HARDWARE/SOFTWARE PARTITIONING FOR CUSTOM INSTRUCTION PROCESSORS

by

Kubilay Atasu

B.S., in Computer Engineering, Boğaziçi University, 2000

M. E., in Embedded System Design, Università della Svizzera italiana, 2002

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering

Boğaziçi University

2007

ACKNOWLEDGEMENTS

The support of Boğaziçi University Research Projects (05HA102D) and UK EPSRC (EP/C509625/1, EP/C549481/1) for this work is gratefully acknowledged. I also would like to express my gratitude for the support of the Scientific and Technological Research Council of Turkey (TUBITAK) under National Ph.D. Scholarship Program.

I would like to thank my thesis advisors Prof. Günhan Dündar and Assoc. Prof. Can Özturan for their invaluable guidance and support throughout the development of this thesis. In particular, I am grateful to Assoc. Prof. Can Özturan for encouraging me to do graduate studies since my undergraduate years. His love in mathematics and theoretical computer science has been a constant source of inspiration for me. I am indebted to Prof. Günhan Dündar for dedicating some of his valuable time for me weekly. His trust in me and his continuous help with all kinds of difficulties I faced during my Ph.D. years helped me keep my motivation always high and made this thesis possible. Additionally, I would like to thank Assoc. Prof. Oskar Mencer and Prof. Wayne Luk for giving me the opportunity to work with the Custom Computing and Computer Architecture groups at Imperial College London in the last two years. The prolific environment and the insightful discussions have greatly contributed to the quality of this thesis. I also would like to thank Prof. Cem Ersoy and Prof. Çetin Kaya Koç for taking part in my thesis committee. Their helpful comments and suggestions have greatly improved this work. I am grateful to Prof. Nadir Yücel and Prof. Mariagiovanna Sami as well, for their support and advices throughout my studies.

I would like to thank all of my friends who made my life enjoyable during the hard Ph.D. years. In particular, I would like to mention Başkan Kalezade, Kemal Kaplan, Tamer Demir, Miljan Vuletic, and Carlos Tavares. Finally, I am most grateful to my mother Perihan Atasu and my brother Atalay Atasu. Without their help and support this thesis would not be possible. This thesis is dedicated to my mother who always believed in me and helped me overcome all the hardships with her endless love.

ABSTRACT

HARDWARE/SOFTWARE PARTITIONING FOR CUSTOM INSTRUCTION PROCESSORS

In this thesis, we describe an integer linear programming (ILP) based system called CHIPS for identifying custom instructions given the available data bandwidth and transfer latencies between the base processor and the custom logic. Our approach, which involves a baseline machine supporting architecturally visible custom state registers, enables designers to optionally constrain the number of input and output operands for custom instructions. We describe a comprehensive design flow to identify the most promising area, performance, and code size trade-offs. We study the effect of the constraints on the number of input/output operands and on the number of register file ports. Additionally, we explore compiler transformations such as if-conversion and loop unrolling. Our experiments show that, in most of the cases, the highest performing solutions are identified when the input/output constraints are removed. However, input/output constraints help our algorithms identify frequently used code segments, reducing the overall area overhead. We provide detailed results for eleven benchmarks covering cryptography and multimedia. We obtain speed-ups between 1.7 and 6.6 times, code size reductions between six per cent and 72 per cent, and area costs that range between 12 adders and 256 adders for maximal speed-up. We demonstrate that our ILP based solution scales well, and benchmarks with very large basic blocks consisting of up to 1000 instructions can be optimally solved, most of the time within a few seconds. We show that the state of the art techniques fail to find the optimal solutions on the same problem instances within reasonable time limits. We provide examples of solutions identified by our algorithms that are not covered by the existing methods.

ÖZET

ÖZELLEŞTİRİLEBİLİR KOMUT KÜMELİ İŞLEMCİLER İÇİN YAZILIM/DONANIM BÖLÜŞTÜRMESİ

Bu tezde doğrusal tamsayı programlama (ILP) tabanlı, CHIPS adı verilen bir araç zinciri tarif edilmektedir. Temel bir işlemci ile adanmış bir mantıksal devre arasındaki veri bandı genişliği verildiğinde, CHIPS özelleştirilmiş komutları bulur. Özelleştirilmiş durum yazmaçlarını destekleyen bir temel işlemci mimarisi üzerine kurulu bu yöntem, tasarımcıların isteğe bağlı olarak komutların işlenen girdi ve çıktılarının sayılarını sınırlandırmalarına olanak verir. Bu tezde, en vaad edici alan, başarım ve kod büyüklüğü ödünleşimlerinin bulunması için kapsamlı bir tasarım akışı anlatılmaktadır. Girdi/çıktı sayısı ve yazmaç dosyası kapı sayısı üzerindeki sınırlandırmalar ile birlikte if-dönüştürmesi ve döngü açılması gibi derleyici dönüşümleri değerlendirilmektedir. Deneylerimizin önemli bir çoğunluğunda en yüksek başarımlı çözümlerin girdi/çıktı sınırlamaları kaldırıldığında bulunduğu gözlemlenmiştir. Fakat, girdi/çıktı sınırlamaları sık kullanılan kod kısımlarının tanımlanmasını sağlamıştır. Şifreleme ve çoklu ortam alanlarını kapsayan on bir denektaşı testi üzerinde detaylı sonuçlar sunulmaktadır. Denektaşı testleri 1.7 ve 6.6 kat arasında hızlandırılmış, kod büyüklükleri yüzde altı ve yüzde 72 oranları arasında azaltılmış, en yüksek başarım için 12 ile 256 toplayıcı alanı arasında değişen mantıksal devre alanlarına ihtiyaç duyulmuştur. Yöntemimizin büyük problemler üzerinde de etkili olduğu, 1000 kadar komuttan oluşan temel kod blokları içeren denektaşı testlerinin eniyi şekilde, çoğu zaman sadece bir kaç saniye içinde çözülebildikleri gösterilmiştir. Aynı testler üzerinde var olan en ileri yöntemlerin kabul edilebilir süreler içinde eniyi sonuçlara ulaşamadıkları görülmüştür. Çalışmamız diğer yöntemler tarafından bulunamayan çözüm örnekleri ile de desteklenmektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . ix
LIST OF TABLES . . . xiii
LIST OF SYMBOLS/ABBREVIATIONS . . . xiv
1. INTRODUCTION . . . 1
1.1. Motivation . . . 1
1.2. Contributions of the Thesis . . . 8
2. BACKGROUND AND RELATED WORK . . . 10
2.1. Compilers . . . 10
2.1.1. Intermediate Representation . . . 11
2.1.2. Definitions for Custom Instruction Identification . . . 13
2.2. Architectural Approaches for Custom Logic Integration . . . 16
2.2.1. Attached and External Processing Units . . . 17
2.2.2. Tightly Coupled Coprocessors . . . 17
2.2.3. Custom Instruction Processors . . . 21
2.3. Technology Choices for Custom Instruction Processors . . . 22
2.3.1. Synthesizable ASIC Processors . . . 23
2.3.2. Soft Core FPGA Processors . . . 24
2.3.3. Hard Core Processors with Reconfigurable Datapaths . . . 25
2.4. Related Work on Hardware/Software Partitioning Algorithms . . . 26
2.5. Summary . . . 31
3. THE CHIPS APPROACH . . . 32
3.1. Problem Formulation . . . 34
3.2. Integer Linear Programming Model . . . 36
3.2.1. Calculation of the Input Data Transfers . . . 38
3.2.2. Calculation of the Output Data Transfers . . . 40
3.2.3. Convexity Constraint . . . 42

3.2.4. Critical Path Calculation 44
3.2.5. Objective 44
3.2.6. Scalability of the Model 45
3.2.7. Support for Statements with Multiple Destination Operands . . 45
3.3. Template Generation 46
3.4. Template Selection 49
3.5. Machine Description and Code Generation 52
3.6. Summary 52
4. EXPERIMENTS AND RESULTS 53
4.1. Experiment Setup 53
4.1.1. Base Processor Configuration 53
4.1.2. Synopsys Synthesis 54
4.1.3. Benchmarks 54
4.1.4. Run-time Environment 55
4.2. If-conversion Results 55
4.3. Examples of Custom Instructions 56
4.3.1. AES Encryption 56
4.3.2. DES Encryption 60
4.4. Effect of Input and Output Constraints 62
4.5. Effect of Register File Ports 65
4.6. Effect of Loop Unrolling 67
4.7. Granularity vs. Reusability 70
4.8. Run-time Results 71
4.9. Performance and Code Size Results 74
4.10. Processor Synthesis Results 75
4.11. Summary 78
5. A SIMPLIFIED MODEL 80
5.1. Motivation 80
5.2. Formulation of the Simplified Problem 81
5.3. An Upper Bound on the Size of the Search Space 83
5.4. Related Work 86
5.5. Summary 87

6. CONCLUSIONS 88
6.1. Summary and Conclusions 88
6.2. Future Work 90
APPENDIX A: IMPLEMENTATION DETAILS 93
A.1. If-Conversion Implementation 93
REFERENCES 95

LIST OF FIGURES

Figure 1.1. Design complexity (transistors per chip) vs. designer productivity (transistors per man-month). 3

Figure 1.2. Efficiency vs. programmability. 4

Figure 1.3. The custom instruction processor. 5

Figure 1.4. Datapath of the custom instruction processor. 6

Figure 2.1. The control flow graph, the nodes represent basic blocks. 12

Figure 2.2. The DAG representation of a basic block. 14

Figure 2.3. The custom instruction template extracted from Figure 2.2, having four input and two output operands. 14

Figure 2.4. The DAG of Figure 2.2 after replacing the template shown in Figure 2.3 with a custom instruction. 15

Figure 2.5. A custom instruction template that is not convex. 15

Figure 2.6. A classification of integration methods. 17

Figure 2.7. A tightly coupled coprocessor has direct access to the main processor through dedicated control and data transfer channels. 18

Figure 2.8. Xilinx MicroBlaze processor. 20

Figure 2.9. Integration of custom functional units (FUs) into the pipeline of a MIPS type single issue processor with five pipeline stages. 21

Figure 3.1. CHIPS: we integrate our algorithms into Trimaran compiler. . . . 33

Figure 3.2. The register file supports RF_{in} read ports and RF_{out} write ports. Custom instructions might have an arbitrary number of input and output operands, $INST_{\text{in}}$ and $INST_{\text{out}}$. 35

Figure 3.3. Templates are defined based on the assignment of ones and zeros to the binary decision variables associated with the DAG nodes. . 37

Figure 3.4. We iteratively solve a set of ILP problems. A good upper bound on the objective value can significantly reduce the solution time. . 47

Figure 4.1. We apply an if-conversion pass before identifying custom instructions. 56

Figure 4.2. The AES round transformation. 57

Figure 4.3. A 32-bit implementation of the MixColumn transformation. . . . 58

Figure 4.4. An optimized AES encryption implementation 59

Figure 4.5. Optimal custom instruction implementing the DES rounds. . . . 60

Figure 4.6. A fully unrolled DES encryption implementation. 61

Figure 4.7. 32-bit implementation of a DES round in C. 62

Figure 4.8. AES decryption: per cent reduction in the execution cycles. Register file supports four read ports and four write ports (i.e., $RF_{in} = 4$, $RF_{out} = 4$). An input constraint of MAX_{in} and an output constraint of MAX_{out} can be imposed on custom instructions. 63

Figure 4.9. DES: per cent reduction in the execution cycles. Register file supports two read ports and one write port (i.e., $RF_{in} = 2$, $RF_{out} = 1$). An input constraint of MAX_{in} and an output constraint of MAX_{out} can be imposed on custom instructions. 64

Figure 4.10. Register file supports two read ports and one write port (i.e., $RF_{in} = 2$, $RF_{out} = 1$). Speed-up (with respect to the base processor) improves with the relaxation of input/output constraints. 65

Figure 4.11. DES: effect of increasing the number of register file ports (i.e., RF_{in} and RF_{out}) on the performance. 66

Figure 4.12. djpeg: effect of increasing the number of register file ports (i.e., RF_{in} and RF_{out}) on the performance. 66

Figure 4.13. djpeg: increasing the number of register file ports (i.e., RF_{in}, RF_{out}) improves the performance. First four columns depict the achievable speed-up for the four most time consuming functions of djpeg. . . 67

Figure 4.14. Loop unrolling improves the performance, and enables coarser grain custom instructions. 68

Figure 4.15. Loop unrolling increases the number of instructions in the code. Code compression due to the use of custom instructions often compensates for this effect. 69

Figure 4.16. Granularity vs. reusability. 70

Figure 4.17. AES decryption: run-time performance of our template generation algorithm. 71

Figure 4.18. DES: run-time performance of our template generation algorithm. 72

Figure 4.19. All benchmarks: increasing the number of register file ports improves the performance. 75

Figure 4.20. All benchmarks: increasing the number of register file ports reduces the number of instructions in the code. 76

Figure 4.21. AES Decrypt processor layout. 0.307 mm^2 area generated using a 130nm process technology. 77

Figure 4.22. The ASIC area and the worst case negative timing slack with a 200 MHz constraint on the clock rate. 78

Figure 5.1. Example DAG: v_4 and v_5 are invalid nodes. 85

Figure 5.2. Incompatibility graph for the DAG of Figure 5.1. 87

Figure A.1. The initial code contains branches (left). If-conversion eliminates the branches (right). 94

LIST OF TABLES

Table 2.1. A comparison with some state-of-the-art techniques 30

Table 4.1. Relative latency and area coefficients for various operators based on synthesis results on UMC's 130nm process. 54

Table 4.2. Information on benchmarks: BB represents basic block. 55

Table 4.3. Relaxed problem: size of the largest basic block (BB), the number of integer decision variables (Vars), the number of linear constraints (Constrs), and the solution times associated with the first iteration of the template generation algorithm. 73

Table 4.4. Original problem with the convexity constraint: solution times associated with the first iteration of the template generation algorithm. 73

Table 4.5. Run-time comparison with the exact algorithm of [24]. We show the solution times in seconds for four (MAX_{in},MAX_{out}) combinations. 74

Table 5.1. Solutions for the DAG of Figure 5.1 85

LIST OF SYMBOLS/ABBREVIATIONS

$C(T)$	Communication latency of template T
DT_{in}	The number of input data transfers
DT_{out}	The number of output data transfers
G	Directed acyclic graph representing a basic block
h_i	The hardware latency of node i
$H(T)$	Hardware latency of template T
$INST_{\text{in}}$	The number of input operands for a custom instruction
$INST_{\text{out}}$	The number of output operands for a custom instruction
MAX_{in}	The maximum number of input operands allowed
MAX_{out}	The maximum number of output operands allowed
RF_{in}	The number of register file read ports
RF_{out}	The number of register file write ports
s_i	The software latency of node i
$S(T)$	Software latency of template T
T	Custom instruction template
x_i	Binary decision variable representing node i
$Z(T)$	Objective value of template T

AES	Advanced encryption standard
ALU	Arithmetic logic unit
ASIC	Application-specific integrated circuit
ASIP	Application-specific instruction-set processor
CFG	Control flow graph
CPLD	Complex programmable logic device
DAG	Directed acyclic graph
DES	Data encryption standard
DSP	Digital signal processor
FIFO	First in, first out
FPGA	Field programmable gate array

FSL	Fast simplex link
GHz	gigahertz
GPP	General purpose processor
ILP	Integer linear programming
IR	Intermediate representation
ISA	Instruction set architecture
MDES	Machine description
MHz	megahertz
MPSoC	Multiprocessor systems-on-chip
RAM	Random access memory
RISC	Reduced instruction set computer
RTL	Register transfer level
SHA	Secure hash algorithm
SIMD	Single instruction multiple data
SoC	Systems-on-chip
SSA	Static single assignment
TIE	Tensilica instruction extension
VLIW	Very long instruction word

1. INTRODUCTION

Information processing has become a part of our daily life. Nowadays, special purpose computer systems are embedded into everyday objects, such as automobiles, consumer electronics, telecommunications, home automation and office automation devices. We are often involved in multiple computational activities at the same time, sometimes even without being aware. The new computational model, known as ubiquitous computing, depends heavily on embedded systems for processing and distribution of the ubiquitous information. Unlike general purpose computer systems, embedded systems are dedicated to specific application domains. Such systems are often produced in high volume, and built under strict cost and performance constraints. Today, embedded processors constitute the majority of the sales in the microprocessor market. More than a billion embedded processors are used in embedded devices every year.

This thesis introduces novel techniques for the efficient design space exploration of embedded processors with customizable instruction-sets, which can be tailored for specific application domains. The rest of this chapter is organized as follows: Section 1.1 describes the motivation for our research and Section 1.2 covers our contributions.

1.1. Motivation

The highly competitive end-market for consumer electronics, multimedia, and communications devices is continuously forcing the producers to release new products with improved functionality and performance. The complexity of the embedded systems-on-chip (SoC) devices is constantly increasing, while the life cycles of the end products are getting shorter. In the late nineties, a typical embedded SoC contained only one or two programmable components, i.e., a microcontroller and possibly a digital signal processor (DSP), in addition to some dedicated logic, memories and peripherals. New generation embedded SoCs are increasingly replacing the complex dedicated logic with programmable processing units and off-the-shelf standard components in order to reduce the design times and nonrecurring engineering costs. Today, a modern SoC

device can contain tens of programmable processing units.

In 1965 Gordon Moore observed that the number of transistors placed in an integrated circuit doubled every one to two years, and predicted that the same trend would continue for a long period of time. The trend Moore projected, later known as Moore's law [1], set the pace for the chip developers. The continuous improvement in the process geometries and circuit characteristics has so far allowed semiconductor manufacturers to keep up with Moore's law, doubling the circuit densities approximately every two years. Although signal integrity, interconnect delay, and power dissipation problems are getting increasingly harder to cope with, and further scaling of the technology requires significant innovation, the exponential rate of increase in the circuit densities is expected to continue at least for 15 more years [2].

New process technologies make it possible to build SoC devices with hundreds of millions of transistors. A diversity of applications with enormous functionalities can now be integrated into the same chip. Within a few years, SoC devices with more than a billion transistors are expected to be on the market. SoC designers are continuously faced with the challenge of designing more and more versatile systems using more and more logic gates. The tight silicon integration is continuously reducing the size of the end products, making them small enough to be portable. New technologies are reducing the energy consumption as well, allowing more and more end products to be battery powered. The increasing use of wireless and hand-held devices in the daily life has made the energy efficiency a critical factor in the SoC design.

A major problem for SoC producers of today is that the designers are not able to keep up with the growth in the logic complexity. Although chip design tools for high level simulation, logic synthesis, placement, and routing are continuously being improved, and systematic design reuse helps in reducing the design effort, the designer productivity improves at a much lower rate compared with the growth in the logic complexity. This phenomenon, known as the SoC design gap [3], is illustrated in Figure 1.1. The design gap necessitates new approaches in the design of complex SoCs. First of all, the new generation SoCs should be sufficiently programmable so that the

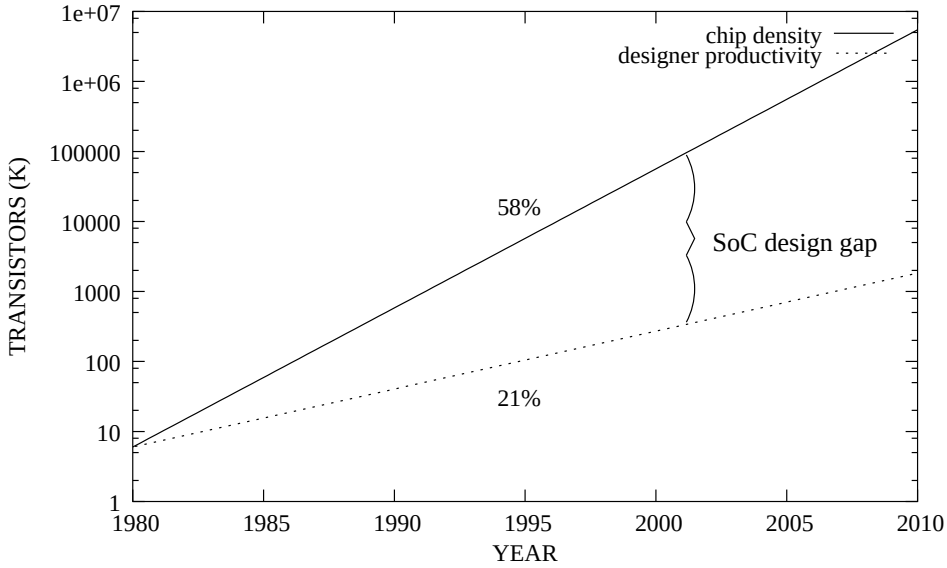


Figure 1.1. Design complexity (transistors per chip) vs. designer productivity (transistors per man-month). Source: Semiconductor Research Corporation [4].

same chip design can be used to realize multiple applications, amortizing the chip design costs. Secondly, there is a need for mechanisms that facilitate optimization of the performance, area, and power efficiency of the SoC designs for specific applications.

Efficiency and programmability are two conflicting goals in the design of complex SoCs for embedded systems. Application-specific integrated circuit (ASIC) technology allows optimization of hardware resources in order to maximize performance and minimize area overhead and power consumption for a specific application at the design phase. However, ASIC devices are not programmable. The ability of ASICs to adapt to different applications or revisions in the system specification is very limited. In addition, implementing a complete application on ASIC technology is not always feasible due to the growing complexity of SoC designs, time to market pressure, and chip fabrication costs. On the other hand, general purpose processors (GPPs) and DSPs can implement a wide range of applications of arbitrary complexity because of their programmable nature. However, both alternatives lack the efficiency of ASICs and often fail to satisfy the strict performance, area, and power consumption requirements of embedded applications. Reconfigurable devices, such as those based on field programmable gate arrays (FPGAs), offer viable solutions combining efficiency and programmability. The computational resources and the interconnections between re-

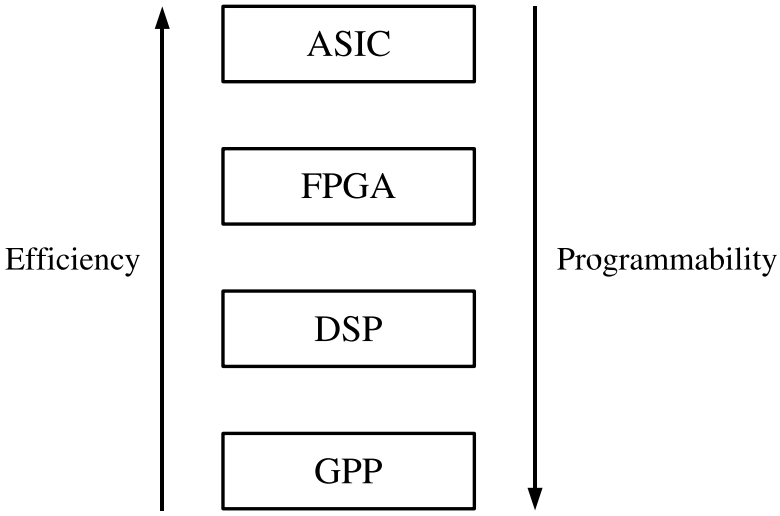


Figure 1.2. Efficiency vs. programmability.

sources can be configured to exploit the inherent parallelism within an application, enabling hundreds of arithmetic and logic operations to be performed concurrently. However, reconfigurable devices are not as performance efficient as ASICs, and their unit costs are higher. Furthermore, reconfigurable devices have a relatively low logic density and a limited resource capacity. Mapping a complete application on reconfigurable logic can be impracticable depending on the size of the application and the volume of the production. Figure 1.2 compares different implementation alternatives based on the relative efficiency and the degree of programmability.

The well known 90/10 law [5] states that a computer program spends 90 per cent of its execution time in only 10 per cent of the code. Therefore, mapping a complete application to dedicated logic, such as ASICs and FPGAs, is very cost inefficient. A common approach in the design of embedded SoCs is to implement only the most computation intensive parts of the code on dedicated logic and the rest of the code on a general purpose processor. Data transfers between the general purpose processor and the dedicated logic is often achieved by means of a bus interface, which can be subject to potential resource contentions. The limited data bandwidth between the processor and the custom logic is often a bottleneck in the overall performance of the system. Application-specific instruction-set processors (ASIPs) try to overcome this bottleneck by integrating custom logic directly into the processor.

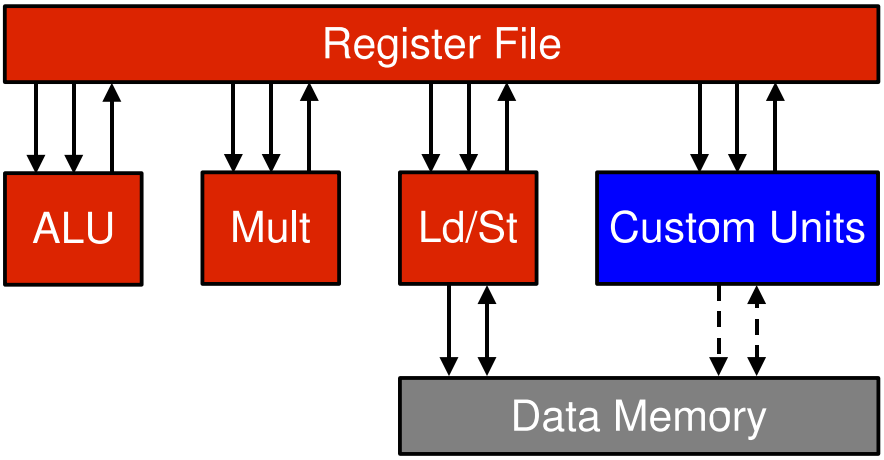


Figure 1.3. The custom instruction processor. Register file ports can be shared between the standard execution units and the custom functional units.

ASIPs provide a good compromise between general-purpose processors and custom ASIC or FPGA designs. The instruction processor preserves the programmability, while the application-specific instructions enhance the processor efficiency. The traditional approach in the design of ASIPs involves the definition of a complete instruction set architecture (ISA) for a given application. The processor and the compiler are synthesized based on a high level ISA description. Target's CHESS compiler [6] based on the nML [7] architecture description language (ADL), and Coware's LISATek based on LISA [8], are among the commercial examples in this area. The more recent approach, however, assumes a pre-verified, pre-optimized base processor with a basic instruction-set. The base processor is extended with custom functional units that implement application-specific instructions. A dedicated link between custom units and the base processor provides an efficient communication interface. Re-using a pre-verified, pre-optimized base processor reduces the design complexity and the time to market. Several commercial examples exist, such as Tensilica Xtensa [9], ARC 700 [10], Altera Nios II [11], MIPS Pro Series [12], and Stretch S6000 [13]. In this thesis, we target the processors of the second type, which we refer to as *custom instruction processors*. We call the extensions to the basic instruction-set as *custom instructions*.

Figure 1.3 illustrates the way custom functional units are integrated into a base processor. The base processor is typically a reduced instruction set computer (RISC) [5]

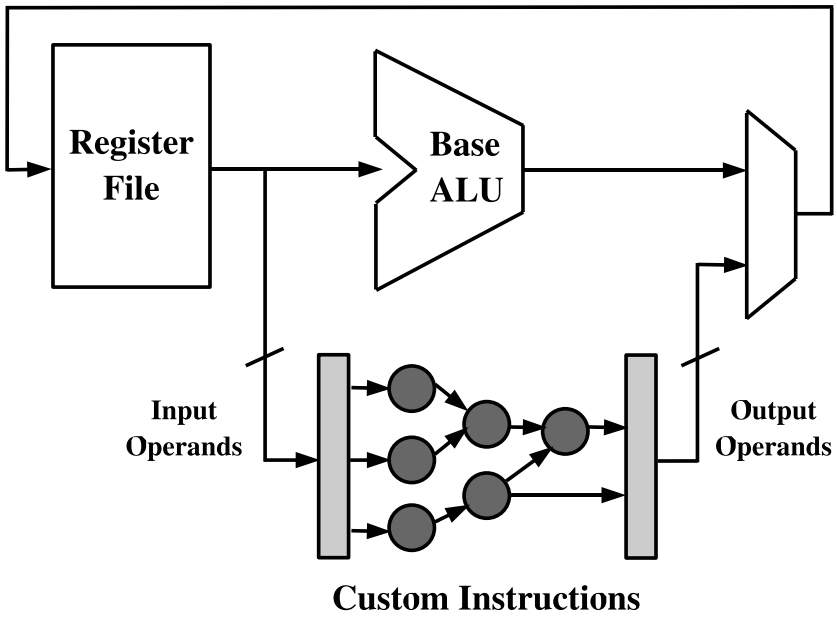


Figure 1.4. Datapath of the custom instruction processor: the data bandwidth may be limited by the register file ports or by the dedicated data transfer channels.

with a load/store architecture. Custom functional units share the base processor resources, such as register file ports and operand buses with the standard execution units of the base processor. Direct access from custom functional units to the memory system can be enabled through dedicated load/store units and data transfer channels.

Custom instruction processors offer new opportunities in the hardware/software codesign of complex SoCs. In the context of custom instruction processors, hardware/software partitioning is done at the instruction level granularity. Given the description of an application in a high level programming language, custom instructions provide efficient hardware implementations for the frequently executed code segments. The partitioning of an application into base processor instructions and custom instructions is done under certain constraints. First of all, the size of the custom logic area is limited, and custom functional units must fit into the available area. Secondly, the data bandwidth between the base processor and the custom functional units is limited either by the available register file ports or by the dedicated data transfer channels as shown in Figure 1.4. Thus, the cost of the data transfers between the base processor and the custom units have to be explicitly evaluated. Next, only a limited number of input and output operands can be encoded within a fixed-length instruction word. Finally,

additional restrictions on the structure of the custom instructions can be necessary in order to guarantee a feasible schedule for the instruction stream.

The objective of the hardware/software partitioning is often formulated as maximizing the performance of the custom instruction processor under area constraints [14]. Similarly, given constraints on the performance, the objective could be minimization of the area overhead [15]. Power consumption and code size are among other important metrics. In fact, improving the performance allows decreasing the clock rate of the processor, which enables a reduction in the power consumption. Ideally, the partitioning algorithms should enable efficient exploration of the design space in order to identify the most promising trade-offs in terms of several different metrics.

Custom instruction processors are emerging as the basic building blocks in the design of complex SoCs, replacing dedicated register transfer level (RTL) hardware blocks. The new multiprocessor SoC (MPSoC) design approach [2] assumes that the application can be decomposed into a set of communicating tasks, and that the functionality of each task can be defined in software using a high level programming language. The processors in the system are then tailored for specific tasks, enhancing the performance, area, and power efficiency. The software based MPSoC approach is expected to reduce the SoC development effort and allows adaptation of the design to changes in the system specification that occur late in the design process, even after the chip fabrication. The MPSoC design problem [2] involves multiple steps: (1) decomposition of an application into a set of tasks; (2) mapping of the tasks to a set of customizable processors; (3) optimization of each processor for the tasks assigned to it; (4) optimization of the communication between the processors. This work targets step (3) only. Given the description of a specific task or a set of tasks (or perhaps a complete application), we explore the design space of a single customizable processor by generating and evaluating custom instructions. The results of this thesis provide key components in the design space exploration for MPSoC devices.

The availability of automated tools for the synthesis of custom instructions reduces the time to market pressure and affects the nonrecurring engineering costs in the

design of custom instruction processors for SoC and MPSoC devices. Recent years have witnessed a significant progress in terms of formalizations and automation techniques for the synthesis of custom instructions. The advance in the theory is also reflected by the availability of successful commercial tools offering automated solutions, such as Tensilica's XPRES Compiler [16] and CoWare's CORXpert [17].

1.2. Contributions of the Thesis

In this thesis, we propose algorithms for automatic identification of the most profitable custom instructions for a baseline architecture given the functional specification of an application in C/C++. In addition, we develop a framework that allows us to explore the design space of custom instruction processors under various constraints and for different metrics. Our contributions can be summarized as follows:

(1) An Integer Linear Programming (ILP) Based Approach: We contribute a formulation of the problem of identifying custom instructions under constraints on the number of input and output operands as an integer linear programming (ILP) problem [18]. Alternatively, the constraints on the number of input and output operands can be relaxed, and the ILP model can evaluate the data transfer costs between the base processor and the custom logic given the data bandwidth of the interface [19, 20, 21]. Section 3.2 describes our ILP formulation in detail. Section 3.2.6 shows that the number of integer variables and the number of linear constraints used in our ILP model grow only linearly with the size of the problem. Section 4.8 demonstrates on a set of benchmarks that our ILP approach optimally solves very large problem instances representing basic blocks consisting of up to 1000 statements, with and without input/output constraints, most of the time within only a few seconds. We show that the state of the art techniques [22, 23, 24, 25] fail to find the optimal solutions on the same problem instances within reasonable time limits. In Section 4.3, we provide examples of solutions identified by our algorithms that are not covered by the existing methods.

(2) Custom Instruction Template Generation and Selection Algorithms: are contributed by integrating our ILP based solution into an iterative template generation

algorithm (Section 3.3). As part of our algorithms, structurally equivalent instruction templates are identified, behavioral descriptions of the custom datapaths are produced, and area and delay estimates are generated (Section 3.4). Finally, the most profitable candidates under area constraints are selected based on a Knapsack model.

(3) A Custom Hardware Instruction Processor Synthesis (CHIPS) Tool Chain for Design Space Exploration: We integrate our algorithms into an academic compiler infrastructure [26], automatically producing high level processor descriptions supporting the custom instructions and the assembly code utilizing the custom instructions. Based on the compiler feedback, we evaluate the impact of input/output constraints (Section 4.4), register file port constraints (Section 4.5), and code transformations such as if-conversion (Section 4.2) and loop unrolling (Section 4.6) on the performance and code size for a range of area constraints on eleven multimedia and cryptography benchmarks (Section 4.9). We integrate our compilation flow that generates behavioral descriptions of the custom datapaths into an academic custom processor synthesis infrastructure [27]. We observe that around half of the custom instruction processors automatically generated from C/C++ source codes meet the target clock frequency using fully automated ASIC synthesis flows (Section 4.10).

(4) A Simplified Model: In Chapter 5, we develop a simplified model for custom instruction identification based on the observation that the highest performing solutions are systematically found when the input/output constraints are removed (Section 4.4 and Section 4.9). We derive a practical upper bound on the worst case time complexity of the solution algorithms for the simplified problem.

2. BACKGROUND AND RELATED WORK

The CHIPS (Custom Hardware Instruction Processor Synthesis) tool chain described in this thesis is built on a compiler infrastructure. Therefore, we first review the compiler technology terminologies, and we introduce the notation we use in the thesis for representing application basic blocks and custom instructions (Section 2.1). Next, we give a review of architectural approaches for integrating a general purpose processor with custom logic (Section 2.2), and a review of technology choices for custom instruction processors (Section 2.3). Finally, we present the related work in automated hardware/software partitioning and custom instruction synthesis (Section 2.4).

2.1. Compilers

Compilers translate a program written in a high level language, such as C/C++ into an equivalent program in the assembly language of a target machine. A compiler consists of three main phases:

- Front end: The front end carries out lexical, syntax and semantic analysis of the source program, and converts it into a machine-independent intermediate representation (IR). A common approach is to transform the source code into a sequence of three address statements, where every statement is a label, a branch, or an assignment with at most two source operands and one destination operand.
- IR optimizer: The IR generated by the front end is not optimized. It may contain redundancies that are either introduced by the software programmer, or by the front end itself. Classical compiler optimizations, such as constant propagation, common subexpression elimination, dead code elimination, and jump optimization remove major redundancies and improve the code quality.
- Back end: The back end operates on the optimized IR, and generates the assembly code for the target machine. The back end implements code selection, register allocation, and instruction scheduling for the target machine. The back end often includes additional machine specific optimizations.

Compilers are key tools in the behavioral synthesis domain where functional descriptions of the applications are automatically mapped to hardware blocks. Compilers provide a formal representation of the application on which various analyses and optimizations can be carried out. Compilers are of paramount importance in the design of custom instruction processors as well. In this work, the compiler IR is the main input for the algorithms we propose for custom instruction identification. Furthermore, compilers provide various utilities that help in transforming programs into forms that are more suitable for our analysis. Moreover, once the custom instructions are identified and the custom datapaths are synthesized, the compiler has to be adapted so that the machine code utilizing custom instructions can be efficiently and automatically generated. Finally and most importantly, compilers play a key role in the design space exploration of custom instruction processors, providing feedback on the choice of custom instructions and in the evaluation of different program transformations.

2.1.1. Intermediate Representation

Several studies have targeted design of efficient compiler IR, since the choice of representation has an immediate effect on the complexity of the compiler optimization algorithms. In the most basic form, the IR is a sequence of three address statements. The typical approach transforms the IR into control flow graphs (CFG) and def-use chains [28] that represent the data flow. The unified control/data flow graph representation is widely used in the behavioral synthesis domain [29]. The control flow information is used in the generation of the control units, and the data flow information is used in the synthesis of the datapaths, allowing the exploration of various area and performance trade-offs. Other well known representations include program dependence graphs [30], static single assignment (SSA) form [31], and dependence flow graphs [32].

Given a three address statement S, which assigns a value to the destination operand x based on the values of the source operands y and z, we say that S defines x, and uses y and z. A data dependence exists between two statements if and only if one of the statements defines a storage (whether a scalar variable or a memory location) that is used or defined by the other. Assume two statements $S1$ and $S2$, where $S1$ is

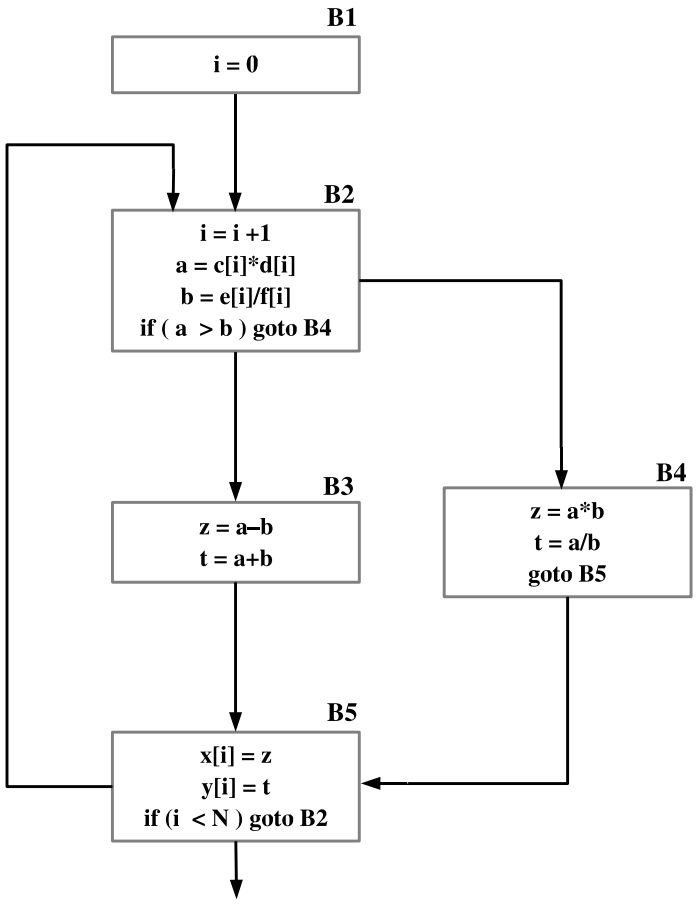


Figure 2.1. The control flow graph, the nodes represent basic blocks.

executed before $S2$. If $S2$ uses a storage defined by statement $S1$, we say that $S2$ is *flow dependent* on $S1$. If statement $S2$ redefines a storage used by statement $S1$, we say that $S2$ is *anti dependent* on $S1$. If statement $S2$ redefines a storage defined by statement $S1$, we say that $S2$ is *output dependent* on $S1$.

This thesis assumes that the compiler IR is transformed into a CFG, where the nodes of the CFG are the basic blocks of the application. A *basic block* is a sequence of consecutive statements in which the control flow always enters at the first statement and exits at the last statement. The edges of the CFG represent the flow of control across basic blocks. Figure 2.1 shows the control flow graph for a hypothetical application.

Within a basic block control dependencies do not exist, the only dependencies that exist are the data dependencies. A basic block can be viewed as a directed acyclic graph (DAG), where the nodes represent statements, and the edges represent

data dependencies between statements. A variable x is called *live* at a point p in the program if the value of x at p could be used by a statement that resides on some execution path of the program starting at p. The variables that are live coming into the basic block and the statements using these values before a redefinition within the basic block can be identified via liveness analysis and def-use chains [28]. The variables that are live out of the basic block and the statements defining these values within the basic block can be computed based on a similar analysis.

SSA form [31] ensures that each variable is defined only once in the IR, effectively eliminating the anti and output dependencies. If there are multiple definitions of the same variable in the code, the destination operands of the associated assignment statements are renamed with different versions of that variable. The statements using the variable are also modified so that their source operands match the version that is most recently defined. Although it is straightforward to rename the variables within basic blocks, multiple versions of the same variable may reach to the same point in the program from different basic blocks in the presence of conditionals or loops. In such cases, the so-called ϕ-functions, which are similar to multiplexers in hardware, merge different versions of the variable into a new version. Insertion of the ϕ-functions increases the number of statements in the IR, but the effect on the size is not significant.

2.1.2. Definitions for Custom Instruction Identification

For simplicity, we assume that the compiler IR is in SSA form. Therefore, each assignment statement is associated with a unique output variable that is assigned exactly once in the IR. We represent a basic block using a DAG $G\left(V_b \cup V_b^{in}, E_b \cup E_b^{in}\right)$ where the nodes V_b represent statements within the basic block, and the edges E_b represent flow dependencies between statements. The nodes V_b^{in} represent the input variables of the basic block, i.e., variables that are live coming into the basic block, and used within the basic block. The edges E_b^{in} connect V_b^{in} to the statements in V_b using these variables. The nodes $V_b^{out} \subseteq V_b$ represent statements defining variables that are live out of the basic block, i.e., the output variables of the basic block. The nodes $V_b^{invalid} \subseteq V_b$ represent statements that cannot be implemented by custom instructions

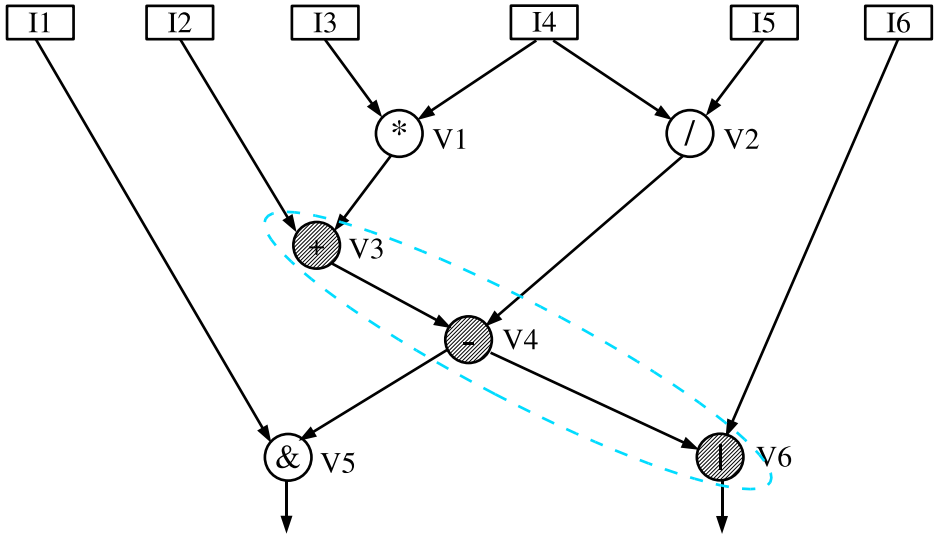


Figure 2.2. The DAG representation of a basic block.

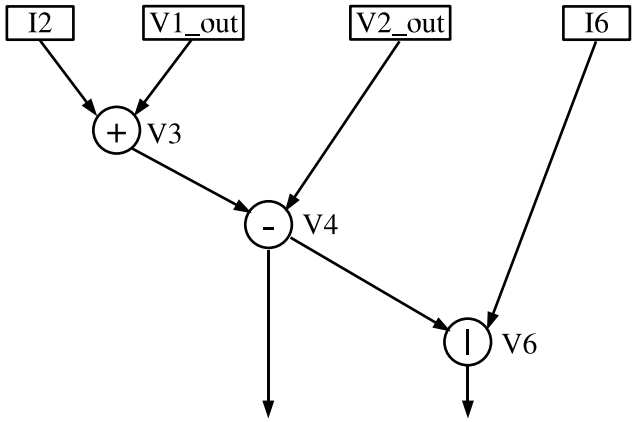


Figure 2.3. The custom instruction template extracted from Figure 2.2, having four input and two output operands.

either because of the limitations of the base processor architecture, or because of the limitations of the custom datapath, or by the choice of the designer.

Given a basic block $G\left(V_b \cup V_b^{in}, E_b \cup E_b^{in}\right)$, a custom instruction template can be uniquely defined by a set of statements $V_t \subseteq V_b / V_b^{invalid}$. A custom instruction template $T\left(V_t \cup V_t^{in}, E_t \cup E_t^{in}\right)$ is a *subgraph* of G. The nodes V_t represent statements included in the template, and the edges E_t represent flow dependencies between these statements. The nodes V_t^{in} represent the input operands of the template, i.e., variables that are

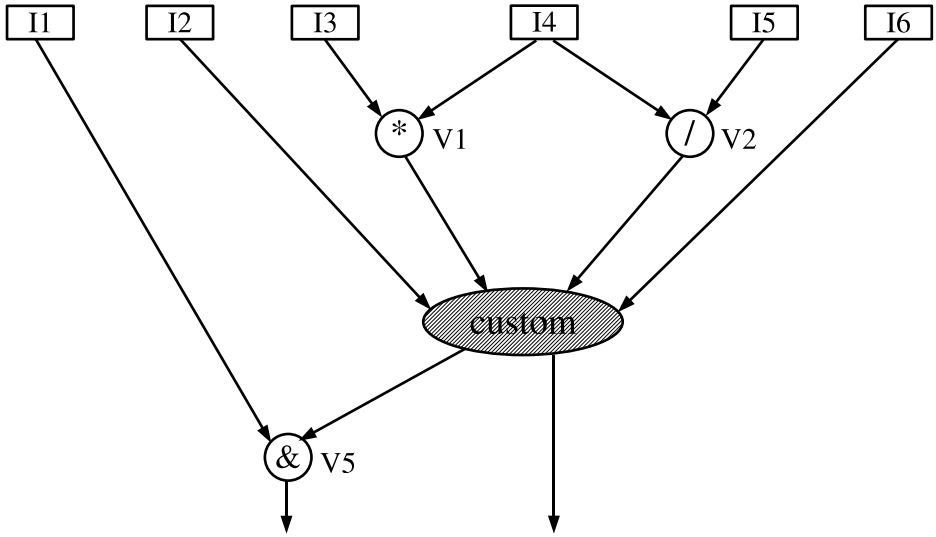


Figure 2.4. The DAG of Figure 2.2 after replacing the template shown in Figure 2.3 with a custom instruction.

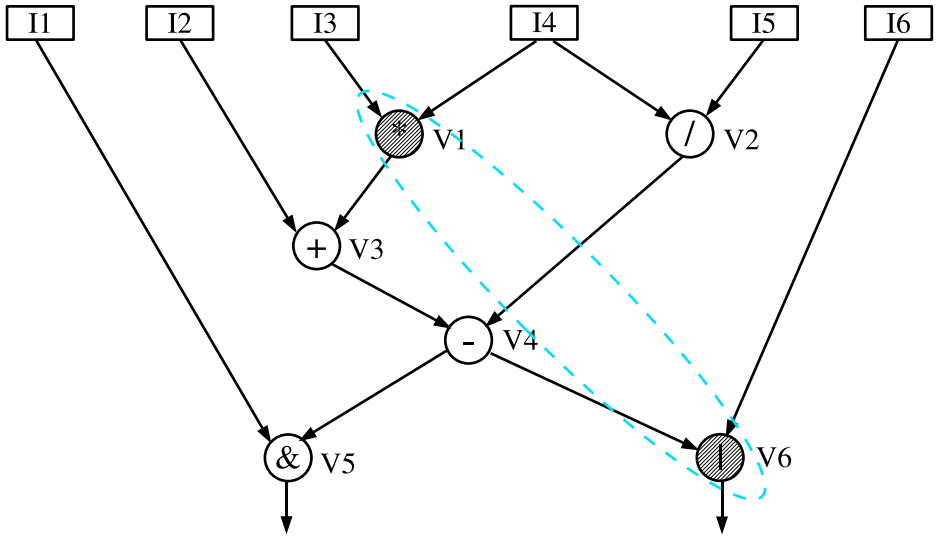


Figure 2.5. A custom instruction template that is not convex. The path from V1 to V6 that goes through V3 and V4 violates the convexity property.

live coming into the template, and used within the template. The input operands of T are either the input variables of G, or are defined by the statements in G that are not in T, i.e., $V_t^{in} \subseteq V_b^{in} \cup (V_b/V_t)$. The edges E_t^{in} connect V_t^{in} to the statements in V_t using these operands. The nodes $V_t^{out} \subseteq V_t$ represent statements defining variables that are live out of T, i.e., the output operands of T. A statement in V_t defining an output variable of G, defines also an output operand of T, i.e., $V_t \cap V_b^{out} \subseteq V_t^{out}$.

Figure 2.2 shows the flow data within an example basic block. Nodes I1 to I6 (i.e., V_b^{in}) represent variables that are defined outside the basic block, and used within the basic block. Nodes V1 to V6 (i.e., V_b) represent the statements of the basic block. Nodes V5 and V6 (i.e., V_b^{out}) define variables that are used outside the basic block. A subset of the statements (i.e., nodes V3, V4, and V6) are circled to define a custom instruction template. Figure 2.3 describes the extracted template with its internal data flow, input and output operands. The template has four input and two output operands. Inputs I2 and I6 are among the input variables of the basic block. Inputs V1_out and V2_out are defined by nodes V1 and V2 of the basic block. Nodes V4 and V6 define the output operands of the template. Finally, Figure 2.4 illustrates how the basic block looks like after the template is replaced by a custom instruction.

A template T is *convex* if there exists no path in G from a node $u \in V_t$ to another node $w \in V_t$ which involves a node $v \notin V_t$. The convexity constraint is imposed on the templates to ensure that custom instructions can be atomically executed on the processor datapath and that a feasible schedule can be achieved for the instruction stream. Figure 2.5 shows a template that is not convex. The path between the two template nodes V1 and V2 that goes through V3 and V4 violates the definition of the convexity. Replacing the template with a custom instruction introduces a cyclic dependency in the data flow graph. Such a dependency is not supported by standard compilers and architectures, and interpreted as an infeasible schedule.

2.2. Architectural Approaches for Custom Logic Integration

In this section, we give an overview of the architectural approaches that integrate a general purpose processor with custom logic for application acceleration. Figure 2.6 shows a classification of different integration methods according to the degree of coupling between the main processor and the custom logic:

- attached and external processing units that communicate with the main processor through a general purpose bus interface;
- tightly coupled coprocessors having direct access to the main processor;

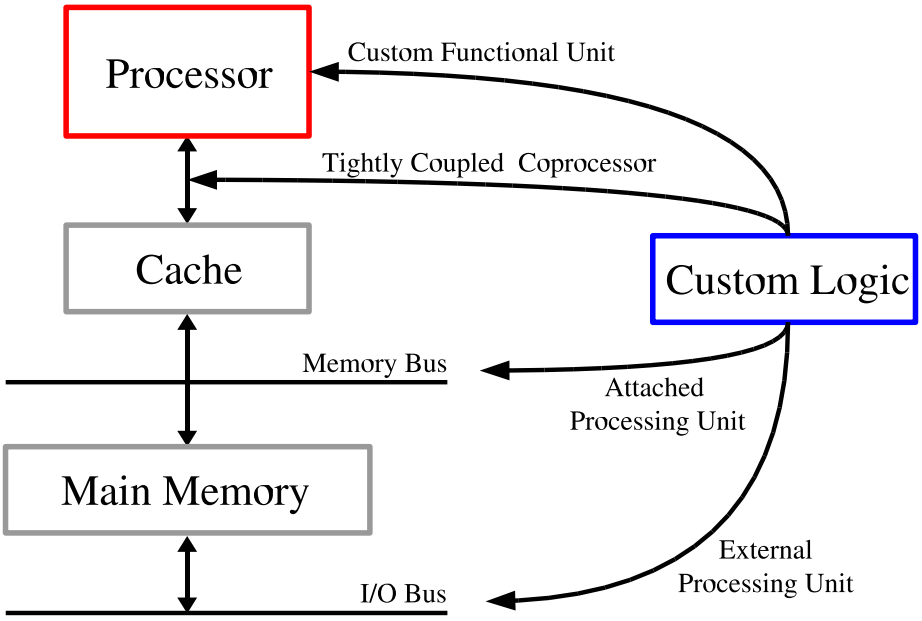


Figure 2.6. A classification of integration methods based on the degree of coupling. Attached and external processing systems are outside the scope of this work.

- processors with custom functional units (i.e., custom instruction processors).

2.2.1. Attached and External Processing Units

Attached and external processing units have no direct access to the main processor. The integration with the main processor is done through a general purpose bus interface, and it is relatively easy. However, the performance of such systems often suffers from the high communication overhead due to the bandwidth and latency limitations of the general purpose bus. This type of systems [33] can be beneficial in accelerating certain types of applications having a high computation to communication ratio, such as stream-based applications. Attached and external processing units, sometimes also called loosely coupled coprocessors, are outside the scope of this thesis.

2.2.2. Tightly Coupled Coprocessors

Unlike attached and external processing units, tightly coupled coprocessors are directly connected to the local bus or to the dedicated pins of the main processor.

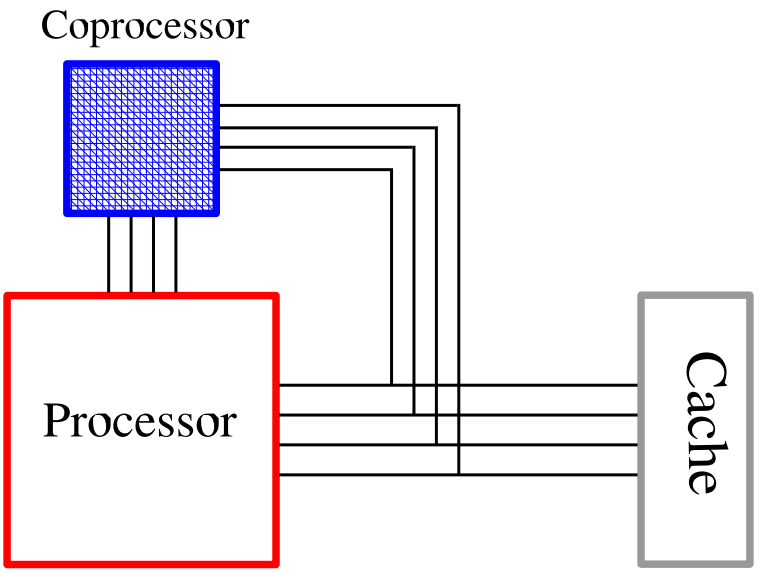


Figure 2.7. A tightly coupled coprocessor has direct access to the main processor through dedicated control and data transfer channels.

Tightly coupled coprocessors have the same view of the memory hierarchy as the main processor. The coprocessor has its own set of registers, control logic, and datapath. A coprocessor access protocol allows information exchange and synchronization with the main processor. The low-latency, high-bandwidth connection between the main processor and the coprocessor increases the number of program sections that can be profitably run on the coprocessor. A typical way of integrating a tightly coupled coprocessor with a main processor is shown in Figure 2.7.

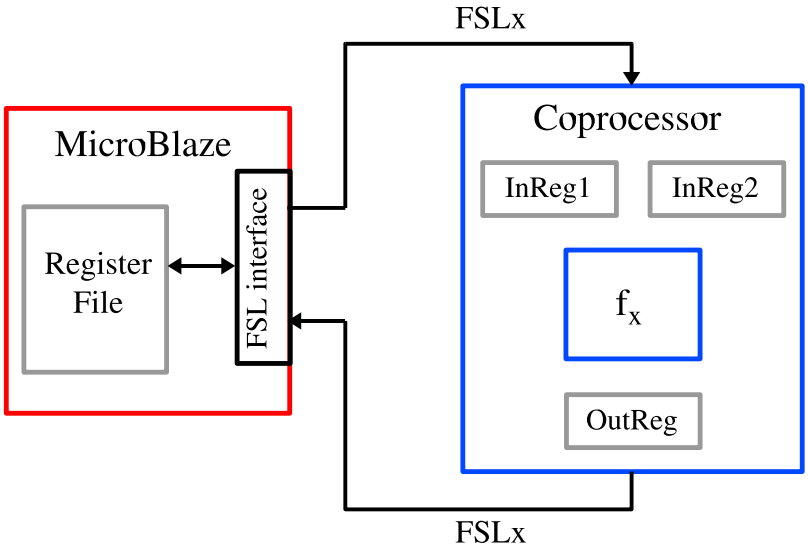
The ARM7TDMI processor [34], widely used in the smart cards due to its compactness and low power consumption characteristics, supports a highly flexible coprocessor interface. Operations that are computationally expensive on the processor, such as encryption and decryption functions, can be migrated to the coprocessor. The coprocessor has access to the instruction stream from the main memory and continuously keeps track of the instructions executed in the processor pipeline. Dedicated control signals allow handshaking between ARM7TDMI and the coprocessor. When a coprocessor instruction is decoded, the processor pipeline is stalled until the coprocessor completes the execution of the instruction. Coprocessor data processing instructions immediately activate the coprocessor datapaths. Coprocessor register transfer instruc-

tions enable low latency data transfers between processor registers and coprocessor registers through the data bus of the processor. Coprocessor load and store instructions allow data transfers between the data memory and the coprocessor registers.

The Garp approach [35] tightly couples a MIPS based processor [5] with a reconfigurable array consisting of logic blocks similar to those found in conventional FPGAs, with additional control blocks and routing channels that enable efficient configuration and efficient integration with the processor and the memory system. The processor controls the reconfigurable array by loading and executing configurations and issuing register transfer instructions. Unlike ARM7TDMI, the Garp approach allows the reconfigurable coprocessor to initiate memory accesses without processor intervention.

The ADRES architecture [36] couples a very long instruction word (VLIW) processor with a coarse-grained reconfigurable matrix. The VLIW processor can execute multiple instructions in parallel using functional units connected through a multiported register file. The reconfigurable matrix is composed of reconfigurable cells, which are made up of ALU like configurable functional units and local register files. The computation intensive loop kernels of the applications are mapped to the reconfigurable matrix based on an adaptation of the modulo scheduling algorithm [37]. The VLIW processor and the reconfigurable matrix cannot execute concurrently. This allows resource sharing between the main processor and the reconfigurable matrix. The multiported register file, the memory interface, and the functional units of the main processor are shared resources. Sharing of the multiported register file results in an efficient communication interface between the processor and the reconfigurable matrix.

The MicroBlaze [38] processor is an intellectual property of Xilinx optimized for embedded applications. MicroBlaze is offered as a soft core, i.e., it is implemented using the logic resources of an FPGA. The remaining logic resources of the FPGA can be configured to implement user defined functions. The Fast Simplex Link (FSL) interfaces of the MicroBlaze processors provide low-latency, high-bandwidth data transfer channels between MicroBlaze and user defined coprocessors. The FSL channels are unidirectional, point-to-point, dedicated first in, first out (FIFO) interfaces. MicroBlaze



Example Code

```
// send the input operands of function x through FSLx
put Ra, FSLx
put Rb, FSLx
// receive the output operands of function x through FSLx
get Rc, FSLx
```

Figure 2.8. Xilinx MicroBlaze [38]: contents of the MicroBlaze registers can be transferred to the coprocessor registers, and the results generated by the coprocessor can be transferred back to the MicroBlaze registers through the FSL channels.

supports up to eight input and eight output FSL channels. Contents of the MicroBlaze registers can be transferred to a coprocessor through the FSL channels using the *put* instructions of the MicroBlaze ISA. Similarly, the results generated by a coprocessor can be transferred back to the MicroBlaze registers using the *get* instructions of the MicroBlaze ISA. Both instructions can be blocking or nonblocking, and can be used in transferring data or control information. Nonblocking communication is a key feature of the FSL interfaces that allows Microblaze processors and custom coprocessors to operate asynchronously at different clock rates. Figure 2.8 illustrates the integration of a coprocessor with a MicroBlaze processor through the FSL channels.

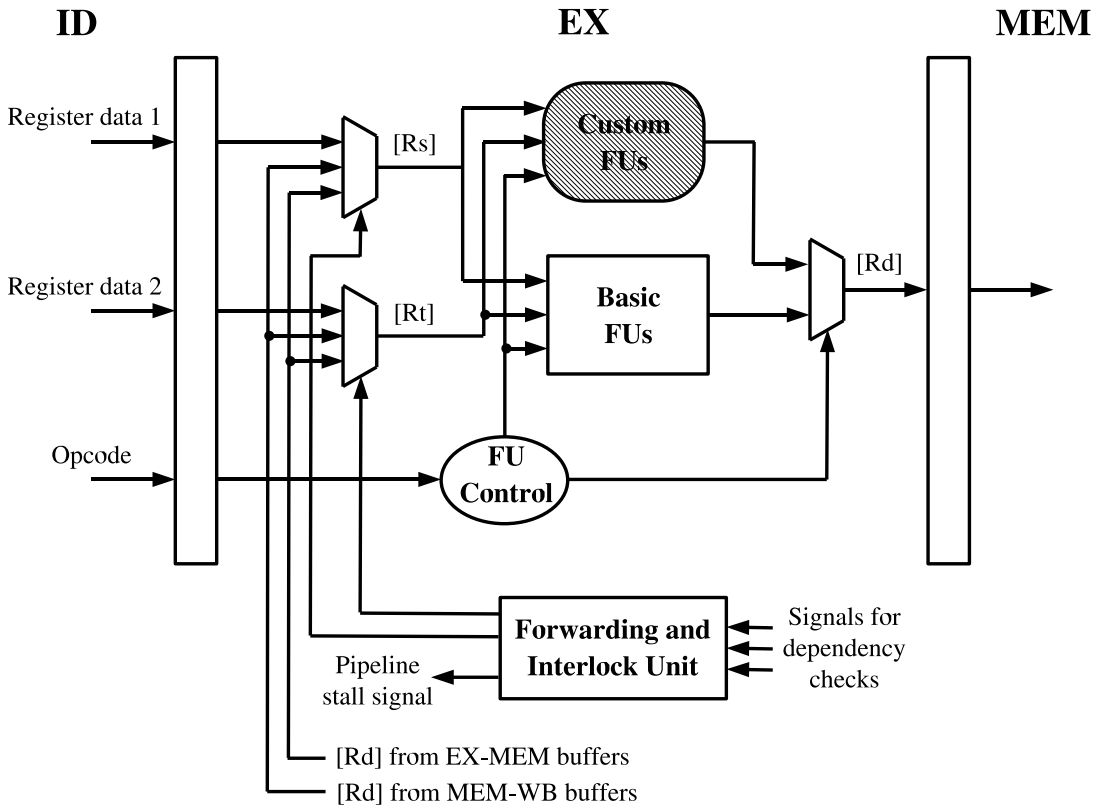


Figure 2.9. Integration of custom functional units (FUs) into the pipeline of a MIPS type single issue processor [5] with five pipeline stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write back (WB).

2.2.3. Custom Instruction Processors

Custom instruction processors further improve the data bandwidth between the processor and the custom logic and avoid the overhead of the coprocessor access protocol. Custom functional units are integrated into the regular datapath of the base processor in parallel to the basic execution units. Register file ports, operand buses, as well as the forwarding and the interlock logic can be shared between the basic execution units and the custom functional units. Figure 2.9 demonstrates a typical method of integrating the custom functional units into the pipeline of a MIPS like base processor.

Custom instruction processors are not limited to streaming type applications and can accelerate a wide range of applications. The improved data bandwidth and the reduced communication latency allows simple sequences of dataflow operations to be

profitably executed on custom logic as custom instructions. In fact, any combinational or sequential circuit could be implemented by custom instructions given a handshake mechanism between custom functional units and the base processor that allows the base processor to determine when the custom instruction execution is complete.

Modern custom instruction processors comprise parallel, deeply pipelined custom functional units including state registers, local memories, and wide data buses to local and global memories. These features make it possible for modern custom instruction processors to achieve a computational performance that is comparable to the performance of custom RTL blocks. In the near future, custom instruction processors may supersede the rigid RTL blocks in the design of embedded SoC devices.

An RTL block is typically composed of a datapath and a finite state machine based control logic. The datapaths are often regular blocks of computation, and can be reused in the implementation of multiple different applications. On the other hand, the control logic is completely application specific and can be highly irregular. Therefore, most of the risk in the design and verification of complex RTL blocks is associated with the control logic. Furthermore, a change in the system specification that occurs late in the design process affects the control logic most, whereas the elements of the datapath often remain unchanged and can still be used. Custom instruction processors can significantly reduce the design and verification effort by replacing the hardwired control logic with a software programmable base processor. The control flow is directed by the software running on the processor, and the instruction decoding logic generates the necessary control signals for the custom datapaths. The software based approach makes the design much more resilient against the changes in the system specification.

2.3. Technology Choices for Custom Instruction Processors

Custom instruction processors can be realized using different technology choices. In the following sections we describe the three main approaches in detail with some selected examples from the industry and the academia.

2.3.1. Synthesizable ASIC Processors

Synthesizable processors naturally match the standard ASIC design flows and can be easily integrated into complex SoCs. In addition, synthesizable processors provide the flexibility of fabrication in multiple foundries and processes. They can be quickly ported to a new foundry or process technology. Moreover, the same synthesizable processor description can be used to generate different circuits with different optimization goals in mind. The same design can be optimized for performance, power, or area by simply changing the target standard cell libraries and the synthesis constraints provided to the ASIC tools. Most importantly, the synthesizable processors can be customized to match the specific requirements of differerent applications. In particular, Tensilica Xtensa [9], ARC 600/700 [10], and MIPS Pro Series [12] processors allow the customization of their instruction-sets.

Synthesizable processors cannot reach the operating frequency of the processors designed with full custom circuit design techniques due to the limitations of standard cell libraries and ASIC synthesis tools. However, very high performance synthesizable processors are already commercially available. The recently introduced MIPS32 74K [39] synthesizable processor with a superscalar out-of-order pipeline can operate at a frequency of one gigahertz (GHz) and provides support for custom instructions.

The synthesizable Tensilica Xtensa processor allows the designers to express the custom instructions in a high level language called the Tensilica Instruction Extension (TIE) language [40]. Unlike architecture description languages, such as nML [7] and LISA [8], TIE is not intended to describe a complete ISA. TIE allows the designers to define a wide variety of custom instructions for extending a selected base processor configuration. The formats and the encoding of custom instructions, and the custom datapaths that implement custom instructions are specified using high level constructs. The TIE compiler automatically generates the synthesizable processor descriptions. Custom instructions can be single cycle or multi-cycle. The TIE compiler automatically generates the pipeline registers, the interlock and the forwarding logic. Custom instructions may involve architecturally visible custom state registers, as well.

State registers can store some of the temporary variables in the custom logic during the program execution and allow custom instructions to operate on more input and output operands than what the core register file supports. Finally, the TIE compiler generates the scheduled machine code that makes good use of the custom datapaths.

In addition to the custom instruction support, the Tensilica Xtensa processors allow the customization of their ISA using VLIW and vector instructions [14]. A VLIW instruction is composed of multiple independent operations that can be issued in parallel. The performance increase is at the expense of hardware cost, such as parallel decoders, multiple independent execution units, and a multiported register file that supports simultaneous access. Vector or SIMD (Single Instruction Multiple Data) instructions can improve the performance by operating on more than one data element at the same time. The improvement in the performance is again at the expense of hardware cost, such as vector register files for storing vectors of data elements and the additional logic for executing vector operations. Custom instructions can compress several dependent and independent operations into a single hardware optimized operation. Latency of the custom instructions can be significantly lower than the combined latency of the simple operations. Moreover, custom instructions can significantly reduce the code size, the number of register file ports, and the issue width for VLIW processors. However, custom instructions have an area overhead, too. The Xtensa approach combines all three techniques (i.e., VLIW, vector, and custom instructions) in order to obtain the maximal performance improvement at a given area cost.

2.3.2. Soft Core FPGA Processors

Shrinking process geometries increase the fabrication costs of ASICs. In addition, the process variability and the signal integrity problems are getting harder to cope with, and this makes the design cycles of ASICs longer [41]. FPGAs eliminate the nonrecurring engineering costs in the fabrication of the chip and significantly shorten the design cycles and the time to market. The unit costs of FPGA devices are higher than the unit costs of ASIC devices, and the logic density and the performance of FPGAs cannot match those of ASICs. However, the reprogrammable nature of FPGA

devices allows FPGA based designs to quickly adapt to the changes in the system specification and increases their life times. The re-programmability and the time to market advantages of FPGAs make them highly competitive solutions for the industry.

A soft core FPGA processor is provided as a hardware description language source code or as a structural netlist that can be mapped to the reconfigurable resources of an FPGA device. Modern FPGA devices have sufficient density and resources that enable efficient implementation of several soft core processors. Altera's Nios II [11] processor is among the most well-known commercial examples. Nios II custom instructions are custom logic blocks connected directly to the operand buses of the ALU in the processor's datapath (similar to Figure 2.9). The custom instruction can have a single cycle (combinational) or multi-cycle (sequential) execution latency. Multi-cycle instructions can have a fixed or variable duration. Custom logic blocks with a variable duration inform the base processor of the completion of their execution using dedicated handshake signals. Nios II architecture allows definition of up to 256 custom instructions. Nios II also supports definition of internal register files and provides access to external logic or memories through FIFO channels. These features can significantly improve the capabilities of custom instructions and the efficiency of the Nios II processors.

Seng et al. [42] describe a customizable soft core processor that can adapt to the application running on it by modifying its instruction-set through run-time reconfiguration of the FPGA resources. Dimond et al. [43] introduce a customizable soft processor with multi-threading support that can hide the latency of the memory accesses. In [27], it is shown that the energy consumption of a softcore processor can be significantly reduced by aplication specific encoding and re-ordering of the instructions.

2.3.3. Hard Core Processors with Reconfigurable Datapaths

The soft core processors operate at a limited frequency and cannot achieve the performance efficiency of their counterparts produced in ASIC technology or using full custom circuit design techniques. An alternative way of introducing processor functionality efficiently within FPGA devices is to use embedded hard processor cores.

Such cores can operate at very high frequencies while occupying a relatively small area on the die. Commercial examples exist, such as Altera's Excalibur devices integrating ARM922T cores and Xilinx Virtex-II FPGAs integrating PowerPC 405 cores. Although these architectures do not support custom instructions, several research groups have already studied possible ways of integrating a high performance processor with reconfigurable functional units [44, 45, 46, 47] for efficient custom instruction support. The Chimaera approach [46] introduces a special purpose reconfigurable array with partial run-time reconfiguration support. The ConCISe architecture [47], on the other hand, proposes the use of a complex programmable logic device (CPLD) that enables extensive logic minimization and improves the utilization of the resources.

The Stretch S6000 family of processors [13] can be classified into this category. The baseline processor is a Tensilica Xtensa processor configured as a dual issue VLIW with vector instruction support fabricated in ASIC technology. Custom instructions are mapped to a special purpose coarse grain reconfigurable fabric that includes numerous ALUs, distributed registers, multipliers, and RAM blocks.

2.4. Related Work on Hardware/Software Partitioning Algorithms

Automatic hardware/software partitioning is a key problem in the hardware and software co-design of embedded systems. The traditional approach assumes a processor and a coprocessor integrated through a general purpose bus interface [48, 49, 50, 51]. Hardware/software partitioning is done at the task or basic block level. The system is represented as a graph, where the nodes represent tasks or basic blocks, and the edges are weighted based on the amount of communication between the nodes. Gupta et al. [48] initially allocate all nodes in hardware. Area cost is reduced by iterative movements from hardware to software while trying not to exceed a constraint on the schedule length. Ernst et al. [49] propose a simulated annealing based methodology. Niemann et al. [50] formulate the hardware/software partitioning problem under area and schedule length constraints as an ILP problem. In [51], Vahid and Le extend the Kernighan-Lin heuristic for hardware/software functional partioning. Hardware/software partitioning problem under area and schedule length constraints is shown to be NP-hard in [52].

Custom instruction processors are emerging as an effective solution in the hardware and software co-design of embedded systems. In the context of custom instruction processors, hardware/software partitioning is done at the instruction level granularity. Application basic blocks are transformed into DAGs, where the nodes represent instructions similar to those in assembly languages, and the edges represent data dependencies. Profiling analysis identifies the most time consuming basic blocks. Code transformations, such as loop unrolling and if-conversion, selectively eliminate control flow dependencies and merge application basic blocks. Custom instructions provide efficient hardware implementations for frequently executed dataflow subgraphs.

Custom instruction identification is seemingly similar to traditional microcode compaction techniques [53]. In both cases, the optimization goals include reducing the schedule length and the code size of a given application. The main difference is that in the context of custom instruction processors we derive specialized processor datapaths in order to optimize the application execution. On the other hand, microcode compaction techniques assume a fixed processor datapath, and try to identify combinations of microoperations that can fully exploit the parallelism within that fixed datapath.

Automatic identification of custom instructions has remained an active area of research for more than a decade. The automation effort is motivated by manual design examples, such as [42, 54], which demonstrate the importance of identifying coarse grain and frequently used code segments. The mainstream approach divides the custom instruction identification problem into two phases: (1) generation of a set of custom instruction templates; (2) selection of the most profitable templates under area or schedule length constraints. Most of the early research and some of the recent work [55, 56, 57, 58, 59, 60, 61] rely on incremental clustering of related DAG nodes in order to generate a set of custom instruction templates. In [62], Alippi et al. introduce the MaxMISO algorithm, which partitions a DAG into maximal input, single output subgraphs in linear run-time. In [15], Binh et al. propose a branch-and-bound based algorithm for the selection problem in order to minimize the area cost under schedule length and power consumption constraints. In [63], no explicit constraint is imposed on the number of input or output operands for custom instructions. The generation

and the selection of custom instruction templates are based on greedy techniques.

Cheung et al. generate the custom instruction templates based on exhaustive search in [64]. The exhaustive search approach is not scalable since the number of possible templates grow exponentially with the size of the DAGs (given a DAG with N nodes there exists 2^N distinct DAG subgraphs induced by the nodes). In [22, 23], Atasu et al. introduce constraints on the number of input and output operands for subgraphs in order to reduce the exponential search space. Application of a constraint propagation technique reduces the number of enumerated subgraphs significantly. A greedy algorithm iteratively selects non-overlapping DAG subgraphs having maximal speed-up potential based on a high level metric. The proposed technique is often limited to DAGs with a few hundred nodes, and the input/output constraints must be tight enough to reduce the exponential worst case time complexity. The work by Atasu et al. shows that clustering based approaches (e.g. [59]) or single output operand restriction on the custom instructions, (e.g. [62]) can severely reduce the achievable speed-up.

Cong et al. propose a dynamic programming based algorithm in [65], which enumerates single output subgraphs with a given number of inputs. In [66], Yu et al. show that subgraph enumeration under input and output constraints can be done much faster if the additional connectivity constraint is imposed on the subgraphs. In [24], the algorithm of [22, 23] is further optimized, and it is shown that enumerating connected subgraphs only can substantially reduce the achievable speed-up. Biswas et al. propose an extension to the Kernighan-Lin heuristic again based on input and output constraints in [67]. This approach does not evaluate all feasible subgraphs. Therefore, an optimal solution is not guaranteed. In [68], Bonzini et al. derive a polynomial bound on the number of feasible subgraphs if the number of inputs and outputs for the subgraphs are fixed. However, the complexity grows exponentially as the input/output constraints are relaxed. Performance of the proposed algorithm is reported to be similar to the performance of the algorithm described in [24].

In [69], Leupers et al. describe a code selection technique for irregular datapaths with complex instructions. In [61], Clark et al. formulate the problem of matching

a library of custom instruction templates with application DAGs as a subgraph isomorphism problem. Peymandoust et al. propose a polynomial manipulation based technique for the matching problem in [70]. In [65], Cong et al. use isomorphism testing to determine whether enumerated DAG subgraphs are structurally equivalent. In [71], Cheung et al. use model equivalence checking to verify whether generated subgraphs are functionally equivalent to a pre-designed set of library components.

Clark et al. propose a reconfigurable array of functional units tightly coupled with a general purpose processor that can accelerate dataflow subgraphs in [72]. A microarchitectural interface and a compilation framework allows transparent instruction-set customization. DAG merging techniques that can exploit structural similarities across custom instructions for area efficient synthesis are proposed in [73, 74, 75].

The speed-up obtainable by custom instructions is limited by the available data bandwidth between the base processor and custom logic. Extending the core register file to support additional read and write ports improves the data bandwidth. However, additional ports result in increased register file size, power consumption, and cycle time. The Tensilica Xtensa [14] processor uses custom state registers to explicitly move additional input and output operands between the base processor and custom units. Hauck et al. propose the use of shadow registers to increase the data bandwidth in [46]. Shadow registers duplicate a subset of the base processor registers in the custom logic area. Contents of the shadow registers can be read without any limitation on the bandwidth. The mapping between the base processor registers and the shadow registers can be fixed or can be established at compile time [76]. Jayaseelan et al. demonstrate that up to two additional input operands for custom instructions can be supplied free of cost by exploiting the forwarding paths of the base processor in [77]. Pozzi et al. [78] show that the data transfer overhead of multi-cycle custom instructions can be reduced by overlapping data transfers cycles with execution cycles.

Another potential complexity in the design of custom instruction processors is the difficulty of encoding multiple input and output operands within a fixed length instruction word. Issuing explicit data transfer instructions to and from custom state

Table 2.1. A comparison with some state-of-the-art techniques

	[22, 23, 24]	[65]	[66]	Our work
Controllability of inputs	√	√	√	√
Controllability of outputs	√		√	√
Support for disconnectedness	√			√
Removal of I/O constraints				√

registers is a way of encoding the additional input and output operands. An orthogonal approach proposed by Lee et al. in [79] restricts the input and output operands for custom instructions to a subset of the base processor registers. This approach effectively reduces the bit-width necessary for operand encoding. However, additional data movement instructions between the base processor registers may be necessary. Tensilica Xtensa LX processors [80] introduce flexible instruction encoding support for multi-operand instructions, known as FLIX, in order to address the encoding problem.

In this thesis, we assume a baseline machine that supports architecturally visible custom state registers, and dedicated instructions that can transfer data between base processor registers and custom state registers. We optionally constrain the number of input and output operands for custom instructions, and we explicitly account for the data transfer cycles between the base processor and the custom logic if the number of inputs or outputs exceed the available register file ports. We explore compiler transformations, such as if-conversion [81] and loop unrolling that can partially eliminate control dependencies, and we apply our algorithms on predicated basic blocks.

Today's increasingly advanced ILP solvers such as CPLEX [82] are often able to solve problems with thousands of integer variables and tens of thousands of linear constraints efficiently. To take advantage of this widely used technology, we formulate the custom instruction identification problem as an ILP in Chapter 3. We show that the number of integer variables and the linear constraints used in our ILP formulation grow only linearly with the size of the problem. In Section 3.3, we integrate our ILP based solution into an iterative algorithm, used also in [22, 23, 24], which reduces the

search space based on a *most profitable subgraph first* approach. Table 2.1 compares our approach with some state-of-the-art techniques in terms of the supported features.

2.5. Summary

In this chapter, we first introduce the compiler terminology and the notation for automated custom instruction identification. After that, we review the architectural approaches that integrate custom logic with a base processor for application acceleration. Next, we describe possible technology alternatives for implementing custom instruction processors. Finally, we provide an overview of the existing work on automated hardware/software partitioning and custom instruction synthesis algorithms and we compare and contrast our approach with the existing techniques.

3. THE CHIPS APPROACH

Today, the most widely used language for programming embedded devices is C/C++. Therefore, we assume that the functionality of the application to be accelerated is specified in C/C++. We target customizable processor architectures, similar to Tensilica Xtensa [9], where the data bandwidth between the base processor and the custom logic is constrained by the available register file ports. Our approach is also applicable to processor architectures that support tightly coupled coprocessors, where the data bandwidth is limited by dedicated data transfer channels, such as the Fast Simplex Link channels of the Xilinx MicroBlaze processor [38]. Given the available data bandwidth and the transfer latencies, we identify the most profitable custom instructions based on an integer linear programming (ILP) model. Figure 3.1 depicts our tool chain called CHIPS (Custom Hardware Instruction Processor Synthesis).

We use the Trimaran [26] compiler infrastructure to generate the control and data flow information, and to achieve basic block level profiling of a given application. Specifically, we work with Elcor, the back-end of Trimaran. The Impact front-end and the Elcor back-end already support most of the well-known compiler optimizations. Additionally, we implement an if-conversion pass adapted to our purposes, which selectively eliminates control flow dependencies due to branches. We apply our algorithms for identifying custom instructions immediately after code selection and prior to register allocation. Therefore, the statements in the IR are mapped to the instructions supported by the base processor, and no register spills are visible at this stage.

Section 3.2 describes a scalable ILP formulation that identifies the most promising data flow subgraphs as custom instructions templates. Our ILP formulation guarantees a feasible schedule for the generated templates. Furthermore, our ILP formulation explicitly calculates the data transfer costs and the critical path delays, and identifies the templates that reduce the schedule length most. We use CPLEX Mixed Integer Optimizer [82] within our algorithms to solve the ILP problems we generate.

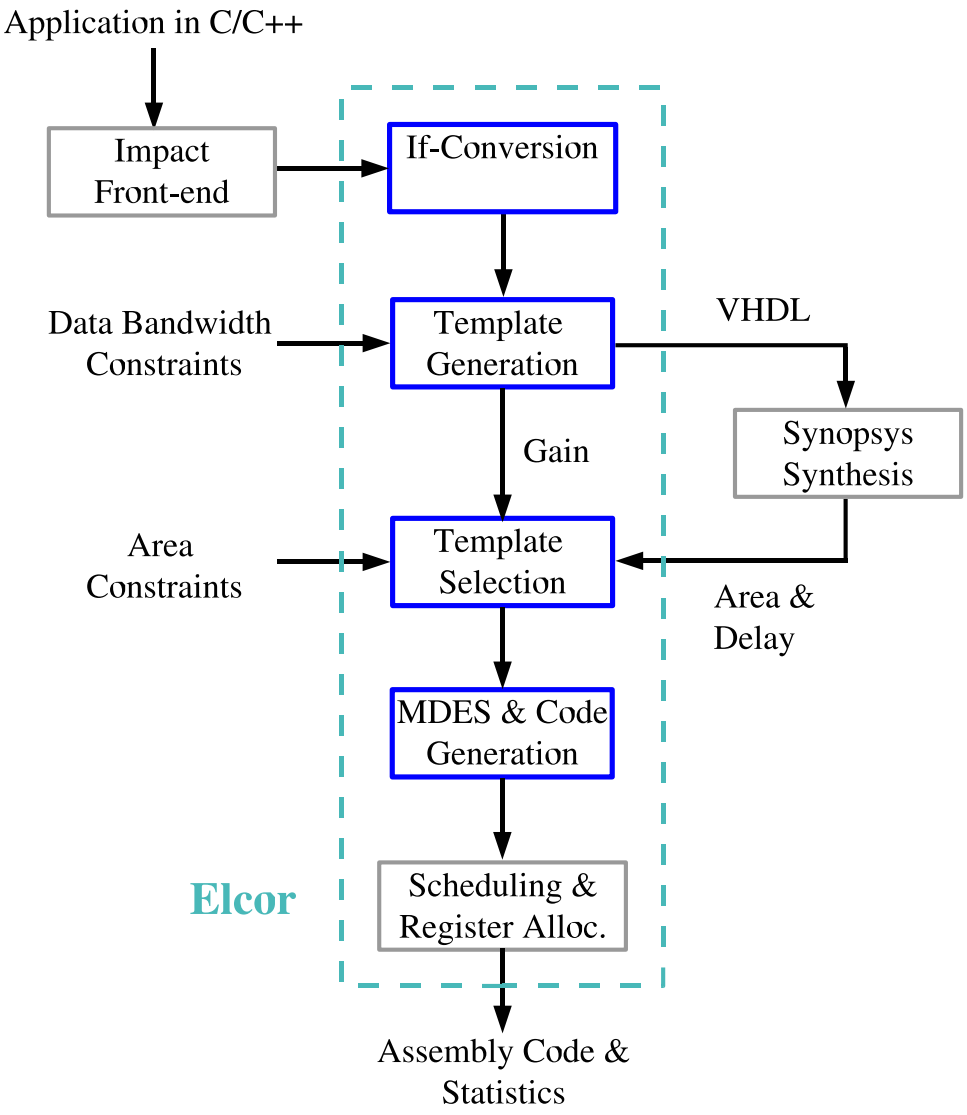


Figure 3.1. CHIPS: we integrate our algorithms into Trimaran [26]. Starting with C/C++ code, we automatically generate behavioral descriptions of custom instructions in VHDL, a high level machine description (MDES), and assembly code.

Given the available data bandwidth and the transfer latencies between the base processor and the custom logic, our *template generation* algorithm iteratively solves a set of ILP problems and produces a set of custom instruction templates that can be profitably run on the custom logic. Section 3.3 describes our template generation algorithm in detail. Our *template selection* algorithm groups structurally equivalent templates within isomorphism classes as custom instruction candidates. Given the behavioral descriptions of the custom instruction candidates in VHDL, Synopsys Design Compiler [83] produces area and delay estimates. Given constraints on the available

custom logic area, the template selection algorithm identifies the most profitable candidates. Section 3.4 describes our template selection algorithm in detail.

Once the most profitable custom instruction candidates are selected under area constraints, we automatically generate high level machine descriptions (MDES [84]) supporting the selected candidates. Next, we insert the custom instructions in the code, and replace the matching DAG subgraphs. Finally, we apply Trimaran scheduling and register allocation passes, and we produce the assembly code and scheduling statistics.

3.1. Problem Formulation

In this section, we introduce our problem formulation using the notation described in Section 2.1.2. We represent a basic block using a DAG $G\left(V_b \cup V_b^{in}, E_b \cup E_b^{in}\right)$ where V_b represent statements, and the edges E_b represent flow dependencies between statements. The nodes V_b^{in} represent the input variables, and the edges E_b^{in} connect V_b^{in} to the statements in V_b. The nodes $V_b^{out} \subseteq V_b$ represent statements defining the output variables. The nodes $V_b^{invalid} \subseteq V_b$ represent statements that are not permitted to take part in custom instructions. A custom instruction template $T\left(V_t \cup V_t^{in}, E_t \cup E_t^{in}\right)$ is a *subgraph* of G. The nodes V_t represent statements included in the template, and the edges E_t represent flow dependencies between these statements. The nodes V_t^{in} represent the input operands of the template and the edges E_t^{in} connect V_t^{in} to the statements in V_t. The nodes $V_t^{out} \subseteq V_t$ represent statements defining the output operands of T.

We assume that the ports of the core register file are shared between the base ALU and the custom units. Given RF_{in} read ports and RF_{out} write ports supported by the core register file, we assume that RF_{in} input operands can be read, and RF_{out} output operands can be written back free of cost by the custom instructions. Furthermore, we assume that RF_{in} input operands and RF_{out} output operands can be encoded within a single instruction word. We denote the number of input operands for custom instructions as $INST_{\text{in}}$, and the number of output operands as $INST_{\text{out}}$. Figure 3.2 duplicates Figure 1.4 and shows the RF_{in}, RF_{out}, $INST_{\text{in}}$ and $INST_{\text{out}}$ parameters of our model on the datapath of the custom instruction processor.

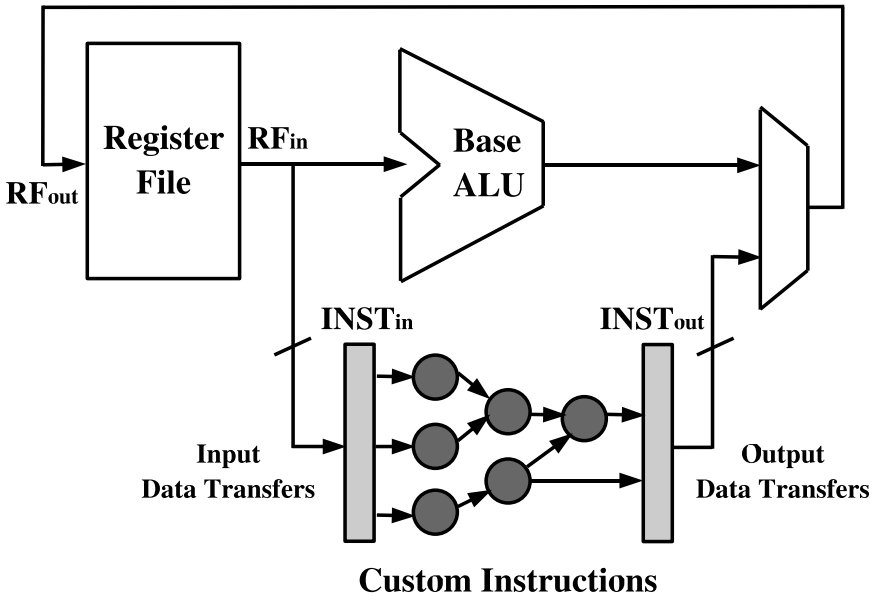


Figure 3.2. The register file supports RF_{in} read ports and RF_{out} write ports. Custom instructions might have an arbitrary number of input and output operands, $INST_{\text{in}}$ and $INST_{\text{out}}$, marshalled in and out through dedicated data transfer channels.

If the baseline machine supports architecturally visible custom state registers and dedicated data transfer instructions, $INST_{\text{in}}$ and $INST_{\text{out}}$ can be arbitrarily large. If $INST_{\text{in}}$ is larger than RF_{in} or $INST_{\text{out}}$ is larger than RF_{out}, we issue additional data transfers between the core register file and custom state registers. We assume that the cost of transferring additional RF_{in} input operands from the core register file to the custom state registers is c_1 cycles, and the cost of transferring additional RF_{out} output operands from the custom state registers to the core register file is c_2 cycles, where $c1, c2 \in Z^+ \cup \{0\}$. We denote the number of additional data transfers from the core register file to the custom state registers as DT_{in} and the number of additional data transfers from the custom state registers to the core register file as DT_{out} respectively.

If the baseline machine does not support architecturally visible custom state registers and dedicated data transfer instructions, $INST_{\text{in}}$ and $INST_{\text{out}}$ is practically limited by the available register file ports or by the finite operand encoding space available in the instruction word. The number of input and output operands may also be bounded explicitly for the purpose of exploring the design space. In this work, we support a constraint of MAX_{in} on the maximum number of input operands, and a

constraint of MAX_{out} on the maximum number of output operands for the templates.

We associate with every DAG node $v_i \in V_b$ a software latency $s_i \in Z^+$, and a hardware latency $h_i \in R^+ \cup \{0\}$. Software latencies give the time in clock cycles that it takes to execute the operations on the pipeline of the base processor. Hardware latencies are given in a unit of clock cycles. Hardware latencies are calculated by synthesizing the operators and normalizing their delays to a target cycle time. We estimate the software latency $S(T)$ of a template T (i.e., the execution latency on the base processor) by accumulating the software latencies of the nodes that are included in V_t. We estimate the hardware latency $H(T)$ (i.e., the execution latency on the custom logic) by quantizing its hardware critical path length. We represent the data transfer cost (i.e., the communication latency) induced by a template T as $C(T)$.

Our objective is to minimize the schedule length of the application by moving templates from software to hardware (i.e., from the base processor to the custom logic) as custom instructions. We search for templates, which when moved from software to hardware, maximize the reduction in the schedule length. Given a template T we estimate the schedule length reduction as the difference between the software latency $S(T)$ and the sum of hardware latency $H(T)$ and the communication latency $C(T)$.

3.2. Integer Linear Programming Model

We associate with every DAG node $v_i \in V_b$ a binary decision variable x_i that represents whether the node is included in the template ($x_i = 1 \Leftrightarrow v_i \in V_t$) or not ($x_i = 0 \Leftrightarrow v_i \notin V_t$). For $v_i \in V_b^{invalid}$ we set $x_i = 0$. We use x_i' to denote the complement of x_i ($x_i' = 1 - x_i$). In this way, it is possible to encode all $2^{|V_b|}$ possible templates given a DAG with $|V_b|$ nodes. Figure 3.3 shows the assignment of binary variables to the nodes for the template shown in circle. Setting x_3, x_4, and x_6 equal to one, and setting x_1, x_2, and x_5 equal to zero defines a template T, where $V_t = \{v_3, v_4, v_6\}$.

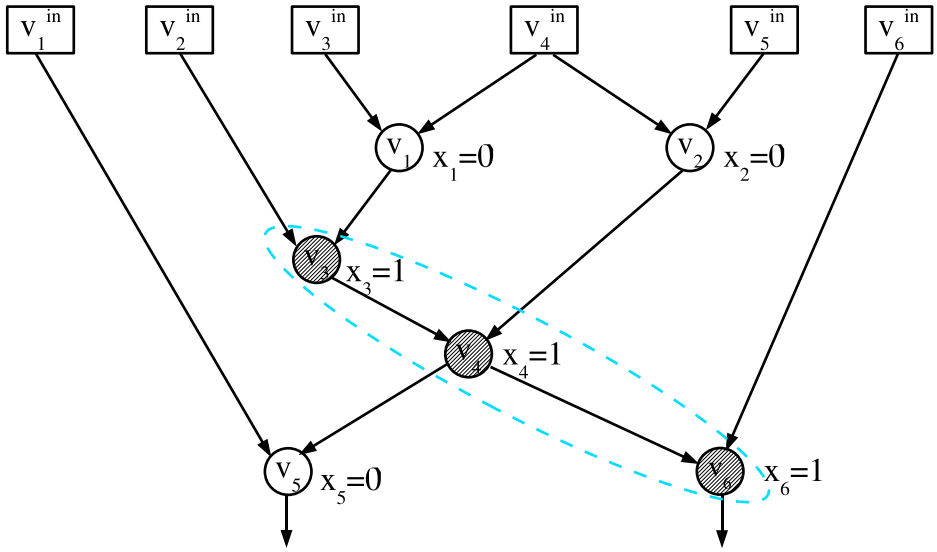


Figure 3.3. Templates are defined based on the assignment of ones and zeros to the binary decision variables associated with the DAG nodes.

We use the following indices in our formulations:

$$I_1: \quad \textit{indices for nodes } v_i^{in} \in V_b^{in}$$
$$I_2: \quad \textit{indices for nodes } v_i \in V_b$$
$$I_3: \quad \textit{indices for nodes } v_i \in V_b^{out}$$
$$I_4: \quad \textit{indices for nodes } v_i \in V_b / V_b^{out}$$

We represent the set of immediate successors of the nodes in V_b^{in} defined by E_b^{in} as follows:

$$Succ(i \in I_1) = \left\{ j \in I_2 \mid \exists\ e \in E_b^{in} : e = (v_i^{in}, v_j) \right\}$$

We represent the set of immediate successors, and the set of immediate predecessors of the nodes in V_b defined by E_b as follows:

$$Succ(i \in I_2) = \{ j \in I_2 \mid \exists\ e \in E_b : e = (v_i, v_j) \}$$
$$Pred(i \in I_2) = \{ j \in I_2 \mid \exists\ e \in E_b : e = (v_j, v_i) \}$$

3.2.1. Calculation of the Input Data Transfers

We use the integer decision variable $INST_{\text{in}}$ to represent the number of input operands of a template T. An input variable $v_i^{in} \in V_b^{in}$ of the basic block is an input operand for T if it has at least one immediate successor in V_t. A node $v_i \in V_b$ defines an input operand of T if it is not in V_t, and it has at least one immediate successor in V_t. The total number of input operands for T can be computed as follows:

$$INST_{\text{in}} = \sum_{i \in I_1} \left(\bigvee_{j \in Succ(i)} x_j \right) + \sum_{i \in I_2} \left(x_i' \wedge \left(\bigvee_{j \in Succ(i)} x_j \right) \right) \tag{3.1}$$

We calculate the number of additional data transfers from the core register file to the custom logic as DT_{in}:

$$DT_{\text{in}} \geq INST_{\text{in}}/RF_{\text{in}} - 1, \qquad DT_{\text{in}} \in Z^+ \cup \{0\} \tag{3.2}$$

A constraint of MAX_{in} on the maximum number of input operands can be imposed as follows:

$$INST_{\text{in}} \leq MAX_{\text{in}} \tag{3.3}$$

In Figure 3.3, the input variable v_2^{in} of the basic block has its immediate successor v_3 in V_t (i.e., $x_3 = 1$). Similarly, v_6^{in} has its immediate successor v_6 in V_t (i.e., $x_6 = 1$). Therefore, both v_2^{in} and v_6^{in} are input operands for T. On the other hand, none of the immediate successors of v_4^{in} are in V_t. Therefore, v_4^{in} is not an input operand for T. Among the basic block nodes, v_1 is not in V_t, and its immediate successor v_3 is included in V_t (i.e., $x_1' \wedge x_3 = 1$). Similarly, v_2 is not in V_t, and its immediate successor v_4 is included in V_t (i.e., $x_2' \wedge x_4 = 1$). Therefore, both v_1 and v_2 define input operands of T. The total number of input operands of T is calculated as four.

Linearization of the logical constraints is carried out as follows: Equation (3.1) is rewritten as:

$$INST_{\text{in}} = \sum_{i \in I_1} z_i^{in} + \sum_{i \in I_2} z_i \tag{3.4}$$

where, we define the auxiliary binary decision variables z_i^{in}, z_i, and t_i as follows:

$$z_i^{in} = \bigvee_{j \in Succ(i)} x_j, \qquad i \in I_1 \tag{3.5}$$

$$z_i = x_i' \wedge t_i, \qquad i \in I_2 \tag{3.6}$$

$$t_i = \bigvee_{j \in Succ(i)} x_j, \qquad i \in I_2 \tag{3.7}$$

Equation (3.5) can be written as follows:

$$\begin{aligned} z_i^{in} &\geq x_j, \qquad j \in Succ(i), \qquad i \in I_1 \\ z_i^{in} &\leq \sum_{j \in Succ(i)} x_j, \qquad i \in I_1 \end{aligned} \tag{3.8}$$

Equation (3.6) can be written as follows:

$$\begin{aligned} z_i &\leq (1 - x_i), \qquad i \in I_2 \\ z_i &\leq t_i, \qquad i \in I_2 \\ z_i &\geq -x_i + t_i, \qquad i \in I_2 \end{aligned} \tag{3.9}$$

Finally, we can rewrite Equation (3.7) as follows:

$$\begin{aligned} t_i &\geq x_j, \qquad j \in Succ(i), \qquad i \in I_2 \\ t_i &\leq \sum_{j \in Succ(i)} x_j, \qquad i \in I_2 \end{aligned} \tag{3.10}$$

We introduce $|V_b^{in}|+2|V_b|$ new binary decision variables in Equations (3.5), (3.6), and (3.7). Additionally, we use $|E_b^{in}|+|V_b^{in}|$ linear constraints to represent Equation (3.5), $3|V_b|$ linear constraints to represent Equation (3.6), and $|E_b|+|V_b|$ linear constraints to represent Equation (3.7). Some of the variables and some of the constraints may be redundant and can be eliminated by a preprocessing step. In total, we need $\mathrm{O}(|V_b|+|V_b^{in}|)$ binary variables and $\mathrm{O}(|E_b|+|V_b|+|E_b^{in}|+|V_b^{in}|)$ linear constraints to convert Equation (3.1) into linear form.

3.2.2. Calculation of the Output Data Transfers

We use the integer decision variable $INST_{\text{out}}$ to represent the number of output operands of a template T. A node $v_i \in V_b^{out}$, defining an output variable of the basic block, defines an output operand of T if it is in V_t. A node $v_i \in V_b/V_b^{out}$ defines an output operand of T if it is in V_t, and it has at least one immediate successor not in V_t. The total number of output operands for T can be computed as follows:

$$INST_{\text{out}} = \sum_{i \in I_3} x_i + \sum_{i \in I_4} \left(x_i \wedge \left(\bigvee_{j \in Succ(i)} x_j' \right) \right) \tag{3.11}$$

We calculate the number of additional data transfers from the custom logic to the core register file as DT_{out}:

$$DT_{\text{out}} \geq INST_{\text{out}}/RF_{\text{out}} - 1, \qquad DT_{\text{out}} \in Z^+ \cup \{0\} \tag{3.12}$$

A constraint of MAX_{out} on the maximum number of output operands can be imposed as follows:

$$INST_{\text{out}} \leq MAX_{\text{out}} \tag{3.13}$$

In Figure 3.3, the node $v_6 \in V_b^{out}$ defines an output operand of T as it is in V_t (i.e., $x_6 = 1$). Among the basic block nodes that are not in V_b^{out}, v_4 is in V_t and has its immediate successor v_5 not in V_t (i.e., $x_4 \wedge x_5'$). As a consequence, v_4 defines an output operand of T. On the other hand, v_3 is also in V_t and has no immediate successor that is not in V_t. Therefore, v_3 does not define an output operand of T. The total number of output operands of T is calculated as two.

We rewrite Equation (3.11) as follows in order to linearize logical constraints:

$$INST_{\text{out}} = \sum_{i \in I_3} x_i + \sum_{i \in I_4} p_i \tag{3.14}$$

where, we define the auxiliary binary decision variables p_i, and q_i as follows:

$$p_i = x_i \wedge q_i, \qquad i \in I_2 \tag{3.15}$$

$$q_i = \bigvee_{j \in Succ(i)} x_j', \qquad i \in I_2 \tag{3.16}$$

Equation (3.15) can be written as follows:

$$\begin{aligned} p_i &\leq x_i, && i \in I_2 \\ p_i &\leq q_i, && i \in I_2 \\ p_i &\geq x_i + q_i - 1, && i \in I_2 \end{aligned} \tag{3.17}$$

We can then rewrite Equation (3.16) as follows:

$$\begin{aligned} q_i &\geq (1 - x_j), \qquad j \in Succ(i), && i \in I_2 \\ q_i &\leq \sum_{j \in Succ(i)} (1 - x_j), && i \in I_2 \end{aligned} \tag{3.18}$$

We introduce $2|V_b|$ new binary decision variables in Equations (3.15) and (3.16).

We use $3|V_b|$ linear constraints to represent Equation (3.15) and $|E_b| + |V_b|$ linear constraints to represent Equation (3.16). In total, we need $O(|V_b|)$ binary variables and $O(|E_b| + |V_b|)$ linear constraints to convert Equation (3.11) into linear form.

3.2.3. Convexity Constraint

We make use of the following theorem when dealing with the convexity constraint:

Theorem 1. *A template T is convex if and only if there is no node in V_b/V_t having both an ancestor and a descendant in V_t.*

Proof. Assume that T is convex. Suppose that there is a node v in V_b/V_t having an ancestor u in V_t and a descendant w in V_t. Then, there exists a path from u to w going through v, contradicting with the definition of convexity.

Conversely, assume that there is no node in V_b/V_t having both an ancestor and a descendant in V_t. Suppose that T is not convex. Then, there exists a path in G from a u in V_t to a w in V_t that goes through a v in V_b/V_t . In this case, v would have both an ancestor (i.e., u) and a descendant (i.e., w) in V_t, contradicting with the initial assumption. □

For each node $v_i \in V_b$ we introduce two new binary decision variables a_i and d_i. a_i represents whether v_i has an ancestor in the template T ($a_i = 1$) or not ($a_i = 0$). Similarly, d_i represents whether v_i has a descendant in T ($d_i = 1$) or not ($d_i = 0$).

A node has an ancestor in V_t if it has at least one immediate predecessor that is already in V_b or that has an ancestor in V_t.

$$a_i = \begin{cases} 0 & \text{if } Pred(i) = \emptyset \\ \left(\bigvee_{j \in Pred(i)} (x_j \vee a_j) \right) & \text{otherwise} \end{cases}, \qquad i \in I_2 \tag{3.19}$$

A node has a descendant in V_t if it has at least one immediate successor that is already in V_t or that has a descendant in V_t.

$$d_i = \begin{cases} 0 & \text{if } Succ(i) = \emptyset \\ \left(\bigvee_{j \in Succ(i)} (x_j \vee d_j) \right) & \text{otherwise} \end{cases}, \qquad i \in I_2 \tag{3.20}$$

Based on Theorem 1, there should be no node in V_b/V_t having both an ancestor and a descendant in V_t to ensure convexity:

$$x_i' \wedge a_i \wedge d_i = 0, \qquad i \in I_2 \tag{3.21}$$

We rewrite Equation (3.19) as follows in order to linearize the logical constraints:

$$\begin{aligned} a_i &\geq x_j, \qquad j \in Pred(i), \qquad i \in I_2 \\ a_i &\geq a_j, \qquad j \in Pred(i), \qquad i \in I_2 \\ a_i &\leq \sum_{j \in Pred(i)} x_j + \sum_{j \in Pred(i)} a_j, \qquad i \in I_2 \end{aligned} \tag{3.22}$$

Similarly, we rewrite Equation (3.20) as follows:

$$\begin{aligned} d_i &\geq x_j, \qquad j \in Succ(i), \qquad i \in I_2 \\ d_i &\geq d_j, \qquad j \in Succ(i), \qquad i \in I_2 \\ d_i &\leq \sum_{j \in Succ(i)} x_j + \sum_{j \in Succ(i)} a_j, \qquad i \in I_2 \end{aligned} \tag{3.23}$$

Finally, Equation (3.21) is written as follows:

$$a_i + d_i - x_i \leq 2 \tag{3.24}$$

We use $|V_b| + 2|E_b|$ linear constraints to represent Equations 3.19) and (3.20). In total, we need $O(|E_b|+|V_b|)$ additional constraints to linearize the convexity constraint.

3.2.4. Critical Path Calculation

We estimate the execution latency of a template T on the custom logic by quantizing its critical path length. We calculate the critical path length by applying an ASAP (as soon as possible) scheduling without resource constraints.

We associate with every DAG node v_i a real decision variable $l_i \in R$, which represents the time in which the result of v_i becomes available when T is executed on custom logic, assuming that all of its input operands are available at time zero.

$$l_i \geq \begin{cases} h_i x_i & \text{if } Pred(i) = \emptyset \\ l_j + h_i x_i, \qquad j \in Pred(i) & \text{otherwise} \end{cases}, \qquad i \in I_2 \tag{3.25}$$

The largest l_i value gives us the critical path length of T. We use $H(T) \in Z^+$ to represent the quantized critical path length.

$$H(T) \geq l_i, \qquad i \in I_2 \tag{3.26}$$

3.2.5. Objective

We estimate the software cost $S(T)$ of a template T as the sum of the software latencies of the nodes included in V_t.

$$S(T) = \sum_{i \in I_2} (s_i x_i) \tag{3.27}$$

$H(T)$ provides the estimated execution latency of T on the custom logic once all of its inputs are ready. $C(T)$ represents the number of cycles required to transfer the input and output operands of T from and to the core register file ($c_1 DT_{\text{in}}$ and $c_2 DT_{\text{out}}$ respectively) when T is executed on the custom logic.

$$C(T) = c_1 DT_{\text{in}} + c_2 DT_{\text{out}} \tag{3.28}$$

Our objective is to maximize the reduction in the schedule length by moving T from software to the custom logic.

$$Z(T) = \max S(T) - H(T) - C(T) \tag{3.29}$$

3.2.6. Scalability of the Model

Our ILP model scales linearly with the size of the problem. The number of decision variables and the number of linear constraints required to represent the overall problem is found by accumulating the corresponding numbers from Section 3.2.1, Section 3.2.2, Section 3.2.3, Section 3.2.4, and Section 3.2.5. We observe that the overall problem can be represented using $\mathrm{O}(|V_b \cup V_b^{in}|)$ binary decision variables, $\mathrm{O}(|V_b|)$ real decision variables, and a few additional integer decison variables. In addition, we make use of $\mathrm{O}(|E_b| + |V_b| + |E_b^{in}| + |V_b^{in}|)$ linear constraints. If the compiler IR consists of three address statements only (i.e., every statement has two input operands and a single output operand), we know that $|E_b| + |E_b^{in}| = 2|V_b|$ and $|V_b^{in}| \leq 2|V_b|$. In this case, both the number of decision variables and the number of linear constraints are $\mathrm{O}(|V_b|)$.

3.2.7. Support for Statements with Multiple Destination Operands

Some IRs are not composed entirely of three address statements and involve assignment statements with multiple destination operands. We handle such statements by adapting our ILP formulation. In the new representation, every destination operand

of every statement defines its own set-of data dependencies. Assuming that node v_i defines K_i destination operands, we associate with every destination operand of every node $k \in \{1 \dots K_i\}, i \in I_2$, a set of successor nodes $Succ(i,k)$.

We rewrite Equation (3.1) as follows:

$$INST_{\text{in}} = \sum_{i \in I_1} \left(\bigvee_{j \in Succ(i)} x_j \right) + \sum_{i \in I_2} \sum_{k=1}^{K_i} \left(x'_i \wedge \left(\bigvee_{j \in Succ(i,k)} x_j \right) \right) \tag{3.30}$$

We associate with every DAG node v_i a set $K_i^S \subseteq K_i$, which represents a subset of the destination operands of node v_i that are among the output operands of the basic block. We rewrite Equation (3.11) as follows:

$$INST_{\text{out}} = \sum_{i \in I_2} \sum_{k \in K_i^S} x_i + \sum_{i \in I_2} \sum_{k \notin K_i^S} \left(x_i \wedge \left(\bigvee_{j \in Succ(i,k)} x'_j \right) \right) \tag{3.31}$$

The unified set of successors of the DAG nodes are defined as follows:

$$Succ(i) = \bigcup_{k=1}^{K_i} Succ(i,k), \qquad i \in I_2 \tag{3.32}$$

Given $j \in Succ\{i\} \Leftrightarrow \ i \in Pred\{j\}$ for $i, j \in I_2$, formulations of the convexity constraint (Section 3.2.3), critical path calculation (Section 3.2.4), and the objective (Section 3.2.5) remain unchanged.

3.3. Template Generation

Our template generation algorithm iteratively solves a set of ILP problems in order to generate a set of custom instruction templates. For a given application basic block, the first template is identified by solving the ILP problem on the DAG repre-

```
1: ALGORITHM: Template Generation
2: Given G : the set of application basic blocks
3: Generate T : the set of custom instruction templates
4: G : the current basic block
5: T : the current custom instruction template
6: Objective : the current objective value
7: Upper_Bound : the current upper bound
8: for G in G do
9:    Objective ← MAX_INT
10:   Upper_Bound ← MAX_INT
11:   while Objective > 0 do
12:      Generate the relaxed ILP problem for G
13:      Add the constraint (Objective ≤ Upper_Bound)
14:      Solve ILP, extract Objective and T
15:      Upper_Bound ← MIN(Objective, Upper_Bound)
16:      if T is NOT convex then
17:         Add the convexity constraint
18:         Add the constraint (Objective ≤ Upper_Bound)
19:         Solve ILP, extract Objective and T
20:         Upper_Bound ← MIN(Objective, Upper_Bound)
21:      end if
22:      if Objective > 0 then
23:         T ← T ∪ {T}
24:         Collapse T into a single node in G
25:         Mark the new node as an invalid node
26:      end if
27:   end while
28: end for
```

Figure 3.4. We iteratively solve a set of ILP problems. A good upper bound on the objective value can significantly reduce the solution time.

senting the basic block as defined in Section 3.2. After the identification of the first template, the nodes included in the template are collapsed into a single node in the DAG, and the data structures representing the DAG are updated. The convexity constraint guarantees that the graph remains acyclic after node collapsing. We mark the new node as an invalid node and iterate the same procedure on the updated DAG. Marking of the new node as invalid ensures that the node will not be identified as part of any other templates. The overall process guarantees that there will be no overlap between the generated templates. We apply the same procedure until no more profitable templates can be found in the DAG. We note here that the new nodes inserted into the DAG can have multiple destination operands, in which case we have to use the formulation described in Section 3.2.7. Finally, we apply the same procedure on all application basic blocks and generate a unified set of custom instruction templates.

Providing a good upper bound on the value of the objective function can greatly enhance the performance of the ILP solver without affecting the optimality of the solution. ILP solvers rely on well-known optimization techniques such as branch-and-bound and branch-and-cut for exploring the search space efficiently. These techniques build a search tree, where the nodes represent subproblems of the original problem. Given a good upper bound on the value of the objective function, the number of branches and the size of the search tree can be significantly reduced. In our experiments, we have observed that the relaxation of the convexity constraint simplifies the problem, and allows us to obtain good upper bounds on the objective value of the unrelaxed problem within short time. In addition, the objective value of the previous iteration provides a second and sometimes a tighter upper bound.

A formal description of our approach is given in Figure 3.4. We first solve the relaxed problem, where the convexity constraint is not imposed on the templates. If the identified template is convex, we add it to our template pool. Otherwise, the solution identified provides an upper bound on the objective value of the unrelaxed problem. We solve the problem once more with the convexity constraint imposed using the improved upper bound. Initially, the upper bound is set to the value of the maximum integer (MAX_INT). As the iterations proceed, the DAG gets smaller, the upper bound gets

tighter, and these factors usually decrease the solution time.

The objective of the iterative template generation algorithm is to generate custom instruction templates covering application DAGs as much as possible while avoiding the exponential computational complexity of the subgraph enumeration techniques. Not allowing overlapping between templates guarantees that the number of iterations will be $O(N_{tot})$, where N_{tot} represents the total number of instructions in an application. In practice, the number of iterations is much smaller than N_{tot} as the templates we generate are often coarse grain.

At each iteration, we choose the template that provides the highest objective value (*i.e., the most profitable subgraph*). Although our approach is heuristic, recent research shows that our approach results in good overall code coverage. Clark et al. [85], combine the subgraph enumeration algorithm of [22, 23, 24] with a unate covering based code selection approach. The improvement in the speed-up over the iterative approach is reported as one per cent only. On the other hand, the same work reports that the use of a locally optimal solution at each iteration, such as in [22, 23, 24], provides significant gains over the use of heuristic clustering based techniques, such as in [61].

3.4. Template Selection

Once the template generation is done, we calculate the isomorphism classes. We apply pairwise isomorphism checks using the Nauty package [86]. The nauty package makes use of a backtracking algorithm that produces automorphism groups of a given graph, as well as the canonically labeled isomorph. A set of templates are isomorphic if and only if their canonically labeled isomorphs are equivalent.

We assume that the set of templates $\mathcal{T}$ generated by the algorithm of Section 3.3 is partitioned into N_G distinct isomorphism classes:

$$\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup ... \cup \mathcal{T}_{N_G} \tag{3.33}$$

An isomorphism class defines a custom instruction candidate that can implement all the templates included in that class. Once isomorphism classes are formed, we generate behavioral hardware descriptions of the custom instruction candidates in VHDL. We apply high level synthesis, and we associate an area estimate $A(\mathcal{T}_i)$, and a normalized critical path estimate $D(\mathcal{T}_i)$ with each custom instruction candidate $\mathcal{T}_i$.

The value of the objective function $Z(T)$ described in Section 3.2.5 provides an initial estimation of the reduction in the schedule length by a single execution of the template $T \in \mathcal{T}_i$ on the custom logic. We replace the estimated critical path length $H(T)$ with the more accurate result $D(\mathcal{T}_i)$ we obtain from high level synthesis in order to generate a refined estimation of the reduction in the schedule length for the custom instruction candidate $\mathcal{T}_i$:

$$Z(\mathcal{T}_i) = Z(T) + H(T) - D(\mathcal{T}_i) \tag{3.34}$$

Given that a template $T \in \mathcal{T}_i$ is executed $F(T)$ times by a typical execution of the application, the total number of executions of a custom instruction candidate $\mathcal{T}_i$ is calculated as follows:

$$F(\mathcal{T}_i) = \sum_{T \in \mathcal{T}_i} F(T), \qquad i \in \{1..N_G\} \tag{3.35}$$

The overall reduction in the schedule length of the application by implementing $\mathcal{T}_i$ as a custom instruction is estimated as follows:

$$G(\mathcal{T}_i) = Z(\mathcal{T}_i) * F(\mathcal{T}_i) \tag{3.36}$$

We formulate the selection of the most profitable custom instruction candidates under an area constraint A_{MAX} as a Knapsack problem, and solve it using the ILP solver. We associate a binary decision variable y_i with each custom instruction candi-

date $\mathcal{T}_i$, which represents whether $\mathcal{T}_i$ is selected ($y_i = 1$) or not ($y_i = 0$).

The Knapsack formulation is constructed as follows:

$$\begin{aligned} max & \sum_{i \in \{1..N_G\}} G(\mathcal{T}_i) y_i \\ s.t. & \sum_{i \in \{1..N_G\}} A(\mathcal{T}_i) y_i \leq A_{MAX} \\ & y_i \in \{0,1\}, \qquad i \in \{1..N_G\} \end{aligned} \tag{3.37}$$

Naturally, the number of custom instruction candidates is smaller than the number of custom instruction templates generated. Hence, the number of binary decision variables used in Equation (3.37) is $O(N_{tot})$, and in practice much smaller than N_{tot}. This allows us to solve the Knapsack problem optimally using ILP solvers for all practical examples.

We note that, the $F(\mathcal{T}_i)$ values we use in our formulations might not accurately represent the potential of a custom instruction candidate since there might be other instances of the candidate in the application DAGs not identified by our algorithms. A truly optimal solution could be possible by enumerating all possible subgraphs within the application DAGs. However, this approach is not computationally feasible since: (1) the number of subgraphs grows exponentially with the size of the DAGs; (2) allowing overlapping between selected subgraphs significantly complicates the mathematical model for template selection. In [87], Yu and Mitra enumerate only connected subgraphs having up to four input and two output operands, and do not allow overlapping between selected subgraphs. Although the search space is significantly reduced by these restrictions, Yu and Mitra report that optimal selection using ILP solvers occasionally fails to complete within 24 hours and propose a heuristic solution for selection.

3.5. Machine Description and Code Generation

The Trimaran [26] infrastructure provides a baseline machine that implements the HPL-PD architecture [88] based on a high level machine description model (MDES [84]). MDES requires specifications of the operation formats, resource usages, scheduling alternatives, execution latencies, operand read and write latencies, and reservation table entries for the instructions supported by the architecture. We automatically generate the MDES entries for the custom instruction candidates selected by our algorithms. We implement custom instruction replacement using a technique similar to the one described by Clark et. al. in [61]. Finally, we apply standard Trimaran scheduling and register allocation passes, and we produce the assembly code and scheduling statistics.

3.6. Summary

In this chapter, we describe an ILP model that identifies the most profitable custom instructions given the available data bandwidth and transfer latencies between the base processor and the custom logic. Our ILP model can optionally constrain the number of input and output operands as well. We show that the number of decision variables and the number of linear constraints used in our ILP formulation grows only linearly with the size of the problem, making our solution highly scalable. We integrate our ILP based solution into an iterative template generation algorithm, which aims to maximize the amount of code covered by the custom instruction templates. Next, we group structurally equivalent custom instruction templates within isomorphism classes as custom instruction candidates. We generate behavioral descriptions of the custom instruction candidates, and we produce area and delay estimates. We select the candidates that reduce the schedule length of the application most under area constraints based on a Knapsack model. Finally, we generate a high level machine description supporting custom instructions, the assembly code and the scheduling statistics.

4. EXPERIMENTS AND RESULTS

In this chapter, we first describe our experiment setup (Section 4.1). We provide examples of custom instructions automatically found by our algorithms (Section 4.3). Based on the compiler feedback, we evaluate the impact of input/output constraints (Section 4.4), register file port constraints (Section 4.5), and code transformations such as if-conversion (Section 4.2) and loop unrolling (Section 4.6) on the performance and code size for a range of area constraints on eleven multimedia and cryptography benchmarks (Section 4.9). In Section 4.8, we show the run-time results of our algorithms. In Section 4.10 we integrate our compilation flow into an academic custom processor synthesis infrastructure [27], and we provide custom ASIC processor synthesis results.

4.1. Experiment Setup

We evaluate our technique using Trimaran [26] scheduling statistics to estimate the execution cycles, and Synopsys [83] synthesis to estimate the area and delay for custom instructions. In the following sections, we provide information on the base processor configuration, synthesis tools, benchmarks, and the run-time environment.

4.1.1. Base Processor Configuration

We define a single-issue baseline machine with predication support including 32 32-bit general purpose registers and 32 one-bit predicate registers. We assume two-cycle software latencies for integer multiplication instructions, and single-cycle software latencies for the rest of the integer operations. We do not allow division operations to be included in custom instructions due to their high area overhead. We exclude support for memory access instructions as part of custom instructions as well, in order to avoid non-deterministic latencies due to the memory system and the necessary control circuitry. We assume single-cycle data transfer latencies between general purpose registers and custom units ($c_1 = c_2 = 1$). Finally, we assume single-cycle copy and update operations for transferring the predicate register file contents to and from the custom logic.

Table 4.1. Relative latency and area coefficients for various operators based on synthesis results on UMC's 130nm process.

Operator	Latency	Area
32-bit + 32-bit adder	1.000	1.000
32-bit * 32-bit multiplier	1.524	18.463
32-bit and	0.010	0.236
32-bit xor	0.029	0.415
32-bit shifter	0.295	1.977
32-bit shifter (constant)	0.000	0.000
32-bit comparator (eq)	0.095	0.512
32-bit comparator (geq)	0.552	0.632

4.1.2. Synopsys Synthesis

We calculate the hardware latencies of various arithmetic and logic operations (i.e., h_i values described in Section 3.1.) by synthesizing on UMC's 130nm standard cell library using Synopsys Design Compiler [83], and normalizing to the delay of a 32-bit ripple carry adder (RCA). Table 4.1 shows the relative latency and area coefficients for some selected operators. Once our algorithms identify the custom instruction candidates, we automatically generate their VHDL descriptions, and synthesize on the same library. If the critical path delay of a candidate is larger than the delay of a 32-bit RCA, it is pipelined to ensure a fixed clock frequency.

4.1.3. Benchmarks

We apply our algorithms on a number of cryptography and media benchmarks. We use highly optimized 32-bit implementations of Advanced Encryption Standard (AES) encryption and decryption described in [89], a FIPS-46-3 compliant fully unrolled Data Encryption Standard (DES) implementation [90], a loop based and a fully unrolled Secure Hash Algorithm (SHA) implementation from MiBench [91], and several other benchmarks from MediaBench [92].

Table 4.2. Information on benchmarks: BB represents basic block.

Benchmark	# of BBs	# of Instrs	Largest BB
AES encryption	27	735	317
AES decryption	28	1011	501
DES	45	1235	822
SHA (loop)	38	302	24
SHA (fully unrolled)	30	1339	1155
IDEA	65	595	96
djpeg	957	5503	92
g721encode	85	892	131
g721decode	79	864	131
mpeg2enc	147	2861	568
rawcaudio	13	119	54
rawdaudio	11	102	45

4.1.4. Run-time Environment

We carry out our experiments on an Intel Pentium IV, 3.2 GHz workstation with one gigabyte memory running Linux. Our algorithms are implemented in C/C++ and compiled with gcc-3.4.3 using -O2 optimization flag.

4.2. If-conversion Results

We implement an if-conversion pass to selectively eliminate the control flow dependencies. This improves the scope of our algorithms, and enables us to identify coarser grain custom instructions. We apply if-conversion only on the most time consuming functions of the application. We partition control flow graphs into maximal single entry, single exit regions, and convert each region into a predicated basic block.

The results of our if-conversion pass on five MediaBench benchmarks are shown in Figure 4.1. We observe that the number of execution cycles and the number of

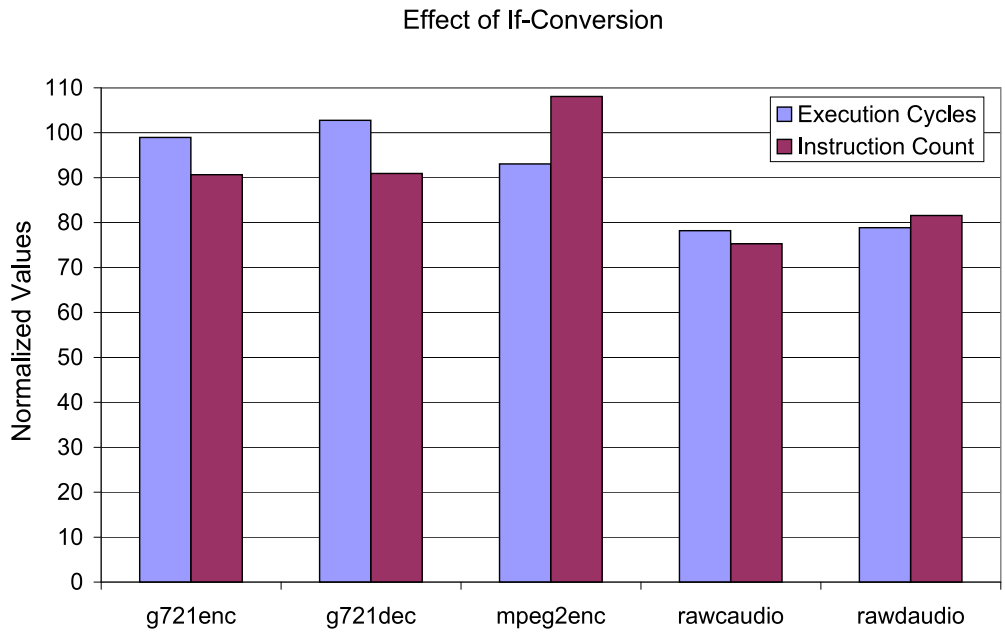


Figure 4.1. We apply an if-conversion pass before identifying custom instructions. This reduces the number of execution cycles and the code size in most of the cases.

instructions in the code are reduced in most of the cases, although this is not our main objective. The remaining benchmarks are only marginally affected, particularly because they already consist of large basic blocks and contain few control flow changes.

Table 4.2 shows the total number of basic blocks, the total number of instructions, and the number of instructions within the largest basic block for each benchmark. This information is collected after the application of if-conversion and considers basic blocks with positive execution frequencies only.

4.3. Examples of Custom Instructions

4.3.1. AES Encryption

Our first example is the Advanced Encryption Standard (AES). The core of the AES encryption is the round transformation (see Figure 4.2), which operates on a 16-byte state. The state can be considered as a two dimensional array of bytes having four rows and four columns. The columns are often stored in four 32-bit registers, and are

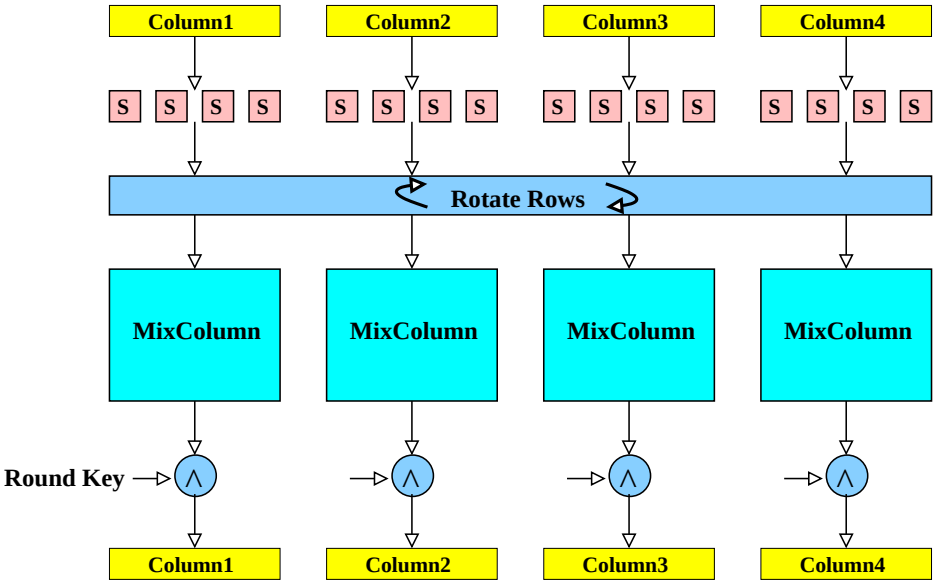


Figure 4.2. The AES round transformation. Given an input constraint of four and an output constraint of four, our algorithms successfully identify the four parallel MixColumn Transformations as the most promising custom instruction candidate. None of the algorithms described in [24] are able to identify this solution.

inputs and outputs of the round transformation. First, a nonlinear byte substitution is applied on each of the state bytes by making table lookups from S-Boxes stored in the memory. Next, the rows of the state array are rotated over different offsets. After that, a linear transformation called MixColumn transformation is applied on each column. The final operation of the round transformation is an EXOR with the round key. The output of a round transformation, becomes the input of the next round transformation. Very often, several round transformations are unrolled within a loop, resulting in very large basic blocks consisting of several hundreds of operations.

The most compute-intensive part of AES encryption is the MixColumn transformation. The MixColumn transformation is a single input, single output transformation consisting of around 20 bitwise operations with frequent constant coefficients. In fact, the MixColumn transformation is the most likely choice for a manual designer as a custom instruction as shown by Seng et al. in [42]. The data flow between the operations implementing MixColumn transformation is depicted in Figure 4.3.

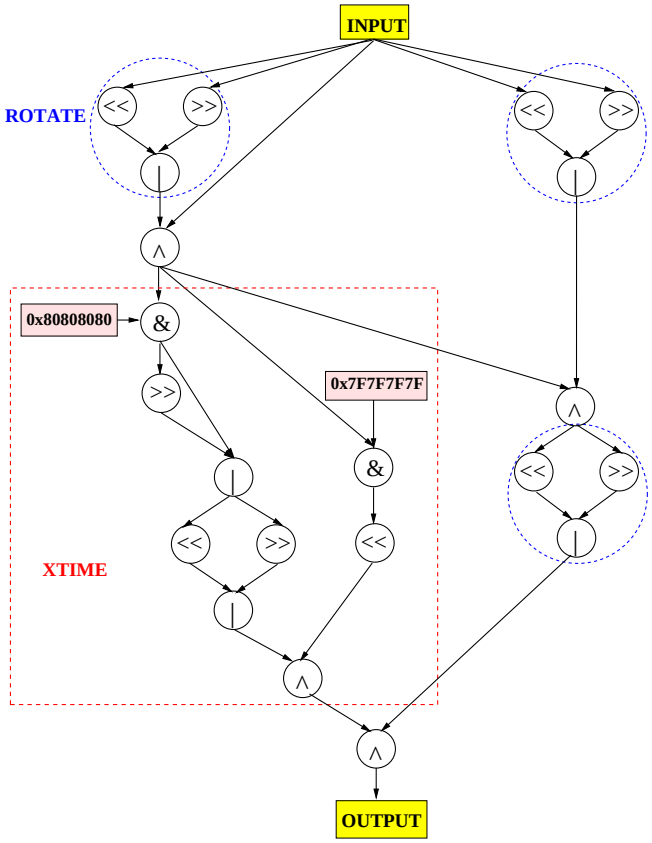


Figure 4.3. A 32-bit implementation of the MixColumn transformation [89]

We used a 32-bit implementation of AES optimized for memory constrained embedded platforms described in [89]. As shown in Figure 4.4, two round transformations are unrolled within a loop, resulting in the largest basic block of the application consisting of 317 operations. A second basic block consists of a single round transformation followed by the final round transformation, which does not incorporate the MixColumn transformations. The code also includes an initialization stage, where the encryption state is read from main memory and reorganized for fast processing, and a finalization stage, where the encryption state is written back to main memory in its original format.

Given an input constraint of one, and an output constraint of one, our algorithms successfully identify all 12 instances of the MixColumn transformation in the code as the most promising custom instruction for the application. Given an input constraint of two and an output constraint of two, our algorithms successfully identify two parallel MixColumn transformations within a round transformation as the most promising custom instruction, finding all six instances in the code. Given an input constraint of four

```
FUNCTION ENCRYPTBLOCK
BEGIN
    Initialize(State)
    FOR i in {1..num_rounds/2}
    BEGIN
        Round(State, RoundKey++) ;
        Round(State, RoundKey++) ;
    END
    Round(State, RoundKey++) ;
    FinalRound(State, RoundKey) ;
    Finalize(State)
END
```

Figure 4.4. An optimized AES encryption implementation [89]

and an output constraint of four, our algorithms successfully identify the four parallel MixColumn transformations within a round transformation, and all three instances of the four-input four-output custom instruction are matched in the code. In all cases, our algorithms identify optimal solutions within a few seconds. On the other hand, the subgraph enumeration algorithm of [24] fails to complete output constraint given an output constraint of four, and none of the approximate algorithms described in [24, 25] are able to identify four MixColumn transformations in parallel.

The synthesis results show that the critical path delay of the MixColumn transformation is around one-fourth of the critical path delay of a 32-bit ripple carry adder, and its area cost is less than the area cost of two 32-bit ripple carry adders. Therefore, the complete transformation can be implemented as a single cycle instruction. Given a register file with two read ports and two write ports, we could as well implement two parallel MixColumn transformations as a single cycle instruction. Obviously, this solution would incur two times more area overhead. Given a register file with four read ports and four write ports, we could even implement four parallel MixColumn transformations as a single cycle instruction depending on our area budget.

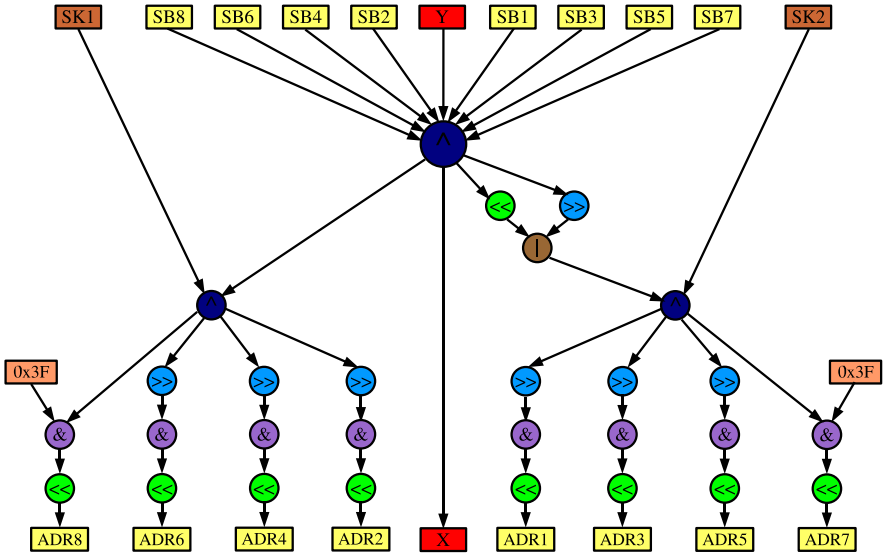


Figure 4.5. Optimal custom instruction implementing the DES rounds. Eight of the inputs (SBs) are substitution table entries, and eight of the outputs are indices of the substitution table entries that should be fetched for the next round. SK1 and SK2 contain the round key. The input Y represents the second half of the current state, and the first half of the state for the next round is generated in X. 15 instances of the same instruction are automatically identified from the reference C code.

4.3.2. DES Encryption

In Figure 4.5 we show the most promising custom instruction our algorithms automatically identify from the DES C code when no constraints are imposed on the number of input and output operands. An analysis reveals that the custom instruction implements the complete data processing within the round transformations of DES. It has eleven inputs and nine outputs, 15 instances of it are automatically matched in the C code. To our knowledge, no other automated technique has been able to achieve a similar result. Subgraph enumeration algorithms, such as [22, 23, 24, 66] are impracticable when the input/output constraints are removed or loose, and cannot identify custom instructions such as the one in Figure 4.5.

Figure 4.6 shows the actual implementation of the DES encryption. DES operates on a 64-bit state stored in two 32-bit variables (X, Y). After an initial permutation of the bytes, DES round transformation is applied 16 times on the state, followed by a

```
DES_ENCRYPT( SK, X, Y )
BEGIN
  DES_IP( X, Y );
  DES_ROUND( SK, Y, X ); DES_ROUND( SK, X, Y );
  ⋮
  DES_ROUND( SK, Y, X ); DES_ROUND( SK, X, Y );
  DES_FP( Y, X );
END
```

Figure 4.6. A fully unrolled DES encryption implementation.

final byte permutation. Figure 4.7 demonstrates a 32-bit implementation of the DES round transformation. Every DES round consumes a 64-bit round key stored in a 32-bit array (SK). Every round makes eight table look-ups to the byte substitution tables (SBs) stored in the main memory. The DES round transformation is the most time consuming part of the DES encryption, and would be the most likely choice for a manual designer as a custom instruction.

An analysis shows that eight of the inputs of the custom instruction of Figure 4.5 are static look-up table entries (SBs), and eight of the outputs (ADRs) contain indices of the look-up table entries that should be fetched for the next round. Two of the inputs (SK1, SK2) contain the DES round key, the input Y and the output X represents the DES encryption state. The custom instruction implements 35 base processor instructions, which are mostly bitwise operations. The synthesis results show that the critical path of the custom instruction is around one-eight of the critical path of a 32-bit ripple carry adder. Hence, the custom instruction can be executed within a single cycle. However, as the register file has a limited number of read and write ports, we need additional data transfer cycles to transfer the input and output operands between the core register file and the custom units. In practice, the granularity of the custom instruction is coarse enough to make it profitable despite the data transfer overhead, and this overhead is explicitly calculated by our algorithms.

```
DES_ROUND( SK, X, Y )
BEGIN
   T = *SK++ ^ X;
   Y ^ = SB8[ (T) & 0x3F ] ^
         SB6[ (T >> 8) & 0x3F ] ^
         SB4[ (T >> 16) & 0x3F ] ^
         SB2[ (T >> 24) & 0x3F ];
   T = *SK++ ^ ((X << 28) | (X >> 4));
   Y ^ = SB7[ (T) & 0x3F ] ^
         SB5[ (T >> 8) & 0x3F ] ^
         SB3[ (T >> 16) & 0x3F ] ^
         SB1[ (T >> 24) & 0x3F ];
END
```

Figure 4.7. 32-bit implementation of a DES round in C.

4.4. Effect of Input and Output Constraints

In this work, we use input/output constraints to control the granularity of the custom instructions, and to locate structural similarities within an application. Our motivation is that applications often contain repeated code segments that can be characterized by the number of input and output operands. When the input/output constraints are tight, we are more likely to identify fine grain custom instructions. As we demonstrate in Section 4.7, fine grain custom instructions often have more reuse potential. Relaxation of the input/output constraints results in coarser grain custom instructions (i.e., larger dataflow subgraphs). Coarse grain instructions are likely to provide higher speed-up, although at the expense of increased custom logic area.

In Figures 4.8 and 4.9 we analyze the effect of different input and output constraints (i.e., MAX_{in}, MAX_{out}) on the speed-up potentials of custom instructions. For each benchmark we scale the initial cycle count down to 100, and we plot the per cent decrease in the cycle count by introducing custom instructions for a range of area

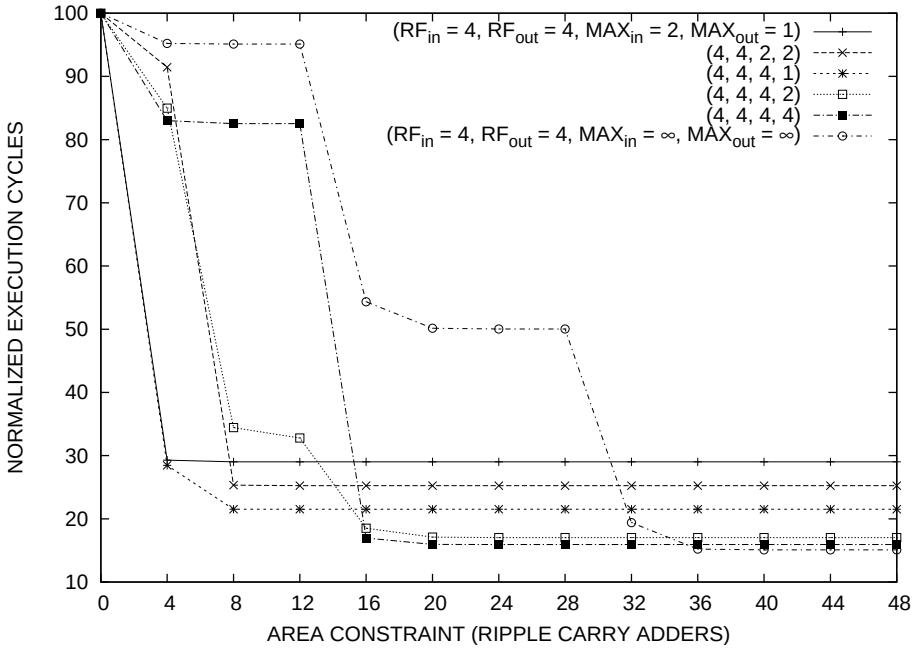


Figure 4.8. AES decryption: per cent reduction in the execution cycles. Register file supports four read ports and four write ports (i.e., $RF_{in} = 4$, $RF_{out} = 4$). An input constraint of MAX_{in} and an output constraint of MAX_{out} can be imposed on custom instructions, or these constraints can be removed (i.e., $MAX_{in} = \infty$, $MAX_{out} = \infty$).

constraints (up to 48 ripple carry adders). At the end of this analysis, we locate the pareto optimal points (i.e., input/output combinations) that maximize the cycle count reduction at each area constraint.

In Figure 4.8, we assume a register file with four read ports and four write ports, and we explore the achievable speed-up for AES decryption. The main difference between AES decryption and AES encryption is the InvMixColumn transformations that replace MixColumn transformations in the round transformation. The area cost of the InvMixColumn transformation is around the area cost of four ripple carry adders. Figure 4.8 shows that at an area constraint of four adders, the pareto optimal solution is obtained using four-input single-output custom instructions. On the other hand, at an area constraint of 16 adders, four-input four-output custom instructions provide the pareto optimal solution. This solution implements four InvMixColumn transformations in parallel as a single cycle instruction. We observe that removing the input/output constraints improves the performance slightly until an area constraint of 40 adders.

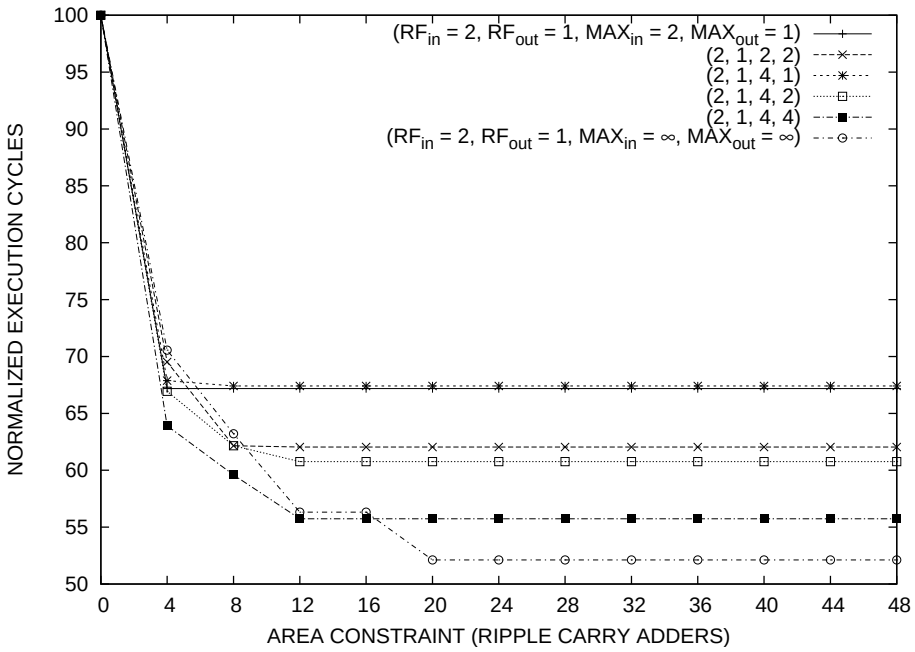


Figure 4.9. DES: per cent reduction in the execution cycles. Register file supports two read ports and one write port (i.e., $RF_{in} = 2$, $RF_{out} = 1$). An input constraint of MAX_{in} and an output constraint of MAX_{out} can be imposed on custom instructions, or these constraints can be removed (i.e., $MAX_{in} = \infty$, $MAX_{out} = \infty$).

In Figure 4.9, we assume a register file with two read ports and single write port, and we explore the achievable speed-up for DES. We observe that when the area budget is below 16 adders, pareto optimal solutions are generated by four-input four-output custom instructions. However, we obtain the highest reduction in the execution cycles when the input/output constraints are removed, at an area cost of 20 adders.

In Figure 4.10, we assume a register file with two read ports and a single write port (i.e., $RF_{in} = 2$, $RF_{out} = 1$), and we show the improvement in speed-up with the relaxation of input/output constraints. The first approach limits the number of inputs and outputs to the available register file ports (i.e., $MAX_{in} = 2$, $MAX_{out} = 1$). The second approach removes the input/output constraints completely (i.e., $MAX_{in} = \infty$, $MAX_{out} = \infty$), improving the speed-up from 1.14 times to 1.28 times for fully unrolled SHA, from 1.49 times to 1.92 times for DES, from 3.45 times to 4.36 times for AES decryption and from 2.55 times to 2.82 times for AES encryption.

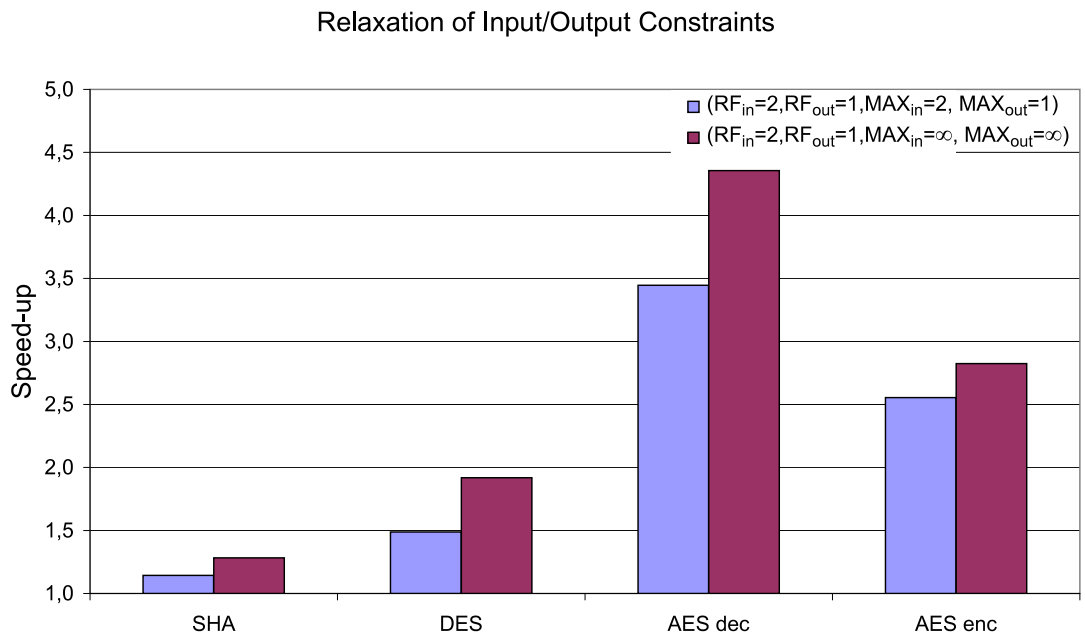


Figure 4.10. Register file supports two read ports and one write port (i.e., $RF_{in} = 2$, $RF_{out} = 1$). Speed-up (with respect to the base processor) improves with the relaxation of input/output constraints.

4.5. Effect of Register File Ports

In Figures 4.11 and 4.12 we demonstrate the improvement in performance using additional register file ports. We scale the initial cycle count down to 100, and we plot the per cent reduction in the execution cycles for a range of area constraints. For each (RF_{in}, RF_{out}) combination, we explore six different (MAX_{in}, MAX_{out}) combinations: (2,1), (2,2), (4,1), (4,2), (4,4), and (∞,∞). At each area constraint we choose the pareto optimal solution given by one of the (MAX_{in}, MAX_{out}) combinations.

A monotonic decrease in the execution cycles with the increasing number of register file ports is clearly visible from Figures 4.11 and 4.12. We observe that a register file with two read ports and two write ports is often more beneficial than a register file with four read ports and a single write port. In addition, a register file with four read ports and two write ports generates favorable design points.

In Figure 4.13, we analyze the four functions that constitute approximately 92

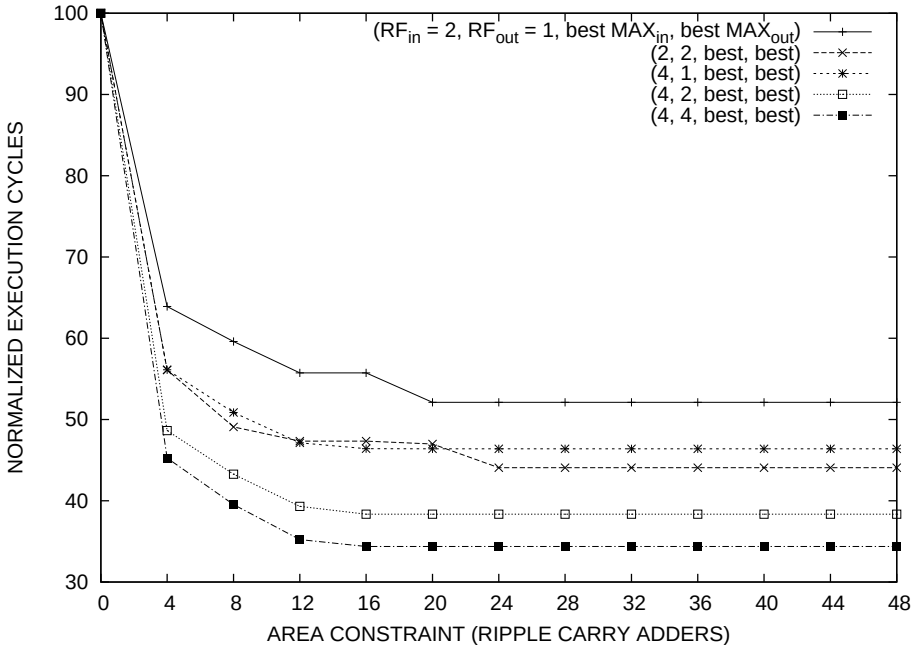


Figure 4.11. DES: effect of increasing the number of register file ports (i.e., RF_{in} and RF_{out}) on the performance. At each area constraint we choose the best MAX_{in}, MAX_{out} combination that minimizes the execution time.

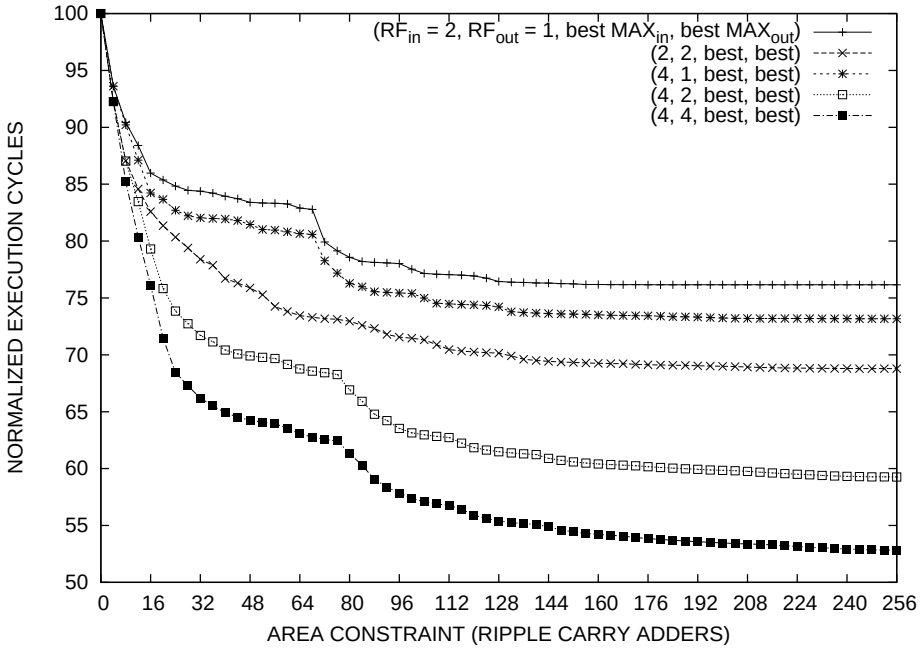


Figure 4.12. djpeg: effect of increasing the number of register file ports (i.e., RF_{in} and RF_{out}) on the performance. At each area constraint we choose the best MAX_{in}, MAX_{out} combination that minimizes the execution time.

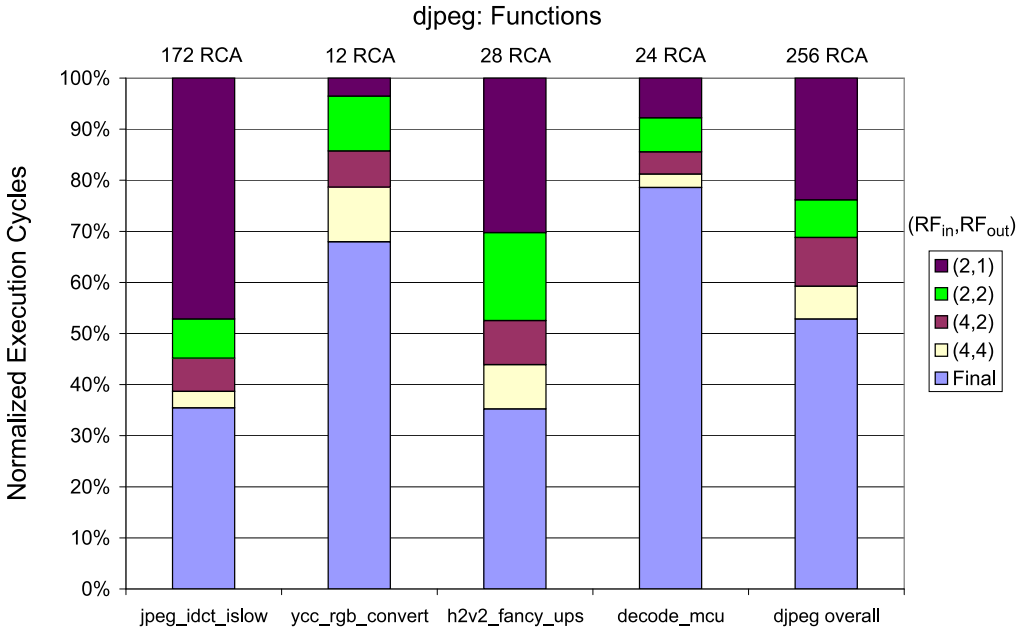


Figure 4.13. djpeg: increasing the number of register file ports (i.e., RF_{in}, RF_{out}) improves the performance. First four columns depict the achievable speed-up for the four most time consuming functions of djpeg. The last column depicts the achievable speed-up for djpeg. Area costs are given in units of ripple carry adders (RCAs).

per cent of the run-time of the *djpeg* benchmark. Given sufficient register file ports and area resources, the custom instructions we identify provide more than 2.8 times speed-up for *jpeg_idct_islow* and *h2_v2_fancy_upsample* functions, whereas acceleration for the other functions is more limited. The last column shows that given four read and four write ports, we achieve a 47 per cent reduction in the execution cycles of the overall application at an area cost of 256 adders. This translates to a 1.89 times overall speed-up. We observe that most of the area is consumed by the *jpeg_idct_islow* function (172 adders for maximal speed-up). Figure 4.12 depicts the area-delay trade-offs in more detail. As an example, using a register file with four read and two write ports, an overall speed-up of 1.63 times can be achieved at an area cost of 128 adders.

4.6. Effect of Loop Unrolling

In Figure 4.14, we demonstrate the effect of loop unrolling on the performance of SHA, and on the quality of the custom instructions our algorithms generate. We

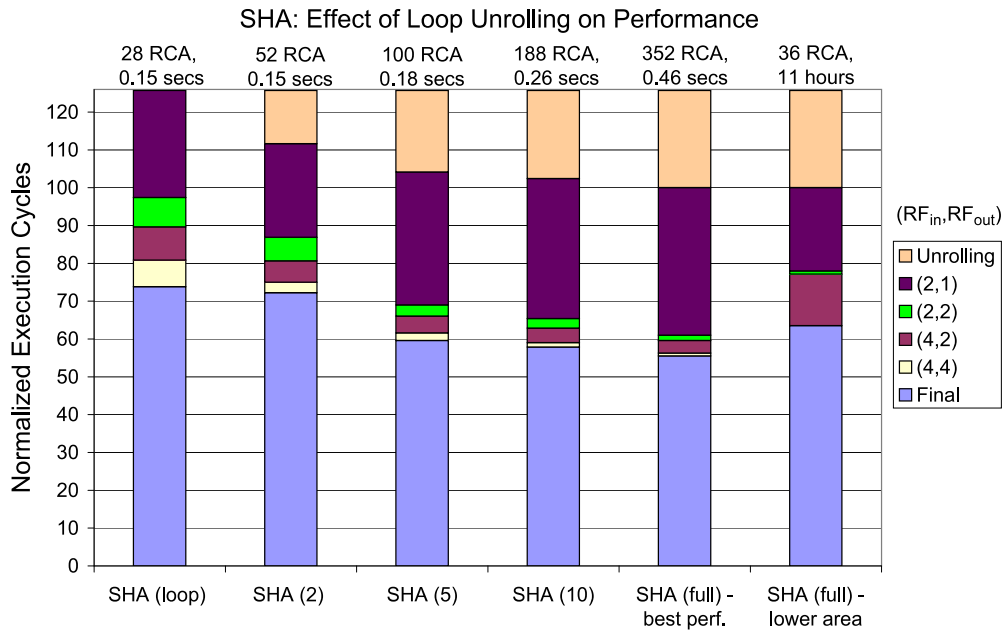


Figure 4.14. Loop unrolling improves the performance, and enables coarser grain custom instructions. SHA (2) represents the SHA implementation where the main loop is unrolled by two. Area costs in units of ripple carry adders (RCAs), and the run-times required to identify the custom instructions are shown.

consider five different SHA implementations: the first implementation does not unroll the loops; the next three implementations have their loops unrolled two, five and ten times. The fifth implementation has all SHA loops fully unrolled. We impose no constraints on the number of inputs and outputs for custom instructions (i.e., $MAX_{in} = \infty$, $MAX_{out} = \infty$) on the first five columns. The last column again targets the fully unrolled implementation and imposes $MAX_{in} = 4$ and $MAX_{out} = 2$.

We observe that the solution time required to identify the custom instructions is less than 0.5 seconds for the first five columns. Loop unrolling increases the size of the basic blocks and results in coarser grain custom instructions. The number of execution cycles monotonically decreases with the amount of unrolling. However, the area overhead increases monotonically, too. We observe that in the last column, by imposing constraints on the number of input and outputs, we identify smaller custom instructions, but we find several equivalent instances of these instructions in the code. The result is a fair speed-up at a reduced area cost. The last column provides a speed-

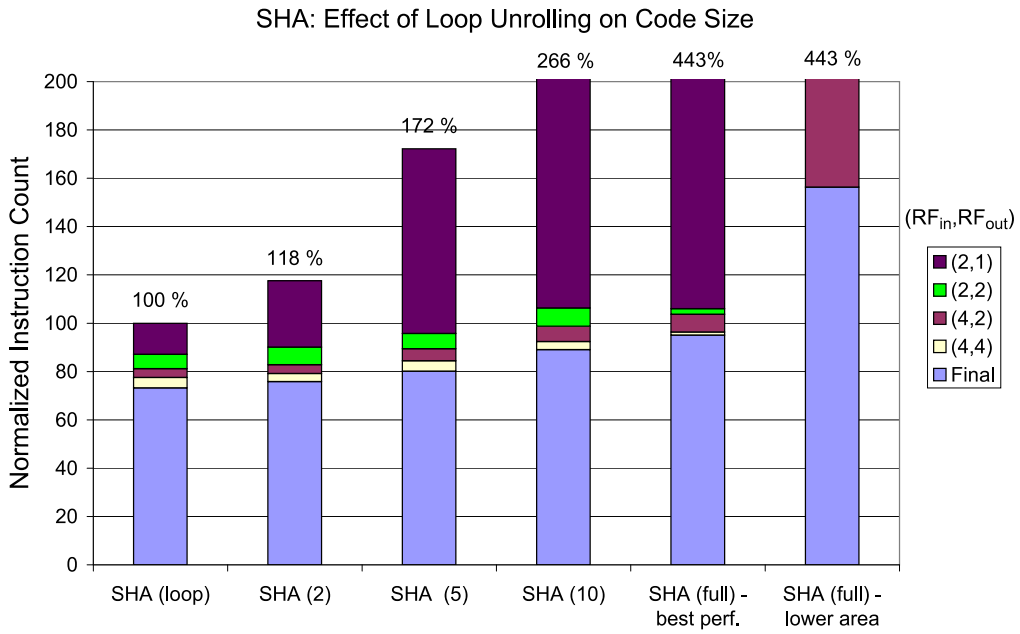


Figure 4.15. Loop unrolling increases the number of instructions in the code (up to 443 per cent). Code compression due to the use of custom instructions often compensates for this effect.

up of 1.56 times over the highest performing software implementation at the cost of only 36 adders. In this case, the time required to identify the custom instructions by our algorithms is around eleven hours.

In Figure 4.15, we consider the same design points shown in Figure 4.14, and we analyze the effect of loop unrolling on the code size. We observe that although loop unrolling increases the number of instructions, the compression due to the use of custom instructions often compensates for this effect. We note that the third column, where the main loop of SHA is unrolled five times, provides a speed-up of 1.68 times over the highest performing software implementation at an area cost of 100 adders, and results in a code size reduction of 20 per cent over the most compact software implementation. The associated solution time is less than 0.2 seconds.

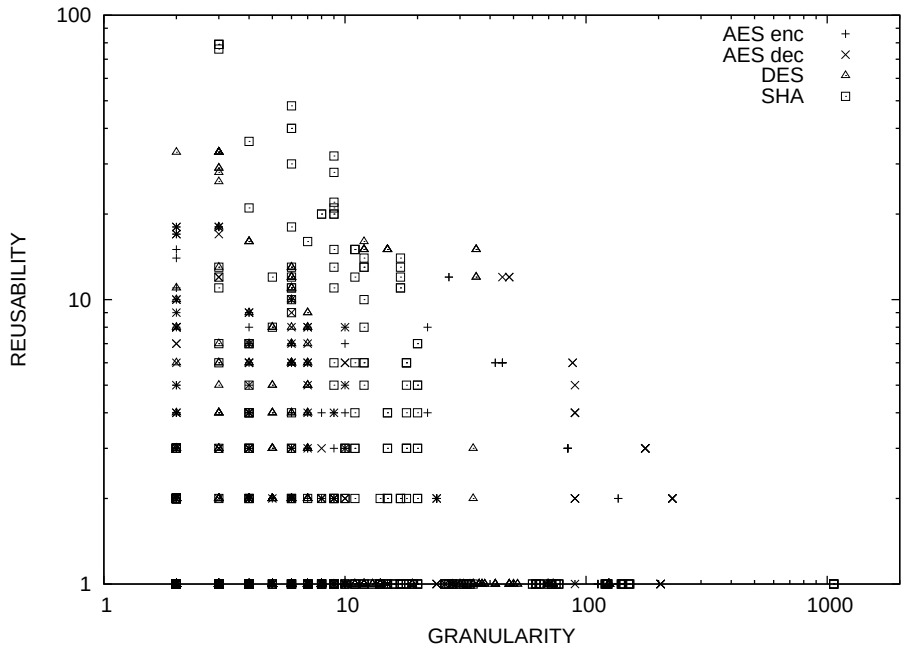


Figure 4.16. Granularity vs. reusability. Each point represents a custom instruction candidate identified by our algorithms.

4.7. Granularity vs. Reusability

We define the granularity of a custom instruction candidate as the number of base processor instructions contained in it. We define the reusability of a custom instruction candidate as the number of structurally equivalent instances of the candidate identified in the application DAGs. Figure 4.16 depicts the granularity of the custom instruction candidates we have generated from four cryptography benchmarks versus their reusability. We observe that candidates with high reusability are often fine grain, and coarse grain candidates usually have limited reusability. In one case, we identify a candidate consisting of 1065 base processor instructions, which has only a single instance in the code. In another case, we identify a candidate consisting of only three base processor instructions, which is reused 80 times. Another candidate identified by our algorithms consists of 45 base processor instructions and it has 12 instances in the code. Exploring different granularities in this manner allows us to identify the most promising area and performance trade-offs within the design space.

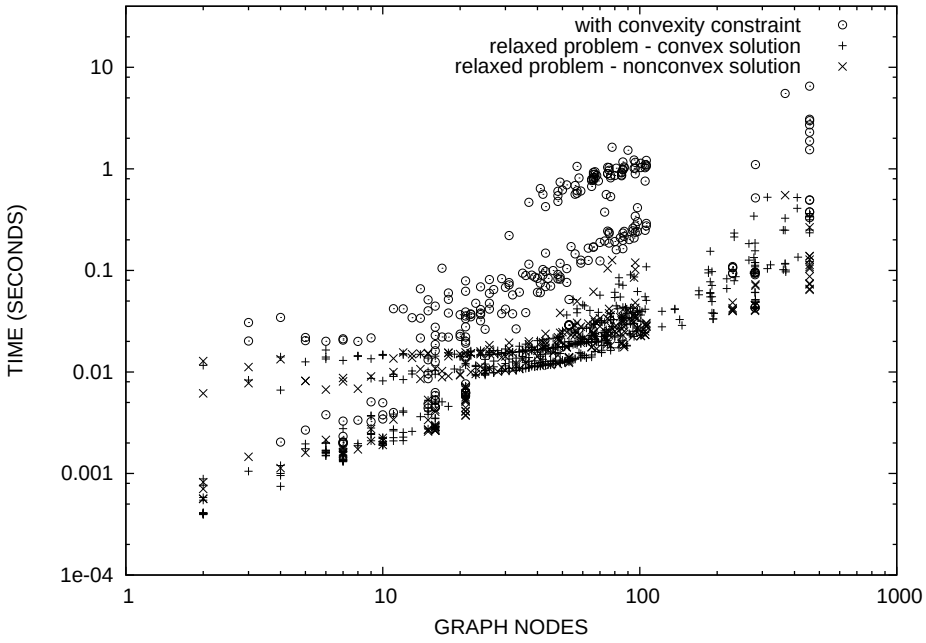


Figure 4.17. AES decryption: run-time performance of our template generation algorithm. Each point represents an iteration.

4.8. Run-time Results

The performance of our algorithms is quite notable. We observe that the run-time of our tool chain is dominated by the template generation algorithm described in Figure 3.4, which iteratively solves a series of ILP problems. In Figures 4.17 and 4.18 we plot the time taken to solve the ILP problems generated throughout the iterations of the template generation algorithm on AES decryption and DES benchmarks. In the figures we merge the run-time results for all (RF_{in}, RF_{out}, MAX_{in}, MAX_{out}) combinations we use in our experiments. The largest run-time we observe for a single iteration of AES decryption is around six seconds, and the largest run-time we observe for a single iteration of DES is around 100 seconds.

Often, the first iteration of the template generation algorithm is the most time consuming one. The DAG shrinks after each iteration, as identified templates are collapsed into single DAG nodes and excluded from further iterations. Note that at each iteration we first solve a relaxed problem, where the convexity constraint is not imposed on the custom instruction templates. In general, the time required to solve

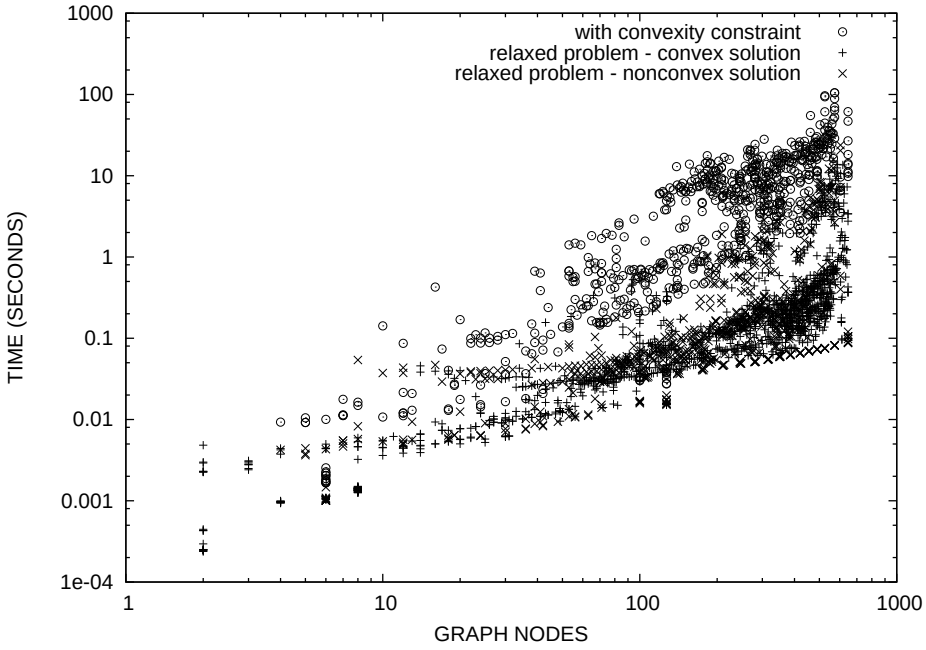


Figure 4.18. DES: run-time performance of our template generation algorithm. Each point represents an iteration.

the relaxed problem is much smaller, and the generated solution is often convex. If the solution to the relaxed problem is a convex template, we can proceed with the next iteration. If not, we impose the convexity constraint and solve once more. We use the solution to the relaxed problem as an upper bound for the original problem in order to reduce its run-time. We obtain optimal ILP results in all of our experiments.

In Tables 4.3 and 4.4 we describe the ILP statistics associated with the first iteration of the template generation algorithm on the largest basic blocks of four cryptography benchmarks for four (MAX_{in}, MAX_{out}) combinations (i.e., (4,1), (4,2), (4,4), and (∞,∞)). While generating these results, we set ($RF_{in} = 2$, $RF_{out} = 1$) if ($MAX_{in} = \infty$, $MAX_{out} = \infty$). Otherwise, we set ($RF_{in} = MAX_{in}$, $RF_{out} = MAX_{out}$). We observe that the run-time is usually a few seconds. Often, the relaxed problem generates a convex template, and we skip solving the original problem with the convexity constraint. The run-time may exceed one hour in some cases as it happens for fully unrolled SHA when $MAX_{out} = 2$ or $MAX_{out} = 4$. In these two cases, the upper bound provided by the relaxed problem reduces the run-time of the original problem considerably. We note that our algorithms are extremely efficient when the input and output constraints

Table 4.3. Relaxed problem: size of the largest basic block (BB), the number of integer decision variables (Vars), the number of linear constraints (Constrs), and the solution times associated with the first iteration of the template generation algorithm for four (MAX_{in},MAX_{out}) combinations: (4,1), (4,2), (4,4), and (∞,∞).

Benchmark	BB size	Relaxed problem					
		Vars	Constrs	Solution time (seconds)			
				(4,1)	(4,2)	(4,4)	(∞,∞)
AES enc.	317	879	1872	0.06	0.12	0.05	0.03
AES dec.	501	1591	3508	0.25	0.23	0.13	0.06
DES	822	1962	4043	2.75	0.42	0.09	0.08
SHA (full)	1155	3777	8885	193	5633	5116	0.22

Table 4.4. Original problem with the convexity constraint: solution times associated with the first iteration of the template generation algorithm. If the relaxed problem generates a convex template, we skip solving the original problem.

Benchmark	BB size	Original problem with convexity constraints					
		Vars	Constrs	Solution time (seconds)			
				(4,1)	(4,2)	(4,4)	(∞,∞)
AES enc.	317	1403	4124	*skipped*	*skipped*	0.78	0.14
AES dec.	501	2483	7404	*skipped*	0.55	2.98	0.37
DES	822	3417	9760	*skipped*	*skipped*	9.78	11.23
SHA (full)	1155	5899	18524	*skipped*	4122	1839	*skipped*

are removed (i.e., $MAX_{in} = \infty$, $MAX_{out} = \infty$): the run-time of the first iteration of the template generation algorithm on fully unrolled SHA is only 0.22 seconds.

In Table 4.5 we show the run-times of the first iteration of the exact algorithm of [24] on the same benchmarks and for the same (MAX_{in}, MAX_{out}) combinations given in Tables 4.3 and 4.4. We also show the run-times of our own method, which

Table 4.5. Run-time comparison with the exact algorithm of [24]. We show the solution times in seconds for four (MAX_{in},MAX_{out}) combinations. In the majority of the cases, the algorithm of [24] fails to terminate within 24 hours.

Benchmark	[24]				Our work			
	(4,1)	(4,2)	(4,4)	(∞,∞)	(4,1)	(4,2)	(4,4)	(∞,∞)
AES enc.	0.43	397	-	-	0.06	0.12	0.83	0.17
AES dec.	1.05	1417	-	-	0.25	0.78	3.11	0.43
DES	-	-	-	-	2.75	0.42	9.87	11.31
SHA (full)	3.94	317	-	-	193	9755	6955	0.22

are sums of the solution times of the relaxed and original problems from Table 4.3 and Table 4.4. We note that given ($RF_{in} = MAX_{in}$, $RF_{out} = MAX_{out}$), the objective function of our ILP formulation is equivalent to the merit function of [24]. We observe that the algorithm of [24] is generally efficient when the input/output constraints are tight (i.e., $MAX_{out} = 1$ or $MAX_{out} = 2$), although it fails to complete for DES within 24 hours even if $MAX_{out} = 1$. The algorithm of [24] becomes extremely inefficient when the constraints are loose (i.e., $MAX_{out} = 4$) or removed (i.e., $MAX_{out} = \infty$), and fails to complete for all four benchmarks within 24 hours. Our algorithm is faster in most of the cases, and successfully completes in all of the cases.

4.9. Performance and Code Size Results

In Figure 4.19 we describe the reduction in the execution cycle count for all the benchmarks from Table 4.2 while increasing the number of register file ports. The area costs, and the solution times are given on top of the columns for each benchmark. Using only a limited amount of hardware resources, we obtain a speed-up of up to 4.3 times for AES encryption, 6.6 times for AES decryption, 2.9 times for DES, 5.8 times for IDEA, 2.7 times for g721decode, 1.7 times for mpeg2encode and 4.7 times for rawcaudio. *Except for a few cases, we obtain the highest performing solutions when we remove the input/output constraints.* In most cases the highest performing solution is found in only a few seconds, but it may take up to a few minutes as observed for

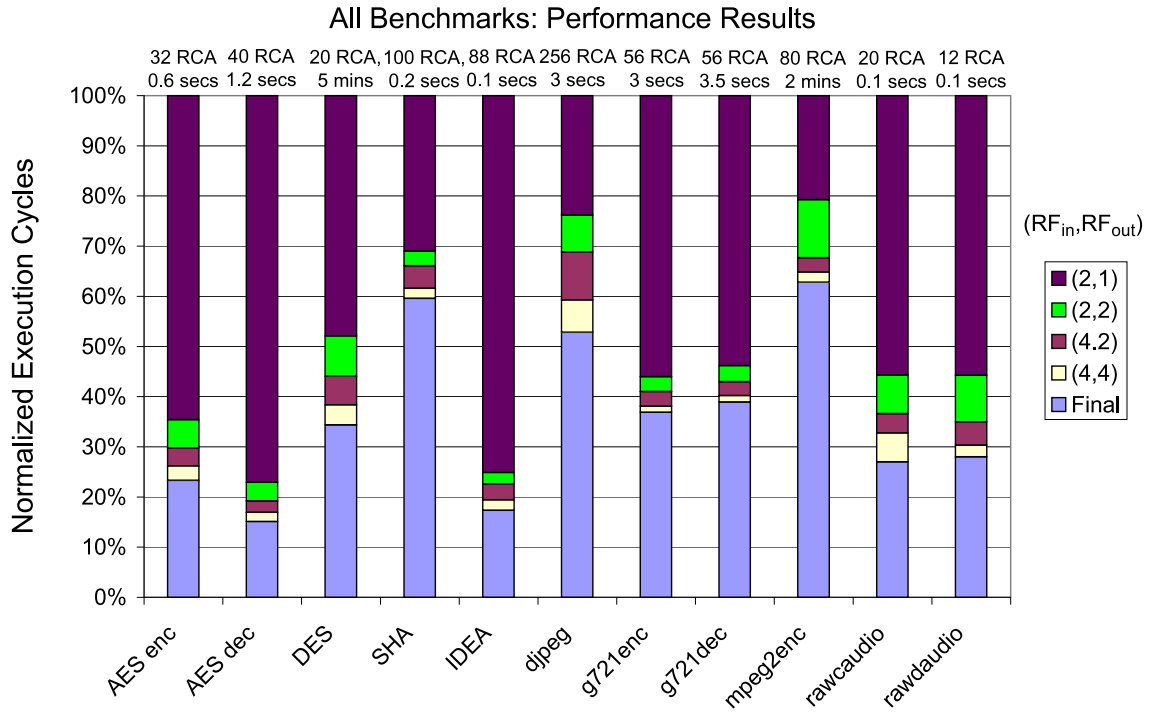


Figure 4.19. All benchmarks: increasing the number of register file ports (i.e., RF_{in}, RF_{out}) improves the performance. Area costs in units of ripple carry adders (RCAs), and run-times are shown above the columns. For each benchmark, the highest performing code with and without compiler transformations is taken as the base.

DES and mpeg2enc. For the same design points, Figure 4.20 shows the reduction in the total number of instructions in the code. A reduction of up to 72 per cent can be reached for AES decryption. For large benchmarks with small kernels, the code size reduction can be as small as six per cent, as it is observed for djpeg.

4.10. Processor Synthesis Results

We integrate our optimizing compiler that generates VHDL descriptions of the custom datapaths with a parameterizable academic customizable processor infrastructure [27] that implements the MIPS integer instruction set and supports up to 512 custom instructions. The infrastructure supports a core register file with two read ports and a single write port and generates state registers for each custom instruction operand. Dedicated data transfer instructions provide single cycle data transfers between the core register file and custom state registers. The combined framework au-

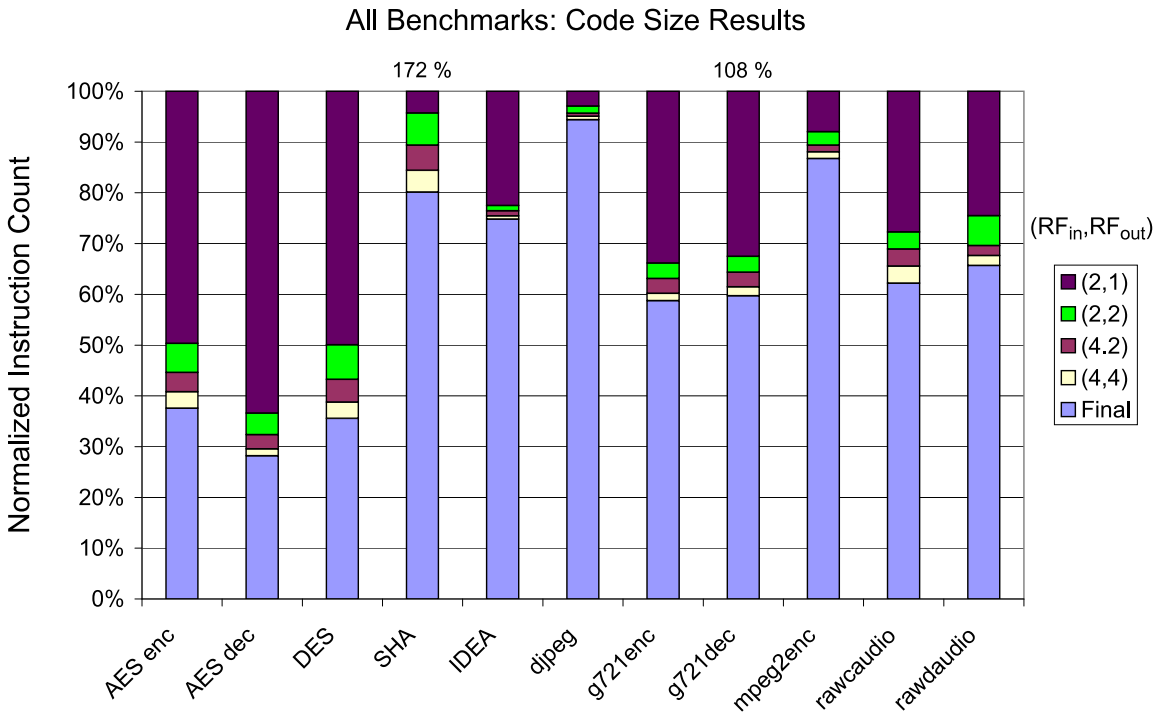


Figure 4.20. All benchmarks: increasing the number of register file ports (i.e., RF_{in}, RF_{out}) reduces the number of instructions in the code. For each benchmark, the smallest code with and without compiler transformations is taken as the base.

tomatically generates synthesizable processor descriptions directly from the application C/C++ source code. The base processor is synthesized to UMC's 130nm standard cell library using Synopsys Design Compiler and routing and layout generation are done using Cadence SoC Encounter. The base processor does not contain a multiplier and its operating frequency is 200 MHz, which is largely determined by the adder provided by standard Synopsys libraries. The area overhead of the base processor is 0.225 mm^2.

Given a register file with two read ports and two write ports (i.e., $RF_{in} = 2$, $RF_{out} = 1$), we evaluate five different constraints on the maximum number of input and output operands: $(MAX_{in}, MAX_{out}) \in \{(2,1),(2,2),(4,1),(4,2),(4,4)\}$ and 12 different area constraints (four to 48 ripple carry adder units, multiples of four only). Therefore, we evaluate 60 different parameter combinations for each benchmark. For each parameter combination we automatically generate a CPU core implementing the custom instructions selected. Some of the parameter combinations may result in equivalent processor configurations having the same set of custom instructions. We evaluate

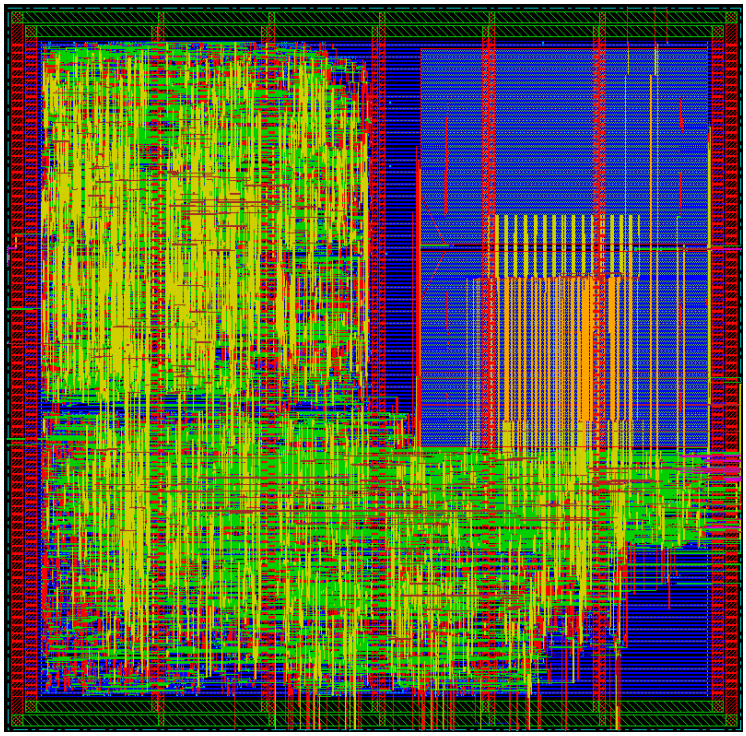


Figure 4.21. AES Decrypt processor layout (taken from [19]). 0.307 mm^2 area generated using a 130nm process technology. The top right area is the register file.

those processor configurations only once. We obtain realistic timing and area results by synthesizing each processor configuration to UMC's 130nm standard cell library using Synopsys Design Compiler and generating the layout and the routing using Cadence SoC Encounter. Figure 4.21 shows an example layout generated automatically from the AES C source code. In Figure 4.21, custom instructions incur only a 35 per cent increase over the area of the unextended processor while offering a speed-up of 4.3×.

Figure 4.22 summarizes timing results for each generated processor (179 in total). The volume of designs prohibits manual optimization, hence we report the worst case negative slack with a 200 MHz constraint for the tool vendor's recommended fully automated flow. Our technique pipelines multi-cycle instruction-set extensions to avoid decreasing the processor clock rate. Figure 4.22 shows that 48 per cent of the customized designs meet timing in the first pass. A further 31 per cent marginally fail to meet timing (<1ns negative slack), and the remainder miss by a greater margin.

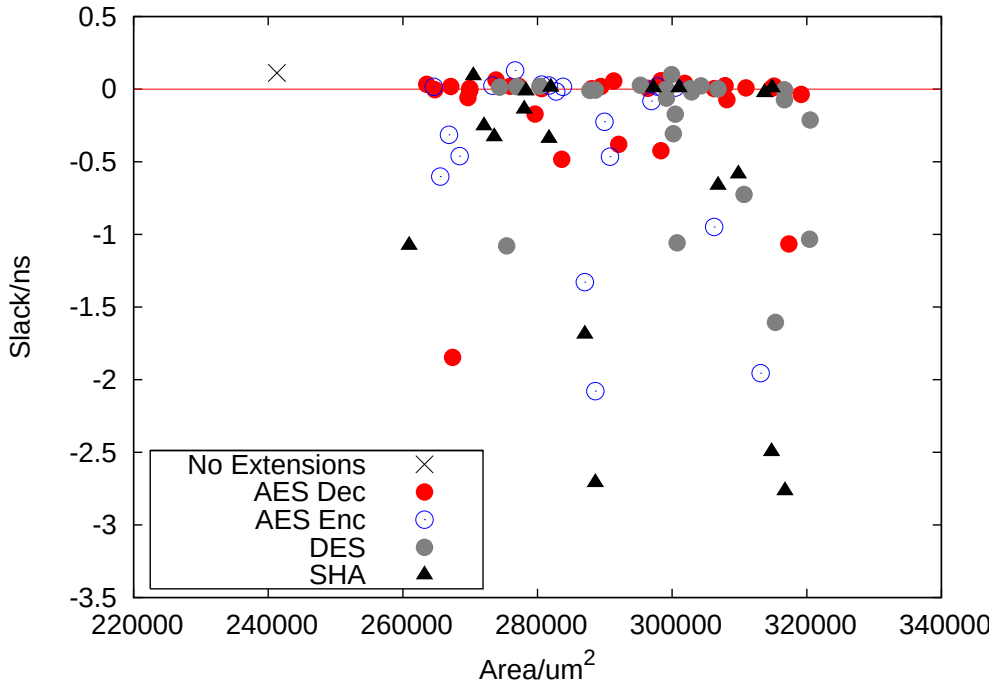


Figure 4.22. The ASIC area and the worst case negative timing slack with a 200 MHz constraint on the clock rate (taken from [19]).

4.11. Summary

We integrate our algorithms into the Trimaran academic compiler [26], automatically producing high level processor descriptions supporting the custom instructions, and the assembly code utilizing the custom instructions. Based on the compiler feedback, we evaluate the impact of the input/output constraints, register file port constraints, and code transformations such as if-conversion and loop unrolling on the performance and on the code size for a range of area constraints on eleven multimedia and cryptography benchmarks. We observe that in most of the cases we obtain the highest performing solutions when we remove the input/output constraints on the custom instructions. However, this happens at a relatively high area overhead. On the other hand, introducing input/output constraints on the custom instructions allows us to control the granularity of the custom instructions and to locate frequently used segments of the code. In this way, we explore a wide range of area, performance, and code size trade-offs in the design space. In addition, we demonstrate that our ILP based approach optimally solves very large problem instances representing basic

blocks having up to 1000 statements with and without input/output constraints, most of the time within only a few seconds. We show that the state of the art techniques fail to find the optimal solutions on the same problem instances. We provide examples of solutions identified by our algorithms that are not covered by the state of the art methods. We also integrate our compilation flow that generates custom datapaths into an academic custom processor synthesis infrastructure [27]. We show that around half of the custom instruction processors automatically generated from C/C++ source code meet the target clock frequency using fully automated ASIC synthesis flows.

5. A SIMPLIFIED MODEL

5.1. Motivation

The results of Section 4.4 and Section 4.9 show that the highest performing solutions are found when the constraints are removed on the number of input and output operands for custom instructions, although this happens at the expense of a high area overhead. In addition, we observe that the run-time of the ILP solver is significantly reduced with the removal of input/output constraints.

Re-evaluating Figure 4.5, we observe that the amount of data transfers can be significantly reduced for the DES custom instruction by applying a post processing step. The size of the static look-up tables (SBs) providing input operands for the custom instructions is only 256 bytes. Storing eight such tables within local memories, we can eliminate all the necessary register file reads, as well as the load instructions that retrieve the data from the main memory to the register file. Moreover, the indices generated at each round (ADRs) are used only within the next round. These values could be stored in custom state registers, avoiding all the related write backs to the register file. The variables X and Y could also be stored in state registers, eliminating all the relevant intermediate register file read and writes. Moreover, we observe that SK1 and SK2 variables are fetched sequentially from an array in the memory. The data within this array could be streamed directly from the main memory to the custom units if dedicated data transfer channels, such as the queue interfaces of Tensilica Xtensa LX [80], are available. Although optimizations of these kind are not yet included in our work, this example clearly demonstrates that removing the constraints on the number of input and output operands provides additional optimization opportunities that can have a significant impact on the achievable speed-up using custom instructions .

In this chapter, we study the problem of identifying maximal convex subgraphs of the application DAGs without imposing any constraints on the number of inputs and outputs. We assume that the the critical path delay and the data transfer costs can

be optimized separately by a post-processing step. In Section 5.2, we introduce a new model with a simplified objective function, and we show that the convexity constraint can be significantly simplified for the proposed objective function definition. Finally, we derive an upper bound on the search space for the simplified problem in Section 5.3.

5.2. Formulation of the Simplified Problem

We first define a simplified objective function as follows:

$$Z(T) = \max \sum_{i \in I_2} (s_i x_i) \tag{5.1}$$

We note that any instance of the objective function (5.1) can be transformed into the objective function (3.29) by setting $c_1 = 0$, $c_2 = 0$, and $h_i = 0$ for $i \in I_2$ in (3.29). In the rest of this section, we assume that objective function (5.1) is used instead of objective function (3.29) in the optimization. We further assume that $MAX_{in} = MAX_{out} = \infty$, and the only constraint imposed while identifying custom instructions is the convexity constraint of section 3.2.3.

Theorem 2. *Given the objective function (5.1), imposing constraint (3.21) only on the invalid nodes ($v^{inv} \in V_b^{invalid}$) is sufficient to guarantee that the optimal solution is convex, i.e., (3.21) holds also for the nodes $v^{val} \in V_b / V_b^{invalid}$ in the optimal solution.*

Proof. Suppose that the constraint (3.21) is imposed only on the invalid nodes. Assume that the optimal solution for the associated problem contains a node $v^{val} \in V_b / V_b^{invalid}$ that violates constraint (3.21). Then, the node v^{val} has both ancestors and descendants in the optimal solution. We are going to show that including v^{val} in the solution does not result in a violation of the constraint (3.21) for any node $v^{inv} \in V_b^{invalid}$.

In a convex solution, there exist three possible choices for each $v^{inv} \in V_b^{invalid}$:

- *The invalid node v^{inv} has ancestors, but no descendants in the optimal solution.*

In this case, we know that v^{val} cannot be a descendant of v^{inv}. If v^{val} was a descendant of v^{inv}, v^{inv} would have descendants in the solution, since v^{val} has descendants in the solution. Because v^{val} is not a descendant of v^{inv}, including v^{val} in the solution does not affect the feasibility of the constraint (3.21) for v^{inv}.

- *The invalid node v^{inv} has descendants, but no ancestors in the optimal solution.* In this case, we know that v^{val} cannot be an ancestor of v^{inv}. If v^{val} was an ancestor of v^{inv}, v^{inv} would have ancestors in the solution, since v^{val} has ancestors in the solution. Because v^{val} is not an ancestor of v^{inv}, including v^{val} in the solution does not affect the feasibility of the constraint (3.21) for v^{inv}.
- *The invalid node v^{inv} has neither an ancestor nor a descendant in the optimal solution.* In this case, we know that v^{val} is neither an ancestor nor a descendant of v^{inv}. Otherwise v^{inv} would have ancestors or descendants in the solution. As a result, including v^{val} in the solution does not affect the feasibility of the constraint (3.21) for v^{inv}.

We have shown that if there exists a $v^{val} \in V_b/V_b^{invalid}$ that violates the constraint (3.21) in the optimal solution, we can safely include it in the solution without violating the constraint (3.21). This contradicts with the optimality of the solution with respect to the objective function (5.1). Therefore, a $v^{val} \in V_b/V_b^{invalid}$ that violates constraint (3.21) cannot exist in the optimal solution if constraint (3.21) is satisfied by all $v^{inv} \in V_b^{invalid}$. Thus, imposing constraint (3.21) only on the invalid nodes ($v^{inv} \in V_b^{invalid}$) is sufficient to guarantee that the optimal solution is convex. □

Based on Theorem 2, we can simplify the formulation of the convexity constraint. For this purpose, we introduce the following additional indices:

$$I_5: \quad \textit{indices for nodes } v_i \in V_b^{invalid}$$
$$I_6: \quad \textit{indices for nodes } v_i \in V_b/V_b^{invalid}$$

We can now rewrite Equation (3.21) as follows:

$$x_i' \wedge a_i \wedge d_i = 0, \qquad i \in I_5 \tag{5.2}$$

Since $x_i = 0$ for $i \in I_5$, we can further simplify Equation (3.21) as follows:

$$a_i \wedge d_i = 0, \qquad i \in I_5 \tag{5.3}$$

The complete ILP formulation for the simplified problem is given as follows:

$$Z(T) = \max \sum_{i \in I_2} (s_i x_i)$$

$$a_i \wedge d_i = 0, \qquad i \in I_5$$

$$a_i = \begin{cases} 0 & \text{if } Pred(i) = \emptyset \\ \left(\bigvee_{j \in Pred(i)} (x_j \vee a_j) \right) & \text{otherwise} \end{cases}, \qquad i \in I_2$$

$$d_i = \begin{cases} 0 & \text{if } Succ(i) = \emptyset \\ \left(\bigvee_{j \in Succ(i)} (x_j \vee d_j) \right) & \text{otherwise} \end{cases}, \qquad i \in I_2$$

$$x_i, a_i, d_i \in \{0, 1\}.$$

5.3. An Upper Bound on the Size of the Search Space

A node v_i, $i \in I_6$ is part of the solution if it does not have an ancestor v_j, $j \in I_5$ for which an ancestor exists in the solution. Otherwise, v_j would have both an ancestor

and a descendant in the solution. Similarly, a node v_i, $i \in I_6$ is part of the solution if it does not have a descendant v_j, $j \in I_5$, for which a descendant exists in the solution.

We represent the set of ancestors, and the set of descendants of the nodes in $V_b/V_b^{invalid}$ that are in $V_b^{invalid}$ as follows:

$$Anc(i \in I_6) = \{j \in I_5 \mid \textit{There exists a path from } v_j \textit{ to } v_i \textit{ in } G\}$$
$$Desc(i \in I_6) = \{j \in I_5 \mid \textit{There exists a path from } v_i \textit{ to } v_j \textit{ in } G\}$$

Once a_j and d_j values are fixed for the nodes v_j, $j \in I_5$, whether a node v_i, $i \in I_6$ is part of the solution can be found by the following equation:

$$x_i = \left(\bigwedge_{j \in Anc(i)} a'_j \right) \wedge \left(\bigwedge_{j \in Desc(i)} d'_j \right), \qquad i \in I_6 \tag{5.4}$$

We note that there exists only three valid a_j, d_j choices for a $v_j \in V_b^{invalid}$: (1) $a_j = 1, d_j = 0$; (2) $a_j = 0, d_j = 1$; (3) $a_j = 0, d_j = 0$. The third choice can be disregarded, since the optimal solution would include as many nodes as possible by the nature of the objective function and only the first and the second choices can improve the objective value. We note that the case where an invalid node v_j has neither an ancestor nor a descendant in the solution can still occur (a) if we choose $a_j = 1, d_j = 0$ and none of the ancestors of the invalid node can be included in the solution either by the properties of the DAG G or because all of the ancestors of the invalid node are prohibited by the choices made for the remaining invalid nodes; (b) if we choose $a_j = 0, d_j = 1$ and none of the descendants of the invalid node can be included in the solution either by the properties of the DAG G or because all of the descendants of the invalid node are prohibited by the choices made for the remaining invalid nodes.

For all $v_j \in V_b^{invalid}$, it is sufficient to evaluate $2^{|V_b^{invalid}|}$ possible choices (i.e., $a_j = 1, d_j = 0$ or $a_j = 0, d_j = 1$, $j \in I_5$). Each choice is associated with a single

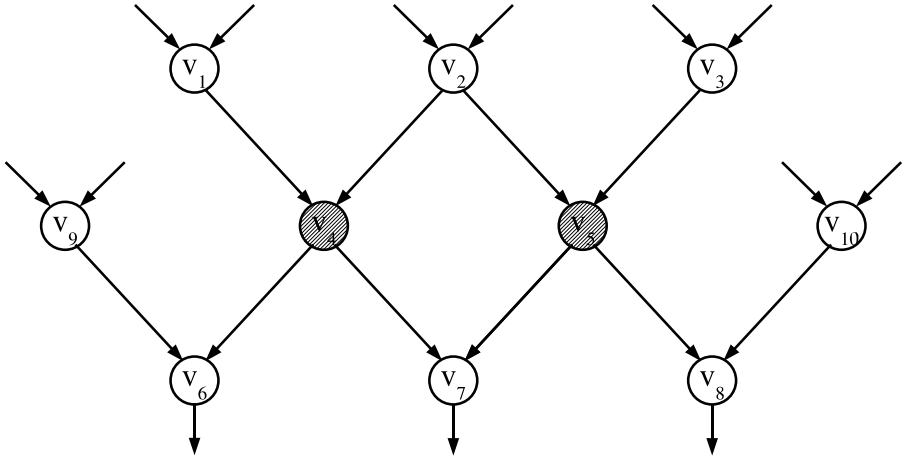


Figure 5.1. Example DAG: v_4 and v_5 are invalid nodes.

optimal solution that includes the maximal number of nodes from $V_b/V_b^{invalid}$, which can be inferred directly from the values of a_j and d_j for $j \in I_5$ using Equation (5.4). As a result, there exists an upper bound of $2^{|V_b^{invalid}|}$ on the size of the search space.

Table 5.1. Solutions for the DAG of Figure 5.1

(a_4, d_4)	(a_5, d_5)	Solution
(1,0)	(1,0)	$\{v_1, v_2, v_3, v_9, v_{10}\}$
(1,0)	(0,1)	$\{v_1, v_8, v_9, v_{10}\}$
(0,1)	(1,0)	$\{v_3, v_6, v_9, v_{10}\}$
(0,1)	(0,1)	$\{v_6, v_7, v_8, v_9, v_{10}\}$

Figure 5.1 depicts an example DAG. The nodes v_4 and v_5 are invalid nodes. Because there exists only two invalid nodes, there exists only $2^2 = 4$ possible choices we need to consider: (1) ancestors of v_4 and ancestors of v_5 can take part in the solution ($a_4 = 1, d_4 = 0$ and $a_5 = 1, d_5 = 0$); (2) ancestors of v_4 and descendants of v_5 can take part in the solution ($a_4 = 1, d_4 = 0$ and $a_5 = 0, d_5 = 1$); (3) descendants of v_4 and ancestors of v_5 can take part in the solution ($a_4 = 0, d_4 = 1$ and $a_5 = 1, d_5 = 0$); (4) descendants of v_4 and descendants of v_5 can take part in the solution ($a_4 = 0, d_4 = 1$ and $a_5 = 0, d_5 = 1$). Table 5.1 shows the solutions associated with these four choices. We note that the nodes v_9 and v_{10} can be included in the solutions associated with all the choices because they have neither ancestors nor descendants among invalid nodes.

5.4. Related Work

Recent work by Pothineni et al. [93] targets the same problem. Given a DAG, Pothineni et al. first define an incompatibility graph, where the edges represent pairwise incompatibilities between DAG nodes. Pothineni et al. define the ancestors and the descendants of an invalid node as incompatible. A node clustering step is applied to identify groupwise incompatibilities and to reduce the size of the incompatibility graph. The incompatibility graph representation allows Pothineni et al. to formulate the maximal convex subgraph enumeration problem as a maximal independent set enumeration problem. Pothineni et al. indicate that the complexity of the enumeration is $O(2^{N_C})$, where N_C represents the number of nodes in the incompatibility graph.

In [94], Verma et al. use maximal clique enumeration instead of maximal independent set enumeration. However, the two problems can be directly transformed into each other. Therefore, the approach of Verma et al. and the approach of Pothineni et al. are the same. We note that (see for example, Garey and Johnson [95, p.54]) for any graph $G(V, E)$ and a subset $V' \subseteq V$, the following statements are equivalent:

- V' is an independent set for G.
- V' is a clique in the complement $G^c(V^c, E^c)$ of $G(V, E)$, where $V^c = V$ and $E^c = \{(u, v) : u, v \in V \ and \ (u, v) \notin E\}$.
- V/V' is a vertex cover for G.

Figure 5.2 shows the incompatibility graph generated by Pothineni's algorithm for the DAG of Figure 5.1. The incompatibility graph contains seven nodes. One of the nodes is disconnected from the rest of the graph, and it can be ignored. According to Pothineni's work, the worst case complexity of maximal convex subgraph enumeration for this graph can be 2^6. On the other hand, we have shown in Section 5.3 that it is possible to enumerate all maximal convex subgraphs in only 2^2 algorithmic steps.

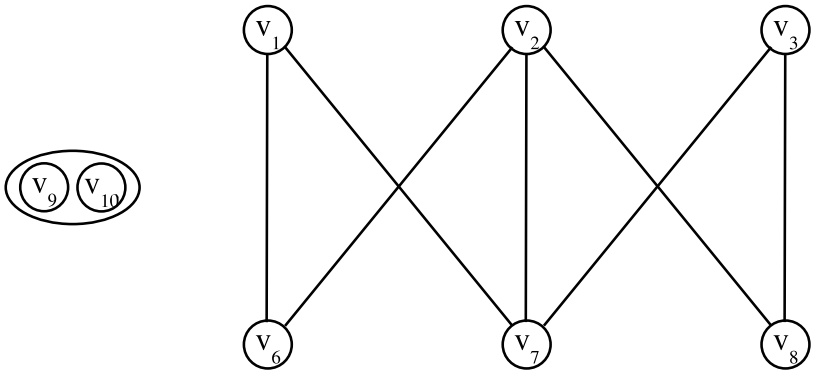


Figure 5.2. Incompatibility graph for the DAG of Figure 5.1.

5.5. Summary

In this chapter, we study a simplified version of the custom instruction identification problem. We do not consider all the subgraphs within application DAGs as potential solutions, but we search for maximal convex subgraphs, assuming that the data transfer costs and the critical path delays of the custom instructions can be independently optimized. We first provide a simplified objective function, and we show that the formulation of the convexity constraint can be significantly simplified as well. Next, we show that there exists an upper bound of $2^{|V_b^{invalid}|}$ on the size of the search space, where $|V_b^{invalid}|$ is the number of invalid nodes in a given DAG. The results of our analysis enable efficient enumeration algorithms with significantly lower worst case time complexity compared with the existing work targeting the same problem [93, 94].

6. CONCLUSIONS

6.1. Summary and Conclusions

The complexity of the SoC designs for embedded systems is continuously increasing. However, the designer productivity is improving at a much lower rate. The increasing time to market pressure in the competitive end-market is stimulating the use of more and more programmable components in complex SoCs. Combining efficiency with programmability, custom instruction processors are emerging as basic building blocks in the design of modern SoCs. In this thesis, we introduce efficient algorithms and tool chain support for the design space exploration of custom instruction processors. Starting with high level application source codes in C/C++, our tools provide a range of area/performance trade-offs, most of the time in only a few seconds. Our tools can be integrated into design space exploration tools for SoC and MPSoC designs, and can help in improving the designer productivity and overcoming the SoC design gap.

In this thesis, we describe an ILP based system called CHIPS for identifying custom instructions given the available data bandwidth and transfer latencies between the base processor and the custom logic. Our approach involves a baseline machine that supports architecturally visible custom state registers and dedicated data transfer channels. Given the data bandwidth between the base processor and the custom logic, our ILP model explicitly evaluates the data transfer costs. We iteratively solve a set of ILP problems in order to generate a set of custom instruction templates. At each iteration, ILP orders feasible templates based on a high level metric, and picks the one that offers the highest reduction in the schedule length. The iterative algorithm aims to maximize the code covered by custom instructions, and guarantees that the number of generated custom instruction templates is at most linear in the total number of instructions within the application. After template generation, we group structurally equivalent templates as custom instruction candidates based on isomorphism testing. We produce the behavioral descriptions of the custom instruction candidates, and we generate area and delay estimates using high level synthesis tools. Finally, we select

the most profitable candidates under area constraints based on a Knapsack model.

We enable designers to optionally constrain the number of input and output operands for custom instructions. We demonstrate that our algorithms are able to handle benchmarks with large basic blocks consisting of up to 1000 instructions with or without the input/output constraints. Our experiments show that the removal of input/output constraints results in the highest performing solutions. We demonstrate that these solutions cannot be covered by the subgraph enumeration algorithms of [22, 23, 24], which rely on input/output constraints for reducing the search space. On the other hand, we show that input/output constraints help us identify frequently used code segments and efficiently explore area/performance trade-offs in the design space.

We integrate our algorithms into an academic compiler infrastructure automatically producing high level processor descriptions supporting the custom instructions and the assembly code utilizing the custom instructions. Compiler feedback enables us to evaluate the impact of input/output constraints, register file port constraints, and code transformations such as if-conversion and loop unrolling on the performance and code size for a range of area constraints on eleven multimedia and cryptography benchmarks. Furthermore, we integrate our compilation flow that generates behavioral descriptions of the custom datapaths into an academic custom processor synthesis infrastructure. We show that a significant percentage of the automatically generated processors meet the target clock frequency using fully automated ASIC synthesis flows.

We develop a simplified model for custom instruction identification, where we ignore the effect of the critical path delays and data transfer costs. We show that the ILP model can be significantly simplified, and we derive a practical upper bound on the worst case time complexity of the solution algorithms. We demonstrate that the tight upper bound enables efficient maximal convex subgraph enumeration algorithms.

Our work improves upon the state-of-the art solution [22, 23, 24] for custom instruction identification that is being used by various research groups in the academia and in the industry [14, 72, 78, 85, 96]. The improved solution proposed in this thesis

has already found its use in the research community [97]. Other ILP based approaches similar to ours have followed our work as well [98].

6.2. Future Work

Our approach does not guarantee a globally optimal solution. Whether a truly optimal algorithmic flow exists is still an open research question. Our solution can still be improved by (1) combining our approach with pattern matching techniques [61, 69, 70, 71], in order to improve the utilization of the custom instructions we generate; (2) integrating our approach with datapath merging techniques [73, 74, 75], in order to exploit partial resource sharing across custom instructions for area efficient synthesis.

Future work includes exploration of architectural mechanisms and automation techniques that enable custom instructions to access the memory hierarchy efficiently. Existing work in this field includes embedding of static look-up tables [25] and relatively small sized arrays [99] within local memories in custom logic. However, a formal approach that partitions the program data across on-chip, off-chip, and local memories under capacity constraints has not yet been proposed. The allocation of the data in local memories directly affects the communication overhead of the custom instructions. Therefore, partitioning of the data between local and global memories should ideally be done simultaneously with the partitioning of the program into software and hardware components (i.e., base processor instructions and custom instructions).

A natural way of improving the memory access efficiency for streaming type applications is to use vector instructions with wide memory access capability. We believe that automatic identification of custom instructions that can operate on vectors of data elements [14] is a promising direction of research. The wide memory access bandwidth provided by the vector register files can significantly improve the efficiency of the custom instructions. From a different point of view, such an approach allows vector operations to represent complex computations, significantly improving their power and efficiency. Additionally, integration of data layout, array access, and control flow transformations within our tool chain in order to increase the number of vectorizable

loops in a given application would be a complementary direction of research.

The techniques proposed in this thesis are applied on the compiler intermediate representation as part of the compiler back-end. However, our technique is applicable at the source code level, binary level or assembly level as well. In fact, our ILP based approach has already been adapted to operate at the source code level [97]. In all of the cases, some of the compiler functionality, such as lexical and syntax analysis, as well as control flow and data flow analysis, have to be carried out on the input. At the source level, information related to loops, arrays, etc. is easily accessible. However, source level statements do not exactly match the instructions of the base processor, and modeling of the software latencies for source level statements may not be precise. On the other hand, working at the binary level enables acceleration of legacy code and libraries for which the source codes are not available. In this case, disassembler tools convert the software binaries into assembly code. However, complete recovery of source level constructs, such as loops and arrays, may not be possible. Coprocessor synthesis methodologies based on binary level hardware/software partitioning have already been proposed in the literature [100]. We believe that synthesis of custom instruction processors from software binaries could also be a promising research direction.

Other possible improvements to our existing approach can be listed as follows:

- support for cyclic data flow graphs to enable more efficient handling of the loop carried data and better optimization of the inner loops [25];
- inclusion of power and energy constraints in the design space exploration [101];
- adaptation of our approach to cover run-time reconfigurable processsors [42, 102];
- integration of our design flow into design space exploration for heterogeneous MPSoC devices [103, 104];
- a study of trade-offs between coprocessors and custom instructions [105];
- a study of trade-offs between VLIW instructions and custom instructions [14].

We believe that methodologies for the design space exploration of custom instruction processors will remain as an active area of research in the following years. Modern

custom instruction processors, such as Tensilica Xtensa LX [80] and Altera Nios II [11] already allow custom instructions to implement the functionality of a coprocessor. Custom functional units can now have internal state and memories, communicate with the base processor through hand shake signals, access the main memory, and send (receive) data to (from) external logic using FIFO channels. On the other hand, including arbitrarily complex logic in the datapath of a processor without increasing its cycle time, and the functional verification of the resulting system remain as grand challenges. We believe that research groups will continue to investigate new techniques that can better exploit the capabilities of customizable architectures, while trying to identify and overcome the architectural and the technological limits on processor customization.

APPENDIX A: IMPLEMENTATION DETAILS

A.1. If-Conversion Implementation

We implement if-conversion to selectively eliminate the control flow dependencies. We transform multiple basic blocks into a single basic block with predicated instructions. This improves the scope of our algorithms, and enables us to identify coarser grain custom instructions. Consider the sample code with if statements shown on the left part of Figure A.1. After applying if-conversion, we obtain the new code shown on the right part of the figure. The branches are eliminated and the whole code is transformed into a predicated basic block, where statements S2 and S3 are conditionally executed based on predicate values c1 and c2.

Given the CFG of a function, our aim is to partition the CFG into maximal single entry, single exit regions, and convert each such region into a predicated basic block. In order to identify the maximal single entry, single exit compiler regions, we apply interval analysis [106] on the CFGs. An interval is a maximal, single entry subgraph of the CFG. A CFG may be partitioned into a unique set of disjoint intervals using an algorithm similar to the one described in [106]. In our work, we impose the additional constraint that intervals may not cross loop boundaries. From each interval we generate, we heuristically remove CFG nodes until the number of exits become one, and we apply if-conversion on the remaining nodes generating a single basic block with predicated instructions. We apply if-conversion only on the most time consuming functions that constitute around 99 per cent of the overall run-time.

Since Trimaran does not support the SSA form, we adapted our data structures and the ILP formulation to support the peculiar data dependencies within predicated basic blocks. Consider again the code in Figure A.1. Although statements S2 and S3 produce the same output (i.e., a), there is no data dependency between the two statements, as they are executed mutually exclusively. Statement S2 generates an input operand of a template if it is not included in the template, and if statement S4 is

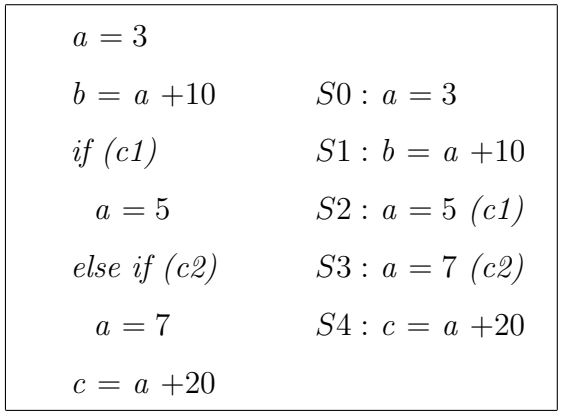


Figure A.1. The initial code contains branches (left). If-conversion eliminates the branches (right): statements S2 and S3 are conditionally executed based on the values of the predicates c1 and c2.

included (i.e., $x_2' \wedge x_4$). Similarly, statement S3 generates an input operand of a template if it is not included in the template, and if statement S4 is included (i.e., $x_3' \wedge x_4$). While calculating the number of input operands generated by statements S2 and S3 together, we cannot use summation as usual, since the two statements produce the same output operand. Instead we need to use the *or* operator (i.e., $(x_2' \wedge x_4) \vee (x_3' \wedge x_4)$). Calculation of the number of output operands can be adapted similarly.

Consider the case where statements S0 and S1 are not included in a template and statements S2, S3, and S4 are included. Although there is no flow dependency between statement S0 and statements S2 and S3, statement S0 provides the default value of the variable a in case both conditions (i.e., c1 and c2) do not hold. On the other hand, consider the case where statements S0, S1, and S2 are not included in the template and statements S3 and S4 are included. The valid value of the variable a as an input of the template is produced by the conditional execution of statement S2, and statement S0 is not relevant after this point. Therefore, statement S0 produces an input of the template if both statements S2 and S3 are included in the template, or statement S1 is included (i.e., $x_0' \wedge (x_1 \vee (x_2 \wedge x_3))$). Evaluation of whether statement S0 produces an output operand of a template is similar.

REFERENCES

1. Moore, G. E., 1965, "Cramming more components onto integrated circuits" *Electronics*, Vol. 38, No. 8, April.

2. Tensilica Inc., 2007, *Catching Up with Moore's Law: How to Fully Exploit the Benefits of Nanometer Silicon*, White Paper, http://www.tensilica.com

3. Henkel, J., 2003, "Closing the SoC Design Gap", *Computer*, Vol. 36, No. 9, pp. 119–121, September.

4. Semiconductor Research Corporation, http://www.src.com

5. Patterson, D. A. and J. L. Hennessy, 1996, *Computer Architecture: A quantitative approach*, Morgan Kaufmann Publishers.

6. Lanneer, D., J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen and G. Goossens, 1995, "CHESS: Retargetable Code Generation for Embedded DSP Processors", *Code Generation for Embedded Processors*, pp. 85–102, Kluwer Academic Publishers.

7. Fauth, A., J. Van Praet and M. Freericks, 1995, "Describing Instruction Set Processors using nML", *Proceedings of the European Design and Test Conference*, pp. 503–507, Paris, France.

8. Hoffmann, A., H. Meyr and R. Leupers, 2002, *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers.

9. Gonzalez, R. E., 2000, "Xtensa: A Configurable and Extensible Processor", *IEEE Micro*, Vol. 20, No. 2, pp. 60–70.

10. ARC International, 2007, *ARC 700 Core Family Brochure*, Product Brief, http://www.arc.com

11. Altera Corp., 2007, *Nios II Processor Reference Handbook*, http://www.altera.com

12. MIPS Technologies Inc., 2006, *MIPS Pro Series Processor Cores*, Product Brief, http://www.mips.com

13. Stretch Inc., 2007, *Stretch S6000 Family*, Product Brief, http://www.stretch.com

14. Goodwin, D. and D. Petkov, 2003, "Automatic Generation of Application Specific Processors", *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, pp. 137–147, November.

15. Binh, N. N., M. Imai, A. Shiomi and N. Hikichi, 1996, "A Hardware/Software Partitioning Algorithm for Designing Pipelined ASIPs with Least Gate Counts", *Proceedings of the 33rd Design Automation Conference (DAC)*, pp. 527–532.

16. Tensilica Inc., 2005, *The XPRES Compiler: Triple-Threat Solution to Code Performance Challenges*, White Paper, http://www.tensilica.com

17. CoWare Inc., http://www.coware.com

18. Atasu, K., G. Dündar and C. Özturan, 2005, "An Integer Linear Programming Approach for Identifying Instruction-Set Extensions", *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 172–177, Jersey City, NJ, September.

19. Atasu, K., R. G. Dimond, O. Mencer and W. Luk, 2006, "Towards Optimal Custom Instruction Processors", *Proceedings of the IEEE HOT Chips Conference*, Stanford, CA, August.

20. Atasu, K., R. G. Dimond, O. Mencer, W. Luk, C. Özturan and G. Dündar, 2007, "Optimizing Instruction-set Extensible Processors under Data Bandwidth Constraints", *Proceedings of the Design Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 588–593, Nice, France, April.

21. Atasu, K., C. Özturan, G. Dündar, O. Mencer and W. Luk, "CHIPS: Custom Hardware Instruction Processor Synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (Accepted in 2007).

22. Atasu, K., L. Pozzi and P. Ienne, 2003, "Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints", *Proceedings of the 40th Design Automation Conference (DAC)*, pp. 256–261, Anaheim, CA, June.

23. Atasu, K., L. Pozzi and P. Ienne, 2003, "Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints" *International Journal of Parallel Programming*, Vol. 31, No. 6, pp. 411–428, December.

24. L. Pozzi, K. Atasu and P. Ienne, 2006, "Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 25, No. 7, pp. 1209–1229, July.

25. Biswas, P., V. Choudhary, K. Atasu, L. Pozzi, P. Ienne and N. Dutt, 2004, "Introduction of Local Memory Elements in Instruction Set Extensions", *Proceedings of the 41st Design Automation Conference (DAC)*, pp. 729–734, San Diego, CA, June.

26. Trimaran: An Infrastructure for Research in Instruction Level Parallelism, http://www.trimaran.org

27. Dimond, R. G., O. Mencer and W.Luk, 2006, "Combining Instruction Coding and Scheduling to Optimize Energy in System-on-FPGA", *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 175–184, Napa Valley, CA, April.

28. Aho, A. V., R. Sethi and J. D. Ullman, 1986, *Compilers: Principles, Techniques and Tools*, Addison–Wesley Publishing.

29. Orailoglu, A. and D. D. Gajski, 1986, "Flow Graph Representation", *Proceedings of the 23rd Design Automation Conference (DAC)*, pp. 503–509.

30. Ferrante, J., K. J. Ottenstein and J. D. Warren, 1987, "The Program Dependency Graph and Its Uses in Optimization", *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319–349, June.

31. Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, 1989, "An Efficient Method of Computing Static Single Assignment Form", *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pp. 25–35.

32. Pingali, K., M. Beck, R. Johnson, M. Moudgill and P. Stodghill, 1990, "Dependence Flow Graphs: An Algebraic Approach to Program Dependecies", *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pp. 67–78.

33. Goldstein, S. C., H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor and R. Laufer, 1999, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration", *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, pp. 28–39, Atlanta, Georgia.

34. ARM Ltd., 2001, *ARM7TDMI Technical Reference Manual*, ARMDDI 0029G.

35. Hauser, J. R. and J. Wawrzynek, 1997, "Garp: A MIPS Processsor with a Reconfigurable Coprocessor", *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, Napa Valley, CA, April.

36. Mei, B., S. Vernalde, D. Verkest, H. De Man and R. Lauwereins, 2003, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix", *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pp. 61–70, September.

37. Rau, B. R., 1995, *Iterative Modulo Scheduling*, HP Labs Technical Report, HPL-

94-115, November.

38. Xilinx Inc., 2007, *MicroBlaze Processor Reference Guide*, http://www.xilinx.com

39. MIPS Technologies Inc., 2007, *Architectural Strengths of the MIPS32 74K Core Family*, White Paper, http://www.mips.com.

40. Wang, A., E. Killian, D. Maydan and C. Rowen, 2001, "Hardware/Software Instruction Set Configurability for System-on-Chip Processors", *Proceedings of the 38th Design Automation Conference (DAC)*, pp. 184–88, Las Vegas, Nev., June.

41. Rutenbar, R. A., M. Baron, T. Daniel, R. Jayaraman, Z. Or-Bach, J. Rose and C. Sechen, 2001, "(When) Will FPGAs Kill ASICs?", *Proceedings of the 38th Design Automation Conference (DAC)*, pp. 321–322, Las Vegas, Nev., June.

42. Seng, S. P., W. Luk and P. Y. K. Cheung, 2002, "Run-Time Adaptive Flexible Instruction Processors", *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Montpellier, France, September.

43. Dimond, R. G., O. Mencer and W. Luk, 2006, "Application-Specific Customisation of Multi-Threaded Soft Processors", *IEE Proceedings - Computers and Digital Techniques*, Vol. 153, No. 3, pp. 173–180, May.

44. Razdan, R. and M. D. Smith, 1994, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", *Proceedings of the 27th International Symposium on Microarchitecture (MICRO)*, pp. 172–80, San Jose, CA, November.

45. Wittig, R. D. and P. Chow, 1996, "OneChip: An FPGA Processor with Reconfigurable Logic", *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pp. 126–135, Los Alamitos, CA.

46. Hauck, S., T.W. Fry, M.M. Hosler and J.P. Kao, 1997, "The Chimaera Reconfigurable Functional Unit", *Proceedings of the IEEE Symposium on FPGAs for*

Custom Computing Machines (FCCM), pp. 87–96, Napa Valley, CA, April.

47. Kastrup, B., A. Bink and J. Hoogerbrugge, 1999, "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator", *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, CA, April.

48. Gupta, R. K. and G. D. Micheli, 1992, "System-level Synthesis Using Re-programmable Components", *Proceedings of the EURO-DAC*, pp. 2–7.

49. Ernst, R., J. Henkel and T. Benner, 1993, "Hardware-Software Cosynthesis for Microcontrollers", *IEEE Design & Test of Computers*, Vol. 10, No. 4.

50. Niemann, R. and P. Marwedel, 1997, "An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming", *Design Automation for Embedded Systems*, Vol. 2, No. 2, pp. 165–193, March.

51. Vahid, F. and T. D. Le, 1997, "Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning", *Design Automation for Embedded Systems*, Vol. 2, No. 2, pp. 237–261 , March.

52. Arato, P., S. Juhasz, Z. A. Mann, A. Orban and D. Papp, 2003, "Hardware-Software Partitioning in Embedded System Design", *Proceedings of the International Symposium on Intelligent Signal Processing*, pp. 197–202, September.

53. Holmer, B. K. and A. M. Despain, 1991, "Viewing Instruction Set Design as an Optimization Problem", *Proceedings of the 24th International Symposium on Microarchitecture (MICRO)*, pp. 153–162.

54. Faraboschi, P., G. Brown, J. A. Fisher, G. Desoli and F. Homewood, 2000, "Lx: A Technology Platform for Customizable VLIW Embedded Processing", *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pp. 203–213, Vancouver, June.

55. Van Praet, J., G. Goossens, D. Lanneer and H. De Man, 1994, "Instruction Set Definition and Instruction Selection for ASIPs", *Proceedings of the 7th International Symposium on High-Level Synthesis*, pp. 11–16.

56. Huang, I.-J. and A. M. Despain, 1995, "Synthesis of Application Specific Instruction Sets", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 14, No. 6, pp. 663–75, June.

57. Choi, H., J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang and C.-M. Kyung, 1999, "Synthesis of Application Specific Instructions for Embedded DSP Software", *IEEE Transactions on Computers*, Vol. 48, No. 6, pp. 603–14, June.

58. Arnold, M. and H. Corporaal, 2001, "Designing Domain Specific Processors", *Proceedings of the 9th International Workshop on HW/SW Codesign*, pp. 61–66, April.

59. Baleani, M., F. Gennari, Y. Jiang, Y. Pate, R. K. Brayton and A. Sangiovanni-Vincentelli, 2002, "HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform", *Proceedings of the 10th International Workshop on HW/SW Codesign*, pp. 151–56, May.

60. Brisk, P., A. Kaplan, R. Kastner and M. Sarrafzadeh, 2002, "Instruction Generation and Regularity Extraction For Reconfigurable Processors", *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, pp. 262–269, Grenoble, France.

61. Clark, N., H. Zhong and S. Mahlke, 2003, "Processor Acceleration Through Automated Instruction Set Customization", *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, pp. 184–88, San Diego, CA, December.

62. Alippi, C., W. Fornaciari, L. Pozzi and M. Sami, 1999, "A DAG Based Design Approach for Reconfigurable VLIW Processors", *Proceedings of the Design Au-*

tomation and Test in Europe Conference and Exhibition (DATE), pp. 778–79, March.

63. Sun, F., S. Ravi, A. Raghunathan and N.K. Jha, 2003, "A Scalable Application-Specific Processor Synthesis Methodology", *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 283–290, San Jose, CA, November.

64. Cheung, N., S. Parameswaran and J. Henkel, 2003, "INSIDE: INstruction Selection/Identification & Design Exploration for Extensible Processors", *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 291–297, San Jose, CA, November.

65. Cong, J., Y. Fan, G. Han and Z. Zhang, 2004, Application-Specific Instruction Generation for Configurable Processor Architectures, *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 183–189, Monterey, CA, February.

66. Yu, P. and T. Mitra, 2004, "Scalable Custom Instructions Identification for Instruction-Set Extensible Processors", *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)* , Washington, DC, September.

67. Biswas, P., S. Banerjee, N. Dutt, L. Pozzi and P. Ienne, 2005, "ISEGEN: Generation of High-Quality Instruction Set Extensions by Iterative Improvement", *Proceedings of the Design Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1246–1251, Munich, Germany, March.

68. Bonzini, P. and L. Pozzi, 2007, "Polynomial-Time Subgraph Enumeration for Automated Instruction Set Extension", *Proceedings of the Design Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1331–1336, Nice, April.

69. Leupers, R. and P. Marwedel, 1996, "Instruction Selection for Embedded DSPs with Complex Instructions", *Proceedings of the EURO-DAC*, pp. 200–205, Geneva, Switzerland.

70. Peymandoust, A., L. Pozzi, P. Ienne and G. De Micheli, 2003, "Automatic Instruction Set Extension and Utilization for Embedded Processors", *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 108–118, The Hague, The Netherlands.

71. Cheung, N., S. Parameswaran, J. Henkel and J. Chan, 2004, "MINCE: Matching INstructions using Combinational Equivalence for Extensible Processor", *Proceedings of the Design Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1020–1027, Paris, France, February.

72. Clark, N., J. Blome, M. Chu, S. Mahlke, S. Biles and K. Flautner, 2005, "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors", *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pp. 272–283, Washington, DC.

73. Geurts, W. , F. Catthoor, S. Vernalde and H. De Man, 1997, *Accelerator Data-path Synthesis for High-throughput Signal Processing Applications*, Kluwer, Boston, MA.

74. Brisk, P., A. Kaplan and M. Sarrafzadeh, 2004, "Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs", *Proceedings of the 41th Design Automation Conference (DAC)*, pp. 395–400, San Diego, CA, June.

75. Moreano, N., E. Borin, C. de Souza and G. Araujo, 2005, "Efficient Datapath Merging for Partially Reconfigurable Architectures", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 24, No. 7, pp. 969–80, July.

76. Cong, J., Y. Fan, G. Han, A. Jagannathan, G. Reinmann and Z. Zhang, 2005,

"Instruction Set Extension with Shadow Registers for Configurable Processors", *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 99–106, Monterey, CA, February.

77. Jayaseelan, R., H. Liu and T. Mitra, 2006, "Exploiting Forwarding to Improve Data Bandwidth of Instruction-Set Extensions", *Proceedings of the 43rd Design Automation Conference (DAC)*, Anaheim, CA, July.

78. Pozzi, L. and P. Ienne, 2005, "Exploiting pipelining to Relax Register-File Port Constraints of Instruction-Set Extensions", *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, September.

79. Lee, J., K. Choi and N. Dutt, 2002, "Efficient Instruction Encoding for Automatic Instruction Set Design of Configurable ASIPS", *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 649–654, San Jose, CA.

80. Martin, G., 2006, "Recent Developments in Configurable and Extensible Processors", *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 39-44, September.

81. Allen, J.R., K. Kennedy, C. Porterfield and J. Warren, 1983, "Conversion of Control Dependence to Data Dependence", *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, January.

82. ILOG CPLEX: High-Performance Software for Mathematical Programming and Optimization, http://www.ilog.com/products/cplex/

83. Synopsys Inc., http://www.synopsys.com

84. Aditya, S., V. Kathail and B. Rau, 1998, *Elcor's Machine Description System*, HP Labs Technical Report, HPL-98-128, October.

85. Clark, N., A. Hormati, S. Mahlke and S. Yehia, 2006, "Scalable Subgraph Map-

ping for Acyclic Computation Accelerators", *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)* , pp. 147–157, Seoul, Korea, October.

86. Nauty Package. http://cs.anu.edu.au/people/bdm/nauty.

87. Yu, P. and T. Mitra, 2005, "Satisfying Real-Time Constraints with Custom Instructions", *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* , pp. 166–171, Jersey City, NJ, September.

88. Kathail, V., M. Schlansker and B. Rau, 1993, *HPL-PD Architecture Specification, Version 1.0*, HP Labs Technical Report, HPL-93-80R1.

89. Atasu, K., M. Macchetti and L. Breveglieri, 2004, "Efficient AES Implementations for ARM Based Platforms", *Proceedings of the ACM Symposium on Applied Computing (SAC)*, March.

90. XySSL - DES and Triple-DES Source Code, http://xyssl.org

91. Guthaus, M. et al. *MiBench: A Free, Commercially Representative Embedded Benchmark Suite*, http://www.eecs.umich.edu/mibench

92. Lee, C., M. Potkonjak and W. H. Mangione-Smith, 1997, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

93. Pothineni, N., A. Kumar and K. Paul, 2007, "Application Specific Datapath with Distributed I/O Functional Units", *Proceedings of the 20th International Conference on VLSI Design*, pp. 551–558, Hyderabad, India, January.

94. Verma, A. K., P. Brisk and P. Ienne, 2007, "Rethinking Custom ISE Identification: A New Processor-Agnostic Method", *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, pp.

125–134, Salzburg, Austria, September.

95. Garey, R. M. and D. S. Johnson, 1979, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., New York.

96. Lam, S. K., T. Srikanthan and C. T. Clarke, 2006, "Rapid Generation of Custom Instructions Using Predefined Dataflow Structures", *Microprocessors and Microsystems: Special Issue on FPGA-based Reconfigurable Computing*, Vol. 30, No. 6, pp. 355–366, September.

97. Bennett, R. V., A. C. Murray, B. Franke and N. Topham, 2007, "Combining Source-to-Source Transformations and Processor Instruction Set Extensions for the Automated Design-Space Exploration of Embedded Systems", *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 83–92, July.

98. Leupers, R., K. Karuri, S. Kraemer and M. Pandey, 2006, "A Design Flow for Configurable Embedded Processors Based on Optimized Instruction Set Extension Synthesis. *Proceedings of the Design Automation and Test in Europe Conference and Exhibition (DATE)*, Munich, Germany, March.

99. Biswas, P., L. Pozzi, P. Ienne and N. Dutt, 2006, "Automatic Identification of Application-Specific Functional Units with Architecturally Visible Storage", *Proceedings of the Design Automation and Test in Europe Conference and Exhibition (DATE)*, Munich, Germany, March.

100. Stitt, G., F. Vahid, G. McGregor and B. Einloth, 2005, "Hardware/Software Partitioning of Software Binaries: A Case Study of H.264 Decode", *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 285–290, Jersey City, NJ, September.

101. Fei, Y., S. Ravi, A. Raghunathan and N. K. Jha, 2004, "A Hybrid Energy-Estimation Technique for Extensible Processors", *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, Vol. 23, No. 5, pp. 652–64, May.

102. Bauer, L., M. Shafique, S. Kramer and J. Henkel, 2007, "RISPP: Rotating Instruction Set Processing Platform", *Proceedings of the 44th Design Automation Conference (DAC)*, pp. 791–796, San Diego, CA, June.

103. Sun, F., S. Ravi, A. Raghunathan and N. K. Jha, 2006, "Application-Specific Heterogeneous Multiprocessor Synthesis Using Extensible Processors", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 25, No. 9, pp. 1589–1602, September.

104. Shee, S. L. and S. Parameswaran, 2007, "Design Methodology for Pipelined Heterogeneous Multiprocessor System", *Proceedings of the 44th Design Automation Conference (DAC)*, pp. 811–816, San Diego, CA, June.

105. Sun, F., S. Ravi, A. Raghunathan and N. K. Jha, 2006, "Hybrid Custom Instruction and Co-processor Synthesis Methodology for Extensible Processors", *Proceedings of the 19th International Conference on VLSI Design*, pp. 473–476, January.

106. Allen, F. E. and J. Cocke, 1976, "A Program Data Flow Analysis Procedure", *Communications of the ACM*, Vol. 19, No.3, pp. 137–147.