reconfiguration of custom instructions, which is the focus of this chapter.

The input to the partitioning step is the set of hot loops $L = \{l_i | i = 1...N\}$. Each loop is associated with multiple custom instruction-set (CIS) versions with a trade-off between hardware area and performance gain. Let $l_{i,j}$ (for $j = 1 \ldots n_i$) be the jth CIS version corresponding to loop l_i where n_i is the number of CIS versions of loop l_i. In addition, let $gain_{i,j}$ and $area_{i,j}$ denote the performance gain and area requirement of $l_{i,j}$. We assume that $l_{i,1}$ corresponds to the software loop without any custom instructions, i.e., $area_{i,1} = 0$ and $gain_{i,1} = 0$. For each loop l_i, only one of its CIS versions will be selected for implementation. For example, if $l_{i,1}$ is selected, loop l_i will be implemented in software without any custom instruction enhancements.

The control flow among the loop kernels is input in the form of a loop trace. Finally, *MaxA* represents the hardware area available for each configuration and ρ represents the time required for a single reconfiguration. In this chapter, we do not consider partial reconfiguration, i.e., a configuration is completely replaced by another configuration in the fabric. Hence both *MaxA* and ρ are constants. Intra-loop reconfiguration incurs high reconfiguration cost. Thus we do not allow custom instructions corresponding to a loop to straddle across configuration boundaries. In other words, the selected CIS version of a loop is completely accommodated within a configuration, i.e., $area_{i,j} \leq MaxA$ (for $i = 1 \ldots N$, $j = 1 \ldots n_i$). Each configuration, however, consists of CIS versions corresponding to one or

more loops. Thus the problem boils down to

1. *Temporal partitioning* of the loops selected for hardware acceleration with CIS into one or more configurations, and

2. *Spatial partitioning* of the loops within a configuration by selecting appropriate CIS version for each loop.

The performance gain of the application is then defined as

$$Performance\ \ gain = \left(\sum_{i=1}^{N} \sum_{j=1}^{n_i} s_{i,j} \times gain_{i,j} \right) - r * \rho \tag{6.1}$$

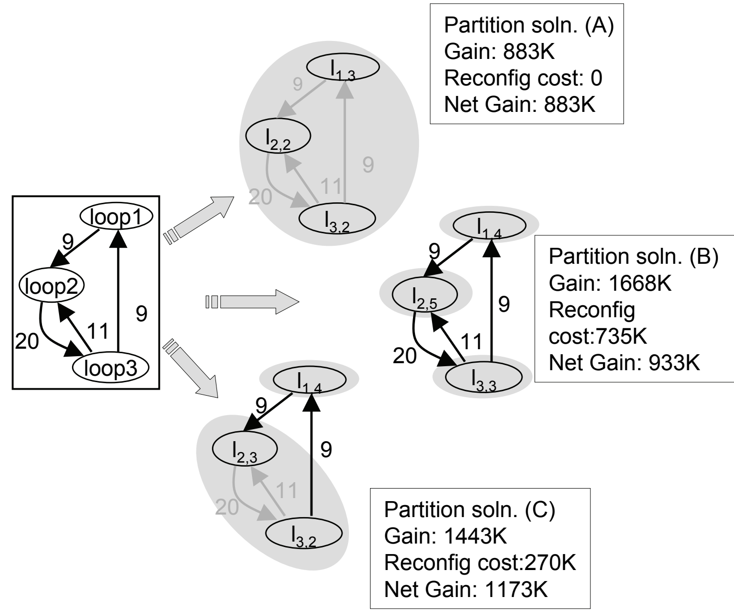
$$\sum_{j=1}^{n_i} s_{i,j} \leq 1 \tag{6.2}$$

where r is the number of reconfigurations given the partitioning and $s_{i,j}$ is a binary variable equal to 1 if CIS version $l_{i,j}$ is selected and 0 otherwise.

Dynamic reconfiguration through temporal partitioning enlarges the available area for the design by increasing the number of configurations. Therefore, each loop can select better CIS version to be implemented in hardware and better performance gain will be achieved. However, this increase in number of configurations may not result in better overall performance due to the reconfiguration cost. On the other hand, if we minimize the number of configurations, the available area is quite restricted. Consequently, each loop will select its CIS version with smaller area and the performance gain of the application is much smaller, especially when the reconfiguration cost is smaller. Our objective is to maximize the performance gain by selecting an appropriate CIS version for each loop and mapping it into an appropriate configuration.

Loop	Version	Area (#AU)	Gain (K cycles)
loop1	1	0	0
	2	257	111
	3	301	160
	4	1612	563
loop2	1	0	0
	2	761	230
	3	1041	387
	4	1321	426
	5	2004	556
loop3	1	0	0
	2	967	493
	3	1249	549

(a) CIS versions for 3 loops



(b) Some partitioning solutions

Figure 6.4: Motivating Example.

Motivating Example

Let us consider an application with three hot loops: `loop1`, `loop2` and `loop3`. Figure 6.4 (a) shows the performance/silicon area tradeoff of different custom instruction-set versions for each loop. In particular, the table shows the hardware requirement in terms of arithmetic units (AU) and corresponding performance gain in terms of K cycles. For example, `loop3` has three CIS versions. Version 1 of each loop is the software version (without any custom instructions enhancements) with zero area and performance gain. We need to select appropriate CIS versions for the three loops and club them into one or more configurations. Let the hardware area constraint for a single configuration be 2048 AUs. The cost for a single reconfiguration is 15K cycles. The graph on the left-hand side of Figure 6.4 (b) shows control flow information among the loops for this example. The actual input to our algorithm is the loop trace. We use the graph here (derived from the loop trace)

for illustration purposes. We will, however, use a similar graph (called reconfiguration cost graph) later in our temporal partitioning algorithm.

If the system does not support dynamic reconfiguration, the best partitioning solution (solution (A) in Figure 6.4 (b)) under the hardware area constraint is the selection of version 3 of `loop1`, version 2 of `loop2`, and version 2 of `loop3`. Total performance gain is $160+230+493 = 883$K cycles and there is no reconfiguration cost.

However, in the presence of dynamic reconfiguration, we can improve the solution. A trivial solution is to put each loop into one configuration (solution (B) in Figure 6.4 (b)). We can then select the CIS version of a loop with the largest area less than or equal to the area of a configuration: version 4 for `loop1`, version 5 for `loop2` and version 3 for `loop3`. Total performance gain is $563+556+549 = 1668$K cycles and the total reconfiguration cost is $(20+11+9+9) \times 15 = 735$K cycles. Therefore the resulting net performance gain after subtracting the reconfiguration cost is $1668-735 = 933$K cycles. While the net performance gain is better than the case when dynamic configuration is not supported, it is not the optimal solution.

The optimal solution is to put `loop2` and `loop3` into one configuration and `loop1` into a different configuration (solution (C) in Figure 6.4 (b)). CIS versions 4, 3, and 2 will be selected for `loop1`, `loop2`, and `loop3`, respectively. The performance gain is 1443K cycles, while reconfiguration cost is $(9+9) \times 15 = 270$K cycles. Hence, the net performance gain is $1443-270 = 1173$K cycles.

6.3 Partitioning Algorithm

Finding the optimal combination of temporal and spatial partition is a difficult problem. Given N loops, the number of possible configurations is 2^N. However, the number of ways

to partition N loops into mutually-exclusive configurations corresponds to the $N+1^{th}$ Bell number. According to de Brujin [27], asymptotic limits of Bell numbers is $O(e^{N\ln(N)})$.

Our partitioning algorithm needs to makes three choices: (1) optimal number of configurations k, (2) temporal partitioning of the loop kernels into k configurations, and (3) spatial partitioning of the loop kernels in each configuration, i.e., choosing the appropriate custom-instruction set (CIS) version for each loop kernel. Clearly, these choices are inter-dependent. The selection of CIS versions for the loops determines the partitioning solution and vice versa.

Algorithm 6: Iterative Partitioning Algorithm

Input: Set of hot loops with custom instruction-set versions: L

Loop Trace: T

Maximum Area of a configuration: $MaxA$

Reconfiguration Cost: ρ

Result: Partition with the best net performance gain

for *k = 1 to |L| in steps of 1* **do**

  C := global_spatial_partition($L, k \times MaxA$);

  P := temporal_partition_with_CIS(C, T, k);

  P' := temporal_partition_wo_CIS(L, T, k);

  $soln$:= local_spatial_partition($L, P, MaxA$);

  $soln'$:= local_spatial_partition($L, P', MaxA$);

  if *net_gain(soln') > net_gain(soln)* **then** $soln := soln'$;

  if *net_gain(soln) > net_gain(bestSoln)* **then** $bestSoln := soln$;

return $bestSoln$;

6.3.1 Overview

We propose an iterative algorithm (Algorithm 6) for joint temporal and spatial partitioning of the custom instruction-sets corresponding to the hot loop kernels. The algorithm iterates from a constraint of having exactly 1 configuration (i.e., no reconfiguration) to the upper bound of having $|L|$ configurations where L is the set of hot loops. The solutions (A) and (B) in our motivating example (see Figure 6.4) represent the two extremes ($k = 1$ and $k = |L|$), while the remaining iterations explore the rest of the design space.

For the iteration with k configurations, we would like to identify the k-way partitioning solution with the optimal net performance gain. Unfortunately, temporal and spatial partitioning are again dependent on each other due to the reconfiguration cost. To break this cycle, we apply a heuristic technique. The heuristic first assumes that we have a continuous area of $k \times MaxA$ available to us where $MaxA$ is the maximum area for a configuration. The assumption of continuous area allows us to tentatively select optimal CIS versions for the loops in an ideal (but un-realizable) situation where reconfiguration cost is zero. This assumption provides an upper bound on the performance achievable with k configurations. In reality, however, we have k distinct configurations with $MaxA$ area each. So we partition the loop kernels with selected CIS versions into k configurations such that each configuration has *roughly* $MaxA$ area and the reconfiguration cost is minimized. As we break up the continuous area into k distinct areas, some configurations end up being bigger than $MaxA$, while some other configurations are smaller than $MaxA$. To fix this problem, we have a final patch-up stage that performs spatial partitioning within each configuration to re-distribute $MaxA$ space among the constituent loop kernels. Figure 6.5 illustrates the three phases of the iterative partitioning algorithm corresponding to the iteration with 2 configurations. The input is the three loops in the motivating example and their CIS versions.

The first phase, global_spatial_partition, partitions the area $k \times MaxA$ (where k is the

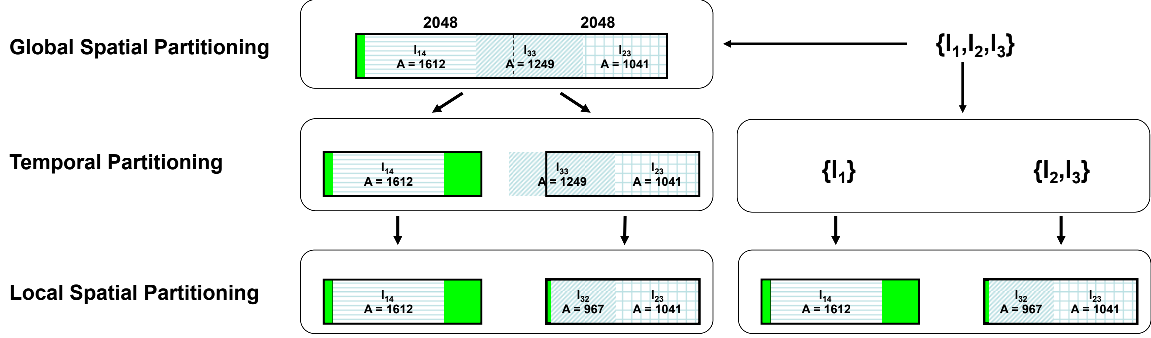


Figure 6.5: Three phases of iterative partitioning algorithm for number of configurations = 2

number of configurations for that iteration) among the loops by selecting the CIS versions such that the performance gain is optimal. This phase disregards the reconfiguration cost. It also assumes that a continuous area of size $k \times MaxA$ is available for hardware acceleration of all the loops. We have developed a *dynamic programming* algorithm for this phase. This phase may choose to select the software version for some loops. For our running example, the first phase in Figure 6.5 chooses CIS versions $l_{1,4}, l_{2,3}, l_{3,3}$.

After the first phase, we have the set of selected CIS versions C for the hot loops. However, we cannot implement this solution as (1) the reconfiguration cost has not been considered, and (2) the loops still need to be partitioned into different configurations. In the second phase temporal_partition_with_CIS, we perform temporal partitioning of the selected loops into k configurations such that the reconfiguration cost is minimized and the partitions are roughly equal in size. This phase returns the partitioning solution P for the set of loops selected for custom instructions enhancements from the first phase.

In the second phase, we also find an alternative partitioning solution P' for the original set of hot loops, i.e., it disregards the results of the first phase. This partitioning, temporal_partition_wo_CIS, only considers the reconfiguration cost and ignores the CIS versions.

Partition P is a better choice when performance gain of the CIS versions is high relative to the reconfiguration cost. On the other hand, partition P' is a better choice when the reconfiguration cost is high relative to the performance gain. P and P' complement each other in the search for the best partitioning solution. We model the temporal partitioning as a *k-way weighted graph partitioning problem*, which is well studied [55, 56].

In Figure 6.5, the left hand side shows the partition P and the right hand side shows the partition P'. For P, the second phase partitions the three loops with selected CIS versions into two configurations: $l_{1,4}$ in the first configuration and $l_{2,3}, l_{3,3}$ in the second configuration. On the other hand, P' simply partitions the three loops based on reconfiguration cost into two configurations. In this example, P and P' return the same temporal partitioning. However, due to the reconfiguration cost, P and P' may be different.

We now have k configurations for each partitioning solution P and P'. The k-way weighted graph partitioning produces partitions with roughly equal size. Therefore for partition P, the area requirement of some of the configurations may exceed the maximum area $MaxA$. Partitioning solution P', on the other hand, does not select any CIS version a-priori. Thus, for each configuration in P and P', the third phase, local_spatial_partition, locally selects the CIS versions for the loops in that configuration to maximize performance gain under area constraint $MaxA$. We again use dynamic programming to perform optimal spatial partitioning for each configuration.

In Figure 6.5, for partition P, the area requirement of the second configuration exceeds the maximum area budget. Hence phase 3 for this partition replaces CIS version $l_{3,3}$ with $l_{3,2}$. Phase 3 keeps the CIS version for loop l_1 unchanged even though there is additional area available (the green part) as $l_{1,4}$ is the best version for l_1. However, in general, the additional area can lead to the selection of better versions for some loops. The third phase of P' simply selects CIS versions of the loops in each configuration for the first time. Finally,

the net performance gains of P and P' are compared to select the best partitioning solution for k configurations.

If the net performance gain of the current solution (with k configurations) is better than the best solution obtained so far (with less than k configurations), we update the best solution. Then we start a new iteration with $k = k+1$. The algorithm terminates when in the current solution, each loop has been assigned its CIS version with the best performance gain. In the worst case, the algorithm runs for $|L|$ iterations. With the motivating example, our algorithm returns the optimal solution, which has two configurations (see Figure 6.5) and the performance gain is 1173K cycles.

Let us now proceed to describe the spatial and temporal partitioning algorithms.

6.3.2 Spatial Partitioning

We propose a pseudo-polynomial time dynamic programming algorithm to select the appropriate CIS versions for the loops such that the performance gain is optimal under a hardware area budget. This algorithm is employed in the first phase and the third phase of our iterative solution with different parameters.

Let $G_i(A)$ be the *maximum* performance gain of loops $l_1 \dots l_i$ under an area budget A. Then $G_i(A)$ can be defined recursively.

$$G_i(A) = \max_{\substack{j=1 \dots n_i \\ area_{i,j} \leq A}} \big(gain_{i,j} + G_{i-1}(A - area_{i,j})\big) \tag{6.3}$$

In Equation 6.3, given an area A, we explore all possible CIS versions for l_i and choose the one that results in maximum performance gain for loops $l_1 \dots l_i$. The base case for loop l_1 is

$$G_1(A) = \max_{\substack{j=1 \dots n_1 \\ area_{1,j} \leq A}} (gain_{1,j}) \tag{6.4}$$

The maximum performance gain for loops $l_1 \ldots l_N$ under area budget $AREA$ then corresponds to $G_N(AREA)$.

Algorithm 7: Spatial Partitioning

Input: Set of loops $\mathrm{l_1, l_1, \ldots, l_N}$ with CIS versions;

Area constraint: *AREA*

Result: Maximum performance gain

for A = 0 to AREA in steps of Δ **do**

$\quad \mathrm{G_1(A)} \leftarrow \max_{\substack{\mathrm{j=1 \ldots n_1} \\ \mathrm{area_{1,j} \leq A}}} (\mathrm{gain_{1,j}})$

end

for A = 0 to AREA in steps of Δ **do**

$\quad$ **for** i=2 to N **do**

$\quad \mathrm{G_i(A)} \leftarrow \max_{\substack{\mathrm{j=1 \ldots n_i} \\ \mathrm{area_{i,j} \leq A}}} \left(\mathrm{gain_{i,j}} + \mathrm{G_{i-1}(A - area_{i,j})}\right)$

end

return $\mathrm{G_N(AREA)}$;

Algorithm 7 encodes this recursion as a bottom-up dynamical programming algorithm. The step value Δ determines the granularity of area. It is chosen as the greatest common divisor of the area requirements of all CIS versions and AREA. The time complexity of this algorithm is $O(N \times \frac{Area}{\Delta} \times x)$ where $x = max_{i=1 \ldots N}(n_i)$.

6.3.3 Temporal Partitioning

We map our temporal partitioning problem to k-way weighted graph partitioning problem. The k-way weighted graph partitioning problem is defined as follows. Given an undirected graph $G = (V, E)$ with weights both on the vertices and the edges, partition V into k subsets $V_1, V_2, \ldots V_k$ such that $V_i \bigcap V_j = \emptyset$ for $i \neq j$, $\bigcup_i V_i = V$, the sum of the vertex-weights in each

subset is roughly equal, and the sum of the edge-weights whose incident vertices belong to different subsets (edge-cut weights) is minimized.

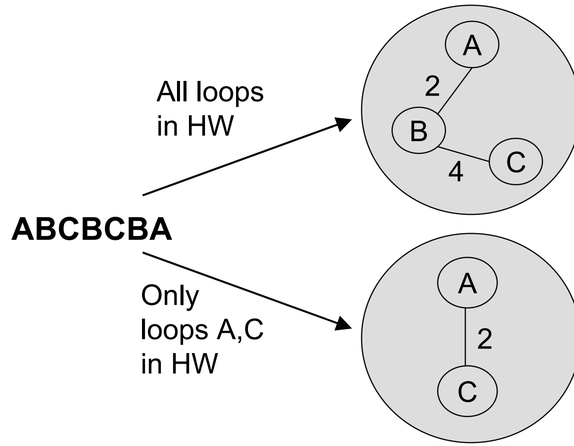


Figure 6.6: Reconfiguration cost graph from loop trace

We generate a *Reconfiguration Cost Graph (RCG)* from the loop trace to model our temporal partitioning problem as a k-way weighted graph partitioning problem. After the first phase, we have tentatively selected CIS versions for the loops. Each vertex in the RCG represents a hot loop selected for hardware acceleration in the first phase. In other words, we do not consider the loops for which the first phase selects software-only version. Given a vertex v associated with loop l, we assign the area of the CIS version selected for l as the weight of the vertex v. When CIS versions from the first phase are ignored (in temporal_partition_wo_CIS), the RCG includes all the loops and we assume unit hardware cost for each vertex.

The edge weight between vertex v (corresponding to loop l) and v' (corresponding to loop l') is defined as the reconfiguration cost between loop l and loop l' if they are mapped to two different configurations. The edge between v and v' exists if and only if control can flow from loop l to l' or l' to l without passing through any other hot loops. The weight on the edge between v and v' represents the number of times control flows from loop l to

l' and l' to l (without passing through any other selected loop). This weight can be derived from the loop trace as follows. If we eliminate the software-only loops from the loop trace, then the weight is the the number of times the string ll' and $l'l$ appear in the loop trace. The time complexity of creating RCG is linear in the size of the hot loop trace.

Figure 6.6 shows an example of RCG generation from the loop trace. It shows a loop trace *ABCBCBA* of three hot loops A, B, C. If all the loops are selected to be placed in hardware, then there are 2 reconfiguration points between loops A and B if they are partitioned into different configurations. Similarly, there are 4 reconfiguration points between loops B and C if they are partitioned into different configurations. However, there are no reconfiguration points between loops A and C directly as the control transfers between them always pass through B. However, if we choose to implement B in software in the first phase, then B is eliminated from the RCG. In this case, there are 2 reconfiguration points between loops A and C if they are partitioned into different configurations.

The objective now is to partition the RCG into k configurations such that the configurations have roughly equal area (or the configurations have roughly equal number of loops when area is ignored) and the reconfiguration cost (edge-cut weights) is minimized. If the configurations have roughly equal area, then the loops have higher probability of retaining the optimal CIS versions selected in the first phase regardless of the third phase. As a result, total performance gain (excluding reconfiguration cost) after the third phase is expected to be near the optimal performance gain in the first phase. The rationale behind having roughly equal number of loops in each configuration when CIS versions are ignored (by assigning unit cost to each vertex in the RCG), is to create a balanced temporal partition. It ensures that equal number of loops compete for each configuration space during subsequent spatial partitioning.

We use multilevel k-way partitioning scheme by Karypis and Kumar [56]. The multi-

level partitioning scheme consists of three phases: coarsening phase, partitioning phase and uncoarsening phase. During coarsening phase, a sequence of smaller graphs $G_i = (V_i, E_i)$, each with fewer vertices, is constructed from the original graph $G_0 = (V_0, E_0)$ such that $|V_i| < |V_{i-1}|$. The coarsening phase ends when the coarsest graph G_m has a small number of vertices or the reduction in the size of successively coarser graph becomes too small. Then, the partitioning phase computes a k-way partitioning P_m of the coarse graph $G_m = (V_m, E_m)$ such that each partition contains roughly $|V_0|/k$ vertex weight of the original graph. The k-way partitioning of G_m is computed using multilevel bisection algorithm [55]. During the uncoarsening phase, the partitioning P_m of the coarser graph G_m is projected back to the original graph by going through the graphs $G_{(m-1)}, G_{(m-2)}, ..., G_1$. At each intermediate level, the partitioning is refined based on Kernighan-Lin [59] partitioning algorithm and their variants. Figure 6.7 shows how the temporal partitioning problem is solved by

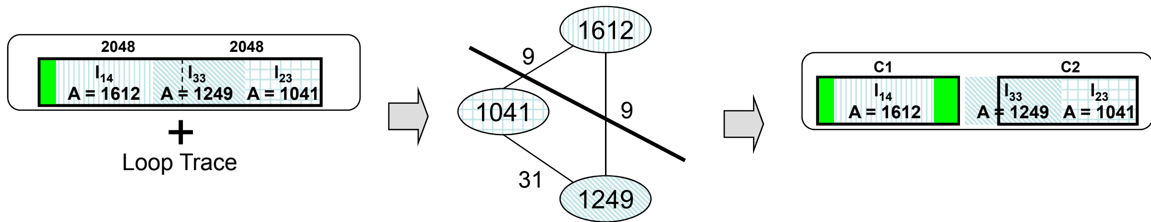


Figure 6.7: Modeling the temporal partitioning problem as k-way graph partitioning problem.

modeling it as a k-way weighted graph partitioning problem for our running example. The edge weights of the RCG are generated from the loop trace. The area of the CIS version selected for each loop in the global partitioning phase is assigned as the weight of the corresponding vertex. Now we perform 2-way partitioning of this graph with minimum edge-cut weights and roughly equal vertex weights in each partition. This partitioning gives us the configurations with roughly equal area while minimizing reconfiguration cost.

6.4 Experimental Evaluation

To compare the accuracy and scalability of our iterative partitioning algorithm, we have developed two other algorithms — exhaustive search and greedy search. The results of the two algorithms are compared with our proposed algorithm in two different sets of experiments. In the first set of experiments, we run the three algorithms using synthetic input to evaluate the scalability and efficiency of the algorithms. We generate input data with 5 to 100 hot loops for this set of experiments. In the second set of experiments, we conduct a case study of the JPEG application with custom instructions implemented on a commercial platform Stretch [38] that supports runtime reconfiguration.

Exhaustive Search The exhaustive search algorithm computes the optimal results by evaluating all possible temporal and spatial partitioning. We use the algorithm described in Kreher and Stinson [63] to enumerate all possible partitioning solutions. We then find the optimal implementation of each configuration in each partitioning solution by choosing CIS versions of the constituent loops through our spatial partitioning algorithm. The net gain of each enumerated partition is then estimated through a brute force computation of the reconfiguration cost by traversing the loop trace. The partition with the maximum net performance gain is then the optimal solution. Our experiments show that the exhaustive search algorithm cannot scale with increasing number of hot loops.

Greedy Search The greedy search algorithm (see Algorithm 8) constructs a solution by building one configuration at a time until no more CIS version can be added without causing a degradation in performance. The input is the set of hot loops with custom instruction-set versions L, loop trace T, area constraint $MaxA$, and single reconfiguration cost ρ. A solution consists of one or more configurations. The algorithm begins with an empty solution

Algorithm 8: Greedy Search Algorithm

Input: Set of hot loops with custom instructions: L

Loop Trace: T

Maximum Area of a configuration: MaxA

Reconfiguration Cost: ρ

Result: Partitioning solution

current := **new_configuration**();

continue := *true*;

while continue = true **do**

  C := **compute_reconfig_cost_for_unselected_loops**($L, T, solution, current$);

  $l_{i,j}$:= **select_most_profitable_feasible_CIS**($C, L, MaxA, current$);

  if $l_{i,j}$ *is not found* **then**

    if current *is not empty* **then**

      update *solution* by adding *current*;

      current := **new_configuration**();

    else

      continue := *false*;

    end

  else

    update *current* with $l_{i,j}$;

    remove from L all CIS versions of loop l_i;

  end

end

return *solution*

and an empty current configuration.

In each iteration, we pre-compute a reconfiguration cost array C. For any unselected loop l_i, the array C gives the expected additional reconfiguration cost if l_i is added to the current configuration. Given C, the current solution and the current configuration, we can now compute the expected performance gain of each CIS version if we add it to the current configuration. For CIS version $l_{i,j}$, this expected performance gain is estimated by subtracting from $gain_{i,j}$, the additional reconfiguration cost for loop l_i (available from array C). We now select the CIS version with the maximum expected *positive* performance gain that can be added to the current configuration without violating the area constraint. The selected CIS version is then added to the current configuration. All the other CIS versions of the same loop are subsequently removed from the set L.

In the event that no CIS version can be selected, there are two possibilities. The first possibility is that no more loops can be added to the current configuration without violating the area constraint (*current* configuration is not empty in Algorithm 8). In this case, we update the solution with the current configuration and re-start the process of selecting CIS versions with an empty configuration. The second possibility is that no more loops can be added to the current solution without decreasing its net performance gain (*current* configuration is empty, i.e., we are trying to select the CIS version under maximum area constraint). In this case, the algorithm stops and returns the solution built so far.

6.4.1 Efficiency and Scalability of Algorithms

For this set of experiments, we generate synthetic inputs with number of hot loops ranging from 5 to 100. The number of CIS versions for each loop is generated randomly and ranges between 1 to 10. The performance gain of each CIS version ranges between $1,000$ to $10,000$ time units. The hardware area is between 1 to 100 units. The performance gain

increases with hardware area for each loop.

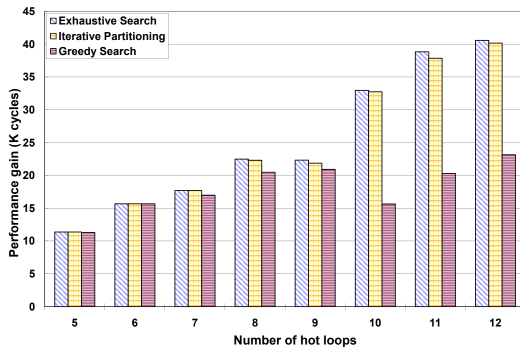
Number of Hot Loops	Running time (sec)		
	Exhaustive search	Greedy search	Iterative partitioning
5	0.26	0.01	0.07
6	1.34	0.02	0.07
7	7.84	0.01	0.07
8	43.91	0.01	0.09
9	283.22	0.04	0.07
10	1788.20	0.01	0.11
11	12604.33	0.01	0.13
12	86338.37	0.01	0.15
20	N.A.	0.02	0.48
40	N.A.	0.04	4.30
60	N.A.	0.07	18.25
80	N.A.	0.11	55.61
100	N.A.	0.16	118.76

Table 6.1: Running time of the algorithms for synthetic input.

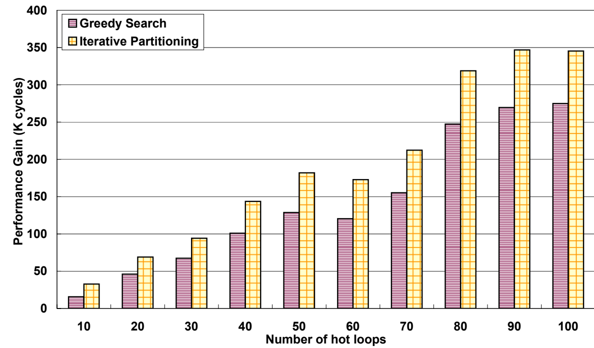
The reconfiguration costs between two loops, if they are assigned to different configurations, are generated randomly. They are in the range 0 to *maxCost* where *maxCost* is approximately 40-50% of the average performance gain of all the CIS versions of all the loops $\frac{\sum_{i=1}^{N}\sum_{j=1}^{n_i} gain_{i,j}}{\sum_{i=1}^{N} n_i}$. The value of *maxCost* ensures that the reconfiguration cost is neither too high nor too low. Both the extremes reduce the search space considerably. If the reconfiguration cost is too high, we should only consider partitions with a small number of

configurations. If the reconfiguration cost is too low, then the solution is to simply select the CIS version with the highest speedup for each loop and construct as many configurations as required. The hardware area constraint $MaxA$ is approximately 20-30% of the sum of the average area requirements of the CIS versions of all the loops $\sum_{i=1}^{N} \frac{\sum_{j=1}^{n_i} area_{i,j}}{n_i}$. This set-up ensures that all the loops with their CIS versions cannot fit under the area constraint.

Table 6.1 shows the running times of the three algorithms for synthetic input with different number of hot loops. The running time of the exhaustive search algorithm, while relatively small with smaller number of loops, increases by almost an order of magnitude each time one more loop is considered. The results of exhaustive search for more than 12 loops cannot be obtained even after waiting for a day. On the other hand, although iterative partitioning algorithm is slower than greedy search in general, its running time is quite acceptable (less than 2 minutes). This result demonstrates the scalability of our approach. Moreover, iterative partitioning generates much better quality solutions compared to greedy search as presented in the following.



(a) Comparison of performance gain of the algorithms for input with 5-12 hot loops.



(b) Relative performance gain of iterative partitioning compared to greedy search.

Figure 6.8: Comparison of the quality of the solutions returned by the algorithms for synthetic input. Exhaustive search fails to return any solution with more than 12 hot loops.

Figure 6.8(a) compares the quality of the solutions returned by the three different algorithms with number of hot loops varying from 5 to 12. The performance gain obtained using our approach is close to the optimal gain obtained with exhaustive search while greedy search falls far behind. Figure 6.8(b) presents the comparison between the performance gain of iterative partitioning and greedy search for input with more than 12 hot loops. We cannot report the results for exhaustive search algorithm here as exhaustive search fails to return any solution for more than 12 loops (even after running for more than a day). The iterative algorithm consistently outperforms greedy search in terms of performance gain by a factor of 1.26 to 2.09.

6.4.2 Case Study of JPEG Application

We present a case study of the JPEG image compression algorithm. In this study, we envision a scenario in which an image is encoded and then decoded subsequently. The hot loops are profiled and the loop trace is generated using an in-house tool based on OpenImpact [2], an open source compiler. The profiling works in two phases. The timing information of each loop is collected by inserting appropriate time stamps at the entry and exit points of the loops. After the first pass, loops which take up more than 1% of the computation time can be detected. During the second pass, the compiler inserts appropriate code to capture the entry point of the hot loops. The resulting application, when executed, generates a trace of the hot loops.

Our loop profiler identifies more than 15 hot loops for the JPEG application. For our experimental purposes, we select the top 10 loops and manually generate custom instruction set versions for each loop on the Stretch S5 platform [88]. Figure 6.9 shows an example of exploiting custom instructions on Stretch processor for performance enhancement of an application. The original loop is shown on the left of the figure. It performs conversion

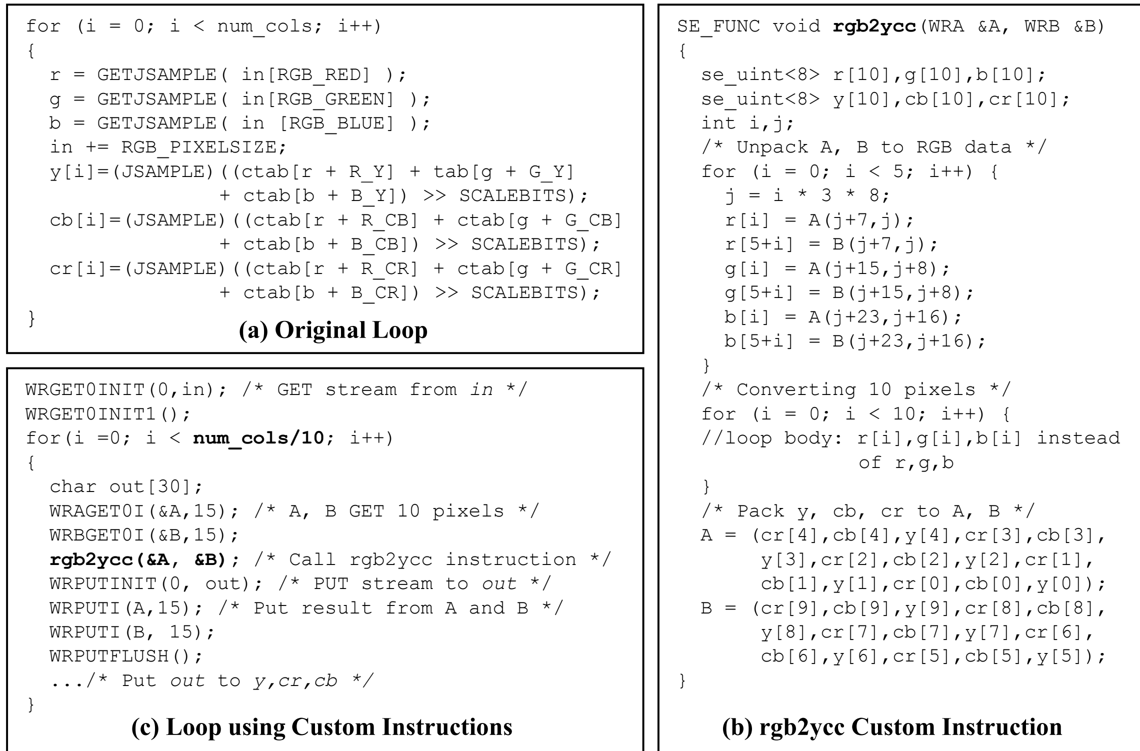


Figure 6.9: An example of custom instruction for Stretch processor.

from RGB color space to YCbCr color space. The original loop converts one pixel at a time. However, the main benefit of custom instructions in Stretch comes from exploiting instruction-level parallelism. Therefore the original loop is unrolled X times ($X = 10$ in our example) to expose more instruction-level parallelism. We can achieve different custom instruction versions (or CIS versions) by changing the unroll factor. The higher unroll factor results in larger hardware area requirement and better performance gain.

Figure 6.9(b) shows an example of using Stretch C language to define a custom instruction corresponding to the loop body (unrolled 10 times). Stretch C is a variant of C language that allows the designer to define custom instructions. The Stretch C compiler can automatically synthesize the custom instructions into the fabric. The custom instruction, called `rgb2ycc`, has two 128-bit wide registers, A and B, as in-out arguments. A and B contain 10 RGB pixels that will be converted to YCC pixels. First, input data in A and B are

unpacked to the local RGB pixel variables. Then RGB pixels are converted to YCC pixels through a `for` loop. The Stretch C compiler, while synthesizing the custom instruction into hardware, will unroll this `for` loop within the custom instruction. As a result , the 10 pixels will be converted in parallel in hardware. Finally, YCC pixels are packed into *A* and *B* registers as the output. After the new custom instruction is defined, we have to change the source code of the original loop to use the newly defined custom instruction (see Figure 6.9(c)). The wide register arguments of `rgb2ycc`, *A* and *B*, get 10 RGB pixels at a time from stream *in*. However, constant tables (such as `ctab`) used in `rgb2ycc` can be hard code into the fabric. Finally, we execute `rgb2ycc` and extract the output from *A* and *B*. For number of RGB pixels less than 10, we perform normal computation without custom instruction.

Loop ID	**(#AUs, #MUs, Gain (K cycles))**		
0	(2249, 4096, 32)		
1	(1612, 2880, 563)	(257, 704, 111)	(389, 2176, 254)
2	(2004, 6272, 556) (761, 1504, 230)	(1041, 2048, 387)	(1321, 2592, 426)
3	(207, 0, 493)	(424, 2, 549)	
4	(2515, 1536, 1094)		
5	(1530, 3584, 1669)	(1300, 3584, 1643)	
6	(981, 4480, 1095)	(491, 2240, 739)	(393, 1792, 590)
7	(1059, 2880, 511)		
8	(1089, 2880, 910)		
9	(1764, 1280, 194)	(1114, 768, 188)	

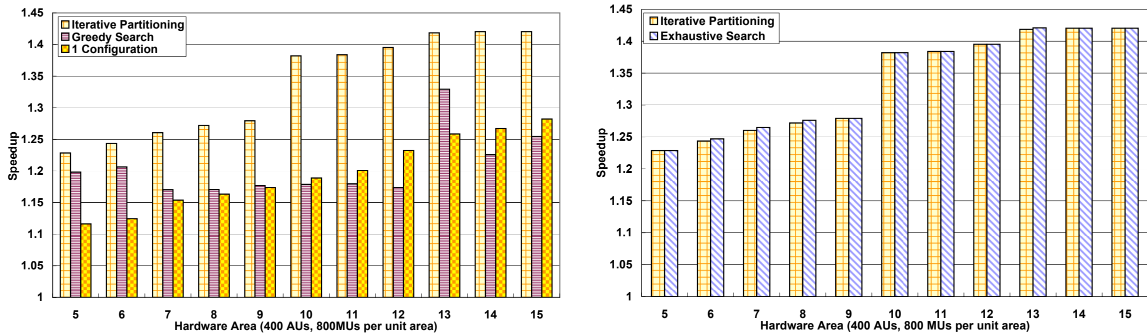
Table 6.2: CIS versions for JPEG application.

The profiler in Stretch IDE can now provide us the performance gain and hardware area of the CIS versions of each loop. Table 6.2 shows the various CIS versions for each loop and their respective area requirements and performance gain. It is worth noting that the performance gain of the CIS versions do not commensurate with area increase in general. For example, loop 0 takes up 2249 arithmetic units and 4096 multiplier units but only gives 32K cycles of performance gain. In contrast, the CIS versions of loop 3 use far less area but give much better performance. This result is because the parallelism that can be exploited varies from one loop to another.

The configuration time of the whole fabric of Stretch development board, which includes 4096 4-bit arithmetic units (AUs) and 8192 4-bit $\times$ 8-bit multiplier units (MUs) is approximately 100μs. Given that the CPU runs at 300MHz, the configuration time translates to roughly 30K CPU cycles. We define *one hardware area unit* to be a tuple of 400 AUs and 800 MUs. Since the configuration time is proportional to the size of the fabric, configuration time of one hardware area unit is approximately 3K CPU cycles. By scaling the configuration time according to the fabric size, we can easily compute the configuration time for any fabric size.

It is possible to fit CIS versions of all the hot loops from our JPEG application in a suitably-sized fabric. For our experimental purposes, we assume that the hardware area constraint varies from one hardware area unit to 20-30% of the sum of maximum hardware area for all the loops ($5 - 15$ hardware units for JPEG application). This set-up will lead to the necessity of dynamic reconfiguration. We run all the three algorithms (exhaustive search, greedy search and iterative partitioning) under these different area constraints. Our profiling data indicates that the application takes around 20 million cycles on Stretch CPU without custom instructions enhancements. It should be noted, however, that the speedup we obtain for a particular application depends on the quality of the custom instructions

generated in the first place. Our focus in this experiment is to evaluate our proposed algorithm in comparison with greedy search and exhaustive search. The purpose is, we are only concerned about comparing the performance gain obtained using the different algorithms starting with the same set of CIS versions. Our results show that the our proposed algorithm is always optimal or near-optimal and produces much better results than greedy search most of the time.



(a) Comparison of iterative partitioning, greedy search, and static configuration (1 configuration).

(b) Comparison of exhaustive search and iterative partitioning.

Figure 6.10: Comparison of the quality of solutions for the case study of JPEG application.

In Figure 6.10(a), we evaluate the performance gain possible if dynamic reconfiguration is exploited. We compare the performance gain obtained using iterative partitioning and greedy search with the case when no reconfiguration is allowed. Clearly, iterative partitioning and greedy search can choose to use more than one configuration. However, the algorithm for static configuration only performs spatial partitioning. If we compare the results of our algorithm with that of static configuration, the advantage of exploiting dynamic reconfiguration decreases as the hardware area increases. This result is to be expected, as more custom instructions can fit into the larger area to gain suitable speedup, thus reducing the need to virtualize hardware through run-time reconfiguration. The graph demonstrates

that our algorithm increases the performance gain over and above static configuration by at least 34% and as much as 78%.

On the other hand, the simple heuristic of the greedy search algorithm fails to achieve substantial performance gain over static configuration. Often, the greedy search performs as good as the static configuration, and in some cases, even worse. Our proposed iterative partitioning algorithm always performs better than the greedy search, being at least 14% and as much as 91% better than greedy search.

Figure 6.10(b) measures how closely our proposed algorithm approximates the optimal results obtained through exhaustive search. The graph shows that our algorithm returns solution that coincides with the optimal solution most of the cases, and falls short of the optimal by at most 1% in the remaining cases.

6.5 Summary

We have presented an algorithm to exploit dynamically configurable custom functional units for optimal performance gain. Given an input application, the algorithm selects and partitions the custom instructions corresponding to the loop kernels into different configurations that are reconfigured at run-time. The experimental results show that our algorithm is highly scalable while producing optimal or near-optimal performance gain.

Chapter 7

Runtime reconfiguration of custom instructions for multi-tasking embedded systems

In Chapter 6, we presented a framework for runtime reconfiguration of custom instructions in the context of sequential application. In this chapter, we return to customization for multi-tasking embedded applications and explore runtime reconfiguration in this context. We assume that the application is specified as a set of task graphs (consisting of a number of tasks with dependencies among them), each associated with a period and a deadline. We only consider static non-preemptive schedules. Our objective is to minimize the processor utilization through appropriate selection of custom instructions for each task and the reconfiguration points while ensuring that all the timing constraints are satisfied.

Our problem formulation corresponds to choosing a design point from a large design space due to (a) the choice of multiple custom instruction candidates per task from which only a subset is selected, and (b) dynamic reconfiguration opportunity that leads to both

spatial and temporal partitioning of the selected set of custom instructions. More importantly, the selected design points should respect real-time constraints. Previous works in processor customization as well as coarse-grained hardware acceleration with reconfigurable logic consider only some restricted versions of our problem. Therefore, the design space exploration approaches proposed in the literature are not directly applicable in our context.

We decide to decouple the task scheduling problem from custom instructions selection. We first employ standard techniques to come up with a task schedule without considering optimizations through custom instructions. Given this task schedule as input, we propose a pseudo-polynomial time algorithm to select custom instructions for each task and the reconfiguration points with the objective of minimizing processor utilization, while meeting the deadline constraints. Our algorithm returns the optimal feasible solution (if one exists) corresponding to the input task schedule.

The highlights of our solution can be summarized as follows.

- Given a fixed task schedule, our pseudo-polynomial time algorithm returns the optimal selection of custom instructions that is feasible (satisfies real-time constraints) and minimizes the processor utilization.

- We exploit both spatial partitioning (allocation of custom instructions to the tasks within a configuration) and temporal partitioning of the tasks (partitioning of the tasks into multiple configurations).

- We account for the reconfiguration cost.

- Our decoupled task scheduling and custom instructions selection approach is scalable and still returns close to the optimal solution.

The remainder of this chapter is structured as follows. In Section 7.1, we present the problem formulation. Section 7.2 details our partitioning algorithm. Experimental setup and evaluation are described in Section 7.3.

7.1 Problem Formulation

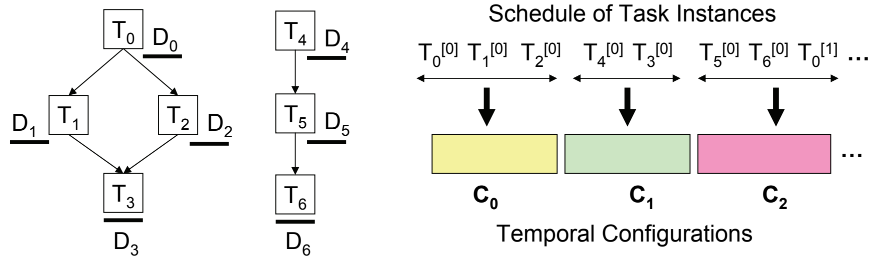


Figure 7.1: A set of periodic task graphs and its schedule

We model the application as a set of periodic task graphs (refer Figure 7.1), which has been widely used in previous works [26, 30, 62, 72]. Each task graph is a directed acyclic graph consisting of a number of tasks. Let $\{T_0, \ldots, T_{N-1}\}$ be the set of N tasks corresponding to all the task graphs. A directed edge between two tasks $T_i \rightarrow T_j$ in a task graph denotes that task T_j can start execution only after task T_i completes execution. Let e_i denote the execution time of T_i in software, i.e., without any optimization through custom instructions. Each task graph has a deadline less than or equal to its period. The deadline D_i of T_i is the latest finish time of T_i derived from a backward topological search of the corresponding task graph starting with the sink node (whose deadline coincides with the task graph deadline).

The underlying processor platform allows optimized implementation of the tasks by exploiting custom instructions. Multiple custom instruction-set (CIS) versions are generated

for each task with a trade-off between hardware area and performance gain. A CIS version consists of a set of custom instructions extracted from the corresponding task under an area constraint. In general, the performance gain of a CIS version increases with larger area. Let $\{v_i^0, \ldots, v_i^{M_i}\}$ denote the possible CIS versions of task T_i. In addition, let g_i^k and a_i^k denote the performance gain and area requirement of the version v_i^k. We assume v_i^0 corresponds to the software implementation, i.e., $g_i^0 = 0$ and $a_i^0 = 0$. In other words, for each task T_i, we have a choice of one software implementation and M_i implementations accelerated with custom instructions. The area A available for implementation of the CFUs can be reconfigured at runtime to support a different set of custom instructions. In this chapter, we focus on inter-task reconfiguration and do not consider intra-task reconfiguration. So the CIS version of a task must fit into the available area without reconfiguration, i.e., $a_i^k \leq A$.

Our objective is to come up with a static non-preemptive schedule of the task set that minimizes processor utilization by exploiting (a) processor customization and (b) runtime reconfiguration of the custom instructions, while satisfying deadline constraints. We need to construct our static schedule for the *hyper-period (HP)*, which is the least common multiple of the task graph periods. All the tasks in a task graph have the same period. Let P_i denote the period of task T_i. Clearly, a task T_i has $\frac{HP}{P_i}$ instances within the hyper-period. The sth instance of T_i, denoted as $T_i^{[s]}$, has the deadline

$$deadline(T_i^{[s]}) = D_i + s \times P_i \tag{7.1}$$

In a feasible schedule, all the task instances meet their deadlines.

To minimize processor utilization, we need to assign appropriate CIS version to each task instance in the schedule. However, as we can exploit runtime reconfiguration of the custom instructions, we need not restrict ourselves to the area constraint A. Instead, we can perform *temporal partitioning* of the schedule into C configurations, where area constraint A is imposed on each configuration. For example, Figure 7.1 illustrates an initial portion

of the schedule and its partitioning into three configurations. Note that each configuration contains a disjoint subsequence of task instances from the original schedule. Temporal partitioning allows us to work with a larger *virtual area* at the cost of a delay ρ per reconfiguration. The area A within a configuration is *spatially partitioned* among the task instances assigned to it by choosing appropriate CIS version for each task instance.

A feasible solution to this problem is a static, non-preemptive schedule of the task instances over the hyper-period where (a) the schedule is partitioned into C configurations, (b) each task instance is assigned to an appropriate CIS version, (c) the total area requirement of the chosen CIS versions within a configuration satisfies area constraint A, (d) each task instance satisfies its deadline constraint given by Equation 7.1, and (e) task dependence constraints are satisfied. The processor utilization U over the hyper-period HP for this solution can be expressed as

$$U = \frac{\left(\sum_{i=0}^{N-1} \frac{HP}{P_i} \times e_i\right) - \left(\left(\sum_{i=0}^{N-1} \sum_{s=0}^{\frac{HP}{P_i}-1} gain(T_i^{[s]})\right) - \rho * (C-1)\right)}{HP} \tag{7.2}$$

where $gain(T_i^{[s]})$ is the performance gain of the sth instance of task T_i based on its assigned CIS version. As stated before, our objective is to construct the solution that minimizes U. In other words, we try to maximize the performance gain minus the reconfiguration cost

$$maximize \quad \left(\sum_{i=0}^{N-1} \sum_{s=0}^{\frac{HP}{P_i}-1} gain(T_i^{[s]})\right) - \rho * (C-1) \tag{7.3}$$

7.2 Algorithm

The problem defined in Section 7.1 consists of two sub-problems, namely, task scheduling and CIS version assignment. Design of optimal task scheduling algorithm is not the focus of this chapter. Instead, we employ *list scheduling* and use deadlines of task instances as the scheduling priority, i.e., a task instance with earlier deadline has higher priority. Still temporal partitioning of the resulting schedule into multiple configurations and assigning appropriate CIS versions to the task instances within each configuration with the objective of minimizing processor utilization (Equation 7.2), while satisfying all deadline constraints (Equation 7.1) is a non-trivial problem. In this section, we present an elegant solution based on dynamic programming.

List scheduling employed on the task graphs (as shown in Figure 7.1) over the hyper-period constructs a linear schedule of the task instances with possible idle periods in between. Let $\langle T_0, T_1, \ldots, T_X \rangle$ be the resulting schedule of task instance where $X = \sum_{i=0}^{N-1} \frac{HP}{P_i}$. *For simplicity of exposition, we ignore the superscripts for task instances in the rest of the chapter.* If all the task instances can meet their deadlines in this schedule without any hardware acceleration, then we can guarantee that reduction in execution time of a task with custom instructions will still maintain schedulability. The problem gets a little simplified in this case. But if some of the task instances fail to meet deadlines, then our first priority is to ensure schedulability through hardware acceleration.

Running Example Throughout this section, we use a simple example to illustrate our algorithm and convey the intuition behind it. Let us assume a schedule consisting of three task instances $\langle T_0, T_1, T_2 \rangle$. The deadlines D_i and execution times e_i of the software implementation of the tasks appear in the table in Figure 7.2. Clearly, all the tasks will miss their deadlines with the software implementations as shown in Figure 7.2 (a.1). Therefore, we

would like to explore processor customization so as to reduce execution times of the tasks and meet the deadlines. The table in Figure 7.2 also shows that each task T_i has three CIS versions v_i^0, v_i^1, v_i^2 with varying area and performance gain (e.g., 3,2 for v_0^2 denotes 3 units of area and 2 units of performance gain). The version v_i^0 is the software implementation (zero hardware area and zero performance gain).

7.2.1 A Simple Solution

Let us for the moment ignore reconfiguration cost, configuration boundaries, and deadline constraints. Our objective is to find an assignment of CIS versions to the tasks to achieve maximum performance gain (given by Equation 7.3) under a *virtual area* constraint. Given a virtual area *area*, let us define the maximum performance gain of the sequence $\langle T_0, \ldots, T_i \rangle$ as $G_i(area)$. If we ignore reconfiguration cost and configuration boundaries, we can compute $G_i(area)$ for different values of *area* through dynamic programming. $G_i(area)$ can be defined recursively as

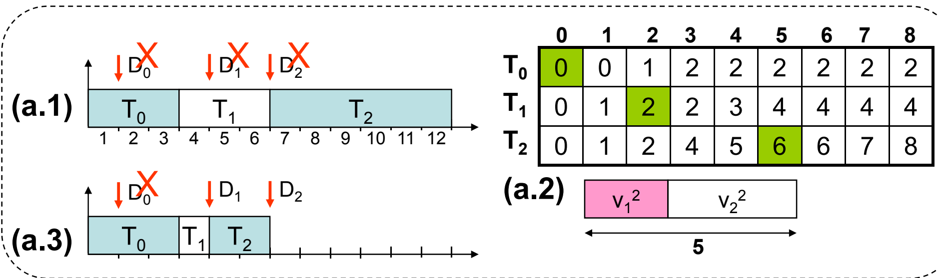
$$G_i(area) = \max_{\substack{k=0,\ldots,M_i \\ a_i^k \leq area}} \left(g_i^k + G_{i-1}(area - a_i^k) \right) \tag{7.4}$$

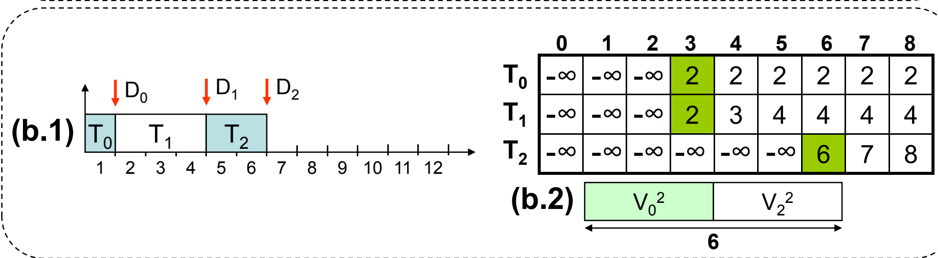
That is, given a virtual area *area*, we explore all possible CIS versions of T_i and choose the one that results in maximum performance gain for $\langle T_0, \ldots, T_i \rangle$. The base case for task T_0

$$G_0(area) = \max_{\substack{k=0,\ldots,M_0 \\ a_0^k \leq area}} g_0^k \tag{7.5}$$

Example Figure 7.2 (a.2) shows the performance gains for the tasks under area constraints 0 to 8 where 8 is the area required to implement the best CIS versions of all the tasks. Total execution time of T_0, T_1, T_2 in software is 12 time units whereas the entire sequence should complete execution within 6 time units. The solution table indicates that we

Task Instance (T_i)	Deadline	Execution Time	v_i^0 (a_0^k, g_0^k)	v_i^1 (a_1^k, g_1^k)	v_i^2 (a_2^k, g_2^k)
T_0	1	3	0,0	2,1	3,2
T_1	4	3	0,0	1,1	2,2
T_2	6	6	0,0	1,1	3,4





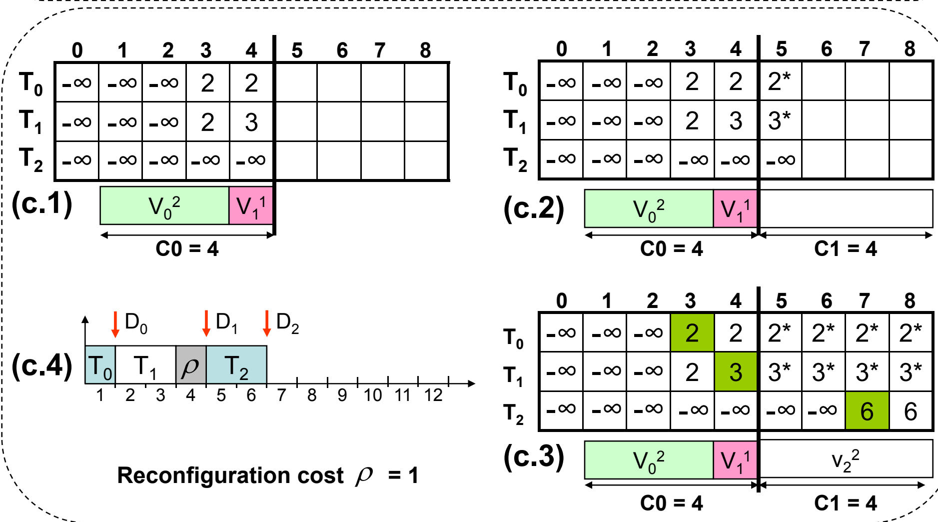


Figure 7.2: Running Example

can obtain a performance gain of 6 time units for the task sequence with 5 units of area. The solution cells corresponding to this performance gain are shaded in Figure 7.2 (a.2); the CIS versions chosen are v_0^0, v_1^2, and v_2^2. Unfortunately, the first task T_0 fails to meet its deadline because it is implemented purely in software as shown in Figure 7.2 (a.3) (execution time = 3 while deadline = 1). This example clearly shows that we cannot ignore the deadline constraints of the individual tasks (T_0 and T_1) while constructing the solution to maximize performance gain.

7.2.2 Deadline Constraints

The recurrence defined by Equation 7.4 does not take into account the deadline constraints. Let us now proceed to modify this equation so as to maximize performance gain while satisfying deadline constraints. We will continue to ignore reconfiguration at this point.

Given a virtual area constraint *area*, we find the solution with the maximum performance gain $G_i(area)$ and each task $T_0,\ldots,T_i$ is assigned one of its CIS versions. The solution is *feasible* if all the tasks $T_0,\ldots,T_i$ can meet their deadlines with the CIS version assignments in the solution. To satisfy the deadline constraints, we modify the construction of dynamic programming solution table with the following consideration. While exploring CIS versions of task T_i, we need to choose the solution that returns best $G_i(area)$ and $T_0,\ldots,T_i$ meet their deadlines. So how do we impose this constraint? Equation 7.4 is now modified as

$$G_i(area) = \max_{\substack{k=0,\ldots,M_i \\ a_i^k \leq area \\ is_schedulable(T_i)}} \left(g_i^k + G_{i-1}(area - a_i^k)\right) \tag{7.6}$$

Here, is_schedulable(T_i) simply checks the deadline constraints for $T_0,\ldots,T_i$. In fact, as we ensure the sequence $\langle T_0,\ldots,T_{i-1}\rangle$ is already schedulable, we only need to check that

T_i meets its deadline. If we cannot find any CIS version assignment for T_i to make the sequence $\langle T_0, \ldots, T_i \rangle$ schedulable, we set $G_i(area) = -\infty$.

Example In our example, the feasible schedule is shown in Figure 7.2 (b.1) and the solution table is shown in Figure 7.2 (b.2). When $area < 3$, $G_0(area) = -\infty$ which shows T_0 misses its deadline. Clearly, $G_1(area)$, $G_2(area)$ are also equal to $-\infty$ when $area < 3$. The difference between Equation 7.4 and Equation 7.6 becomes clear by looking at the last row in Figure 7.2 (b.2). For example, when $area = 5$, we cannot find any CIS version for T_2 to make it schedulable and $G_2(5) = -\infty$ instead of $G_2(5) = 6$ in Figure 7.2 (a.2). The shaded cells in Figure 7.2 (b.2) provide the optimal solution that satisfies all the deadline constraints. Here T_0 selects v_0^2, T_1 selects v_1^0 (implemented in software) and T_2 selects v_2^2.

7.2.3 Runtime Reconfiguration

So far we assume that the entire virtual area is available as a single continuous configuration. However, in reality, the virtual area is divided into a number of configurations and reconfiguration cost is incurred while switching from one configuration to another. Suppose the area constraint for a single configuration $A = 4$ in our example. Let us now investigate the optimal solution returned in the previous subsection where T_0 selects v_0^2 (3 unit area), T_1 selects v_1^0 (implemented in software) and T_2 selects v_2^2 (3 unit area) in Figure 7.2 (b.2). This solution is no longer feasible for the following reasons

- A task instance should be mapped to only one configuration; it cannot straddle across configuration boundaries. In our example, task T_2 occupies 1 unit of area in the first configuration and 2 units of area in the second configuration.

- The reconfiguration cost should be taken into account while computing performance gain.

Restricting a task instance to one configuration How do we handle the constraint that a task cannot straddle across configuration boundaries? Given a virtual area *area*, the number of configurations is $C = \lceil \frac{area}{A} \rceil$ and the area available in the last configuration *physical_area* is

$$physical_area = \begin{cases} A & \text{if } area \bmod A = 0 \\ area \bmod A & \text{otherwise} \end{cases} \tag{7.7}$$

When exploring the CIS versions of task T_i under area constraint *area*, we should now impose the constraint that the available area is less than the *physical_area*, i.e., the area of the current configuration. We modify Equation 7.6 to reflect this.

$$G_i(area) = \max_{\substack{k=0,\dots,M_i \\ a_i^k \leq physical_area \\ is_schedulable(T_i)}} \left(g_i^k + G_{i-1}(area - a_i^k) \right) \tag{7.8}$$

Reconfiguration cost We now need to subtract the reconfiguration cost from the total performance gain under the following conditions.

1) If the area requirement of a CIS version is equal to the area of the current configuration, i.e., $a_i^k = physical_area$ and $C > 1$, then T_i is the *first* task in the current configuration. We should subtract the reconfiguration cost from the gain.

2) The reconfiguration cost offsets the performance gain of the CIS version chosen for task T_i. Hence, $T_0, \dots, T_i$ may have obtained greater performance gain when reconfiguration was not involved. That is, it is possible to have $G_i(area) \leq G_i(area - physical_area)$. In this case, it does not make sense to perform reconfiguration before task T_i and we should instead select the solution with gain $G_i(area - physical_area)$. Even if $G_i(area)$ is equal to $G_i(area - physical_area)$, we still prefer the solution $G_i(area - physical_area)$ as it is better not to use the current configuration, if possible. The fact that a solution does not use the current configuration is represented visually with a '*' in Figure 7.2 and maintained

as a binary variable $reconfig_i(area)$. If the solution for tasks $T_0, \ldots, T_i$ under $area$ has not used any portion of the current configuration, then $reconfig_i(area) = false$; otherwise we set $reconfig_i(area) = true$.

3) Suppose in Equation 7.8, we use the partial solution $G_{i-1}(area)$ where $reconfig_{i-1}(area) = false$ (marked with a *), i.e., the solution did not use the current configuration. If we combine this solution with a CIS version of T_i, the implication is that T_i is the first task to use the current configuration. Therefore, reconfiguration cost should be subtracted from the total performance gain.

Algorithm 9: **Compute** $\mathrm{G_i(area)}$

```
1  C ← ⌈area/A⌉;
2  physical_area ← { A               if area mod A = 0
                   { area mod A      otherwise
3  G_i(area) ← −∞;  reconfig_i(area) ← false;
4  for k = 0 to M_i do
5      if a_i^k ≤ physical_area then
6          gain ← g_i^k + G_{i−1}(area − a_i^k);
7          reconfiguration ← false;
8          if C > 1 AND (a_i^k = physical_area OR !reconfig_{i−1}(area − a_i^k)) then
9              gain ← gain − ρ;  reconfiguration ← true;
10         if is_schedulable(T_i) AND gain > G_i(area) then
11             G_i(area) ← gain;  reconfig_i(area) ← reconfiguration;
12 if C > 1 AND G_i(area) ≤ G_i(area − physical_area) then
13     G_i(area) ← G_i(area − physical_area);  reconfig_i(area) ← false;
```

The modification of Equation 7.8 to take reconfiguration cost into account is easier to present in an algorithmic form as shown in Algorithm 9.

Example Now let us get back to our running example. Tasks T_0, T_1, T_2 cannot have a feasible solution when we restrict ourselves to one configuration (4 units of area) and take schedulability constraints into account (Figure 7.2 (c.1)). Let us now look at performance gain with 5 units of area in Figure 7.2 (c.2). Task T_0 cannot obtain any further performance gain. Therefore, its solutions is marked with '*' in the second configuration indicating that T_0 belongs to the previous configuration.

The situation gets interesting with T_1. If T_1 is implemented in the second configuration, it can get a maximum gain of 2 time units. However, we need to subtract reconfiguration cost of 1 time unit. As T_0 has a gain of 2 time units, the total performance gain for T_0, T_1 with two configurations is only $2+2-1=3$. On the other hand, we can easily get a gain of 3 units by implementing both T_0 and T_1 in the first configuration as shown by the shaded cells in Figure 7.2 (c.3). Therefore, it does not make sense to put T_1 into the second configuration and its cell is marked with '*'.

Finally, T_2 fails to meet its deadline in the beginning by using the second configuration as reconfiguration cost overshadows the performance gain. However, when $area = 7$, T_2 can implement its best CIS version in the second configuration with 4 units of performance gain (Figure 7.2 (c.3)). T_0, T_1 gets 3 units of gain from the first configuration. Therefore, total performance gain is $3+4-1=6$. At this point, we have been able to construct a solution that satisfies all the timing constraints as shown in Figure 7.2 (c.4).

7.2.4 Putting It All Together

We can now present our complete dynamic programming (called *DP*) algorithm (Algorithm 10) that satisfies deadline constraints as well as takes into account runtime reconfiguration.

Let $X = \sum_{i=0}^{N-1} \frac{HP}{P_i}$ be total number of task instances over the hyper-period. We vary *area* in steps of Δ to the area required to implement the best CIS versions of all task

Algorithm 10: **Maximize Performance Gain**

```
1 for area = Δ to Max_A in steps of Δ do
2     for i=0 to X − 1 do
3         if ∀j ≤ i  !reconfig_j (⌊area/A⌋ × A) then
4             G_i(area) = G_i (⌊area/A⌋ × A) ;
5         else
6             compute G_i(area);
7     if area mod A = 0   AND   ∀ i   !reconfig_i(area) then
8         break;
9 return G_{X−1}(area);
```

instances Max_A. For each $area$, we do not compute $G_i(area)$ if performance gains of $\langle T_0, \ldots, T_i \rangle$ have no improvement compared to the preceding configuration $\left(\lfloor \frac{\mathrm{area}}{\mathrm{A}} \rfloor \times \mathrm{A}\right)$ (line 3). Therefore, $G_i(area)$ should be filled up with the solution from the preceding configuration (line 4) as performance gain is guaranteed to have no improvement in the current configuration either. In Figure 7.2 (c.3), performance gains of $\langle T_0, T_1 \rangle$ have no improvement in configuration C_1. Therefore, the performance gain of $\langle T_0, T_1 \rangle$ will remain unchanged in all the future configurations. We compute $G_i(area)$ through Algorithm 9 (line 6). Finally, if performance gains of $\langle T_0, \ldots, T_{X-1} \rangle$ have no improvement at the end of the current configuration, we will stop the algorithm (lines 7-8). This is because we cannot get any additional performance gain by exploring further configurations.

Algorithm Complexity For each task instance, we compute $G_i(area)$ (Algorithm 9) with $area = 0 \ldots Max_A$ in steps of Δ. Moreover, for each $G_i(area)$ we have $(M_i + 1)$ choices of CIS version. Let $M_{max} = \max_{i=0 \ldots N-1} (M_i + 1)$. Therefore, the worst case complexity of our algorithm is $O(X \times \frac{Max_A}{\Delta} \times M_{max})$.

7.3 Experimental Evaluation

To evaluate the accuracy and scalability of our dynamic programming algorithm, we also develop an Integer Linear Programming (ILP) solution for our problem. We note that the optimal ILP formulation is non-trivial as it includes task scheduling, CIS version assignment, and runtime reconfiguration. However, the ILP solution is not scalable as will be evident in the experimental evaluation.

7.3.1 ILP Formulation

We define the ILP formulation to find the optimal solution under the maximum number of configurations NC. st_i^s and et_i^s are integer variables denoting the start and end time of task instance T_i^s. Let e_i^k be the execution time of the kth CIS version of task T_i and let $optimal_nc$ be optimal number of configuration returned by ILP solution. In addition, $y_{i,s,k,c}$ and $b_{(i,s),(i',s')}$ are binary decision variables defined as follows.

+ $y_{i,s,k,c}$: is equal to 1 if task instance T_i^s selects its kth CIS version, which is partitioned into configuration c. Otherwise, 0.

+ $b_{(i,s),(i',s')}$: is 0 if T_i^s finishes before $T_{i'}^{s'}$. Otherwise 1.

We impose the following constraints.

7.3.1.1 Uniqueness Constraint

Each task instance must select at most one CIS version (including software version):

$$\forall i, s \quad \sum_{c=0}^{NC-1} \sum_{k=0}^{M_i} y_{i,s,k,c} = 1 \tag{7.9}$$

7.3.1.2 Resource Constraint

Total area used for CIS versions of a configuration must be less than or equal to maximum hardware area A of a configuration:

$$\text{For each c}: \sum_{i=0}^{N-1} \sum_{s=0}^{\left(\frac{HP}{P_i}-1\right)} \sum_{k=0}^{M_i} y_{i,s,k,c} * a_i^k \leq A \tag{7.10}$$

7.3.1.3 Scheduling Constraint

A task instance must be scheduled after its release:

$$\forall i, s: \quad st_i^s \geq s \times P_i + 1 \tag{7.11}$$

Start time of a task instance partitioned into configuration $(c > 0)$ must be greater than the end time of the configuration $(c-1)$ plus the reconfiguration cost ρ. The end time of the configuration c, etc_c, is equal to the maximum end time of task instances partitioned into configuration c:

$$\forall i, s: \ if \sum_{k=0}^{M_i} y_{i,s,k,c} = 1 \ \ then \ \ etc_c \geq \sum_{k=0}^{M_i} y_{i,s,k,c} * e_i^k + st_i^s \tag{7.12}$$

$$\forall i, s \ \ \forall c \in [1..NC-1]:$$

$$if \sum_{k=0}^{M_i} y_{i,s,k,c} = 1 \ \ then \ \ st_i^s \geq etc_{c-1} + \rho \tag{7.13}$$

Start time of task $T_{i'}$ is greater than the end time of task T_i if there is an edge (dependency) between T_i and $T_{i'}$ in the same period, $T_i \rightarrow T_i'$, and task T_i, $T_{i'}$ are partitioned into the same temporal configuration:

$$\forall i, k, \ \forall (T_i \rightarrow T_i'), \ \forall c \in [0 \ldots NC-1]: \ \ et_i^s + 1 \leq st_{i'}^s \tag{7.14}$$

Equation 7.15 computes end time of execution of a task instance in configuration c:

$$\forall i, s: \quad et_i^s = st_i^s + \sum_{c=0}^{NC-1} \sum_{k=0}^{M_i} y_{i,s,k,c} * e_i^k \tag{7.15}$$

Every task instance must finish before its deadline:

$$\forall i,s: \quad et_i^s \leq s * P_i + D_i \tag{7.16}$$

We have to serialize two independent tasks that can be executed in parallel in the same configuration: $\forall c \in [0 \ldots NC-1]$, $\forall i,s,i',s'$:

$$if \quad \sum_{k=0}^{M_i} y_{i,s,k,c} = 1 \;\; and \;\; \sum_{k'=0}^{M_{i'}} y_{i',s',k',c} = 1 \quad then$$

$$b_{(i,s),(i',s')} + b_{(i',s'),(i,s)} = 1 \tag{7.17}$$

$$st_i^s \geq et_{i'}^{s'} - \infty * b_{(i,s),(i',s')} + 1 \tag{7.18}$$

$$st_{i'}^{s'} \geq et_i^s - \infty * b_{(i',s'),(i,s)} + 1 \tag{7.19}$$

The optimal number of configurations is the number of configurations that returns the optimal result. It is the maximum of the configurations containing the last instances of the sink tasks of each task graph. Let T_{jsink} be the sink task of jth task graph. Therefore, the last instance of T_{jsink} is in $\left(\frac{HP}{P_{jsink}} - 1\right)^{th}$ release.

$$\forall j: \quad \sum_{c=0}^{NC-1} \sum_{k=0}^{M_{jsink}} y_{jsink, \frac{HP}{P_{jsink}}-1,k,c} * (c+1) \leq optimal_nc \tag{7.20}$$

7.3.1.4 Objective Function

Our objective is to minimize utilization:

$$\frac{\sum_{i=0}^{N-1} \sum_{s=0}^{\left(\frac{HP}{P_i}-1\right)} \sum_{c=0}^{NC-1} \sum_{k=0}^{M_i} y_{i,s,k,c} * e_i^k + (optimal_nc - 1) * \rho}{HP} \tag{7.21}$$

ILOG Concert Technology can help to linearize the non-linear constraints in the formulation such as if statements. Therefore, we do not discuss the details of linearization here.

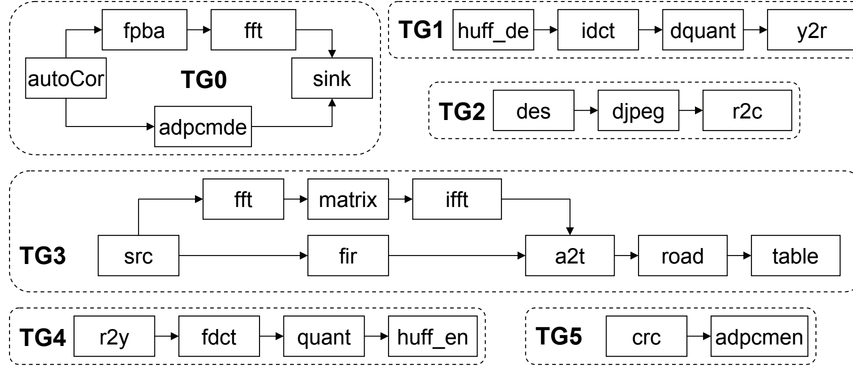


Figure 7.3: Task Graphs

7.3.2 Experimental Setup

Each task graph (see Figure 7.3) used in our experiment combines real kernels from the same application domain to form meaningful benchmarks, such as JPEG decoder (TG1) and encoder (TG4), automotive application (TG3), and consumer electronic applications (TG0, TG2, TG5).

Given each task, custom instruction versions are manually generated for the Stretch S5 platform [88] by using Stetch C language. We can achieve different custom instruction versions (or CIS versions) by changing the unroll factor of the compute-intensive loops within the task or the number of custom instructions (Table 7.1). The higher unroll factor results in larger hardware area requirement and better performance gain. The profiler in Stretch can provide us the performance gain and hardware area of the CIS versions of each task.

We create four combination of task graphs, *A0*, *A1*, *A2*, *A3*, each consisting of two to four task graphs from Figure 7.3 to represent different applications. *A0*, *A1*, *A2*, *A3* consist of {TG1,TG4,TG5}, {TG1,TG3}, {TG0,TG2,TG4,TG5}, and {TG2,TG4,TG5} respectively. To set the periods for the task graphs, we choose a total processor utilization U for the entire system (without any custom instructions) and then select the periods for

Task	CIS versions (#AUs, #MUs, Execution Time K cycles)	
rgb2ycc (r2y)	(0, 0, 1476)	(393, 1792, 886)
	(491, 2240, 737)	(981, 4480, 381)
huff_en	(0, 0, 6683)	(2515, 1536, 5589)
	(1300, 3584, 5040)	(3815, 5120, 3946)
dequant	(0, 0, 900)	(257, 704, 789)
	(389, 2176, 646)	(1612, 2880, 337)
crc	(0, 0, 4462)	(101, 0, 730)
adpcmde	(0, 0, 1338)	(1128, 256, 1107)
djpeg	(0, 0, 2496)	(1430, 4224, 1823)
	(1710, 4768, 1816)	(2933, 5472, 1507)
idct	(0, 0, 473)	(2294, 4096, 441)
adpcmen	(0, 0 , 1882)	(1790, 256, 1328)
rgb2cmyk (r2c)	(0, 0, 2956)	(243, 0, 1320)
	(848, 0, 1312)	
fir	(0, 0, 1369)	(263, 2048, 138)
	(487, 4096, 57)	(1127, 8192, 38)
des	(0, 0, 732)	(1892, 256, 49)
fft	(0, 0, 1285)	(898, 2048, 20)
ifft	(0, 0, 1281)	(898, 2048, 21)
laplacian	(0, 0, 761)	(3122, 1024, 19)

Table 7.1: CIS Versions of the tasks.

each constituent task graph to achieve the corresponding utilization. We vary U between $0.9 - 1.4$ for each scenario. $U > 1$ implies that the application scenario is definitely not schedulable without custom instructions, whereas it may or may not be schedulable with $U \leq 1$.

The configuration time of the whole CFU fabric of Stretch, which includes 4096 4-bit AUs and 8192 4-bit $\times$ 8-bit MUs, is approximately 100μs or roughly 30K CPU cycles at 300MHz core. We define one hardware area unit to be a tuple of 400 AUs and 800 MUs. As configuration time is proportional to fabric size, configuration time of one hardware area unit is approximately 3K CPU cycles. For each application, we vary the CFU fabric size between 10-100% (in steps of 10%) of the maximum area required to implement the best CIS versions of the constituent kernels, Max_A. When maximum area is available, an application explores the limit of speedup achievable though custom instructions without reconfigurations.

Given an application scenario, area constraint and processor utilization, we apply three different techniques to generate a feasible schedule and CIS assignments with minimum processor utilization: (1) our **DP** algorithm proposed in Section 7.2. (2) **Optimal:** an Integer Linear Programming (ILP) formulation that can return the optimal solution. (3) **Static:** this solution restricts itself to a static configuration, i.e., it does not consider dynamic reconfiguration. This is a simplified version of the ILP formulation for *Optimal* that excludes dynamic reconfiguration.

7.3.3 Experimental Results

Figure 7.4 shows the accuracy of our algorithm *DP* compared to *Optimal*. This figure also shows the advantage of runtime reconfiguration (*Optimal* and *DP*) over static configuration (*Static*). *DP* achieves up to 37% better processor utilization compared to *Static* when area

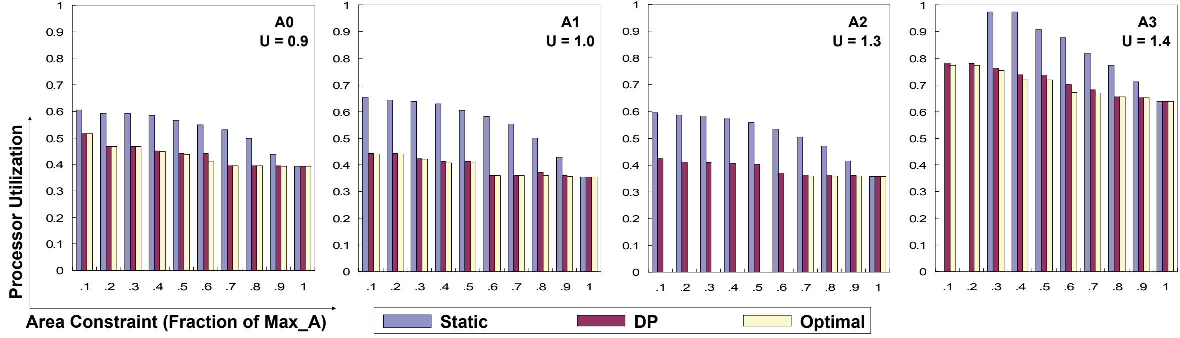


Figure 7.4: Comparison of DP, Optimal, and Static

constraint decreases. This is expected as runtime reconfiguration can fit more custom instructions into the fabric through temporal sharing. Note that for the application *A3*, when the area constraint is really tight, i.e. $0.1 * Max_A$, there does not exist any feasible solution with static configuration *Static*. But feasible solutions can be obtained with runtime reconfiguration. More importantly, the solution returned by *DP* often coincides with the optimal solution. In fact it is mostly within 3% of the optimal processor utilization. Moreover, for the application *A2*, when area constraint is small, we do not get *Optimal* result as ILP solver fails to return any solution.

Running times of both *Optimal* and *DP* depend on the number of task instances in the schedule, *schedule length*. {*A0,A1,A3*} have schedule lengths 11 or 12 while *A2* has schedule length 16. To show the effect of schedule length on running time of both algorithms, we create more task graph sets with schedule lengths varying from 11 to 18. Table 7.2 shows running times of *Optimal* and *DP* on different schedule lengths when input processor utilization is $U = 1$ and area constraint is $0.3 * Max_A$ for different task graphs. The running time of *Optimal*, while relatively small with schedule lengths of {11,12}, shoots up quickly at schedule length 16. The solution for *Optimal* cannot be obtained even after waiting for two days when schedule lengths are greater than 16. Clearly, *DP* is significantly

Schedule Length	Task Graph Sets	Optimal	DP
11	*A3*	19.525920	0.179993
12	*A0*	94.245572	0.246194
13	{TG0,TG2,TG5}	168.509196	0.492661
14	{TG2,TG5}	193.335448	1.182449
15	{TG0,TG1,TG4,TG5}	1273.653911	0.795110
16	*A2*	N/A	0.350204
17	{TG0,TG5}	N/A	0.903613
18	{TG0,TG1,TG2,TG4,TG5}	N/A	1.513959

Table 7.2: Running Time of Optimal and DP in seconds.

more scalable compared to *Optimal*.

7.4 Summary

We propose a pseudo-polynomial time algorithm to efficiently solve the problem of run-time reconfiguration of custom instructions for real-time embedded systems. Minimized processor utilization is achieved through appropriate custom instructions selection as well as temporal partitioning with consideration of reconfiguration cost. Our experiments using real embedded benchmarks on Stretch customizable processor show scalability and accuracy of our algorithm compared to integer linear programming based optimal solutions.

Chapter 8

A case study of processor customization

In the previous chapters, we have concerned ourselves with design methodologies for processor customization. We have evaluated our techniques with benchmark application. We conclude this thesis with a real world case study that exploits processor customization for bio-monitoring application. The increasingly ageing population is posing a major challenge to the overall health-care systems worldwide. Remote and non-obtrusive continuous bio-monitoring of a non-critical patient at home is a viable alternative that can reduce considerable burden on the hospital resources. Wireless body-area sensor networks (or BANs) and related wearable computing technologies promises a convenient platform for such bio-monitoring applications. The recent technological advancements in embedded processors, availability of ultra low-power and lightweight sensor nodes and advances in wireless networking have all paved the ways for wireless BAN platforms. Some of the well-known projects and prototype architectures in this area are the MIThril [29], CustoMed [51], Wearable e-Textiles [31], Wearable Motherboard [77], e-Textile [54], and RFab-Vest [50]. However, continuous monitoring of the vital signs of a patient requires processing a large volume of data streams arriving through multiple sensors. The resource constrained nature of wireless BAN platforms raises significant concerns about meeting both the computation

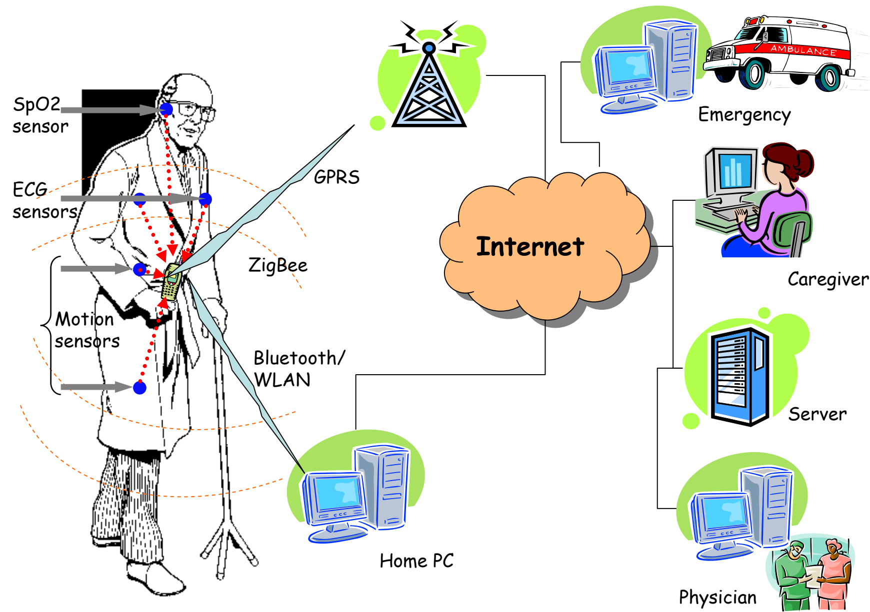


Figure 8.1: Wearable bio-monitoring.

bandwidth and power consumption requirement of high-end bio-monitoring applications. This has fueled lot of interests in designing architectures and software specifically targeted towards wireless BANs in general and wearable bio-monitoring solutions, in particular [32, 51, 52, 60, 61].

Figure 8.1 illustrates the typical architecture of a wearable bio-monitoring platform. Multiple tiny sensor nodes are attached to the different parts of the patient's body. These sensor nodes continuously sample various vital signs, such as ECG (Electrocardiograph), SpO2 (Saturation of Arterial Oxygen) etc., at regular intervals and transmit the collected samples to a gateway device (typically mobile phone or personal digital assistant (PDA)) through wireless medium. The gateway device is also located in the vicinity of the person being monitored such as on his/her body. The sensor nodes communicate with the gateway device through wireless communication protocol such as ZigBee (802.15.4) or Bluetooth (802.15.1). The gateway device is responsible for processing the sampled data streams and detecting emergency conditions (such as a fall) or anomaly in the vital signs. It can employ

mobile telephone networks (GPRS, 3G, etc.) or wireless LAN to reach an Internet access point and thereby trigger an alarm to the care-giver in case of an emergency or anomaly. It also periodically reports the status of the patient to the medical servers.

Clearly, the high-end bio-monitoring applications demand significant computation bandwidth from the gateway device, typically a PDA or smart phone. This is in addition to the computation bandwidth required for running regular applications on the device, such as phone calls or music players. On the other hand, given the small form factor and battery life restrictions, the PDAs include very lightweight processors running at 100-300 MHz. Thus, there is an increasing trend towards building customized gateway devices specifically tailored towards wearable bio-monitoring platforms. As an example, recently an application-specific multiprocessor system-on-chip (MPSoC) design has been proposed for real-time analysis of a 12-lead ECG [60], which requires processing of twelve different signals from the patient's body.

Following this line of development, we focus on *processor customization* [48] to support the computation demand placed on the gateway device by high-end bio-monitoring applications. In this chapter, we choose Stretch customizable processor [38] as the hardware platform. The major obstacle to customization of bio-monitoring applications is that Stretch extensible processor (like many other extensible processors) does not support floating point operations within extension instructions. Unfortunately, profiling of bio-monitoring applications indicate that all the compute-intensive kernels contain significant amount of floating point arithmetic operations. Therefore, we first transform the applications to use fixed point arithmetic instead of floating point. This transformation enables better exploitation of instruction-set customization. We then generate multiple customization options for each compute-intensive kernel with varying area and performance gain. It is obvious that a customization option with larger area will typically provide better perfor-

mance gain.

The remainder of this chapter is structured as follows. Section 8.1 describes bio-monitoring application from the geriatric care domain. In Section 8.2, we present processor customization for bio-monitoring application. Section 8.3 shows the results of our experiments.

8.1 Wearable Bio-monitoring Applications

In this chapter, we choose a concrete bio-monitoring application from the geriatric care domain as a case study. The application consists of two related subsystems: (1) continuous monitoring of vital signs and (2) fall detection.

8.1.1 Continuous Monitoring of Vital Signs

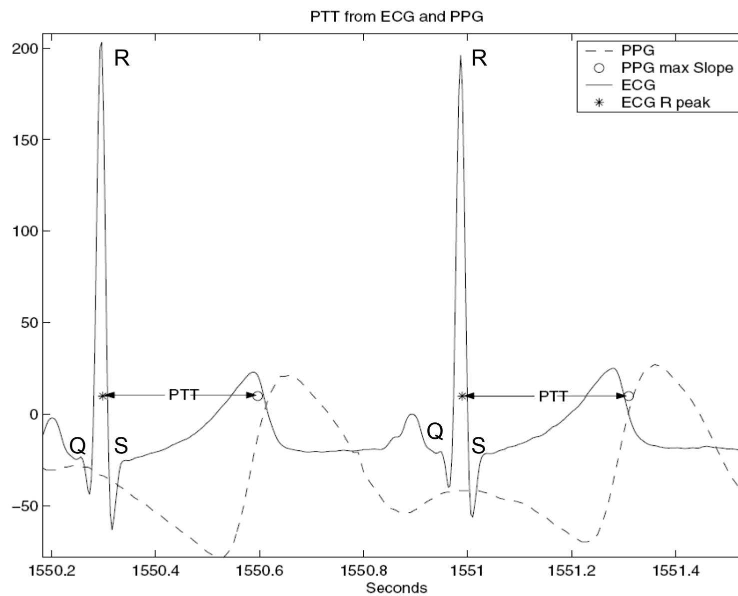
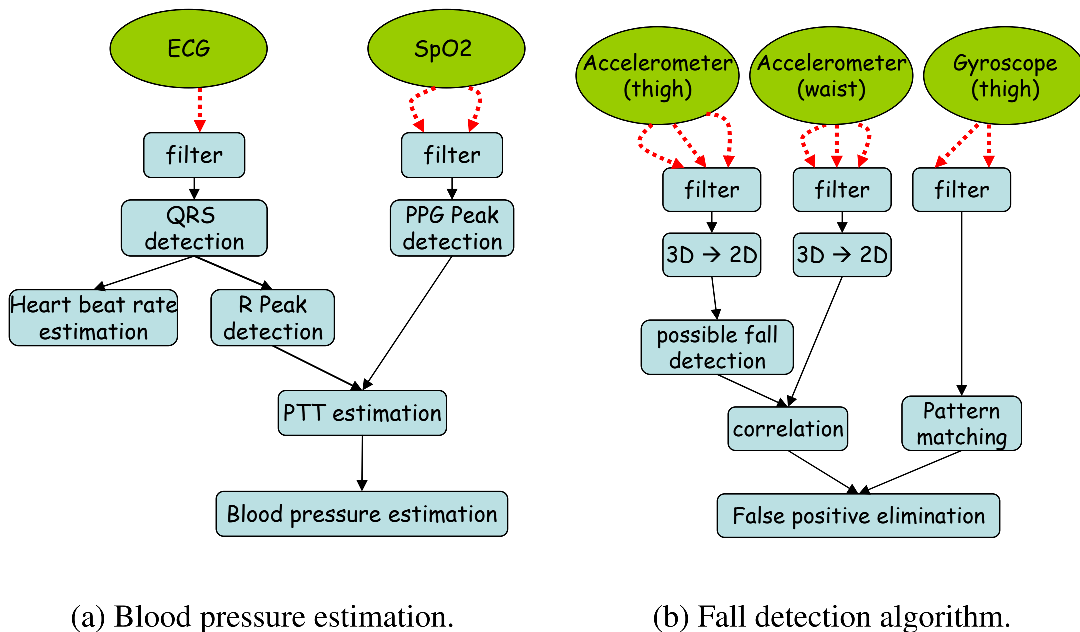


Figure 8.2: Pulse Transmit Time [35].

The subsystem for monitoring vital signs is capable of continuously measuring ECG, SpO2, systolic blood pressure, and heart beat rate. ECG electrodes are attached on the chest to measure the cardiac activities. SpO2 probe irradiates red and infrared light onto earlobe and then records the continuous changes of transmitted intensities, which is called PPG (Photo Plethysmogram). In each cardiac cycle, the ECG R peak indicates the starting of cardiac contraction, and the corresponding maximum inclination in the PPG indicates the arrival of blood at earlobe. The interval between the two kinds of peaks is defined as pulse transit time (PTT) [35] as illustrated in Figure 8.2. That is, PTT is the time it takes for the blood flow to reach from the heart to the earlobe. These vital signs are continuously monitored and transmitted to the gateway device.



(a) Blood pressure estimation. (b) Fall detection algorithm.

Figure 8.3: Bio-monitoring Applications.

The continuous measurement of vital signs requires a real-time systolic blood pressure estimation algorithm as shown in Figure 8.3(a). The detection of pulse transit time (PTT) [35], which is the time it takes for the blood flow to reach from the heart to the earlobe,

involves peak detection in both ECG and differentiated PPG. An Analog to Digital converter samples the ECG signal. The sampled ECG waveform contains some amount of superimposed line-frequency content. This line-frequency noise is removed by digitally filtering the samples through a low-pass FIR filter. This is followed by detection of all the QRS complex in the ECG waveform. The ECG R peaks can be easily derived from the QRS complex. The QRS complex also serves as a definite indicator for every heart beat, hence, it can be used to calculate the heartbeat rate. The PPG signal similarly goes through a FIR filter to remove the noise followed by detection of all the maximum slopes of the PPG. After R peaks of ECG and maximum slopes of PPG are detected, the corresponding pairs are mapped together to compute PTT. Finally, several PTT readings in a time interval are combined together into one blood pressure index.

8.1.2 Fall Detection

Any wearable fall detection system typically employs physical motion sensors such as tri-axial accelerometers and gyroscopes. The fall detection system we examine for case study consists of one tri-axial (3D) MEMS (Micro Electro Mechanical Systems) accelerometer plus one gyroscope on the thigh position and another accelerometer on the waist position. The sensitivity axes of each accelerometer is arranged in lateral, vertical, and antero posterior directions. The gyroscope provides 2D angular (lateral and sagittal) motion information. Overall we have eight streams of sensor signals coming in from the physical motion sensors (lateral, vertical, antero-posterior for each accelerometer and lateral, sagittal for gyroscope) to the gateway device through ZigBee (802.15.4) wireless communication protocol. The fall detection algorithm runs on the gateway device.

The central hypothesis of elderly fall detection approach is that the thigh motion does not go beyond certain threshold angle to forward (lateral) and sideways (sagittal) directions

in normal activities; the abnormal behavior occurs in the onset of falls among the elderly. Moreover, there is a high correlation between thigh and waist angle during fall, but low correlation during normal activities. Thus the algorithm first needs to transform the 3D accelerometer data to 2D angular data (lateral and sagittal). Next, it marks an angular motion of the thigh beyond a threshold as a "possible" onset of fall. For each such possible onset of fall, the correlation between thigh and waist angles as well as pattern matching of gyroscope angle (against reference values obtained from a number of actual falls) are used to eliminate false positives. A high-level overview of the functionalities of the fall detection application appears in Figure 8.3(b).

8.2 Processor Customization

A quick profiling of the fall detection application revealed the floating point arithmetic operations as the main performance bottleneck. Most of the functions are implementations of floating-point arithmetic operations. In fact, more than 80% of the execution time of the application is spent in floating point arithmetic operations. More importantly, the instruction-set extensible processor that we are targeting (i.e., Stretch) does not support floating point arithmetic operations within custom instructions. Indeed, most customizable processors do not support floating-point operations inside custom instructions. Consequently, we get at most 1.04x speedup after we generate Stretch custom instructions for fall detection application. Therefore, we first transform the fall detection application code to use fixed point arithmetic instead of floating point enabling better exploitation of instruction-set customization. On the other hand, blood pressure estimation application mostly uses integer arithmetic. So, we do not need to implement fixed point arithmetic version for the blood pressure estimation algorithm.

8.2.1 Conversion to Fixed Point Arithmetic

We use N-bit binary number $x = x_{N-1}x_{N-2}\ldots x_1x_0$ to present a fixed-point number in the form $U(a,b)$ [99].

$$x = \frac{1}{2^b}\sum_{n=0}^{N-1} 2^n x_n \qquad \text{and} \qquad a = N - b$$

In this representation, a bits on the left correspond to the integer part while b bits on the right correspond to the fractional part. The implied binary point exists between the bth bit x_b and the bit to its right x_{b-1}. The accuracy of the fixed point representation and the results of the corresponding arithmetic operations (compared to the floating point implementation) crucially depend on the appropriate choice of values for a and b. Therefore, we select different values of a and b for different functions depending on the accuracy requirements in our fixed point implementation of the applications. Moreover, we choose $N = 32$ for most of functions and $N = 64$ for certain functions. For our application, $N = 64$ is large enough to maintain the accuracy of floating-point operations when we convert them to fixed-point representation.

We convert each rational number or integer number to fixed-point representation by multiplying it with 2^b, where the value of b is chosen to maintain the appropriate accuracy. A fixed-point representation can be treated as an integer number except that it has the implied binary point separating integer and fractional parts. Therefore, if we ensure that two fixed point operands of an operation (such as addition or division) have the same values for a and b, we can use the normal integer arithmetic operations for fixed-point numbers. Stretch extension instruction can support integer multiplication, subtraction and addition operation but does not support integer division and modulus operations. Therefore, we have to implement integer and fixed-point division operation using basic arithmetic operations (such as shift, or, etc.) [78].

A single custom instruction in Stretch can specify a complete inner loop in the applica-

tion. The developer needs to capture the inner loops as extension instructions in Stretch C, which is a variant of standard ANSI-C language. The Stretch C compiler then fully unrolls any loop with constant iteration counts. There are three main sources of performance gain from the custom instructions in Stretch: (1) Each custom instruction can read up to three 128-bit operands and produce up to two 128-bit operands. This allows a custom instruction to exploit significant data parallelism as multiple data values can be packed together in a single 128-bit operand. (2) A custom instruction can exploit temporal parallelism through a deeply pipelined implementation of up to 27 processor clock cycles. (3) Each custom instruction can be specialized through bit width optimization, constant folding, partial evaluation, and resource sharing. After custom instructions are defined in Stretch, we have to change the source code of the original loop to use the newly defined custom instructions.

8.3 Experimental Results

We write Stretch C instructions for each hot function to explore speed up of bio-monitoring application. Then we used Stretch profiler to get cycle count of each function in the bio-monitoring application. Moreover, after generating bit stream configuration of custom instructions, we get the hardware area (in terms of number of arithmetic/logic units (AU) and multiplier units (MU)) of each custom instruction for each hot function. Different combinations of custom instructions create different custom instruction-set versions for each hot functions.

From custom instruction-set versions generated for hot functions, we choose appropriate custom instruction-set version for each hot function of the bio-monitoring applications. We vary the hardware area constraint from 0 to Max_Area at a hardware unit of 0.1 x Max_Area. The Max_Area is simply the summation of the maximum hardware area re-

quirements of the constituent bio-monitoring kernels. Bio-monitoring application enhanced with custom instructions at Max_Area explores the limit of speedup achievable. In Figure

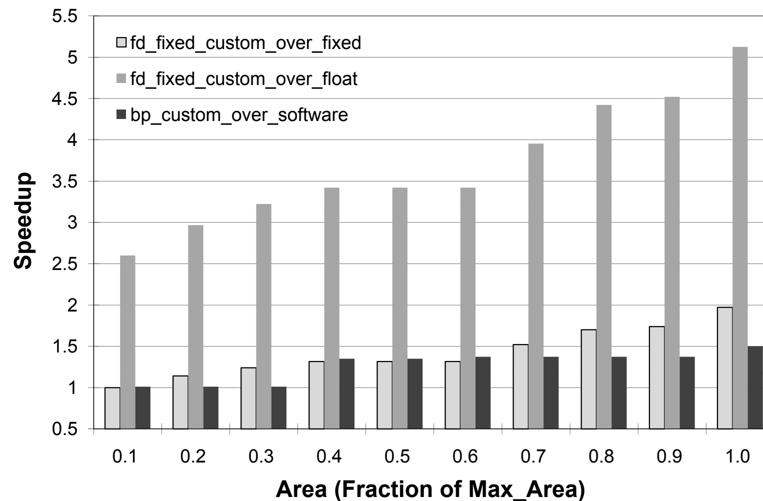


Figure 8.4: Performance Speedup with Customization.

8.4, the X-axis and Y-axis represent area constraints and speedup of the application respectively. Recall that blood pressure estimation application mostly uses integer arithmetic. Therefore, we only enhance blood pressure estimation application with custom instructions and we can get up to 1.5x speedup shown in green bar in Figure 8.4, *bp_sw_custom*. Here, the speedup is the ratio of blood pressure application execution time in software to the execution time (in cycles) of the application enhanced with custom instructions.

On the other hand, we have three implementations for fall detection application: (1) software-fixed-point implements fixed point arithmetic in software. (2) software-floating-point implements floating point arithmetic in software. (3) custom-fixed-point implements fixed point arithmetic with custom instructions. The custom-fixed-point implementation gets up to 1.97x speedup (blue bar in Figure 8.4) compared to the software-fixed-point implementation, *fd_sw_fixed_custom_fixed*. Performance speedup also comes from the fixed point arithmetic implementation instead of the floating point implementation. Red bar

in Figure 8.4 shows final speedup of custom-fixed-point implementation over software-floating-point one, *fd_sw_float_custom_fixed*. We can get nearly 5.2x performance speedup compared to the original floating-point implementation of fall detection application while the accuracy of arithmetic operations is still maintained.

8.4 Summary

In this chapter, we present our work on processor customization for bio-monitoring applications. Our customization is based on fixed point implementation and custom instruction selection. Through customization, we can get high performance gain (5.2x). The result of this work confirms the efficiency of processor customization for compute-intensive application domains such as bio-monitoring applications.

Chapter 9

Conclusions and Future Work

In this thesis, we have presented efficient design methodologies for instruction-set customization in the context of multi-tasking embedded systems. First, we studied instruction-set customization for multi-tasking embedded system with realtime constraint [44]. The results clearly show that enhancing multiple tasks with custom instructions can help these tasks meet their deadline constraints. Second, we successfully extended our work [44] to consider the conflicting tradeoffs among multiple objectives [16]. Our multi-objective framework returns an approximate Pareto curve of different tradeoffs between hardware area and performance. The approximate Pareto curve is very close to the exact Pareto curve while the running time of our algorithm is four magnitudes faster than the exact algorithm. Third, we investigated an efficient iterative custom instructions generation scheme for instruction-set customization for multi-tasking applications. Fourth, we have proposed an efficient framework for runtime reconfiguration of custom instructions for a sequential application [47]. This framework can automatically generate custom instructions for a sequential application code and pack them into different configurations which are used for runtime reconfiguration. The partitioning component which is the key component of our framework returns optimal or near optimal (99%) results with many orders of magnitudes

faster than the optimal solution. Fifth, we extended runtime reconfiguration of custom instructions [47] to multi-tasking applications with real-time constraints [46]. The proposed algorithm mostly returns results within 3% different with the optimal results. Finally, we performed a real world case study that exploits processor customization for bio-monitoring application [45]. The results show that processor customization can return a performance gain of up to 5.2X.

We can extend our study in instruction-set customization for multi-tasking embedded systems in a couple of directions. First, we should take into account the custom instructions sharing among tasks. Second, runtime reconfiguration of custom instructions should be extended to consider partial reconfiguration with pre-fetch capability. Finally, our work can be extended to study instruction-set customization in the context of multi-processor system on chip instead of the single processor context in this thesis.

Bibliography

[1] MIPS technologies. MIPS Configurable Solutions. http://www.mips.com/everywhere/technologies/configurability.

[2] OpenIMPACT Compiler. http://www.gelato.uiuc.edu/.

[3] Altera. Introduction to the Altera Nios II Soft Processor. ftp://ftp.altera.com/up/pub/Tutorials/DE2/Computer_Organization/tut_nios2_introduction.pdf.

[4] ARC. Customizing a Soft Microprocessor Core, 2009. http://www.arc.com/configurablecores/arc700/.

[5] M. Arnold and H. Corporaal. Designing domain-specific processors. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, 2001.

[6] K. Atasu, G. Dündar, and C. Özturan. An integer linear programming approach for identifying instruction-set extensions. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2005.

[7] K. Atasu, O. Mencer, W. Luk, C. Ozturan, and G. Dundar. Fast custom instruction identification by convex subgraph enumeration. In *ASAP '08: Proceddings of Inter-*

national Conference on Application-Specific Systems, Architectures and Processors, 2008.

[8] K. Atasu, C. Ozturan, G. Dundar, O. Mencer, and W. Luk. CHIPS: Custom hardware instruction processor synthesis. In *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, volume 27, March 2008.

[9] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli. Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, 2002.

[10] S. Banerjee, E. Bozorgzadeh, and N. Dutt. Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *DAC '05: Proceedings of the 42nd ACM/IEEE Design Automation Conference (DAC)*, 2005.

[11] L. Bauer, M. Shafique, S. Kramer, and J. Henkel. RISPP: Rotating instruction set processing platform. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, 2007.

[12] E. Bini and G. Buttazzo. The space of rate monotonic schedulability. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium*, 2002.

[13] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne. ISEGEN: An iterative improvement-based ISE generation technique for fast customization of processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(7), July 2006.

[14] K. Bondalapati and V. K. Prasanna. Mapping loops onto reconfigurable architectures. In *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, 1998.

[15] P. Bonzini and L. Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, 2007.

[16] U. D. Bordoloi, H. P. Huynh, S. Chakraborty, and T. Mitra. Evaluating design trade-offs in customizable processors. In *DAC '09: Proceedings of the 46th ACM/IEEE Design Automation Conference*, 2009.

[17] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, 2002.

[18] B. Chakraborty, T. Chen, T. Mitra, and A. Roychoudhury. Handling constraints in multi-objective ga for embedded system design. In *VLSID '06: Proceedings of the 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design*, 2006.

[19] L. N. Chakrapani1, J. Gyllenhaal, W. W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. *Languages and Compilers for High Performance Computing*, chapter Trimran: An Infrastructure for Research in Instruction-Level Parallelism. 2005.

[20] K. S. Chatha and R. Vemuri. Hardware-software codesign for dynamically reconfigurable architectures. In *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, 1999.

[21] X. Chen, D. L. Maskell, and Y. Sun. Fast identification of custom instructions for extensible processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2), Feb. 2007.

[22] N. Cheung, S. Parameswaran, and J. Henkel. Inside: Instruction selection/identification & design exploration for extensible processors. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, 2003.

[23] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang, and C.-M. Kyung. Synthesis of application specific instructions for embedded dsp software. *Computers, IEEE Transactions on*, 48(6), Jun 1999.

[24] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[25] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, 2004.

[26] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: Hardware-software co-synthesis of embedded systems. In *DAC '97: Proceedings of the 34th annual Design Automation Conference*, 1997.

[27] N. G. de Bruijn. *Asymptotic Methods in Analysis*. Dover Publications, 1981.

[28] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.

[29] R. W. DeVaul, M. Sung, J. Gips, and A. Pentland. MIThril 2003: Applications and Architecture. In *ISWC '03: Proceedings of the International Symposium on Wearable Computers*, 2003.

[30] R. P. Dick and N. K. Jha. CORDS: Hardware-software co-synthesis of reconfigurable real-time distributed embedded systems. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, 1998.

[31] J. Edmison, D. I. Lehn, M. Jones, and T. Martin. E-textile based Automatic Activity Diary for Medical Annotation and Analysis. In *BSN '06: Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*, 2006.

[32] E. Farella, A. Pieracci, L. Benini, and A. Acquaviva. A Wireless Body Area Sensor Network for Posture Detection. In *ISCC '06: Proceedings of the IEEE Symposium on Computers and Communications*, 2006.

[33] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC '82: Proceedings of the 19th Design Automation Conference*, 1982.

[34] J. Fisher. Trace scheduling: A technique for global microcode compaction. *Computers, IEEE Transactions on*, C-30(7), July 1981.

[35] P. Fung, G. Dumont, C. Ries, C. Mott, and M. Ansermino. Continuous Noninvasive Blood Pressure Measurement by Pulse Transit Time. In *IEMBS '04: Proceedings of the 26th Annual International Conference of the IEEE on Engineering in Medicine and Biology Society*, 2004.

[36] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis. Automatic selection of application-specific instruction-set extensions. In *CODES+ISSS '06:*

Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, 2006.

[37] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2), 2000.

[38] R. E. Gonzalez. A software-configurable processor architecture. *IEEE Micro*, 26(5), 2006.

[39] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, 2001.

[40] C. Hardnett, K. V. Palem, and Y. Chobe. Compiler optimization of embedded applications for an adaptive SoC architecture. In *CASES '06: Proceedings of the ACM International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[41] T. Hollstein, J. Becker, A. Kirschbaum, and M. Glesner. HiPART: A new hierarchical semi-interactive HW-/SW partitioning approach with fast debugging for real-time embedded systems. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, 1998.

[42] P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. *ACM SIGARCH Comput. Archit. News*, 14(2), 1986.

[43] I. J. Huang and A. Despain. Synthesis of application specific instruction sets. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 14(6), Jun 1995.

[44] H. P. Huynh and T. Mitra. Instruction-set customization for real-time embedded systems. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, 2007.

[45] H. P. Huynh and T. Mitra. Processor customization for wearable bio-monitoring platforms. In *FPT '08: Proceedings of International Conference on Field-Programmable Technology*, 2008.

[46] H. P. Huynh and T. Mitra. Runtime reconfiguration of custom instructions for real-time embedded systems. In *DATE '09: Proceedings of the conference on Design, automation and test in Europe*, 2009.

[47] H. P. Huynh, E. J. Sim, and T. Mitra. An efficient framework for dynamic reconfiguration of instruction-set customization. *Design Automation for Embedded Systems*, 13(1-2), 2009.

[48] P. Ienne and R. Leupers. *Customizable Embedded Processors*. Morgan Kauffman, 2006.

[49] J. A. Jacob and P. Chow. Memory interfacing and instruction specification for reconfigurable processors. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, 1999.

[50] R. Jafari, F. Dabiri, P. Brisk, and M. Sarrafzadeh. Adaptive and Fault Tolerant Medical Vest for Life-critical Medical Monitoring. In *SAC '05: Proceedings of the ACM Symposium on Applied Computing*, 2005.

[51] R. Jafari, A. Encarnacao, A. Zahoory, F. Dabiri, H. Noshadi, and M. Sarrafzadeh. Wireless Sensor Networks for Health Monitoring. In *MobiQuitous '05: Proceedings of the International Conference on Mobile and Ubiquitous Systems*, 2005.

[52] R. Jafari, H. Noshadi, M. Sarrafzadeh, and S. Ghiasi. Adaptive Electrocardiogram Feature Extraction on Distributed Embedded Systems. *IEEE Transactions on Parallel and Distributed Systems*, 17(8), 2006.

[53] H. Javaid and S. Parameswaran. A design flow for application specific heterogeneous pipelined multiprocessor systems. In *DAC '09: Proceedings of the 46th annual Design Automation Conference (DAC)*, 2009.

[54] J.-C. Kao and R. Marculescu. On Optimization of E-Textile Systems using Redundancy and Energy-aware Routing. *IEEE Transactions on Computers*, 55(6), 2006.

[55] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 1998.

[56] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1), 1998.

[57] R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Trans. Des. Autom. Electron. Syst.*, 7(4), 2002.

[58] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. In *DAC '99: Proceedings of the 36th ACM/IEEE Design Automation Conference*, 1999.

[59] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2), 1970.

[60] I. A. Khatib, D. Bertozzi, A. Jantsch, and L. Benini. Performance Analysis and Design Space Exploration for High-end Biomedical Applications: Challenges and

Solutions. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2007.

[61] I. A. Khatib, F. Poletti, D. Bertozzi, L. Benini, M. Bechara, H. Khalifeh, A. Jantsch, and R. Nabiev. A Multiprocessor System-on-chip for Real-time Biomedical Monitoring and Analysis: Architectural Design Space Exploration. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference (DAC)*, 2006.

[62] Y. J. Kim and T. Kim. HW/SW partitioning techniques for multi-mode multi-task embedded applications. In *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, 2006.

[63] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms Generation, Enumeration and Search*. CRC Press Inc, 1998.

[64] A. L. Rosa, L. Lavagno, and C. Passerone. Hardware/software design space exploration for a reconfigurable processor. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, 2003.

[65] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO '97: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997.

[66] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set design of configurable ASIPs. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, 2002.

[67] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. *ACM SIGOPS Oper. Syst. Rev.*, 32(5), 1998.

[68] R. Leupers, K. Karuri, S. Kraemer, and M. Pandey. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, 2006.

[69] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *DAC '00: Proceedings of the 37th ACM/IEEE Design Automation Conference*, 2000.

[70] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[71] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri. A VLIW processor with reconfigurable instruction set for embedded applications. *IEEE Journal of Solid-State Circuits*, 38(11), 2003.

[72] B. Mei, P. Schaumont, and S. Vernalde. A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In *ProRISC '00: Proceedings of the 11th ProRISC Workshop on Circuits, Systems and Signal Processing*, 2000.

[73] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 2001.

[74] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal Of Artificial Intelligence Research*, 7, 1997.

[75] C. H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, 2000.

[76] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5), 1991.

[77] S. Park, K. Mackenzie, and S. Jayaraman. The Wearable Motherboard: A Framework for Personalized Mobile Information Processing (pmip). In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, 2002.

[78] D. A. Patterson and J. L. Hennessy. Computer Organization & Design. Morgan Kauffman, 1998.

[79] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[80] L. Pozzi. Methodologies for the design of application-specific reconfigurable VLIW processors. PhD thesis, Politecnico Di Milano, 2000.

[81] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(7), July 2006.

[82] L. Pozzi, M. Vuletic, and P. Ienne. Automatic topology-based identfication of instruction-set extensions for embedded processor. *Technical Report 01/377*, Swiss Federal Institute of Technology Lausanne (EPFL), 2001.

[83] K. M. G. Purna and D. Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers*, 48(6), 1999.

[84] R. Razdan and M. Smith. A high-performance microarchitecture with hardware-programmable functional units. *MICRO '94: Proceedings of the 27th annual international symposium on Microarchitecture*, 1994.

[85] Y. Shin and K. Choi. Enforcing schedulability of multi-task systems by hardware-software codesign. In *CODES '97: Proceedings of the 5th International Workshop on Hardware/Software Co-Design*, 1997.

[86] J. Shu, T. C. Wilson, and D. K. Banerji. Instruction-set matching and ga-based selection for embedded-processor code generation. In *VLSID '96: Proceedings of the 9th International Conference on VLSI Design: VLSI in Mobile Communication*, 1996.

[87] F. Stappert. WCET benchmarks. http://www.c-lab.de/home/en/download.html.

[88] Stretch Inc. Stretch S5530 software configurable processor. http://www.stretchinc.com/products/s5000.php.

[89] F. Sun, S. Ravi, A. Raghunathan, and N. Jha. Custom-instruction synthesis for extensible-processor platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(2), Feb. 2004.

[90] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha. A scalable application-specific processor synthesis methodology. In *ICCAD '03: Proceedings of International Conference on Computer Aided Design*, 2003.

[91] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha. Application-specific heterogeneous multiprocessor synthesis using extensible processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(9), Sept. 2006.

[92] J. Teich, S. P. Fekete, and J. Schepers. Optimization of dynamic hardware reconfigurations. *The Journal of Supercomputing*, 19(1), 2001.

[93] Tensilica - XPRES Compiler - Optimized Hardware Directly from C. www.tensilica.com/products/devtools/hw_dev/xpres/.

[94] Transmeta-corporation. Tm5400 processor specifications.

[95] A. K. Verma, P. Brisk, and P. Ienne. Rethinking custom ISE identification: A new processor-agnostic method. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, 2007.

[96] M. J. Wirthlin and B. L. Hutchings. A Dynamic Instruction Set Computer. In *FCCM '95: Proceedings of Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1995.

[97] C. Wolinski and K. Kuchcinski. Automatic selection of application-specific reconfigurable processor extensions. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, 2008.

[98] Xilinx. Microblaze Processor. `http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm`.

[99] R. Yates. Fixed-point Arithmetic: An Introduction. Technical report, Digital Signal Labs, 2007.

[100] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ISCA' 00: Proceedings of the 27th annual international symposium on Computer architecture*, 2000.

[101] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, 2004.

[102] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES '04: Proceedings of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2004.

[103] P. Yu and T. Mitra. Disjoint pattern enumeration for custom instructions identification. In *FPL '07: Proceedings of International Conference on Field Programmable Logic and Applications*, 2007.