uOttawa
L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Jonathan Parri
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical and Computer Engineering)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

A Framework for Selection and Integration of Custom Instructions for Hybrid System-on-Chips

TITRE DE LA THÈSE / TITLE OF THESIS

Miodrag Bolic
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

Voicu Groza
CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

Emil Petriu

James Green

Gary W. Slater
Le Doyen de la Faculté des études superieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

A Framework for Selection and Integration of Custom Instructions for Hybrid System-on-Chips

by

Jonathan Parri

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the M.A.Sc. degree in
Electrical and Computer Engineering

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

© Jonathan Parri, Ottawa, Canada, 2010

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-73819-1
Our file Notre référence
ISBN: 978-0-494-73819-1

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

TRADITIONALLY, common processor augmentation solutions have involved the addition of coprocessors or the datapath integration of custom instructions within extensible processors as Instruction Set Extensions (ISE). Rarely is the hybrid option of using both techniques explored. Much research already exists concerning the mutually exclusive identification and selection of custom hardware blocks from hardware/software partitioning techniques. The question of how to best select and use this hardware within a user system where both coprocessors and datapath augmentations are possible and are mutually inclusive remains. Here a system with both types of these custom instructions is denoted as a hybrid SoC.

In this work, both the coprocessor and internal datapath custom instruction design decisions are modeled within a design space exploration framework created to facilitate hybrid SoC development. We explore how to best select and integrate these instructions using available metrics and traditional combinatorial optimization techniques while packaging these ideas together into a complete toolchain framework. This framework is integrated into industry design flow tools in an attempt to achieve significant performance gains over existing methodologies.

Acknowledgements

A special thanks to all the members of the Computer Architecture Research Group for their tireless support and dedication to difficult research avenues. I would especially like to acknowledge the support of Daniel, Michael and Saurabh along with my supervisors Dr. Miodrag Bolic and Dr. Voicu Groza for not only their research support but friendship as well.

Finally, my family for instilling the belief that perseverance and dedication can conquer any feat imaginable I am most grateful. I am sure they will appreciate the completion of this work as much as myself.

Contents

1 Introduction **1**

1.1 Design Space Exploration 1

1.1.1 Overview 1

1.1.2 Modeling the DSE Problem 2

1.2 The Hybrid SoC 4

1.3 Motivation and Contributions 5

1.4 Scope 6

2 Background **8**

2.1 Introduction to Instruction Set Extensions 8

2.1.1 Overview of ISEs 8

2.1.2 ISE Identification 9

2.1.3 Automated Algorithms 10

2.2 Coprocessors 11

2.2.1 Introduction to Coprocessors 11

2.2.2 Architecture 11

2.2.3 Coprocessor Instruction Identification Algorithms 15

2.3 Internal Datapath Augmentations 16

2.3.1 Architecture 16

2.3.2 Current Methods and Considerations 17

2.4 Combinatorial Optimization-The Integer Linear Program 18

2.4.1 Integer Linear Programs 18

2.4.2 Finding an Optimal Solution 19

3 Prior Art **22**

3.1 Overview 22

3.2 Shared Ideas 22

3.2.1 Similarities with Multiprocessing System on Chip Modeling Efforts 23
3.2.2 Separate Problems-Independent Directions 24
3.3 Beginning of the Hybrid Custom Instruction Configuration 24

4 Introduction to the Knapsack Problem 26
4.1 The Generic Knapsack Problem 26
4.2 Knapsack Variants 27
4.3 Solving the Multiple Knapsack Problem 28

5 Knapsack Extensions for Hybrid Selection 30
5.1 Overview 30
5.2 Selection Considerations 30
5.3 ALU Datapath Extension-Variable Cycle 32
5.4 Coprocessor versus ALU 32
5.5 Quantifying Additional Considerations 34
5.6 The Hybrid SoC Model 40
5.6.1 Separate and Known Hardware Constraints 40
5.6.2 Known Global Constraints-General Form 41
5.6.3 ISA Limitations 41
5.6.4 Summary 42

6 Relaxing Constraints 44
6.1 Resolving Input/Output Limitations 44
6.2 Finite Target Hardware 46
6.3 Multiple Hardware Designs 47

7 Algorithmic Perspective 50
7.1 Collecting Candidate Instructions 50
7.2 ILP Solver Selection 51
7.3 Formulating a Hybrid SoC Design Flow Algorithm 51
7.4 Multi-Threading for Multiple Executions 54
7.5 Formulating a Full FPGA-based Hybrid SoC Toolchain 55
7.5.1 Configuring the Working Directory and Path Variables 58
7.5.2 Overview of Found Custom Instructions and Target Configuration 59
7.5.3 Solver and Virtual Machine Settings 60
7.5.4 Target Hardware Constraints 61

7.5.5 Candidate Synthesis 62
7.5.6 Solver Execution 63
7.5.7 Reviewing Results 64
7.6 Software Overview 65
7.7 Compiler Integration 66

8 SoC Hybrid Integration with NIOS II-A Case Study 68
8.1 Overview 68
8.2 Target SoC Environment 70
8.3 Baseline Execution 71
8.4 Custom Instruction Candidates 72
8.5 Candidate Selection 73
8.6 Final Integration 77
8.7 Garnering Further Speedup 78
8.8 Concluding Remarks 79

9 Further Evaluation 81
9.1 Overview 81
9.2 Examining Solver Times 83
9.3 Theoretical Performance Assessment 84
9.4 Xtensa Hybrid SoC with XPRES Identification 87

10 Extensions 90
10.1 Overview 90
10.2 Exploiting DMA within Coprocessors 91
10.3 Changing Objectives 92
10.4 Multi-Objective Directions 92
10.5 Scheduling 94
10.6 Proper Polynomial Time Approximation Scheme 94

11 Conclusion 95
11.1 Overview 95
11.2 Discussion 96
11.2.1 Results 96
11.2.2 Concluding Remarks 96

A Altera Cyclone II Hardware Sizes **98**

B NIOS II Case Study Test Program **101**

C Generated ILP For NIOS II Case Study **104**

D Glossary of Terms **106**

References **109**

List of Tables

1.1 Considerations for Internal and External Custom Instructions. 5

4.1 Generalizations of the Knapsack Problem. 28

5.1 Prerequisite Data 31
5.2 Summary of Model Creation. 43

6.1 Altera FPGA Hardware Sizes as Logic Elements (LE). 46
6.2 Sample Candidate Scenario with Multiple Hardware Configurations. . . . 49

8.1 Generated and Obtained Candidate Instruction Metrics. 72
8.2 Maximum Frequency of Hardware. 74

9.1 Solver Times with Global Area. 85
9.2 FFT Hardware Selection for Hybrid SoC. 86
9.3 CRC32 Hardware Selection for Hybrid SoC. 87
9.4 Hybrid SoC with Tensilica. 89

List of Figures

1.1 Sample of Exploitable Avenues in DSE. 2
1.2 A Design Space as a Directed Graph 3
1.3 Example of a Hybrid SoC. 4
1.4 Overview of Defined Hybrid SoC Flow. 7

2.1 MicroBlaze FSL Coprocessor Architecture. 12
2.2 Coprocessor Integration Models. 14
(a) Bus Only. 14
(b) Register File Only. 14
(c) Mixed Implementation. 14
2.3 Datapath with Custom Hardware Partitions. 16
2.4 Sample MIPS Instruction R-Format. 17
2.5 Physical Representation of an LP. 18
2.6 Physical Representation of an ILP. 19
2.7 Branching, Formation of Subproblems. 20
(a) Sample ILP Program. 20
(b) Extracting Subproblems from x_1. 20

4.1 Knapsack Motivation. 27

5.1 Cycles Saved From Custom Instruction Implementation. 34
5.2 Extracted DFG of Candidate. 36
5.3 Internal Custom Instruction Execution with Accompanying Stalls. 37
5.4 Timing of Coprocessor Implementation using Coprocessor Interface. . . . 38
5.5 Execution with Bus-Based Coprocessor. 38
5.6 Coprocessor Signaling Task Completion. 39

6.1 Typical Register File. 44

6.2 Pipelining to Compensate for IO Limitations. 45
6.3 Generation of Multiple Models with Varying Target Considerations. . . . 47

7.1 Custom Instruction Object List. 51
7.2 Threading Model for Multiple Target FPGA Candidates. 54
7.3 Threaded Execution Results. 55
7.4 Identified 3-1 Custom Instruction for 32-bit Unsigned Integers. 56
7.5 Adder Subtracter for 32-bit unsigned integers, VHDL and constraint file. 57
7.6 Selecting the Custom Instruction Directory and Synthesis Tools. 58
7.7 Reviewing Custom Instruction Files and Configuration of Target FPGA. 59
7.8 Solver and Java Settings. 60
7.9 Target Hardware Constraint Settings. 61
7.10 Getting Area and Latency Information Automatically. 62
7.11 Spawning Threads and Executing the Solvers. 63
7.12 Reviewing Results. 64
7.13 Toolchain Package Diagram. 65
7.14 *control* Class Diagram. 66
7.15 Internal and External Component Interaction. 67

8.1 NIOS II SoC Target Architecture. 70
8.2 Profiled Required Cycles for Fully Accelerated Design versus Software. . 73
8.3 Profiled Required Cycles for Hybrid SoC Design versus Software. 76
8.4 Hybrid SoC Validation on FPGA. 78
8.5 Comparison of 50 and 60MHz **factorial()**. 79

9.1 Solver Time Trials. 84

10.1 DMA Coprocessor. 92
10.2 DAGs with Linear Memory Load Access. 93

Chapter 1

Introduction

1.1 Design Space Exploration

1.1.1 Overview

In the development of embedded systems, *design space exploration* (DSE) is a general term used to describe the generation and analysis of multiple design candidates for a given set of system requirements. The problem of DSE is widely known and studied by scientists and engineers across the globe.

The problem of design space exploration can be simply put as the formulation of potential solutions to the requirements of a desired target system. Suppose an engineer would like to design an embedded system to run a particular algorithm. Before the design phase has even begun the engineer knows what kind of performance the system must adhere to, the maximum power requirements and the cost of the target hardware. DSE is the action of creating and analyzing target system hardware and software design choices to meet these requirements as best as possible.

Design space exploration may involve the consideration of many different design options or decisions. These options may include modification of a processor's Instruction-Set Architecture (ISA), the addition of hardware accelerators or the use of multiple processors. DSE takes place at both the hardware and software level making it a prime facilitator of hardware/software codesign. The various avenues that can be taken when exploring hardware design possibilities are shown in Fig. 1.1

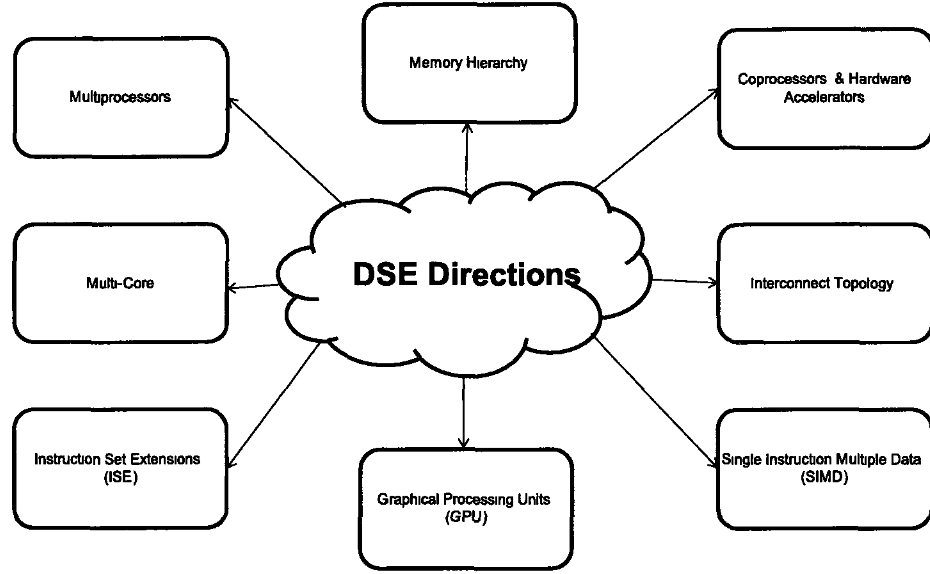


Figure 1.1: Sample of Exploitable Avenues in DSE.

The focus of this work involves the consideration of both internal Instruction-Set Extensions (ISE) and coprocessors together. These two options are only a subset of the possibilities in DSE. The implementation of these two options in a System-on-Chip (SoC) target will yield a *hybrid SoC*.

As we consider more and more candidates and design avenues it becomes difficult and eventually impossible to manually enumerate all possible DSE options. By effectively modeling this problem it can become a key component of *design automation* requiring little input from the end-user and offering a relatively quick solution.

1.1.2 Modeling the DSE Problem

For each consideration we make about a potential design decision a dependency may exist with other decisions with regards to our expectations of the finished system. For example, in Fig. 1.2 the design decision 3 requires the implementation of both 2 and 1 for it to be effective due to inherent performance dependencies.

We can easily model this idea as a directed graph with a generic unmodified design as the root node. Reverse traversal from a leaf node to the root indicates which selections are required for full implementation. This graph, Fig. 1.2, defines the *design space*. The design space is a figurative structured space where all of our design decisions and

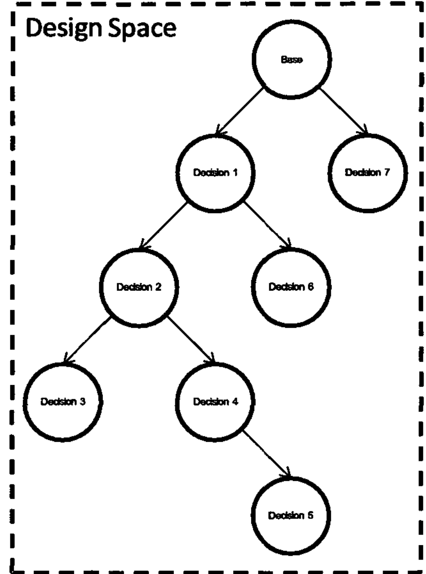


Figure 1.2: A Design Space as a Directed Graph

candidate designs exist. Here we can sort through and select which decisions will be most advantageous for our final design based on user requirements.

As a graph, we may subject this problem to known methods and techniques from the fields of graph theory and combinatorial optimization. Each node contains vital information required to make an effective decision. Such information can include: hardware size, maximum frequency, power consumption or required clock cycles.

With this information we can optimize for certain aspects of the design by selecting advantageous components. The goals of such optimizations generally include:

- Maximized Performance, *Speedup*
- Hardware Minimization
- Power Minimization
- Resource Utilization

By considering only coprocessor and internal ISE decisions in a design space we can construct a hybrid SoC. Next we define the hybrid SoC and its potential role in embedded systems.

1.2 The Hybrid SoC

A hybrid SoC is a System-On-Chip where internal processor datapath customizations are made as ISEs and coprocessors have been added as well. There are many possibilities for configuring these placement options with respect to buses, memory access, register file porting and datapath integration. Fig. 1.3 illustrates a potential hybrid system with internal and external custom instructions.

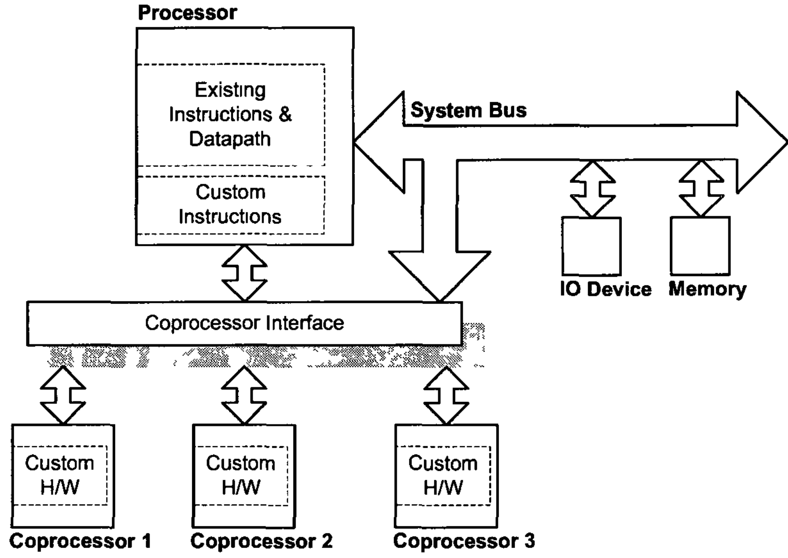


Figure 1.3: Example of a Hybrid SoC.

In this example the coprocessors are available on the system bus along with an interface to the register file. The internal augmentations are found within the processor as an extension of the base ISA as well but with custom execution hardware located within the processor's datapath. Such internal custom instructions require not only custom execution logic but custom control and decode logic.

It is clear that for these two placement options different custom instructions may be better suited to one particular implementation over another. A brief overview of these anecdotes are presented in Table 1.1 and discussed further in subsequent chapters.

Table 1.1: Considerations for Internal and External Custom Instructions.

	Advantages	**Issues**
ISE	Inherent Parallelism through Pipeline Hardware Reuse Negligible Communication Delay	Hazard Detection Clock Frequency Limitations Instruction I/O
Coprocessors	Variable Clock Frequency Direct Memory Access (DMA) Less Integrated Control Logic	Hardware Overhead Instruction I/O Communication from Processor

1.3 Motivation and Contributions

As previously stated the idea of looking at the both the ISE and coprocessor DSE avenue is quite uncommon. Typically each of these solutions is seen as a separate problem. We look to show that both techniques should be analyzed concurrently. By looking at both internal and external candidates together we can effectively model trade-offs and dependent performance gains.

The goal of this work is to maximize performance using a hybrid SoC approach for FPGA targets. Performance is measured as speedup where the greater the speedup the better the performance gain. The hybrid SoC approach is contained within a developed framework that integrates into existing tools and design flows while accepting custom instruction candidates from users.

The contributions of this framework can be summarized as:

1. Consideration of both coprocessor and internal augmentation solutions simultaneously using known metrics.

2. Thorough mathematical trade-off analysis of coprocessor versus ISE implementation decision with varying aspects for consideration.

3. Formulation of these tenets into a Integer Linear Program (ILP) where conventional ILP solvers can be used to select an optimal placement for custom instructions.

4. Integration of these ideas into existing industry hardware design flows with a developed hybrid SoC framework application.

This work has been motivated by the research goals of the Computer Architecture Research Group at the University of Ottawa. Currently, a project is underway to generate

a complete embedded SoC design toolchain which facilitates design automation while exploiting DSE of ISEs, coprocessors[1, 2], multiprocessing and vectorization through SIMD[3]. This work bridges internal ISEs and coprocessors into the global solution space.

1.4 Scope

The scope of this project is quite broad but this document attempts to contain all important concepts of the developed hybrid SoC framework. The flow of the described approach to the hybrid SoC problem is shown in Fig. 1.4. This project looks at the hybrid DSE problem from the user perspective. It is required that a user has an application or algorithm defined as source code. Furthermore, the user needs to have an idea of their target architecture *ie. RISC, VLIW* and their constraints. This work requires that a user has already obtained a set of candidate custom instructions whether manually or through many of the automated means including those addressed in this document. As a defining feature of this work, all candidates are treated equally and not are provided by the user on the basis that they are for external or internal implementation only.

For this project our user constraints are given as maximum target hardware utilization and a set of custom instruction candidates for a RISC architecture. The main objective developed in this document is the maximization of system performance given target restrictions such as hardware size and ISA limitations. The eventual realization of the performance model is presented in such a manner that it can be modified to optimize for other system parameters. See Section 10.3 for more information.

Succinctly defining the scope of this work has been a difficult task but is presented as the existing methods of identification of potential custom instructions and hybrid placement selection through a developed hybrid SoC framework application. The main focus is situated around the development and execution of an ILP model for the hybrid placement problem within the hybrid SoC framework as it is the main discussion point in this document. The scope of this document is summarized as development of a hybrid SoC framework for a given target where a user provides an application and candidate custom instructions. Validation of such a model is done using a NIOS II base architecture and existing theoretical and simulation tools.

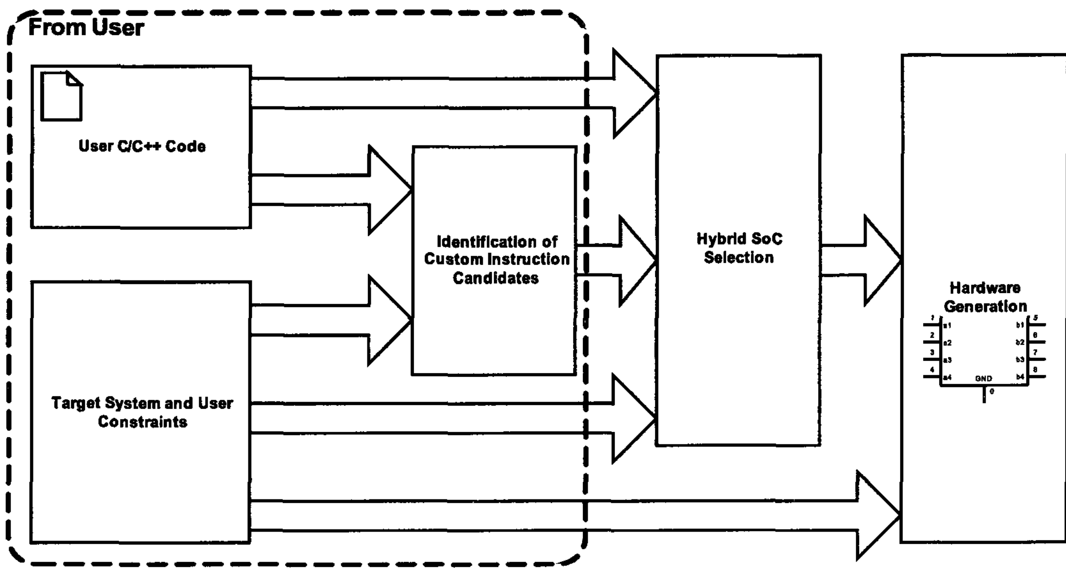


Figure 1.4: Overview of Defined Hybrid SoC Flow.

Chapter 2

Background

2.1 Introduction to Instruction Set Extensions

2.1.1 Overview of ISEs

In recent years the addition of internal ISEs in commercial embedded solutions have become more available under the analogous descriptor configurable cores [4, 5] or simply custom instructions. ISEs form the basis of Application-Specific Instruction Set Processors (ASIP) where a processor is adapted to perform a specific application. The instruction sets in these processors have been specially selected to best meet the required application specifications.

Proponents of ASIP usage have stated that it is the next logical step in microprocessor evolution as conventional processors cannot meet the strict requirements of embedded systems [6]. Counter arguments can be easily deduced including datapath instability and CISC similarity. Datapath instability can occur when new hardware is introduced disrupting existing timing constraints. Despite this, many systems have benefited greatly from the use of ISEs such as a custom FFT ASIP [7] implemented on the Tensilica Xtensa Processor. The speedup achieved in this work did not require the addition of large hardware but a simple change in the underlying ISA with new instructions.

Identifying ISEs that can form the basis of an ASIP can be a difficult task. Clearly it is possible for the user to do this manually but automated solutions are becoming ever more present in the research community.

2.1.2 ISE Identification

As previously mentioned, a user has two options in identifying ISEs that will prove beneficial to his or her application. A user may formulate these internal custom instructions either manually or automatically.

The manual perspective can be approached before a user physically begins coding where these custom instructions are integrated directly into the code or algorithmic realization based on functional complexity. This functional complexity can be defined as a function point in the application where most of the computation time is spent.

Ignoring any manual intervention from the user, we are left with the automatic identification of ISEs. We can group the automated identification into two main groups, functional and low-level. Functional identification exists at the source code level where groups of C or C++ lines can be grouped together to form a single custom instruction. Note that this will not require extensive retargeting of the compiler, assembler, linker and loader. Low-level identification exists at either an intermediate or lower assembly level after the compiler has completed initial optimizations and broken the program into data flow and control flow graphs.

Two main techniques can be applied to the automatic ISE identification problem, profiling and combinatorial optimization. Profiling can be done at both the functional and low-level. Profiling is a dynamic run-time program analysis technique to investigate a program's behavior. It can be done statically off line as well. Profiling can identify hot-spots within a program that may benefit from further attention. Hot-spots may include areas that are frequently called or computationally intensive bottlenecks. As previously mentioned at the low-level we can obtain an accurate map of the physical execution of our program as control and data flow graphs (DFG). Coupled with profile information on subsets of the graph we can utilize combinatorial optimization techniques to optimally determine advantageous node groupings or clusters that will benefit from custom instruction conversion. The selection is based and constrained on custom instruction requirements. Such a constraint could involve the production of one result based on two input values or a maximum hardware value. The goal of such a cluster selection is a maximization of a specified metric, whether it be performance, power conservation *etc.*

Many algorithms exist based on the above ideas, some of which are available as commercial products. These profiling and combinatorial optimization based approaches from research and industry are explored in the next section.

2.1.3 Automated Algorithms

Tensilica has emerged as one of the leaders in ASIP design with their Xtensa platform. Besides providing a base VLIW/RISC processor that supports ISEs Tensilica provides a series of tools to facilitate ASIP design. The XPRES Compiler (AutoTie) [8] can automatically identify ISE candidates that may benefit an end-user's system. Defined are the 4 stages of the XPRES Compiler summarized from [9]:

1. Profile application identifying performance hot-spots, loops and functions. Collect operation mix and dependency graph for each hot-spot.

2. Generate family of potential ASIPs based on application profile.

3. Evaluation of each ASIP based on hardware size and performance impact producing a Pareto based curve of the lowest cost ASIP at different performance levels.

4. Generation of each Pareto-optimal ASIP. Both configuration and physical TIE are generated.

TIE is a HDL-like syntax used by Tensilica to formulate custom instructions from the hardware and software perspective. For completeness, a Pareto-optimal outcome is one in which it is impossible to make another outcome more advantageous without making another outcome less favorable.

The tool has been shown to be quite effective for certain applications but to achieve maximum performance a programmer must optimally model their application for the compiler to provide a top-tier solution. Such steps to be taken include removal of pointers and memory access. Data dependencies related to pointers and data structures can often be approximated and consequently resolved as seen in [10]. As we progress downwards from this abstraction used by XPRES we may obtain a more complete view of our program and potentially obtain better results with a clearer view of the entire program space.

Many low-level automatic instruction-set extension algorithms work off of the manipulation of basic blocks in directed acyclic graphs (DAG) representing our program. A basic block is a graph or code segment that has only one entry and one exit point. Basic blocks are typically the smallest blocks with which compilers perform optimizations upon. Algorithms working this way follow the given structure:

1. Conversion of program into basic blocks.

2. Generation of data flow graphs as DAGs for each basic block. (Nodes=Primitive Operations, Edges=Dependencies)
3. Perform edge cuts on DAG in an effort to optimize a parameter, typically Speedup.
4. Select cuts which maximize global target parameter yielding ISE selection.

Algorithms following such a procedure include [11, 12, 13]. Cut and pruning rules differ between them. A common consideration is the notion of forbidden nodes. These are nodes that cannot be implemented in a custom instruction such as **jump**. We discuss this forbidden node issue later as a potential exploit for coprocessors by permitting load and store instructions through use of DMA.

2.2 Coprocessors

2.2.1 Introduction to Coprocessors

Coprocessors are hardware units that supplement functionality of a primary processor. Early coprocessing units were used to perform floating-point operations but have expanded to cover the requirements of graphics, signal processing and security applications amongst others. The term hardware accelerator is often used interchangeably with coprocessor.

Coprocessors began performing mathematical computations within mainframes and eventually made their way into desktop computers [14]. Currently there exists a large trend in their use within embedded systems in an effort to garner a significant performance gain. The use of coprocessors is the preferred method of custom instruction integration for the Xilinx MicroBlaze soft-core processor [15].

2.2.2 Architecture

The architectural definition of a coprocessor varies greatly. These differences may include how the external custom hardware is wrapped, how it is connected to the main processor, how it communicates with the main processor and what other peripheral connections may exist from it. We will review various implementable architectures and present the ones used for our hybrid SoC model.

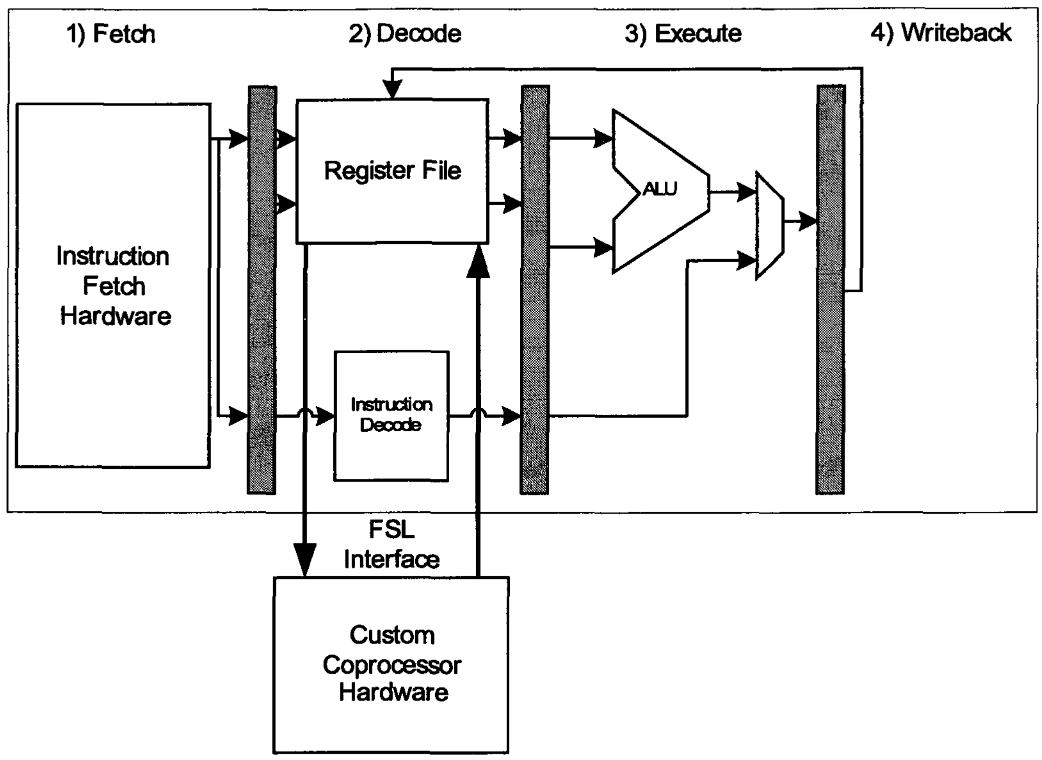


Figure 2.1: MicroBlaze FSL Coprocessor Architecture.

The MicroBlaze soft-core processor provided by Xilinx allows up to 16 coprocessors [15] to be used via Fast Simplex Link (FSL) channels [16]. These channels run directly from the processor's datapath and provide timing critical interaction between the main processor and the custom logic. The register file becomes mapped to the custom hardware through these links. Coprocessors can also be integrated into a system through the On-chip Peripheral Bus which is part of the IBM Core Connect standard. Here our coprocessors lose their direct connection to the processor as they are located on the peripheral bus but we are not so limited with the number of coprocessors we implement.

The similarity of these FSL coprocessor implementations and internal ISE custom instructions may be slightly confusing. The key distinguishing feature that gives this FSL hardware the identity of a coprocessor is that is not integrated directly into the data path. The custom hardware sits outside the decode stage negating the rest of the existing pipeline as shown in Fig. 2.1.

We have just looked at an example of an industrial approach to integration of co-

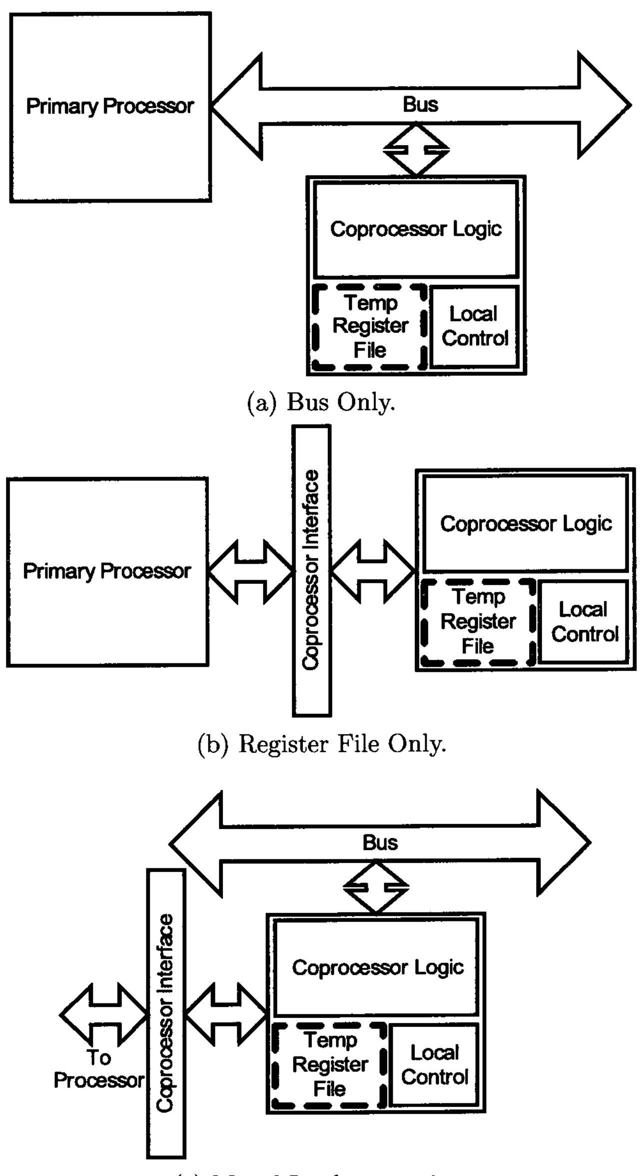
processor hardware blocks. From this we deduce three possible coprocessor integration scenarios that can be applied to abstract system models:

1. Bus Only
   Coprocessor is only connected to the system bus. In this configuration the coprocessor can be a memory-mapped peripheral. Typical interaction with the coprocessor can be accomplished through **mov** commands from the main processor. For example, consider a coprocessor mapped with address 0x80000. The primary processor will move the coprocessor instruction and operands to this address. It can read the resultant values from the bus in a similar fashion. As the coprocessor is on the system bus it may be possible for it to communicate or acquire data from other peripheral devices.

2. Register File Only
   The external coprocessor is connected through a special interface to the processor. This is the same fashion as the previous FSL example. These coprocessors are very similar to internal ISE custom instructions however they lie outside of the datapath of the processor but have access to data and variables stored within the processor. In such instances the interface between the processor and the coprocessor must be made available. Typically a limited number of *dummy* coprocessor instructions will be available within the ISA [17]. These dummy instructions are easily mapped to the custom hardware where the instructions can be inserted into a user's application program.

3. Mix of Both
   It is possible to utilize both of these configurations simultaneously. The advantage of such an implementation is the direct communication possibility with other peripherals along the bus while maintaining access to the vital internals of the main processor including the register file. The main method of communication between the primary processor and the coprocessor must be negotiated and setup into the design to avoid confusion.

Fig. 2.2 on the following page outlines the described coprocessor configurations. Each of these configurations are considered valid coprocessor setups for our hybrid SoC model. The temp register file shown to store temporary values is not required. The local control logic manages the coprocessor logic and its interface to the processor whether it be by bus or direct interface.



(a) Bus Only.

(b) Register File Only.

(c) Mixed Implementation.

Figure 2.2: Coprocessor Integration Models.

Besides the above communication methods, the physical hardware implementation of a coprocessor can also vary. At the very least custom arithmetic logic and control logic are required. The custom control logic controls the physical execution from the initiation to the return of results. Local memory units, scratch pads, register files or other components can be found within coprocessor hardware as well. In an effort to create generic IP cores this hardware can be packaged into universal wrappers similar those of the IBM Core Connect standard. Since coprocessors exist outside of the datapath they can run at a variable clock rate different from the main processor.

Because coprocessors exist in many forms, there are many different algorithms that will attempt to generate compatible coprocessor hardware from a given source code. We briefly discuss these ideas next.

2.2.3 Coprocessor Instruction Identification Algorithms

Similar to the previous automated ISE methodology and algorithms, available techniques can look into a user's source code and make coprocessor suggestions. Cascade [18] provides an interesting approach to coprocessor design where coprocessors are extracted from physical machine code; however, we are more interested in selection before assembling and linking the code, the compilation stage.

For this work we will be extending and relaxing existing ISE algorithms for coprocessors in an attempt to generate candidate custom hardware. This yields a potential design space that includes both coprocessors and internal datapath augmentations for final selection consideration. Many automatic coprocessor identification algorithms such as [19] resemble the low-level ISE algorithms previously discussed. Here the program is again broken into data flow graphs presented as directed acyclic graphs. Maximal single-output subgraphs are identified while pruning the search space with [20].

Despite the fact that ISE implementations have the potential to be implemented as coprocessors it is imperative to point out that both implementations should not be treated and constrained in the same way. For example, in [19] only single-output custom instructions are considered. This may be practical for internal ISE implementations within a processor's datapath, however, coprocessors often are not bound by such restrictions with more relaxed IO constraints.

2.3 Internal Datapath Augmentations

2.3.1 Architecture

In our hybrid configurations we have custom coprocessors outside the primary processor and custom hardware within the existing datapath. We have seen various coprocessor configurations and now look to integration within a RISC datapath.

Integration of custom instructions internally into a datapath is not a trivial task. Datapaths can be complex to modify and any modification runs the risk of breaking existing timing setups. Internal custom hardware integration when done properly can enjoy the instruction level parallelism provided by a pipelined and optimized datapath. Integration does not simply involve adding custom arithmetic logic. An outline of possible custom hardware is shown in Fig. 2.3 where internal datapath changes are made across different pipeline stages.

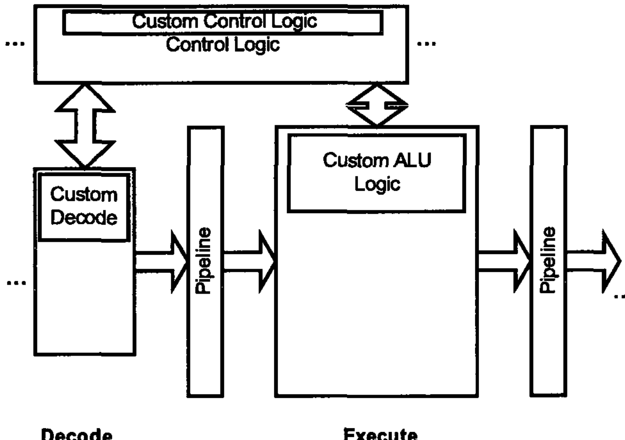


Figure 2.3: Datapath with Custom Hardware Partitions.

A key difficulty with integrated custom hardware if a pipeline progression is to be maintained is the required custom control logic. The custom control logic must deal with each pipeline stage for the instruction along with hazards. A simple less-efficient solution to internal custom instructions is to stall preceding instructions until write back.

Such logic may be provided by the commercial tools used or entered directly by the programmer. Newer programs such as CoWare Processor Designer abstract hardware functionality with LISA[21], a C-like language for ISAs, allowing for easy customization as RTL level changes can be difficult.

2.3.2 Current Methods and Considerations

It seems as though the best way to facilitate datapath integration is to provide existing custom partitions within the datapath. These dummy operations can be integrated into a complete design toolchain from compiler to physical datapath where they can be synthesized away if not used. For example, NIOS II allows for 256 custom instructions [5]. Here the 256 dummy ISA instructions can be activated and customized at will.

When looking at internal custom instructions there exist two large factors that play a part in selection; *is it possible to integrate this block and will it break the datapath?* The breaking of the datapath usually occurs as a violation of timing requirements. If the critical path of a custom hardware block is greater than the existing critical path the processor should not use the custom hardware. It is possible that trade-offs are performed relating to use of the custom block and decreasing the clock frequency of the entire processor datapath, but this corrective avenue is outside the scope of this document. Many instructions can be interfaced into the datapath but will break existing functionality and timing. Instructions that cannot be integrated violate specific constraints of our target architecture. Consider the static 32-bit instruction format presented in Fig. 2.4 from the MIPS ISA[22]. Only 32 bits are available, some of which taken by existing functionality. There exists only so much room in the existing instruction width and used instruction breadth that limitations occur on how many instructions we can implement and how many operands we can address. This problem can be side-stepped with a VLIW architecture as Tensilica does. VLIW is not a stop-gap solution as new problems arise, mainly scheduling.

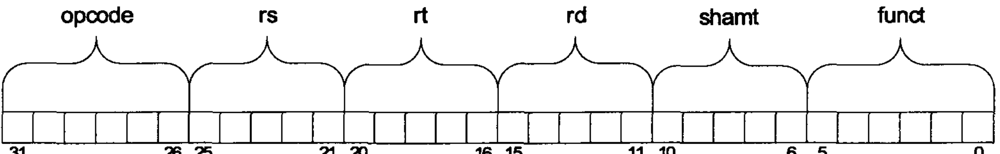


Figure 2.4: Sample MIPS Instruction R-Format.

The register file becomes another potential bottleneck if it is a conventional memory with 2 read and 1 write port. This issue is addressed, examined and solved in Chapter 6.

2.4 Combinatorial Optimization-The Integer Linear Program

2.4.1 Integer Linear Programs

Combinatorial optimization is a field of applied mathematics that pulls from the areas of combinatorics, linear programming and algorithms. The objective of such a field is to solve optimization problems that can be modeled as discrete structures [23]. Common combinatorial optimization problems include the *Traveling Salesman* and *Max Flow*. In these instances the problem is effectively modeled as a graph. Many problems such as these can be converted to linear programs (LP).

Linear programming allows mathematical models to be represented as linear equations and solved for a best outcome. A linear program will have an objective function, what we are trying to optimize, and a series of constraints. The canonical form of a linear program is shown below in Eq. 2.1 where we are trying to solve for x. Note c, x and b are vectors while A is a matrix of coefficients.

$$\begin{aligned} &\textit{maximize} && c^{\mathrm{T}}x \\ &\textit{subject to} && Ax \leq b \end{aligned} \tag{2.1}$$

Fig. 2.5 illustrates a physical representation of a linear program with two linear constraints and a non-negativity constraint. The area that forms from the intersection of these constraints is called the feasible region. All points within this region satisfy our set of constraints. A point found in this region may yield an optimal solution to the given objective function. We look to find this point.

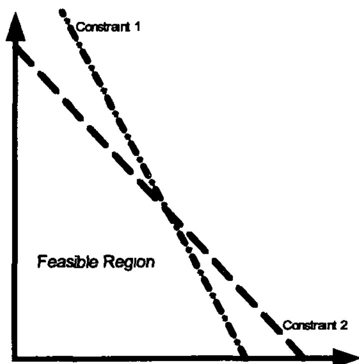


Figure 2.5: Physical Representation of an LP.

By forcing x to be an integer the problem becomes an integer linear program (ILP). The added integer constraint is shown in Eq. 2.2.

$$\begin{aligned} maximize \quad & c^{\mathrm{T}}x \\ subject\ to \quad & Ax \leq b \\ & x \in \mathbb{Z} \end{aligned} \tag{2.2}$$

Once again we can show this graphically, Fig. 2.6. The circular points indicate integer values with the added dark points as integer values within the feasible region. One of these dark points is an optimal value to the given objective function and constraint bounds.

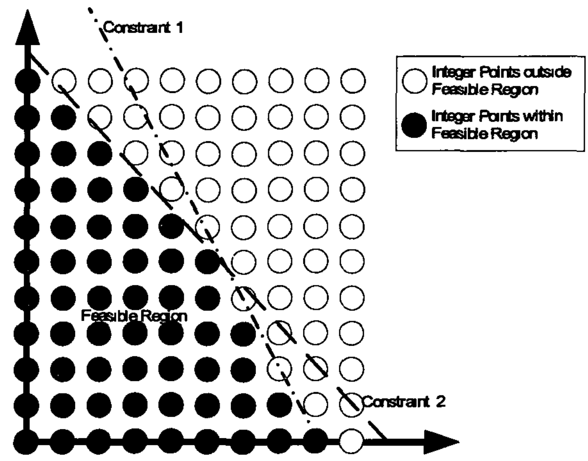


Figure 2.6: Physical Representation of an ILP.

2.4.2 Finding an Optimal Solution

There are many approaches to solving LPs [24] and their subset ILPs. A common wrapper technique in solving ILPs is branch-and-bound. As the name implies the branch-and-bound algorithm is made up two concepts, branching and bounding.

1. **Branching**
   Branching involves the division of the original problem into multiple subproblems. Solving the subproblems will yield an optimal solution to the original problem. See Fig. 2.7 for an example of an initial branching into subproblems. Each subproblem in Fig. 2.7 (b) can be broken down again into additional subproblems.

$$
\begin{aligned}
\textit{maximize} \quad & \boxed{x_1} + 2x_2 + 3x_3 \\
\textit{subject to} \quad & \boxed{x_1} + x_2 \leq 4 \\
& x_2 + x_3 \leq 4 \\
& \boxed{x_1} \leq 2 \\
& x_2 \leq 2 \\
& x_3 \leq 2
\end{aligned}
$$

(a) Sample ILP Program.

If $x_1 = 0$		If $x_1 = 1$		If $x_1 = 2$	
maximize	$2x_2 + 3x_3$	*maximize*	$1 + 2x_2 + 3x_3$	*maximize*	$2 + 2x_2 + 3x_3$
subject to	$x_2 \leq 4$	*subject to*	$x_2 \leq 3$	*subject to*	$x_2 \leq 2$
	$x_2 + x_3 \leq 4$		$x_2 + x_3 \leq 4$		$x_2 + x_3 \leq 4$
	$x_1 = 0$		$x_1 = 1$		$x_1 = 2$
	$x_2 \leq 2$		$x_2 \leq 2$		$x_2 \leq 2$
	$x_3 \leq 2$		$x_3 \leq 2$		$x_3 \leq 2$

(b) Extracting Subproblems from x_1.

Figure 2.7: Branching, Formation of Subproblems.

2. **Bounding**

   Bounding involves the placing of a bound on the optimal solution of an individual subproblem.

The effective bounding of ILPs can be accomplished while maintaining an optimality guarantee. Appropriate bounds are obtained through LP relaxation. The integer constraints of the subproblems are removed and solved strictly as LPs as they are easier to solve. Since all ILP feasible solutions are also feasible for an LP, the optimal solution of the ILP will always be equal to or worse than the optimal solution of the LP. The bounds can also be used to discard entire subproblems if a current solution is better.

The complexity of pure branch-and bound algorithms usually cannot be explicitly described as the physical number of nodes cannot be a bounded priori [25]. If resources or time are very limited, approximation algorithms may be better suited yielding a sub-optimal solution. As the subproblems have been converted into LPs they can be solved individually quite efficiently. The simplex method[24] has a worst case exponential complexity while interior point techniques[26] have a worst case polynomial complexity. The simplex method is generally more efficient excluding special LP families which create the exponential worst case [27].

If we further restrict our ILP solutions to binary values we get Eq. 2.3. These types problems are again NP-hard. NP-hard problems, non-deterministic polynomial-time hard, are a class of problem that are at least equally as hard to solve as the hardest problems in NP. In common terms, there is no efficient way to optimally solve them[28]. The binary ILP, 0-1 Integer programming or binary integer program (BIP) is a known NP complete problem [29] that is a subset of ILP. We will explore BIP modeling uses as we examine the Knapsack Problem in Chapter 4 which will form the basis of our hybrid SoC selection model.

$$\begin{aligned} maximize \quad & c^{\mathrm{T}} x \\ subject\ to \quad & Ax \leq b \\ & x \in \{0,1\} \end{aligned} \tag{2.3}$$

There is often a stigma associated with NP-hard problems which currently favor approximation based algorithms. Such approximation techniques can employ the use of heuristics, genetic algorithms *etc.* NP-hard does not mean that a problem is unsolvable, it means that it is difficult to solve. Genetic algorithms and approximations have been extensively studied and can achieve very good results in different applications; however, they cannot make a guarantee that the solution is optimal. Exact algorithms may not be quick or efficient, but some offer incremental improvement where sub-optimal solutions are presented as the algorithm progresses. These can be used to form a comparison basis with approximate algorithms when a sub-optimal solution is the only feasible option.

An easy solution evangelized in this document is this joint approach. As branch-and-bound executes, its results get progressively better until the optimal value is reached, unless the problem is infeasible. As a result we will never obtain a solution worse than the one that that was previously found. If the solver has taken an unreasonable amount of time a user may specify a time-out for the branch-and-bound algorithm where the current sub-optimal value is taken from the solver and compared to the value from a quick-and-dirty approximation. The best of the two is taken. No best selection guarantees can be made in this case. The solution may be good enough given a system's performance needs.

Chapter 3

Prior Art

3.1 Overview

This chapter provides a brief overview of current research work that has lead to this thesis. Similar efforts are discussed along with revisiting separate and independent techniques. Finally, we will look at the recent introduction of the hybrid custom instruction environment, the hybrid SoC.

3.2 Shared Ideas

The use of ILPs in modeling design space exploration problems can be found everywhere. Such models have found their way into many aspects of hardware-software partitioning including custom instruction identification algorithms to effective scheduling across targets with multiple processors. The way to solve these models differs greatly between researchers and their individual problems. Some prefer exact algorithms guaranteeing the best possible solution for the end user while others take a more quick and direct approach giving a sub-optimal result in a short time frame. Both approaches have their merits and for this reason, the execution of our hybrid model depends on both.

It is more difficult to find models that incorporate multiple DSE facets. Our hybrid SoC model falls into this category as we are attempting to mutually optimize the use of coprocessors and internal custom instructions within the datapath.

A common ILP modeling effort involves the attempted exploitation of multiprocessing System on Chips (MPSoC). We look at the simple case first followed by other research work that incorporates the identification of internal custom instructions along with multiprocessing.

3.2.1 Similarities with Multiprocessing System on Chip Modeling Efforts

An ILP formulation for architectural synthesis and application mapping targeting FPGA based MPSoCs is presented in [30]. Here the ILP based model aids designers in identifying a resultant architectural design, binding schema and scheduling algorithm. The features of the work are defined as:

1. Automatic system generation.
2. Global mapping problem, optimal solution can be found.
3. Different communication path support: P2P, bus.
4. Fine grained processing model to facilitate scheduling.

Each of these features have been included in their mathematical model. The model attempts to minimize hardware size while meeting certain constraints such as performance while selecting the type of network connection to use between processors. DMA is not modeled as all data communication goes through other processing elements. It is imperative to note that their model effectively captures the MPSoC problem; however the model is so complex that for even their simple test program of 7 independent tasks still required 443s to find an optimal configuration. Much work is still required to facilitate results from actual programs such as compiler integration and further study of larger test cases.

Further MPSoC work has been completed involving the use of internal custom instructions forming a heterogeneous MPSoC [31]. Here multiple extensible processors are automatically generated given a system hardware requirement and performance metric. After dividing the application into tasks each task is verified from a performance perspective and custom instructions are added or not.

Both of these approaches seem to be a common direction with respect to design space exploration within many domains. We are more interested in returning to the

individual processor in the hopes that the model described in this document can later be added to MPSoCs. We are attempting a similar feat but on the scale of one processor with different directions. The heterogeneous MPSoC is a similar problem where there are multiple considerations: the number of processors and where to assign the custom instructions.

3.2.2 Separate Problems-Independent Directions

DSE is often studied as independent problems for which to solve. An example of this is the consideration of only coprocessor or internal augmentations, not both. Similar and related directions should be modeled together to maximize our desired outcome. Coprocessors and internal custom instructions have been thoroughly studied but from a separate scope. This section briefly discusses these separate directions.

Various automated ISE and profile based custom instruction algorithms including [11, 12, 13] have been outlined in Chapter 2. A similar ISE based approach was applied to coprocessor instruction selection[19] with promising results.

Here we look at a knapsack-based model used for ISE, [32]. In this paper, candidate templates are generated for custom instructions then they are ranked and given a priority. Templates are grouped together based on priority and are verified. Finally these template combinations are selected for integration as custom instructions using the branch-and-bound technique with a custom cost function forming a knapsack like problem with area constraints.

3.3 Beginning of the Hybrid Custom Instruction Configuration

To the best of our knowledge the first verifiable automated venture into a hybrid custom instruction model was presented in [33, 34]. The papers evangelize the benefits of hybrid custom instruction architectures while demonstrating that this DSE avenue can be beneficial for many applications. Presented is a methodology for achieving the automatic generation of internal and external custom instructions for the Xtensa platform.

The key criteria are different between the work described in [33, 34] and this document. They include:

1. Division of program into tasks forming the base unit of manipulation and analysis. A hierarchical task graph is generated from the application with simple and compound nodes.

2. Distinct requirement of coarse-grained tasks for coprocessor (compound nodes) and fine-grained tasks for internal datapath custom instructions (simple nodes).

3. Multi-objective evolutionary algorithm used to solve selection model.

Despite the differences in the underlying model these papers were able to effectively demonstrate the benefit of a hybrid SoC using a task level model. Our work tackles this problem from a different perspective; *we have a candidate instruction from some means, what should we do with it?* The difference supported in this thesis is that a user presents a list of candidate custom instructions obtained from whatever means they chose. All candidate hardware designs are treated on equal ground. All candidates are considered as having both an internal and external implementation potential all the while letting the solver decide which placement options will yield the greatest speedup. The author's approach already has a preconceived notion of where the custom instructions are to go. This is understandable as that is what they were selected for during the analysis of the program's task hierarchy. Taking this direction forces the user to accept that custom instructions can only come in the form of tasks or a hierarchy of tasks. This voids all other current custom instruction identification methods. Our solution takes a step back and gives control to the user by allowing them to define where the custom instructions are coming from and how they may benefit the end system.

It is important to point out that this work focused on the Xtensa platform which is very forgiving with respect to the integration of both internal and external custom instructions regarding data access, clock frequency and hazards. This assumed functionality is not present in other systems and may require further investigation.

Chapter 4

Introduction to the Knapsack Problem

4.1 The Generic Knapsack Problem

The simplest form of any decision can be modeled in a binary fashion, *yes* or *no*. The knapsack problem deals with a series of alternatives and whether they are selected given certain restrictions. For example, consider a shipping container that can hold a maximum weight of 1000kg. 10 packages are available each with a weight and sales value. The question the knapsack problem models involves how we can best select packages to ship in an effort to maximize the potential sales value, profit, while adhering to the container weight restriction. This example is shown in Fig. 4.1. Since the problem is small the solution can be derived visually. The problem becomes impractical to solve manually with a large number of alternatives and constraints.

The alternative selection is considered a binary decision; it is or is not selected. As discussed before, the BIP effectively models this decision type and is encompassed within the general form of the knapsack problem, Eq. 4.1. Here, we have a capacity value c with a set of n alternatives j with matching profit p_j and weight w_j.

$$\begin{array}{ll} maximize & \sum_{j=1}^{n} p_j x_j \\ subject\ to & \sum_{j=1}^{n} w_j x_j \leq c \\ & x_j \in \{0,1\} \end{array} \tag{4.1}$$

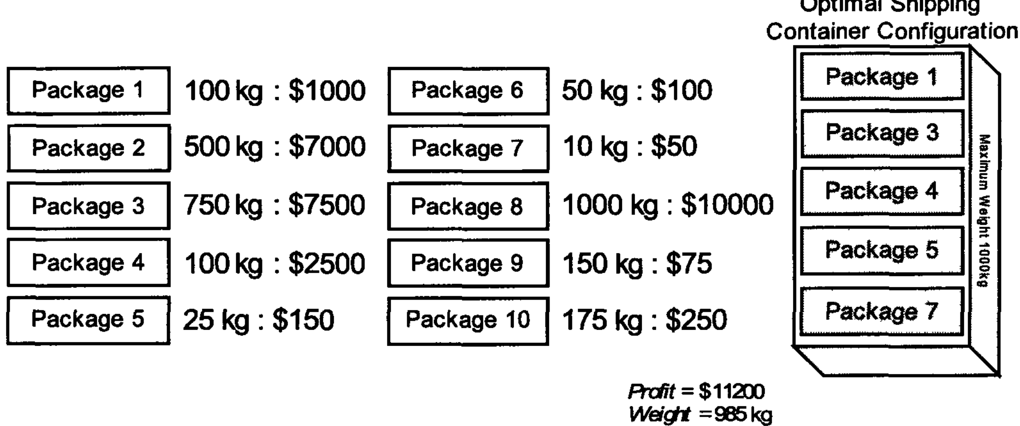


Figure 4.1: Knapsack Motivation.

Essentially, solving the model yields a vector of 1's and 0's indicating which alternatives are to be included in the optimal solution. This knapsack idea forms the basis of our hybrid selection where we attempt to maximize system performance given a target platform and ISA constraints. Coprocessor and internal custom instructions will either be included or not in the hopes of maximizing performance.

The generic knapsack problem is quite simple and may not meet the needs of more complicated problems. Consider the previous packing example but with a volume and weight constraint on the shipping container. The current model does not allow consideration for this dimensional relationship. Next we look at variations of the knapsack problem better suited to our situation.

4.2 Knapsack Variants

There exist many direct generalizations formulated from the general knapsack problem discussed previously. A short-list of these can be found in Table 4.1.

Of particular interest to us is the multiple knapsack problem where we have n items and m knapsacks. We can use the coprocessor design space and the internal custom instruction design spaces as individual knapsacks for the hybrid SoC problem. Each knapsack can have a different capacity or set of requirements. The general form of the multiple knapsack problem is defined as Eq. 4.2.

Table 4.1: Generalizations of the Knapsack Problem.

Problem Name	Description
Bounded Knapsack Problem	Each item or alternative can be chosen a finite number of times.
Unbounded Knapsack Problem	Each item or alternative can be chosen an infinite number of times. No bound is placed on how many times the same item can be selected.
Multidimensional Knapsack Problem	Problem with a collection of different resource constraints or one constraint with multiple attributes.
Multiple Knapsack Problem	Existence of multiple knapsacks with different capacities for selection purposes.
Multiple-Choice Knapsack Problem	Alternatives are divided into classes where an item from each class must be selected.
Quadratic Knapsack Problem	Profit from choosing items is dependent on the selection of other items.
Multi-objective Knapsack Problem	Multiple objective functions are included. Multiple features can be optimized for.

$$\begin{aligned} & \textit{maximize} && \textstyle\sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} \\ & \textit{subject to} && \textstyle\sum_{j=1}^{n} w_j x_{ij} \leq c_i, \quad i = 1, ..., m, \\ & && \textstyle\sum_{i=1}^{m} x_{ij} \leq 1, \quad j = 1, ..., n, \\ & && x_{ij} \in \{0, 1\}, \quad i = 1, ..., m, \quad j = 1, ..., n \end{aligned} \tag{4.2}$$

4.3 Solving the Multiple Knapsack Problem

As previously stated, the general 0-1 knapsack problem is a known NP-hard problem. Unfortunately the multiple knapsack problem continues in this fashion. Several branch-and-bound based exact algorithms address this problem. Widely used algorithms include MTM[35] and Mulknap[36]. The latter is better suited towards problems with large

numbers of items to be placed in each knapsack while the former is better suited for many knapsacks but few items. Each algorithm performs very differently in the establishment of a lower bound while both exploit the surrogate relaxed problem for the upper bound. The relaxed surrogate problem stems from a relaxation of side-constraints yielding a basic knapsack problem for which to establish an upper bound. Each of these exact algorithms is available within our solver used for evaluation.

It is important to note that a fully polynomial-time approximation scheme (FPTAS) does not exist for the multiple knapsack problem when we use 2 knapsacks [25]. A fully polynomial-time approximation scheme requires an algorithm to be polynomial with respect to the problem size and $\frac{1}{\epsilon}$. ϵ is a value indicting the factor at which the solution is of being optimal. Despite this, hope is not lost as a more elementary polynomial-time approximation (PTAS) scheme is available.

A relatively recent accomplishment in the field of combinatorial optimization has been the development of a PTAS for all multiple knapsack problems [37]. Here, a logarithmic number of profits and weights are required with a guessing strategy and extra knapsacks which are removed to give near optimal solutions.

Chapter 5

Knapsack Extensions for Hybrid Selection

5.1 Overview

In this chapter we develop the linear programming model to effectively capture the essence of a hybrid SoC. As the chapter title indicates, the model is an extension of the multiple knapsack problem discussed previously. It is here where we discuss the different tenets of internal and external custom instructions and place them into a stringent physical form. Note that different scenarios and considerations are included here but the model is not exhaustive due to the differing architectures and implementations possible. A formal priori, requirements set, algorithm and output will be defined in a subsequent chapter, Chapter 7.

5.2 Selection Considerations

To decide how a custom instruction should be implemented, if at all, information must be gathered statically from the compiler/profiler and hardware generator. The compiler or profiler creates a list of potential custom instructions as either DFGs or high-level code. A hardware generator maps these custom instructions into hardware blocks. The generator may create multiple hardware candidates for each custom instruction. For this work we

assume a one-to-one hardware mapping is taking place, therefore hardware optimizations such as additional pipelining are not present. Details on effective generation of reusable hardware can be found in [38].

The ultimate goal, our objective function, is a performance gain given the constraints of a target system. This performance is measured by how many clock cycles we will save by implementing selected custom instructions both internally and externally. Amongst other information, each candidate custom instruction will provide a *cycles saved* (CS) metric. It is this CS value that we are looking to maximize.

It is imperative to demonstrate from where we are obtaining our metrics used in the selection process. The required metrics are shown in Table 5.1 along with their sources. These sources are analyzed more closely in Chapter 7.

Table 5.1: Prerequisite Data

Information	**Accuracy**	**Source**
Cycles Saved	Estimate	Compiler/Profiler
I/O for Instruction	True	Compiler
Hardware Size	True	Hardware Generator
Cycles Req'd	True	Hardware Generator
Instruction Mix	True	Compiler/Profiler
Critical Path Latency	True	Hardware Generator

The cycles saved metric CS that we are obtaining from the compiler is an estimate in a sense that we are unsure of potential hazards and communication delays. These issues are heuristically addressed and added to the model. We will treat this estimate as a near true value. The IO metric indicates how many input and output values the custom instruction requires. Depending on the target architecture this can be a very limiting factor. Instruction width and the register file will dictate this limitation. The hardware size is easily obtained from a hardware generator. Each custom instruction candidate will have a matching hardware implementation. The differences between coprocessors and datapath augmentations will be accounted for in the model. This hardware size metric only includes the arithmetic logic portion. Control logic and overhead are not included yet. For each custom instruction we must know how many clock cycles are required for execution. Again this value can be obtained from the hardware generator. The instruction mix will provide an overview how often custom blocks execute, essentially profiling information. Finally, for each custom instruction we must know the maximum

frequency for which it can operate correctly within. This critical path latency metric can be obtained from the hardware generator. Note that it is out of the scope of this work to analyze the trade-offs between implementing a slower instruction within a processor's datapath and lowering the global clock frequency of the processor in an attempt to gain performance. Instructions requiring a lower clock can be clocked variably outside the processor as coprocessors.

5.3 ALU Datapath Extension-Variable Cycle

More often than not, a target SoC's processor will allow for ALU instructions of different required clock cycles, multi cycle implementations of ALU hardware. This simplifies the overall problem by not lowering another aspect of the program's performance when implemented.

Put succinctly, if the critical path latency of an internal custom instruction hardware block is less than or equal to the existing ALU critical path latency and it meets the IO constraints of the register file, the component can be inserted into the datapath without lowering the processor's clock frequency. Knowing the total required clock cycles to execute a program T, instructions saved if custom instruction is used, CS, the speedup calculation simplifies:

$$Speedup \approx \frac{T}{T - CS} \tag{5.1}$$

If $Speedup > 1$ then the program may benefit from such an implementation. It is clear that any custom instruction that has $CS > 1$ may offer a speedup opportunity. It follows that if $CS \geq 1$ then $Speedup > 1$.

To summarize, if the latency of a custom instruction candidate is lower than the processor's latency, IO constraints are respected and a performance gain can be achieved then it is considered for internal datapath selection.

5.4 Coprocessor versus ALU

Coprocessors and internal datapath augmentation based custom instructions are two different entities with different requirements and different potential exploits. Coprocessors are typically used in DSP, encryption, graphics and floating point applications. A common theme with the first three examples is the streaming nature of the operations;

however, all of them involve complex operations. Streaming is addressed in Chapter 10 by suggesting the modification of existing custom instruction selection techniques for DMA. The exploitation of the application differences are often tackled with similarities omitted. Both implementations manifest themselves as a custom operation or instruction and should be treated as similar entities each with a defined scope of functionality. As the coprocessor is a separate unit, the operations may take advantage of an increased clock speed or larger hardware foot print but will be penalized with a communication delay. In the current model used for this work, when a coprocessor is executing the processor stalls until a result is returned. Here we look at custom instructions that have been selected not based upon where they would integrate best but as general candidates. Hybrid agnostic custom instruction candidates are discovered either through a manual or automated algorithm and are then matched to the target architecture.

Since the requirements for integration within the ALU are the most strict while coprocessors are external and can be more liberal with their hardware and timing constraints it is clear that some implementations may fall into both categories. Due to the generality of the coprocessor, any custom instruction should be implementable within if it offers a performance gain. It may not be the most effective solution but it is a candidate for consideration. Working from this generality that any custom instruction can be a coprocessor block, an instruction-set extension within the ALU is a custom instruction, therefore it too can be implemented as a coprocessor. Theorem 1 formalizes this fact.

Theorem 1. *Let C' be the existence of a matching coprocessor implementation where datapath area $\leq$ coprocessor area. For any custom instruction, if an implementation exists as a datapath augmentation (ALU instruction) D, then*

$$D \implies C'.$$

The existence of an ALU implementable instruction-set extension implies that a matching coprocessor instruction exists, however, the converse is not guaranteed to be true. Note that one implementation may be more biased to one category due to factors such as the ability to exploit datapath parallelism, limited communication delay or IO requirements. These considerations are described in the upcoming mathematical model.

Separately, coprocessor and internal implementations will have an effect on the metrics we are using. This is clear from the hardware perspective. Each coprocessor may require additional wrapper hardware to interface with the processor in a standardized

fashion. Individual custom instruction hardware values should be modified accordingly (Eq. 5.2) with w as our final hardware cost related to occupied area for the custom instruction, w_{synth} as our cost derived from our hardware synthesis tool and w_{static} as our static wrapper cost for external implementations.

$$w = \begin{cases} w_{synth} & \text{if ALU,} \\ w_{synth} + w_{static} & \text{if Coprocessor.} \end{cases} \tag{5.2}$$

Next we look to modifying the CS metric to better reflect probable events and delays that can occur while using custom instructions. By doing this we are introducing an affinity aspect to the model while retaining solution optimality based on acquired metrics.

5.5 Quantifying Additional Considerations

The obtained cycles saved CS metric is a direct value which describes in a perfect world how many cycles our program will save if a hardware implementation for a custom instruction is used. The value is based on a static or runtime analysis and a set of static assumptions on a program. Consider the scenario in Fig. 5.1 of a program that executes a shift on a loaded value followed by both a multiplication and an addition to the shifted datum. These three operations take 120 cycles to execute. By creating a custom instruction and corresponding hardware module we can bring the operation down to 50 cycles.

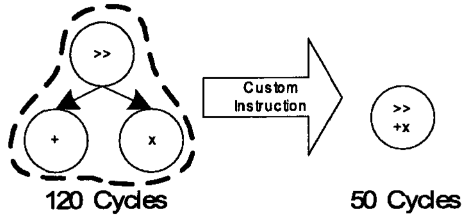


Figure 5.1: Cycles Saved From Custom Instruction Implementation.

It is not practical to go through all this customization work to save only 70 clock cycles which is not actually reflective of the actual savings. Through static program analysis we can see how many times this particular custom instruction will be used. If this particular instruction will be called 1000 times we garner a cycle savings of 70000

cycles. The number of times a custom instruction will be used is available from ISE identification algorithms and profile based custom instruction tools. Our initial cycles saved metric therefore becomes the product of the individual implementation savings and the estimate of execution frequency based on static analysis from an available tool.

The individual CS is available from synthesized hardware information of a custom instruction. The C function below has been extracted from a larger C program. This function is called 150 times.

```
void superOp (int a, int b)
{
  int c,d;
  c = a + b;
  d = c * b;
  printf ("%d", d);
}
```

The two lines shown in red have been selected as a potential custom instruction. Our compiler has converted this into the following assembly:

```
lw $t1, a
lw $t2, b
add $t1, $t1, $t2
mul $t2, $t1, $t2
sw $t2, d
```

The corresponding DFG is shown next in Fig. 5.2. Currently this graph segment takes 120 cycles to execute since the current design does not contain a multiplier. By converting this into a custom instruction (`custop`) and creating a new hardware block to execute it our assembly syntax becomes:

```
lw $t1, a
lw $t2, b
custop $t1, $t1, $t2
sw $t1, d
```

It turns out that Fig. 5.2 implemented directly as hardware will execute in 27 clock cycles. Accounting from the fact that this custom instruction will be used 150 times, we have saved 13950 clock cycles.

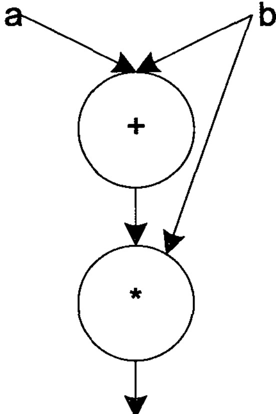


Figure 5.2: Extracted DFG of Candidate.

Unfortunately it is not as clear-cut as this since other factors may influence how many clock cycles we will actually be saving. Here we will look at factors present in either a coprocessor implementation or internal datapath augmentation that will add variability to our cycles saved metric, CS, which is not seen in previous work.

When considering a modern RISC processor one must take into account the use of a pipelined design. Pipelining creates understandable and unavoidable obstacles when adding additional logic to the existing datapath. Conditional jumps or long branching are not permitted within the scope of our custom instruction candidates so we need not worry about hazards of this type, control hazards. On the other hand, data hazards can easily occur and may be addressed from two perspectives:

1. *Modify Compiler* Introduction of stalls at compile time to ensure data is written back before the next instruction.

2. *Modify Hardware* Use of a forwarding hardware unit in the custom hardware block along with custom control logic to trigger redirection to forwarding signal when needed.

The simple solution and the solution advocated in this document to ensure brevity is that of the introduction of stalls (NOP). The introduction of forwarding for custom instructions is a complex task but can be achieved.

If a forwarding mechanism has been introduced to return either a single or multiple values then the cycles saved metric CS will remain unchanged assuming no stalls after

loads or stores. In the more common case of the stall introduction we must look at the number of stages in the pipeline. This concept is shown in Fig. 5.3 with a 5 stage MIPS pipeline. In this architecture the *Instruction Decode* stage is the first point in the

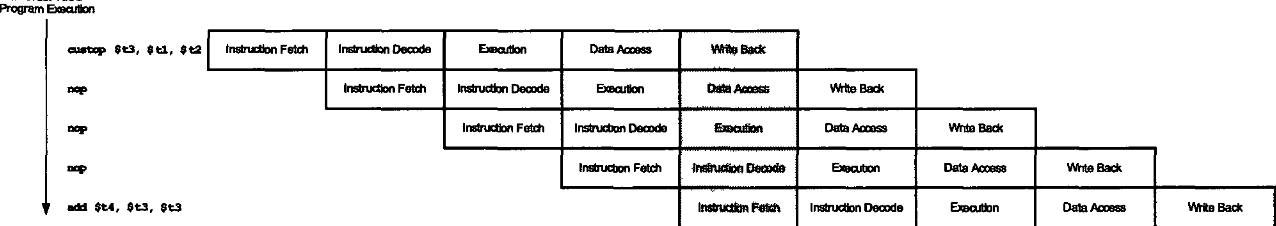


Figure 5.3: Internal Custom Instruction Execution with Accompanying Stalls.

program where a value is extracted from the register file. Since this is the case, another valid tailing instruction cannot reach this point until the custom instruction has written its data back to the register file. The NOPs have been introduced to keep a buffer of three stages so the data from the custom instruction can be written back in time to prevent a data hazard. Program execution can continue normally after the custom operation has left the *Write Back* stage as shown with the grayed column.

Assuming that the second pipeline stage will always be the first instance of register file interaction and the last stage will return a value to the register file as a write back we can calculate how many cycles will be wasted leading to a more accurate CS.

Let L be the number of pipeline stages in the target architecture with a corresponding number of clock cycles for each stage, P_i. Note that the ALU latency in the execution stage usually varies by operation. The CS_{old} is equal to the execution time with no stalls within the execution stage for our custom instruction candidates. The forwarding and stall cases are formalized in Eq. 5.3.

$$CS_{new} = \begin{cases} CS_{old} - \sum_{i=1}^{L} P_i & \text{if stalls are used,} \\ CS_{old} & \text{if forwarding is used.} \end{cases} \tag{5.3}$$

Again, not all candidates can be implemented within the datapath which leaves co-processors and the correction of their cycles saved metric. Depending on the coprocessor architecture a large communication delay may be present.

If a coprocessor interface is available within the instruction decode pipeline stage similar to the MicroBlaze implementation [15] then the write back and completion will

occur within the same point in datapath execution. This means that stalls do not in fact need to be introduced. The pipeline timing for an in-order RISC execution with such an interface is shown in Fig. 5.4. The *Instruction Decode* pipeline stage has been

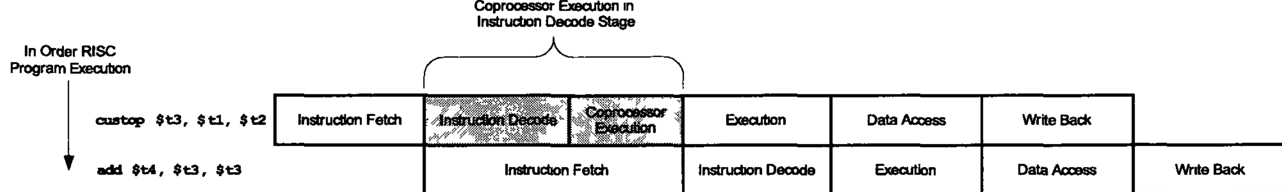


Figure 5.4: Timing of Coprocessor Implementation using Coprocessor Interface.

extended to include the execution of the coprocessor. The coprocessor execution includes a complete run involving pulling data from the register file and returning it.

Since this execution occurs within one stage of the pipeline the only delay incurred involves the cycles required to transfer the data through the coprocessor interface. This communication delay can be measured in cycles and is represented as $D_{interface}$. Therefore CS should be modified accordingly, Eq. 5.4, if a coprocessor interface is used. $D_{interface}$ should be a static value based on instruction width or data required for the custom instruction derived from the physical architecture.

$$CS_{new} = CS_{old} - D_{interface} \tag{5.4}$$

If a bus implementation is used, a similar delay is introduced D_{bus} Here the custom instruction wrapped as a coprocessor will begin in the data access stage where the data is transferred. Fig. 5.5 illustrates where the coprocessor execution takes place in a bus-based implementation.

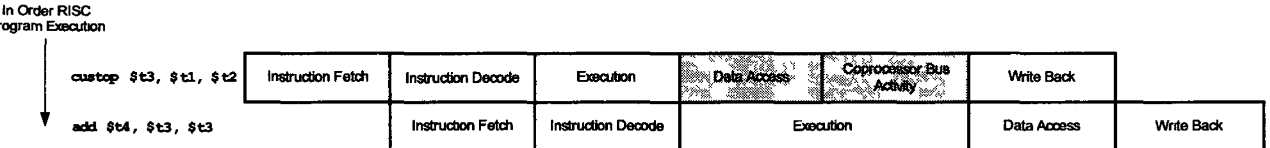


Figure 5.5: Execution with Bus-Based Coprocessor.

If a bus-based write back mechanism is not present the coprocessor may store the data in local memory where it is loaded into the register file in the next instruction.

Here we propose that a signaling mechanism, such as an interrupt, be provided to inform the primary processor when coprocessor execution is complete so that the program may continue.

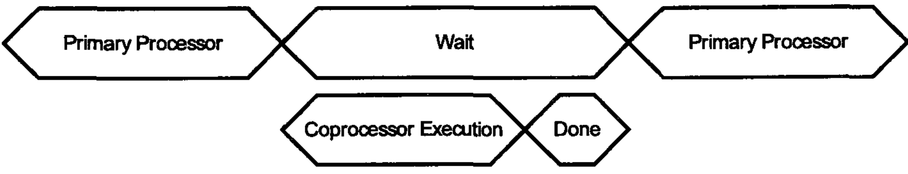


Figure 5.6: Coprocessor Signaling Task Completion.

We must ensure that all dependent data has been written to the register file before the custom instruction reaches the instruction decode stage if custom forwarding is unavailable. Here we can introduce stalls to the pipeline before and after the custom coprocessor instruction to prevent data hazards before the processor goes into a wait mode for coprocessor execution. We can ensure that all pipeline registers are empty and that we have a known delay for bus usage D_{bus} and the write back mechanism D_{write}. Formulation of a new cycles saved metric for bus interaction is shown in CS, Eq. 5.5.

$$CS_{new} = CS_{old} - (2\sum_{i=1}^{L} P_i + D_{bus} + D_{write}) \tag{5.5}$$

If other devices are permitted shared bus usage the problem gains an aspect of probability due to the possibility of contention. We assume the cost of such a delay to be negligible in this application as no simultaneous usage devices are considered. It is clear though if a large number of peripherals, controllers, processors *etc.* are operating simultaneously on a shared bus a severe penalty will be incurred. To calculate this an accurate transactional probability model should be developed.

So far we have looked at special cases where our cycles saved has actually decreased but that is not always the case. Since the coprocessor may be clocked at a different frequency than that of the main processor we can normalize the metric based on the new frequency assuming the hardware to do this is available.

Given a coprocessor hardware candidate we can easily obtain the maximum frequency from any synthesis tool. This frequency can then be applied to the coprocessor device. To normalize the CS metric a simple ratio is used as shown in Eq. 5.6 with *cycles* indicating how many unnormalized cycles the custom instruction currently takes. $f_{original}$ indicates

the original frequency.

$$\frac{\left(\frac{cycles}{f_{original}}\right)}{\left(\frac{cycles}{f_{new}}\right)} \tag{5.6}$$

Normalizing the value can be simplified to Eq. 5.7.

$$cycles_{new} = \left\lceil \frac{(cycles_{old})\,(f_{original})}{f_{new}} \right\rceil \tag{5.7}$$

We now have an accurate value relative to the clock frequency of our primary processor. The cycles saved metric CS can be modified accordingly knowing how many times the custom instruction will be executed.

5.6 The Hybrid SoC Model

5.6.1 Separate and Known Hardware Constraints

Consider the situation where both the external coprocessor and ALU hardware area availability are known separately. This is a simple problem case where we have a static limit for each option. We can solve this problem with ease as it devolves into a multiple knapsack problem with added constraints. The variation involves some items being barred from a specific knapsack or biased towards another.

The linear programming model is shown in Eq. 5.8 where p_{1j} is the cycles saved CS of an internal implementation while p_{2j} is the cycles saved of an external coprocessor. If an internal equivalent does not exist then $p_{1j} = 0$. In this scenario we are constrained with both available internal hardware area W_1 and external available hardware area W_2. Each x_{ij} represents a binary variable that will indicate whether a matching component is to be implemented, 1 or not 0, along with a corresponding custom hardware cost w_{ij}.

$$\begin{array}{ll} maximize & \sum_{i=1}^{2} \sum_{j=1}^{n} p_{ij} x_{ij} \\ \\ subject\ to & \sum_{j=1}^{n} w_{1j} x_{1j} \leq W_1 \\ & \sum_{j=1}^{n} w_{2j} x_{2j} \leq W_2 \\ & \text{Non-Duplication Constraints} \\ & x_{ij} \in (0,1) \end{array} \tag{5.8}$$

No custom instruction can be implemented both within the processor datapath and externally as a coprocessor, therefore the model must be constrained to omit duplicate implementations. These non-duplication constraints are described in Definition 1.

Definition 1. ***Non-Duplication Constraints***
If an equivalent implementation exists both internally a, and externally b they cannot both be selected.

$$x_{1a} + x_{1b} \leq 1$$

5.6.2 Known Global Constraints-General Form

When utilizing targets that are slightly removed from the user such as FPGAs, the physical on-chip implementation can remain a mystery due to *place and route*. It is advantageous to look at these targets as having a global area that can support both coprocessors and internal datapath extensions since the design tools will effectively place and connect them on the chip. The question is now how to best balance the area between both options.

The problem can be modeled in a similar fashion to the first by removing both area constraints and creating a single global constraint using a total available hardware constant GW. Eq. 5.9 models this problem with one global hardware constraint. Note that we must ensure that duplicate assignments do not occur therefore the non-duplication constraints remain.

$$\begin{array}{ll} \textit{maximize} & \sum_{i=1}^{2} \sum_{j=1}^{n} p_{ij} x_{ij} \\ \textit{subject to} & \sum_{i=1}^{2} \sum_{j=1}^{n} w_{ij} x_{ij} \leq GW \\ & \text{Non-Duplication Constraints} \\ & x_{ij} \in (0,1) \end{array} \tag{5.9}$$

5.6.3 ISA Limitations

The more we bind the problem with upper limits the less permutations will exist for the solver allowing for a quicker but still optimal solution.

Many if not all RISC processors are limited by the number of bits available, opcodes, to represent different instructions. Large instruction sets may leave very few unused

opcodes available for later use. The NIOS II soft-core processor from Altera has a custom hardware designated instruction that has its own customizable opcode field which allows up to 256 custom instruction selections. Despite this there are still limits to the support. Tensilica has worked around this problem by changing the underlying architecture with a VLIW-like overlay, but again it is safe to assume that limitations exist. Tensilica only allows a maximum of 8 coprocessors. Despite this, most vendors have no custom instruction support at all or a low and finite number of available slots.

Two constraints have been added (Eq. 5.10) to create an upper bound on the number of external S_2 and internal S_1 available instruction slots.

$$\begin{array}{ll} \textit{maximize} & \sum_{i=1}^{2}\sum_{j=1}^{n} p_{ij}x_{ij} \\ \textit{subject to} & \sum_{i=1}^{2}\sum_{j=1}^{n} w_{ij}x_{ij} \leq GW \\ & \sum_{j=1}^{n} x_{1j} \leq S_1 \\ & \sum_{j=1}^{n} x_{2j} \leq S_2 \\ & \text{Non-Duplication Constraints} \\ & x_{ij} \in (0,1) \end{array} \tag{5.10}$$

The total available slots is $S_1 + S_2 = S$. If the ISA does not differentiate between coprocessor and internal instructions the constraints can be merged as a single global constraint, with S upper-bound. Eq. 5.11 finalizes this concept.

$$\begin{array}{ll} \textit{maximize} & \sum_{i=1}^{2}\sum_{j=1}^{n} p_{ij}x_{ij} \\ \textit{subject to} & \sum_{i=1}^{2}\sum_{j=1}^{n} w_{ij}x_{ij} \leq GW \\ & \sum_{j=1}^{n} x_{1j} + x_{2j} \leq S \\ & \text{Non-Duplication Constraints} \\ & x_{ij} \in (0,1) \end{array} \tag{5.11}$$

5.6.4 Summary

So far the cycles saved CS metric for common architectures and situations has been introduced. Furthermore, we have introduced different target scenarios as knapsack problem variants. Here these ideas and flow are recapped in Table 5.2. Refer to the previous pages for supporting models defined mathematically.

Table 5.2: Summary of Model Creation.

Initial Decision Binning	Metric Formulation	Model Generation
Prerequisites: Critical Path Latency, Hardware Size, IO After gathering custom instruction candidates and running them through a hardware synthesizer we obtain a custom hardware block with a given size and latency. This information is used here in the initial binning of instructions that can be integrated as internal augmentations or both coprocessors as well. Instructions that do not offer a speedup or exceed our IO constraints are removed from contention. Candidate instructions are sorted into two bins, *Coprocessor* or *Coprocessor and Datapath*. 1. Coprocessor- Candidates are entered into this bin if they do not meet the critical path latency requirements of the processor's clock or their hardware size is too large for an internal implementation. 2. Coprocessor & Datapath- Candidates are sorted into this bin if they meet at least the requirements of the internal implementation for latency and hardware size.	**Prerequisites:** Architecture Profile, Blind Cycle Savings At this point we have decided on a base RISC target architecture such as 5-stage MIPS. Furthermore, this base architecture must specify how custom hardware will be integrated for both datapath augmentations and coprocessors. Each candidate at this stage has a corresponding cycles saved value based on frequency of execution and matching hardware implementation. The eventual goal of this modeling effort is to maximize performance by maximizing the number of cycles saved through custom hardware use. The cycles saved metric CS can be modified to more closely reflect how each candidate will function by knowing the architecture base. Common situations include: 1. Datapath stalls to prevent data hazards. 2. Bus or interface communication delay. 3. Variable coprocessor clock rate. All of these situations can be accounted for in creating a more accurate metric for how many cycles are being saved during custom hardware execution.	**Prerequisites:** Target Constraints, Hardware Size By now we have acquired a series of candidate instructions and have separated them and devised an accurate metric to demonstrate performance gains. Knowing the target constraints we can automatically generate a knapsack variant model with an additional number of non-duplication constraints equivalent to the number of custom hardware candidates that can be implemented both as datapath augmentations and coprocessors. This model can be solved to optimize the cycles saved by giving the best performance possible through selecting candidate instructions to meet architecture constraints. Factors used in defining the model include: 1. Global or Separate Hardware Area- Target systems will either have an individual hardware area constraint for each implementation type or a global shared hardware constraint. 2. ISA Limitations- A limited number of coprocessor or internal datapath instructions are available in most systems.

Chapter 6

Relaxing Constraints

6.1 Resolving Input/Output Limitations

Systems considered in this work are RISC based. By targeting RISC, we have been working under the general assumption of a limited IO. One factor that plays a role in this limitation is that of the porting of the register file. Fig. 6.1 below shows a high-level view of a generic register file.

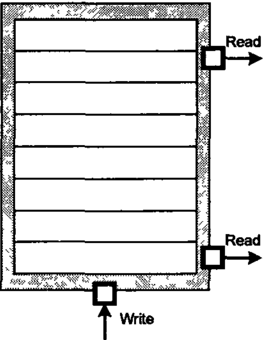


Figure 6.1: Typical Register File.

Register files are statically and finitely ported memories which contain each general register defined in an ISA. This bank of registers is a static RAM with dedicated read and write ports. Due to the static and dedicated nature of the memory ports, the instruction IO becomes constrained limiting available operand usage for instructions within the ISA.

Most register file memory units are designed as 2 input 1 output since many ISA instructions work on the same IO principle, especially with RISC. When considering custom instructions it is possible that custom instructions exist which garner a large performance gain that do not conform to this 2-1 standard. Xilinx offers a hardware solution in the form of the MPMC IP core [39] which connects to a standard memory and can multiplex out up to 8 ports using time division.

This time division idea for memory can be applied to most register files and has been studied for internal datapath augmentations with ISEs using additional internal pipelines [40]. The idea is to complete reads one after another progressing the values through a pipeline then using the contents of the pipeline registers as input to the custom instruction. Fig. 6.2 shows the read process given an internal custom instruction with 4 inputs. Each transaction below is assumed to require 1 clock cycle. Each cycle is denoted as Clock N.

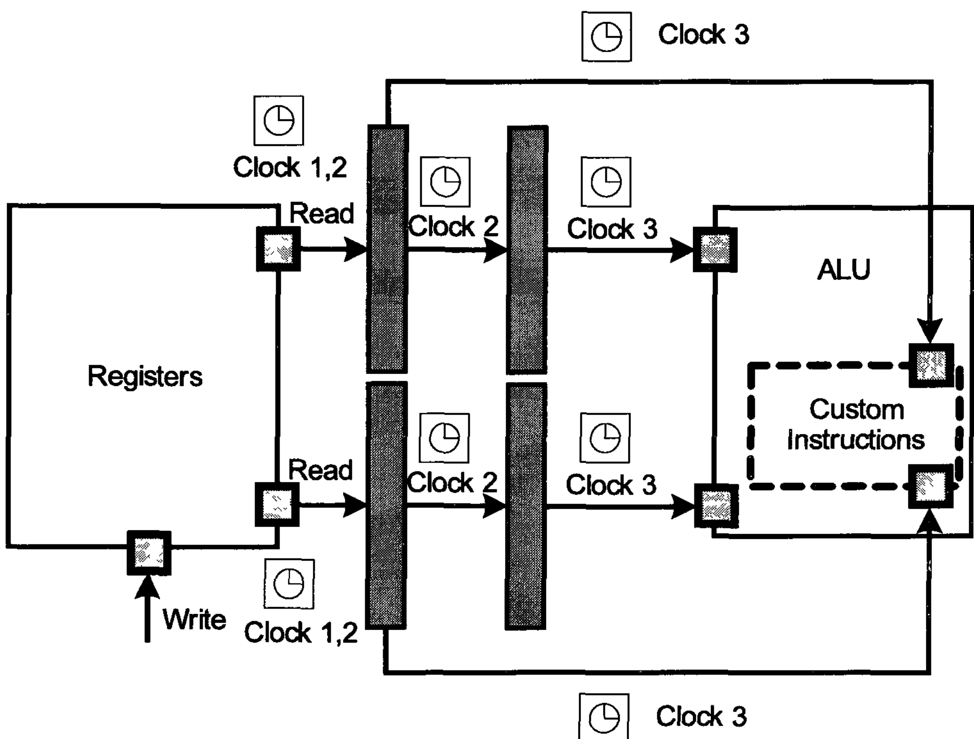


Figure 6.2: Pipelining to Compensate for IO Limitations.

Knowing that such IO limitations for the register file can be overcome we can accept

and bin more custom instruction candidates. Other IO limitations such as instruction width may remain, however, if these instructions are considered the CS value must be modified accordingly. If hardware is available or can be easily constructed to add multi-port functionality then these custom instructions can be included in the hybrid SoC model.

The CS should be modified if any special hardware is added to allow such a relaxation. This delay may be static if an existing solution is used or can be variably dependent on the number of inputs required.

6.2 Finite Target Hardware

So far we have worked under the assumption that we know how much hardware we want to use beforehand. This value has been used as an upper bound for our entire system in our effort to maximize performance. The main hardware platform considered in this document is that of the FPGA. Many applications which require or favor partial re-configuration benefit greatly from the use of a FPGA compared to that of an ASIC. The nature of FPGA chips can be introduced as a way to add variability to performance versus cost aspects of hybrid SoC creation.

The FPGA by nature is a chip with a constrained hardware design space. A selection of Altera FPGAs are presented in Table 6.1 with their matching available logic elements (LE). Hardware constraints are based on atomic FPGA hardware units such as Altera's

Table 6.1: Altera FPGA Hardware Sizes as Logic Elements (LE).

FPGA	Logic Elements (LE)
Cyclone II EP2C5	4608
Cyclone II EP2C8	8256
Cyclone II EP2C50	50528
Cyclone II EP2C70	68416
Stratix II EP2S15	15600
Stratix II EP2S60	60440
Stratix II EP2S90	90960
Stratix II EP2S180	179400

LEs shown here or Xilinx's logic slices. Due to implementation differences it is not

possible to convert exactly between different base units thus it is imperative that the correct base unit be generated with appropriate synthesis targeting the desired FPGA make.

Since FPGAs are finite hardware constructs we can effectively generate various hybrid SoC models based on a limited number of target chips. Essentially the same model will be executed multiple times with different hardware constraints. A user is expected to select a series of potential target chips where models are generated and solved as shown in Fig. 6.3. Unfortunately the runtime is accumulative but the average runtime is shown to be relatively quick in Chapter 9. We will later explore threading this problem for multiple cores to give a performance boost.

Potential Targets

1

2

N-1

N

Model Generator

Model

Sub-Model 1

Sub-Model 2

Sub-Model N-1

Sub-Model N

Select most advantageous sub-model.

Figure 6.3: Generation of Multiple Models with Varying Target Considerations.

6.3 Multiple Hardware Designs

Another issue not addressed thus far is the possibility of different hardware configurations for each individual custom instruction candidate. This document works under the assumption that either a one-to-one mapping is taking place from a DFG selection or

another industry mapping tool (*i.e.*,C2H, XPRES) has come up with one candidate for consideration. For the sake of completeness we will address hardware multiplicity here.

It is entirely possible and probable that multiple hardware configurations exist for candidate custom instructions. Automated hardware synthesis solutions can be tuned for many factors such as minimizing latency or minimizing hardware usage each providing a different RTL and set of constraints. We can easily model this idea as we did before with our *Non-Duplication Constraints.* Each candidate design can be entered into the model where the solver can force only one selection. Consider the example scenario shown next in Table 6.2 where multiple hardware designs are presented for internal and external custom instruction implementations. vn indicates a different configuration. Note that coprocessors and ALU implementations are not distinguished separately as only one vn can be selected ensuring non-duplicity. For example, with regards to the second candidate instruction v1 ≡v3 and v2 ≡v4.

To account for the multiple designs and the selection location we identify each one separately $v1...vn$. The *Non-Duplication Constraints* are modified to allow only one v selection for each custom candidate instruction. Considering each candidate instruction i with all possible designs vq we can modify the constraints as Eq. 6.1.

$$\sum_{q=1}^{Versions} x_{ivq} \leq 1 \tag{6.1}$$

Using this the above example table would have additional non-duplication constraints as follows:

$$x_{1v1} + x_{1v2} + x_{1v3} \leq 1$$
$$x_{2v1} + x_{2v2} + x_{2v3} + x_{2v4} \leq 1$$
$$x_{3v1} + x_{3v2} + x_{3v3} + x_{3v4} + x_{3v5} + x_{3v6} \leq 1$$

Table 6.2: Sample Candidate Scenario with Multiple Hardware Configurations.

	Coprocessor	**ALU**
Custom Candidate Instruction 1	v1 v2 v3	✕
Custom Candidate Instruction 2	v1 v2	v3 v4
Custom Candidate Instruction 3	v1 v2 v3 v4 v5	v6

Chapter 7

Algorithmic Perspective

7.1 Collecting Candidate Instructions

No specific requirement or method has been set out to define our desired custom instruction hardware candidates. As previously discussed there are many ways to obtain candidate instructions. This methodology only requires that candidate instructions can be represented as one instruction. This one instruction is used to identify a single RISC based instruction used for a custom candidate with matching hardware.

For this work and the later evaluation, instruction candidates have been gathered using 3 different methods:

- From Data Flow Graphs
- From Manual Profiling Data
- From Tensilica XPRES Compiler

Each method is a personal preference and selected by the developer. In our case each method was selected due to different target platforms and the availability of existing low-level tools such as customized assemblers.

Instructions that do not meet our basic needs are not selected. This may include instructions violating our IO conditions from before. Furthermore, here instructions are forbidden to include jumps and memory access for now.

7.2 ILP Solver Selection

The selection model itself has been described as an ILP model. Effectively solving an ILP on a computer can be difficult with various solvers available offering different algorithmic approaches and methods in order to offer a quick runtime. Both CPLEX and LINGO were tested but are proprietary and cannot be distributed and therefore were excluded from final implementation. The open-source LPSolve[41] was selected and offered a simple yet efficient interface in the form of an API that integrates well into the constructed Java front-end. LPSolve uses a branch-and-bound method for integer based problems.

The solver has been configured to provide an optimal solution while being permitted to decide whether to pursue the ceiling or the floor on each branch in a greedy fashion. No limit to the depth of the search is specified as default but can be set by the user.

7.3 Formulating a Hybrid SoC Design Flow Algorithm

This work provides ideas and a standard flow for the effective modeling of hybrid SoCs with an end goal of achieving a large speedup. Here we will put these ideas into a standard and algorithmic description. Note that the data structure HDL Custom Instruction List (i) defined below is in fact an array of custom instruction software objects as shown in Fig. 7.1 and is used in the upcoming algorithm.

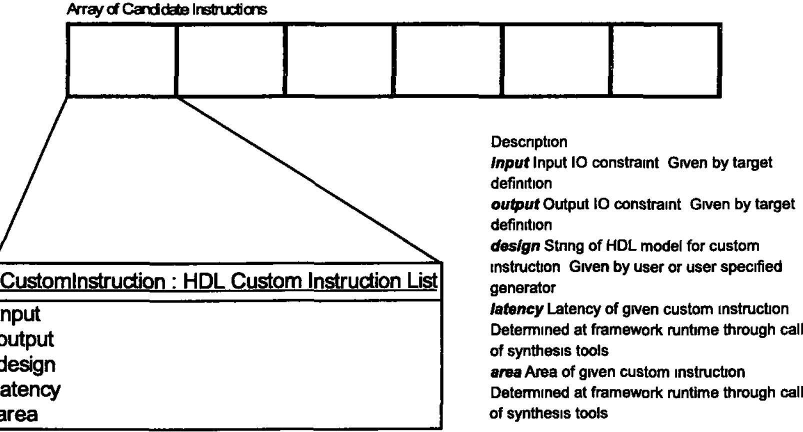


Figure 7.1: Custom Instruction Object List.

Described next is the algorithm for custom instruction selection in hybrid SoC targets. The *solve* function is defined textually after the presentation of the algorithm.

Require: Target Model Type (TM), Input Limitations (IO_{in}), Output Limitations (IO_{out}), Maximum Processor Latency (PL), Maximum Coprocessor Latency (CL), $PL \leq CL$, HDL Custom Instruction List (i), Processor ISA Bound (ISA_{pro}), Coprocessor ISA Bound (ISA_{copro}), Cycles Saved (CS)

1: **for** $x = 0$ to $i.length() - 1$ **do**
2: **if** $i[x].input \leq IO_{in}$ AND $i[x].output \leq IO_{out}$ **then**
3: $ValidInstructions \leftarrow i[x]$
4: **end if**
5: **end for**{Selection of only valid instructions meeting IO constraints.}
6: **for** $x = 0$ to $ValidInstrs.length() - 1$ **do**
7: $[ValidInstrs[x].latency, ValidInstrs[x].area] = SYNTHESIZE(ValidInstrs[x].design)$
8: **end for**{Synthesize each valid instruction to get timing and area information.}
9: **if** TM = Global Area **then**
Require: Global Area Constraint (GW)
10: **for** $x = 0$ to $ValidInstrs.length() - 1$ **do**
11: **if** $ValidInstrs[x].latency \leq PL$ AND $ValidInstrs[x].latency \leq CL$ AND $ValidInstrs[x].area \leq GW$ **then**
12: $DatapathStack \leftarrow ValidInstrs[x]$
13: $CoprocessorStack \leftarrow ValidInstrs[x]$
14: **else if** $ValidInstrs[x].latency > PL$ AND $ValidInstrs[x].latency \leq CL$ AND $ValidInstrs[x].area \leq GW$ **then**
15: $CoprocessorStack \leftarrow ValidInstrs[x]$
16: **end if**
17: **end for**{Verify latency of instruction and sort for global area constraint.}
18: **else if** TM = Separate Area **then**
Require: Internal Area Constraint (PW), External Area Constraint (CW)
19: **for** $x = 0$ to $ValidInstrs.length() - 1$ **do**
20: **if** $ValidInstrs[x].latency \leq PL$ AND $ValidInstrs[x].latency \leq CL$ AND $ValidInstrs[x].area \leq PW$ AND $ValidInstrs[x].area \leq CW$ **then**
21: $DatapathStack \leftarrow ValidInstrs[x]$
22: $CoprocessorStack \leftarrow ValidInstrs[x]$

23: **else if** $ValidInstrs[x].latency \leq CL$ AND $ValidInstrs[x].area \leq CW$ **then**
24: $CoprocessorStack \leftarrow ValidInstrs[x]$
25: **else if** $ValidInstrs[x].latency \leq PL$ AND $ValidInstrs[x].area \leq PW$ **then**
26: $DatapathStack \leftarrow ValidInstrs[x]$
27: **end if**
28: **end for**{Verify latency of instruction and sort for separate area constraint.}
29: **end if**
30: $[ALUInstrs, CoprocInstrs] = solve(CoprocessorStack, DatapathStack, TM, GW, CW, PW, ISA_{pro}, ISA_{copro}, CS)$ {Execute solver and model generation with binned instructions.}

The solve function takes all arguments needed to generate a model and sends these to the model generator. The model generator creates a program script that can be run through the LPSolve ILP solver. The arguments are listed as follows:

- *CoprocessorStack* - Object list of coprocessor candidates. This includes needed area and IO values.
- *DatapathStack* - Object list of internal datapath candidates. Again, this includes needed area and IO values.
- TM - Type of model we are using, whether it be with global area (0) or separate area (1).
- GW - Global area value. Can be null if not used.
- CW - Coprocessor area value. Can be null if not used.
- PW - Processor datapath area value. Can be null if not used.
- ISA_{pro} - ISA bounds on number of internal processor instructions.
- ISA_{copro} - ISA bounds on number of external coprocessor instructions.
- CS - Cycles Saved metric determined through theoretical profiling which defines the number of saved cycles if the instruction is used.

7.4 Multi-Threading for Multiple Executions

Given that the user is able to select multiple FPGA targets, the above algorithm is run multiple times. For the physical implementation, each traversal of the algorithm and solver instantiation has been divided and encapsulated into a separate Java thread. With multicore and multithreading processors ever prevalent these independent runs can be automatically assigned to different CPU cores on the host machine. Fig. 7.2 shown below illustrates this threading model where a thread generator will dispatch threads depending on how many target FPGAs the user would like the system to explore. Each thread will contain an independent execution module which maintains a generated ILP script, a solver instantiation and basic control logic. When data is returned from the threads it is presented to the user and a selection is suggested.

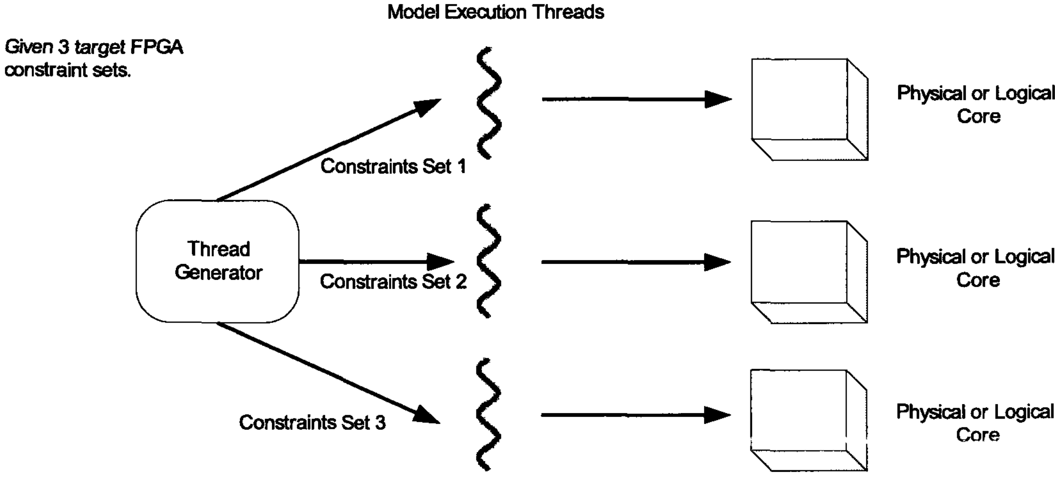


Figure 7.2: Threading Model for Multiple Target FPGA Candidates.

This concept was tested on a Core 2 Quad with 8GB of DDR3 RAM using a sample hardware model with 100 nodes within a front-end Java Application. This processor has 4 physical (not logical) cores. In a perfect world we could expect a maximum speedup of 4. This is never the case due to overhead and resource bottlenecks. The maximum mean speedup achieved was 3.2x. The results are given in Fig. 7.3 after 50 tests on each thread number. As the number of threads grows larger, the demand on the shared memory space increases along with the required space for each thread's data. This is easily seen through the Java Virtual Machine where allotments of memory must be

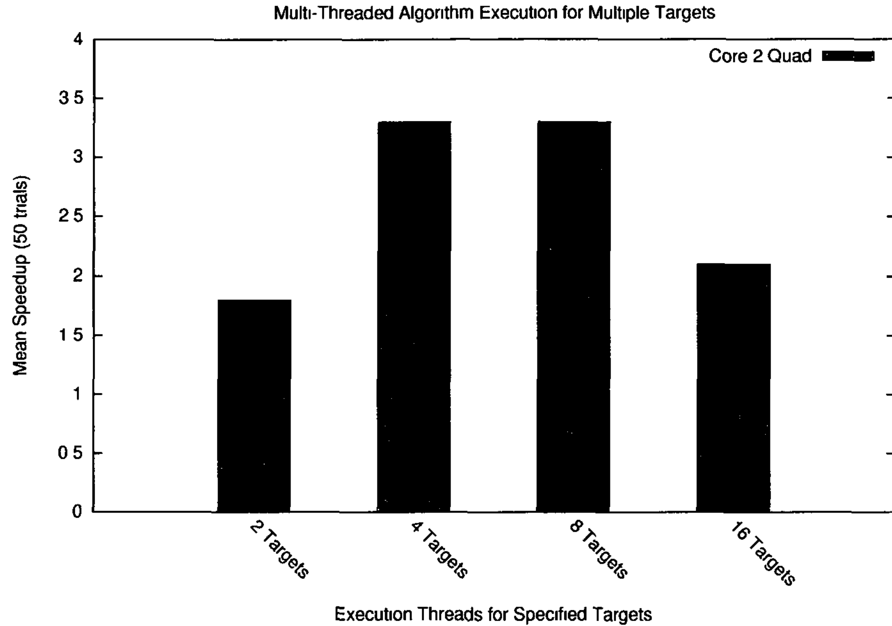


Figure 7.3: Threaded Execution Results.

constantly increased. Experimentation with the garbage collector may improve results but cannot clear native code leaks if JNI is used within the solver libraries. According to these results, a safe thread generation model should generate at maximum 8 threads when using a similar host with regards to processing power and memory. For now, each target environment will generate an individual thread. For the purposes of this work a maximum of 4 target FPGAs, therefore 4 threads, have been extensively tested in the entire user flow.

7.5 Formulating a Full FPGA-based Hybrid SoC Toolchain

A major portion of this work has involved not just the backend of such a design flow but also a user oriented front-end as well. Here we look at both of these from the perspective of the hardware designer as we walk through the user flow framework housed in Java.

Before beginning the task of hybrid SoC analysis a user has obtained a set or possible custom instructions, whether through profiling or lower-level analysis. Each instruction is

required to be available in an HDL format and accompanied by a constraint file specifying basic hardware outlines including the estimated number of cycles that will be saved by using this instruction and the IO interface of the custom block. Consider the identified custom instruction shown in Fig. 7.4 with the matching VHDL implementation *ins1.vhd* and constraint file *ins1.dat* in Fig. 7.5. The instruction in question takes in three unsigned integer operands, adds the first two and subtracts the third from the sum.

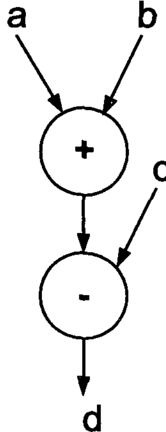


Figure 7.4: Identified 3-1 Custom Instruction for 32-bit Unsigned Integers.

The constraint file is parsed for the following information:

1. `INPUT_PORT`

   The number of input operands required by the custom instruction and specified in the VHDL file.

2. `OUTPUT_PORT`

   The number of output operands required by the custom instruction and specified in the VHDL file.

3. `CYCLES_SAVED`

   The number of cycles that will be saved if this instruction is implemented. This value is an estimate but can be very accurate for static interrupt-less programs.

Figure 7.5: Adder Subtracter for 32-bit unsigned integers, VHDL and constraint file.

```
--filename ins1.vhd
--Identified Custom Candidate, AdderSub
--3 Input Operands (a,b,c) 1 Output d
--Result d=a+b-c

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all ;

ENTITY ins1 IS
GENERIC (n :INTEGER := 32);
PORT (REG1, REG2 :IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
      REG3 :IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
  OUTREG: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
END ins1;

ARCHITECTURE Behavior OF ins1 IS
signal operand1,operand2: unsigned (n-1 DOWNTO 0);
signal operand3,resultOp: unsigned (n-1 DOWNTO 0);
BEGIN
operand1<= unsigned(REG1);
operand2<= unsigned(REG2);
operand3<= unsigned(REG3);
OUTREG <= std_logic_vector(resultOp);

resultOp<=operand1+operand2-operand3;

END Behavior;
```

```
#filename ins1.dat
#Initial Constraints for ins1.vdhl
INPUT_PORT: 3
OUTPUT_PORT: 1
#Cycles Saved Per Call*Number of Calls
CYCLES_SAVED: 54,000
ORIGINAL_TARGET_SPECIFIC_MAKE:ALTERA
ORIGINAL_TARGET_MODEL:CYCLONEII_EP2C20F484C7
```

4. `ORIGINAL_TARGET_MAKE`

   The original target FPGA make for the given HDL. Acceptable flags are `ALTERA` or `XILINX`. The reason for this query is that occasionally an HDL specified block may call target specific libraries such as Altera's LPM[42] modules. If the user wishes to target another platform they will be prompted with a warning specifying that the original HDL was designed for a specific make and may require porting.

5. `ORIGINAL_TARGET_SPECIFIC_MODEL`

   The specific FPGA model that the HDL was generated for. This value does not necessarily need to correspond to the target models that are being explored by the user. This value is simply used as a reference for the original HDL generation. Either a space-less FPGA code is acceptable or the null terminal.

Note that comments in the constraint files should be presented with a preceding # character unlike the VHDL comments of - -.

7.5.1 Configuring the Working Directory and Path Variables

As the user begins the process of analyzing their custom instructions, initial path settings and a working directory must be specified as presented in Fig. 7.6. The user has numerous HDL design files with matching **.dat* constraint files. These files need to be located in their working directory. The user begins by specifying this working directory. The PATH variable on Windows can be mysterious at times so user specified paths can be set here. These paths should include the working directory and TCL interpreter for the target synthesis tool. For example, if Altera Quartus is selected, the bin folder from the installation should be included here. If a path is incorrectly specified the following window will warn the user that the required interpreter was not found.

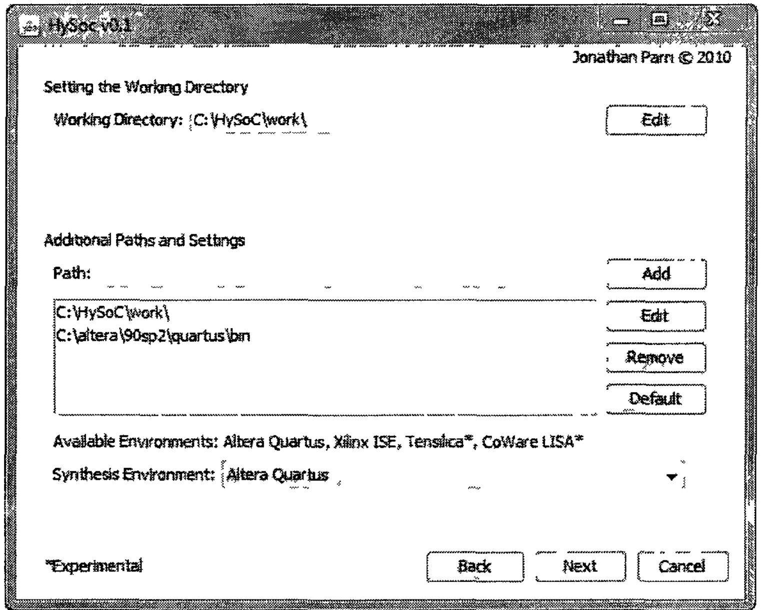


Figure 7.6: Selecting the Custom Instruction Directory and Synthesis Tools.

Synthesis environments that can be selected include Altera Quartus, Xilinx ISE, Tensilica or CoWare LISA. Both Quartus and ISE have been tested extensively but Tensilica and CoWare remain experimental and will be improved with further releases as TCL support improves.

7.5.2 Overview of Found Custom Instructions and Target Configuration

On the next page, Fig. 7.7, the user is prompted with a list of found custom instruction HDL files and matching constraint files from within the working directory. During the previous stage, paths could be specified by the user and are now verified. In the *Tool Discovery* pane, the system path and the specified paths are searched for TCL interpreters. The user should verify that the correct interpreter has been found for the target synthesis tool selected on the previous page.

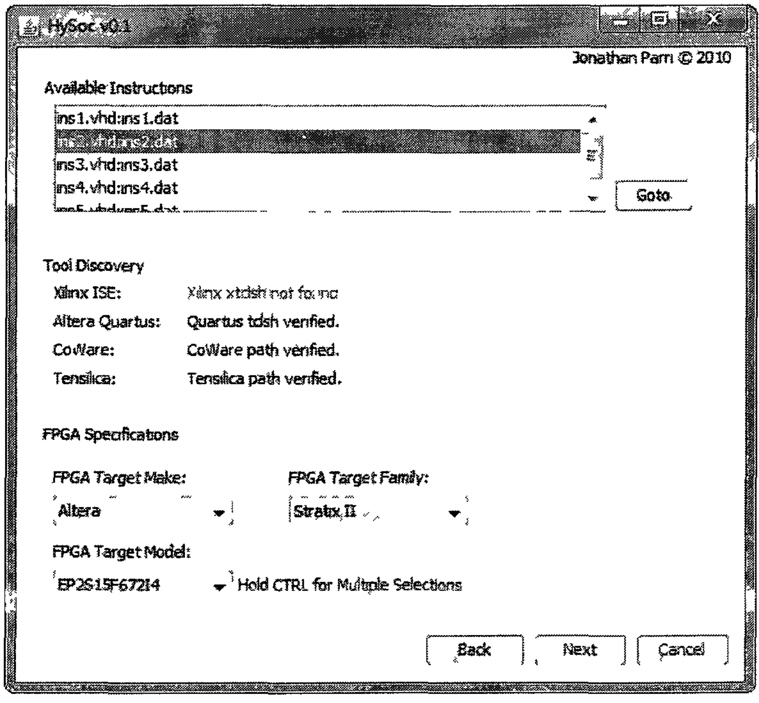


Figure 7 7: Reviewing Custom Instruction Files and Configuration of Target FPGA

Both Xilinx and Altera place their TCL interpreters, **xtclsh** and **tclsh84** respectively, in their installation's bin directories. Both Xilinx and Altera tools are fully automated. CoWare and Tensilica synthesis requires manual intervention not discussed here and is experimentally supported.

The *FPGA Specification Pane* allows the user to select the target FPGA make, family and model. Multiple models can be selected here. The user must ensure that the make selected is supported by the synthesis tool previously specified to avoid an error.

7.5.3 Solver and Virtual Machine Settings

As previously stated, Java provides the front-end presentation logic and back-end dispatch framework. Here the user has the opportunity to allocate more memory to the virtual machine heap or modify the stack size. More importantly the user can specify how many threads they wish the program to dispatch. The smallest thread unit contains one execution stream for a different target FPGA. For example, if the user selects 4 target FPGAs and 2 threads then 2 execution streams will be assigned to one thread. The different execution streams available in one thread will be executed sequentially. Available settings are shown in Fig. 7.8.

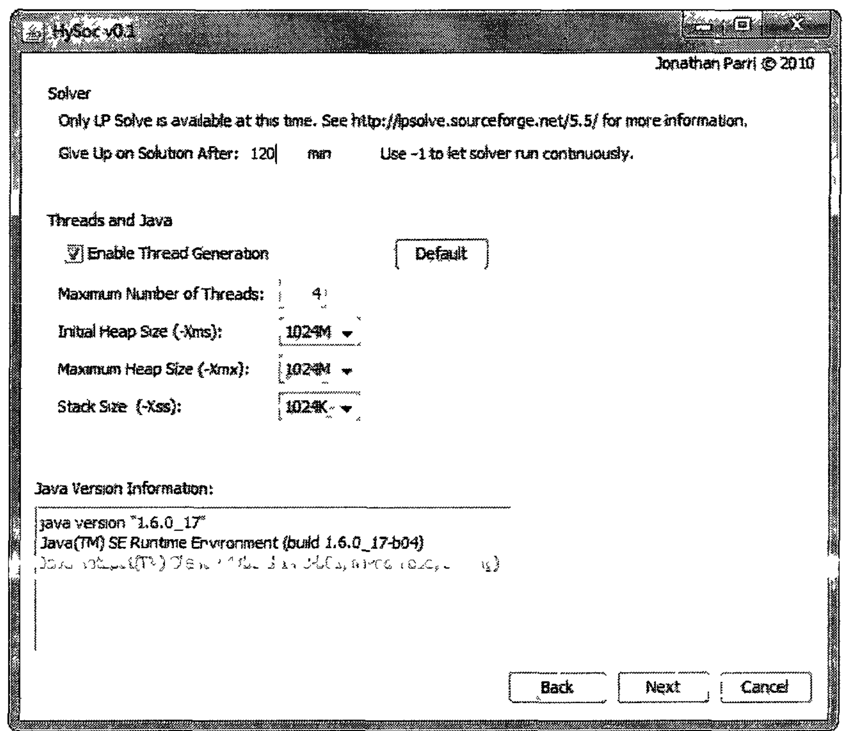


Figure 7.8: Solver and Java Settings.

The currently implemented solver is LPSolve. The solver option most readily available to the user is a watchdog timer. The timer value in minutes specifies a maximum solver runtime. At this time, individual solver settings such as search depth or branch and bound rules cannot be set in the front-end. They can be changed manually using the *ModelGen* hooks discussed in the next section.

7.5.4 Target Hardware Constraints

On the next page the user is given the opportunity to specify constraints imposed by the target system and architecture. These settings are shown in Fig. 7.9. The user must specify whether they wish to have a global or separate area constraint and give values for these bounds. If the user is unsure of what maximum area to set and has multiple targets they can view each target maximum automatically. ISA limitation bounds, latency and IO bounds are all specified here as well. The user must ensure that any settings used here reflect what the target architecture can support. Only one hardware area value is currently accepted. If multiple targets have been selected the difference between the global area given here and the maximum FPGA size are used as a factor. The available area for each other FPGA target is calculated as the difference between its maximum size and this factor. The assumption is that the factor is in fact the area overhead of the existing system.

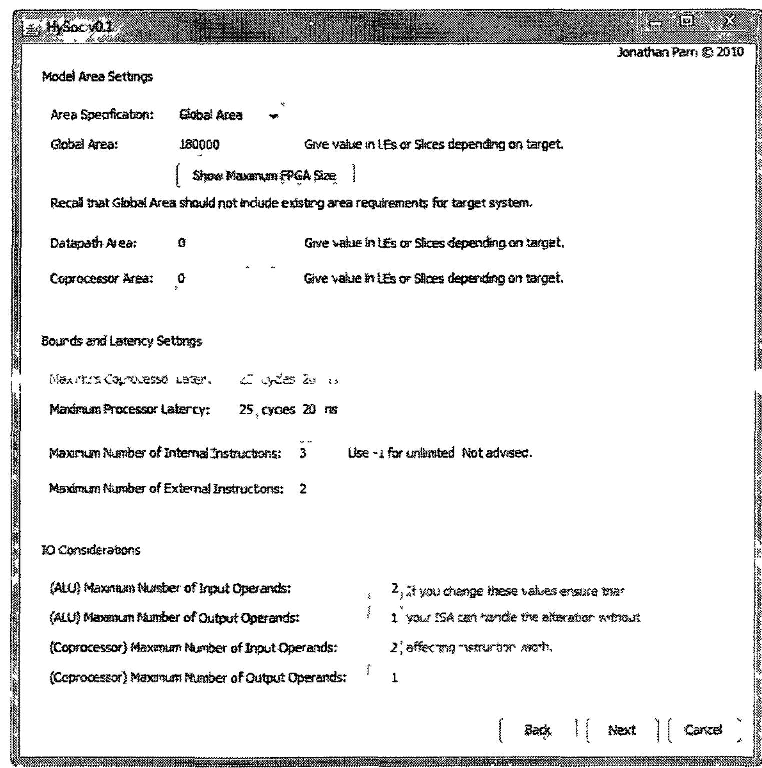


Figure 7.9: Target Hardware Constraint Settings.

7.5.5 Candidate Synthesis

Many FPGA design tools, including those from Xilinx[43] and Quartus[44] can utilize TCL scripts to run internal commands. These scripts can automate testing and building of components or entire systems. They are conveniently used here to get area and latency information about potential candidates.

Fig. 7.10 shows the steps required for obtaining timing and area information. Before getting area and latency/frequency information the user can run a *Quick and Dirty Candidate Removal.* This process will remove all current candidates that do not meet with the specifications dictated on the previous page which do not require synthesis to obtain information. This may reduce the number of HDL candidates we need to synthesize. This is actually covered in the algorithm described previously as validation.

TCL scripts must be generated before running the *Synthesize Candidates* process. The generated scripts include: source file specification, project instantiation with device settings and a variable return for latency and area. On a synthesis run the scripts are sent to the designated interpreters to execute the compilation.

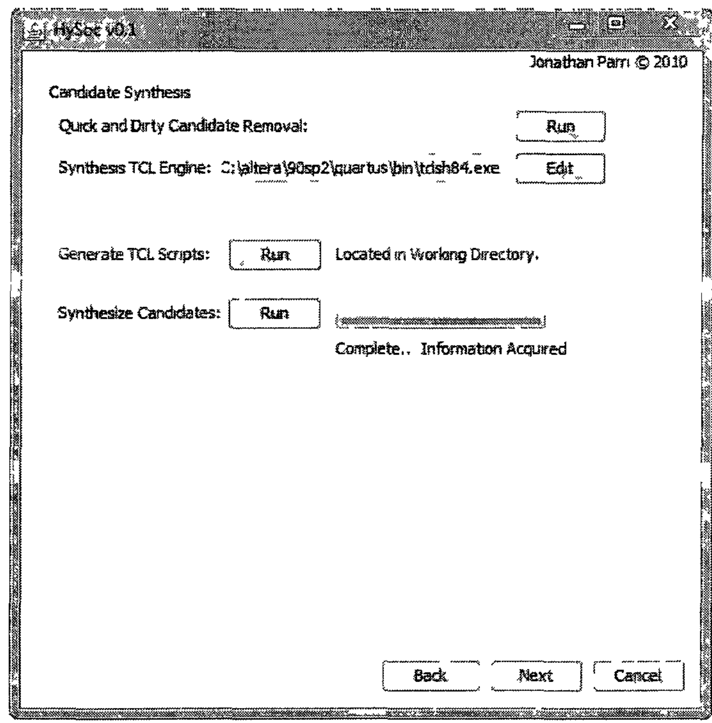


Figure 7.10: Getting Area and Latency Information Automatically.

7.5.6 Solver Execution

The page shown in Fig. 7.11 is where most of the processing time is spent. First the ILP models are generated as CPLEX scripts. CPLEX scripts are used as the syntax appears cleaner than that of default LPSolve script syntax. LPSolve can accept models in LP, MPS, MathProg, CPLEX, LINDO and LPFML using an external language interface.

Next, threads are created containing control logic, LPSolve libraries and model scripts. Upon creation, the threads begin to execute. The process stops when either all threads have terminated or the watchdog timer stops the execution. If the watchdog timer throws the execution and an optimal solution was not found either a sub-optimal value can be selected or an estimate run can be done.

Due to the difficulty of finding an exact and proven PTAS for these problem variations, a simple brute force selection is offered as a backup estimate. Again the user specifies a stop time. The random algorithm will try as many solutions as it possibly can within that time frame and compare the returned selections to the sub-optimal solution. The brute-force solution may be worse or better than the sub-optimal and is presented to the user as an option. Unfortunately, previously searched solutions may be explored again.

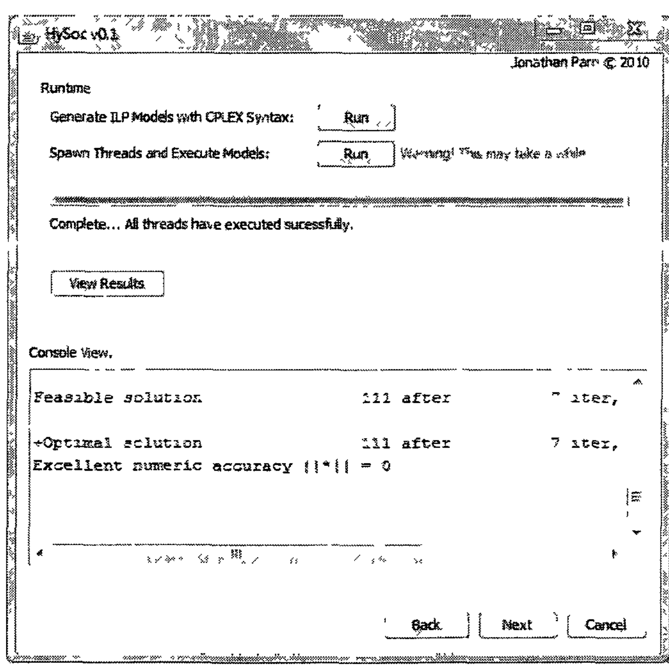


Figure 7.11: Spawning Threads and Executing the Solvers.

7.5.7 Reviewing Results

Once a selection has been made, either optimally or sub-optimally, the user is presented with the custom instructions to be implemented divided into *Internal Processor Datapath* and *External Coprocessors*. This is shown in Fig. 7.12

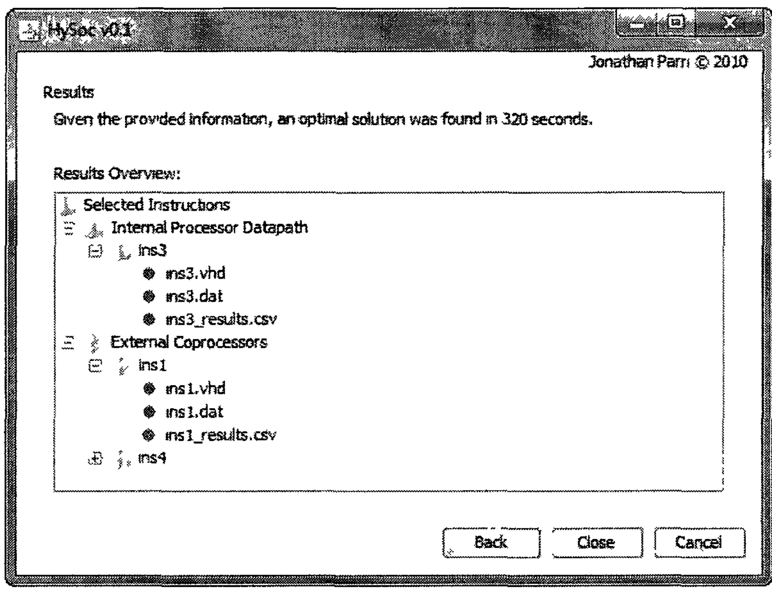


Figure 7.12: Reviewing Results.

Each instruction in the results window has a generated **.csv* that contains metric and solver information. The file contains:

- Hardware placement suggestion.
- Result specified as optimal, sub-optimal, infeasible or unbounded.
- Brute force random estimate comparison, if used.
- Solver execution time.
- Hardware parameters, both from constraint file and synthesis tool.
- Estimated total cycles saved and speedup with all selected instructions implemented.

7.6 Software Overview

For completeness we examine the software architecture of the framework application from a class level illustrating component interaction. From a top-level we have 3 main packages as shown in Fig. 7.13. The *gui* with the presentation logic, *lpsolve* maintaining the solver libraries and API and *control* holding everything together within Java and performing native process calls.

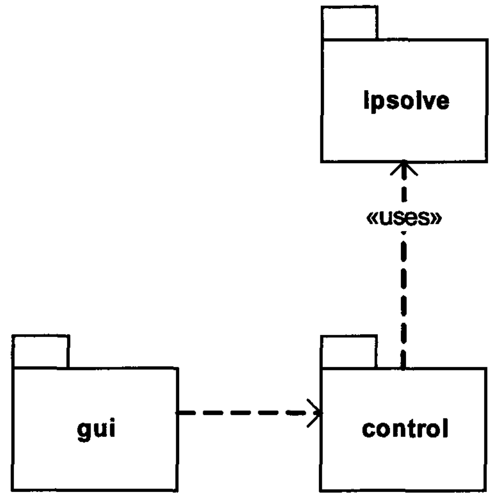


Figure 7.13: Toolchain Package Diagram.

The *control* package maintains the heart of the processing and control logic. Within this package we have 6 main classes. These main classes can be seen in Fig. 7.14. The *Controller* class is a high-level maintenance class which is made up of a model generation class, *ModelGen*, a thread dispatching class, *Dispatcher*, and a class which deals with constraint generation and maintenance, *Synthesis*. The *Synthesis* class can generate many *TCLGenerators* which create the TCL scripts to synthesize design candidates. Furthermore, the *Dispatcher* creates *ExecutionThreads* which hold the solver runtime threads that interact with LPSolve.

The *ModelGen* class is responsible for creating the ILP model script and setting the solver parameters. Bypass hooks are available to use a different model generation algorithm along with altering solver settings. The scripting engine can be turned off through the available API with all constraint variables transparent through it as well.

The interaction between internal components and external processes is key for correct

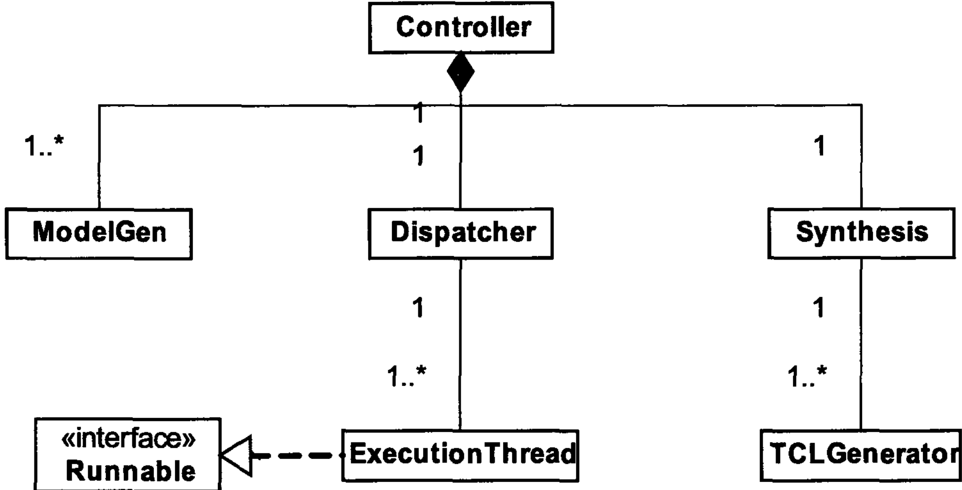


Figure 7.14: *control* Class Diagram.

and efficient operation. The brief architecture overview is concluded with Fig. 7.15 showing the interaction of components both internally and externally. Internal classes, files, folders and external processes are shown.

7.7 Compiler Integration

The first method of gathering custom instruction candidates described previously is that of data flow graph analysis. This data flow graph analysis is typically done during code compilation where the code has been broken down into basic blocks and intermediate representations. The *ModelGen* hooks provided can be integrated into an external solver that is performing passes during compilation. Hybrid analysis can be facilitated during the compiler's runtime.

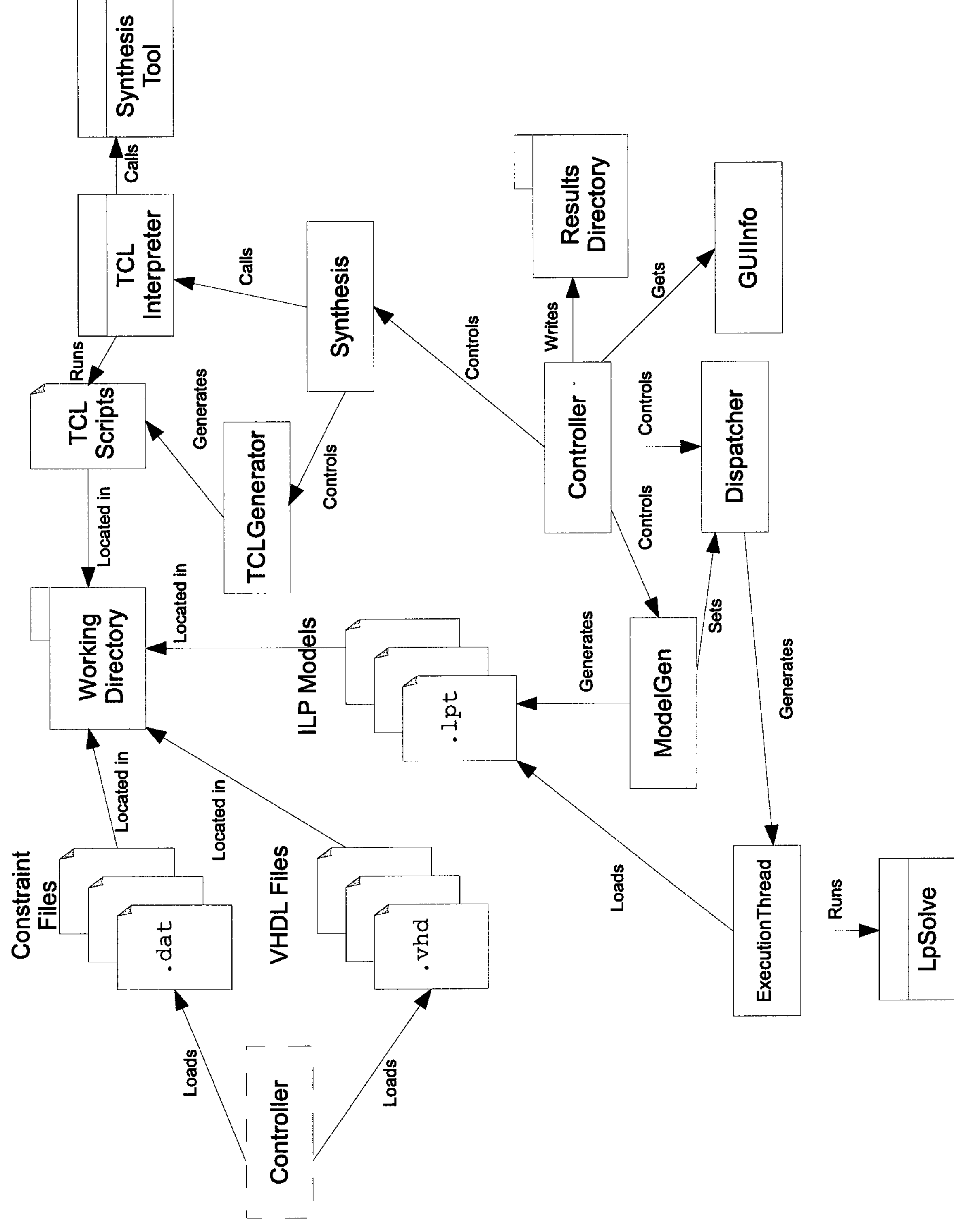


Figure 7.15: Internal and External Component Interaction.

Chapter 8

SoC Hybrid Integration with NIOS II-A Case Study

8.1 Overview

Altera is most widely known for their FPGA hardware; however, they offer many products facilitating hardware/software codesign. One of these products is the NIOS II. The NIOS II is a 32-bit soft-core processor designed for Altera FPGAs. To construct NIOS based systems, Altera provides SOPC Builder. SOPC builder is a Quartus based plug-in used to configure and generate NIOS based systems. SOPC builder can be used to add peripherals to the Avalon bus, memory controllers, or additional processors and configure each individually.

Many different design automation tools are available on the market to aid in the conversion from the algorithmic side to the software side. A famous example of this is Catapult C[45] distributed by Mentor Graphics. Here, a user can specify an application in ANSI C/C++ and have matching VHDL or Verilog hardware automatically generated. Altera provides a similar tool, the C2H compiler. C2H allows acceleration of software algorithms at the functional level. For example, consider the excerpt of a program shown below. This snippet is an entire function which can be accelerated with C2H. You simply highlight the function in the NIOS IDE, then it will attempt to generate a hardware block for SOPC Builder and recompile the source code with the new mapping.

```
/*Acceptable C2H Function*/
int func1 (int a, int b, int c )
{
    int result,temp;
    temp=a+b;
    temp=temp*temp;
    result=temp+c;
    return result;
}
```

Either the C code will be transformed into pure combinatorial logic or a state machine will be introduced. The above statement converts into one combinatorial block with use of `ieee.std_logic_arith.all` VHDL operators simplifying the arithmetic mapping.

C2H supports most standard ANSI C/C++ constructs excluding **float**, **double** and some **struct** declarations. It is also crucial to note that variable declarations cannot occur after any operation takes place in a function block.

Target wise, the C2H compiler generates hardware accelerators or coprocessors which exist outside the NIOS' datapath. Since we are looking at hybrid systems, the problem lies in the fact that C2H only generates hardware accelerators that connect through the Avalon bus. Since the functions from our sample problem given in Appendix B are simple and their corresponding HDL code is rather simple, we are able to extract only the execution logic from the C2H generation. Fortunately, NIOS also offers a tool for adding internal custom instructions into the datapath[46] which will be based on this extracted logic. The logic will also be used in a custom coprocessor wrapper instead of the one provided by C2H.

The NIOS II custom wrappers are defined as:

- **Internal Processor Datapath Custom Instruction Wrapper**
  2-1 IO ported wrapper with multi-cycle support. Only register file interaction is permitted. Stalls are introduced before custom logic initialization to ensure no dependencies remain in the pipeline.

- **External Coprocessor Custom Instruction Wrapper**
  Memory-mapped peripheral with FIFO interface to Avalon Bus. Coprocessor is activated and used by writing to matching memory address with operand data

from register file. A maximum of three input operands and 1 output result has been included in the design.

8.2 Target SoC Environment

The target platform tested for this NIOS based hybrid SoC is the Altera DE1 Development Board. This reference board contains a Cyclone II EP2C20F484C7N FPGA. On a side note, we will be targeting only this physical FPGA therefore only one execution thread will be generated for the model analysis portion of this case study.

A NIOS II/s processor is used with 4 KB instruction cache. The program executable is located in SRAM while the program heap in SDRAM. The system is set to run at a frequency of 50MHz. An outline of the SoC is shown in Fig. 8.1. In this physical and

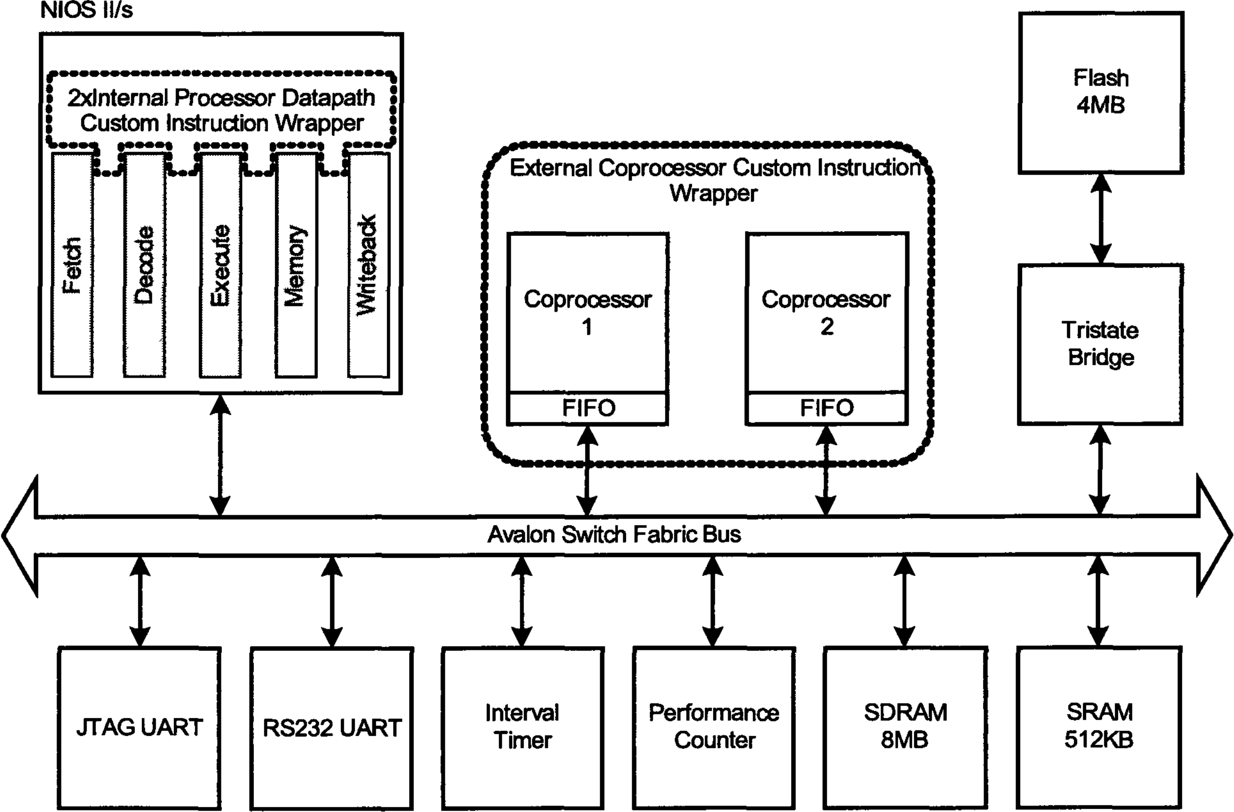


Figure 8.1: NIOS II SoC Target Architecture.

verifiable system we would like to explore our hybrid instruction options for two potential internal instruction and two external coprocessors as the corresponding wrappers have been developed in VHDL and are included in this design. Excluding the coprocessor wrappers our base design utilizes 4,371 logic elements (LE). The total number of LEs available on the FPGA is 18,752 leaving 14,381 LEs. Due to place-and-route considerations we will make available 70% of the difference between the total available and already used elements yielding a global area maximum hardware constraint of 10,066 LEs.

We have already began to gather some constraints and metrics. To review, we have:

1. 50MHz provided by global clock. No PLLs are available.
2. 2-1 IO constraints for internal processor datapath instructions.
3. 3-1 IO constraints for external coprocessor instructions.
4. 2 available coprocessor custom instruction slots.
5. 2 available internal processor datapath custom instruction slots.
6. Known overhead for custom instruction wrappers.
7. Global available area of 10,066 LEs for a target Altera Cyclone II.

8.3 Baseline Execution

The sample program to be accelerated with coprocessors and new internal instructions is given in Appendix B. The program is made up of 10 functions that can be accelerated using C2H. Function contents range from simple arithmetic to loops. Each function is called 1000 times. Profiling the basic software-only program implementation shows a need of 35357445 clock cycles for complete execution.

By turning these functions into hardware we look to garner a significant speedup. The particular functions estimated to offer the greatest gain are those with the loops. C2H is able to effectively unroll them and offer a hardware alternative. At most, the loop based functions require another 1000 internal iterations, totaling $1000 \times 1000 = 1000000$ total iterations spent within the function. There is much room for speedup.

8.4 Custom Instruction Candidates

Each of the 10 functions offers potential as a custom instruction. Each function has been manually run through C2H and profiled (and normalized) to obtain a cycles saved *CS* metric. The custom arithmetic logic has been stripped out of the generated VHDL files and placed into candidate VHDL files to run through the hybrid SoC framework application shown previously. Constraint files have been manually created for each of the corresponding VHDL files.

Each candidate has been automatically run through the Quartus synthesis tool. The area results of the synthesis and CS profiling data are given in Table 8.1.

Table 8.1: Generated and Obtained Candidate Instruction Metrics.

Function	Datapath Area (LE)	Coprocessor Area (LE)	Cycles Saved Datapath	Cycles Saved Coprocessor
power3()	193	267	48418	46218
addUpto()	352	606	15895245	15893015
factorial()	355	609	16854941	16852748
power4()	235	488	61161	58961
mac()	267	521	3032	2048
limit()	87	341	27108	24908
func1()	235	489	1114	-2976
func2()	212	466	1794	11
func3()	221	475	2086	226
func4()	230	484	1869	206

The table clearly shows that the two largest custom instructions with respect to area offer the greatest cycles saved metric. These functions are the largest in the program containing a single loop each. It is interesting to note that the simple arithmetic functions, **func1()** to **func3()** do not see any large improvement as coprocessors. **func1()** actually shows a performance decrease. This occurrence is expected as coprocessors have a large overhead. It is clear that this overhead is larger than a full NIOS pipeline execution for two arithmetic operations. Nevertheless, if all instructions are selected with their best *CS* value we would have an astonishing estimated speedup of 14.37x. Fig. 8.2 shows the number of cycles needed for execution if the accelerators were fully implemented in both internal and external wrappers.

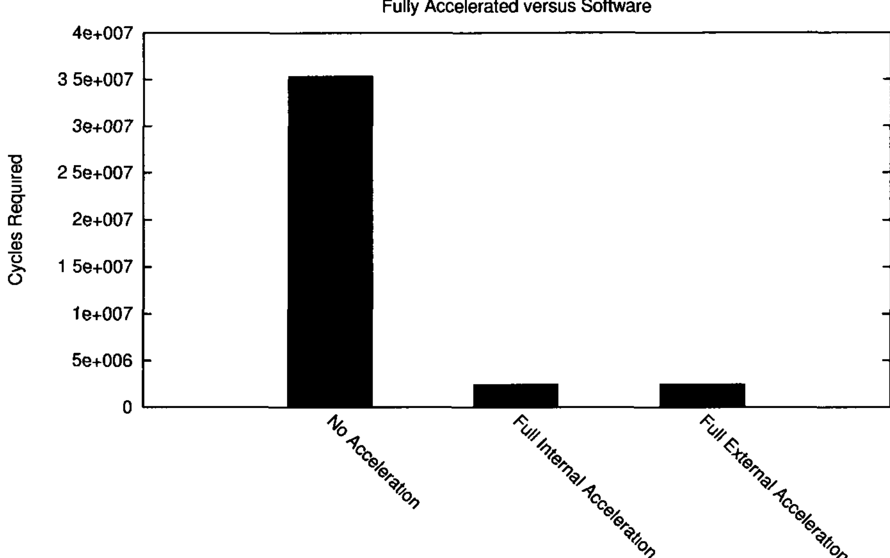


Figure 8.2: Profiled Required Cycles for Fully Accelerated Design versus Software.

In this problem scenario only one global clock is available, 50MHz. Table 8.2 shows the maximum frequency that can be achieved without timing problems for each custom instruction candidate. Recall that the maximum frequency of the base non-augmented system is only 66.17MHz. If we were to implement a Phase-locked-loop (PLL) or utilize an external clock, the coprocessors could be clocked at a variable frequency using the FIFO buffer attached in the wrapper to facilitate proper data transfer with the processor. All hardware generated by C2H meets our requirement of a minimum 50MHz latency and can be included for consideration with regards to latency. The coprocessor implementation of **funcl()** will be automatically excluded since it lowers our end performance. Similarly, the **mac()** violates the IO constraint for the internal custom instruction wrapper and will only be considered as a coprocessor implementation.

8.5 Candidate Selection

As previously stated, **funcl()** as a coprocessor implementation will be automatically removed since it does not benefit the end system. All other candidates will be run through the model generator. The generated model is shown below. Since this model is

Table 8.2: Maximum Frequency of Hardware.

Function	Maximum Frequency (MHz)
power3()	65.24
addUpto()	102.48
factorial()	63.37
power4()	64.15
mac()	84.25
limit()	92.7
func1()	60.96
func2()	62.29
func3()	61.2
func4()	60.73

quite simple due to the limited number of custom instructions our search time is only 0.023s yielding an optimal solution. The model that was automatically generated is shown in Appendix C. The resultant model matrix is shown below in Eq. 8.1 with each row representing constraints and each column representing a custom instruction option.

Columns read left to right as $\text{power3}()_{processor}$, $\text{power3}()_{coprocessor}$, $\text{addUpto}()_{processor}$, $\text{addUpto}()_{coprocessor}$, $\text{factorial}()_{processor}$, $\text{factorial}()_{coprocessor}$, $\text{power4}()_{processor}$, $\text{power4}()_{coprocessor}$, $\text{mac}()_{coprocessor}$, $\text{limit}()_{processor}$, $\text{limit}()_{coprocessor}$, $\text{func1}()_{processor}$, $\text{func2}()_{processor}$, $\text{func2}()_{coprocessor}$, $\text{func3}()_{processor}$, $\text{func3}()_{coprocessor}$, $\text{func4}()_{processor}$ and $\text{func4}()_{coprocessor}$.

$$\begin{bmatrix}
193 & 267 & 352 & 606 & 355 & 609 & 235 & 488 & 521 & 87 & 341 & 235 & 212 & 466 & 221 & 475 & 230 & 484 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1
\end{bmatrix} \quad (8.1)$$

Rows read top to bottom as Area Constraint, power3() Non-Duplication, addUpto() Non-Duplication , factorial() Non-Duplication, power4() Non-Duplication, limit() Non-Duplication, func2() Non-Duplication, func3() Non-Duplication, func4() Non-Duplication, Internal ISA bounds and External ISA bounds.

If we examine the right-hand side (RHS), Eq. 8.2, we see that all constraints have been satisfied after execution of the solver.

$$RHS = \begin{bmatrix} 0 \\ 14381 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 2 \\ 2 \end{bmatrix} \quad (8.2)$$

The resultant solution is shown below in Eq. 8.3. The 1's indicate that the matching instruction has been selected. 4 instructions were automatically selected, 2 as internal processor datapath instructions and 2 as external coprocessors. The selected instructions are given below:

- **Custom Instructions as Internal Datapath Augmentations**
  - **power3()** saving 48418 clock cycles.
  - **addUpto()** saving 15895245 clock cycles.
- **Custom Instructions as Coprocessors**
  - **factorial()** saving 16852748 clock cycles.
  - **power4()** saving 58961 clock cycles.

By utilizing this hybrid SoC selection the total estimated savings is 32855372 cycles with a total additional hardware usage of 1,642 LEs which is well below our upper limit

of 10,066. Our new performance gain is seen in Fig. 8.3 with a 14.13x speedup.

$$\begin{bmatrix} \text{Cycles Saved} \\ \text{power3}_{processor} \\ \text{power3}_{coprocessor} \\ \text{addUpto}_{processor} \\ \text{addUpto}_{coprocessor} \\ \text{factorial}_{processor} \\ \text{factorial}_{coprocessor} \\ \text{power4}_{processor} \\ \text{power4}_{coprocessor} \\ \text{mac}_{coprocessor} \\ \text{limit}_{processor} \\ \text{limit}_{coprocessor} \\ \text{func1}_{processor} \\ \text{func2}_{processor} \\ \text{func2}_{coprocessor} \\ \text{func3}_{processor} \\ \text{func3}_{coprocessor} \\ \text{func4}_{processor} \\ \text{func4}_{coprocessor} \end{bmatrix} = \begin{bmatrix} 32855372 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{8.3}$$

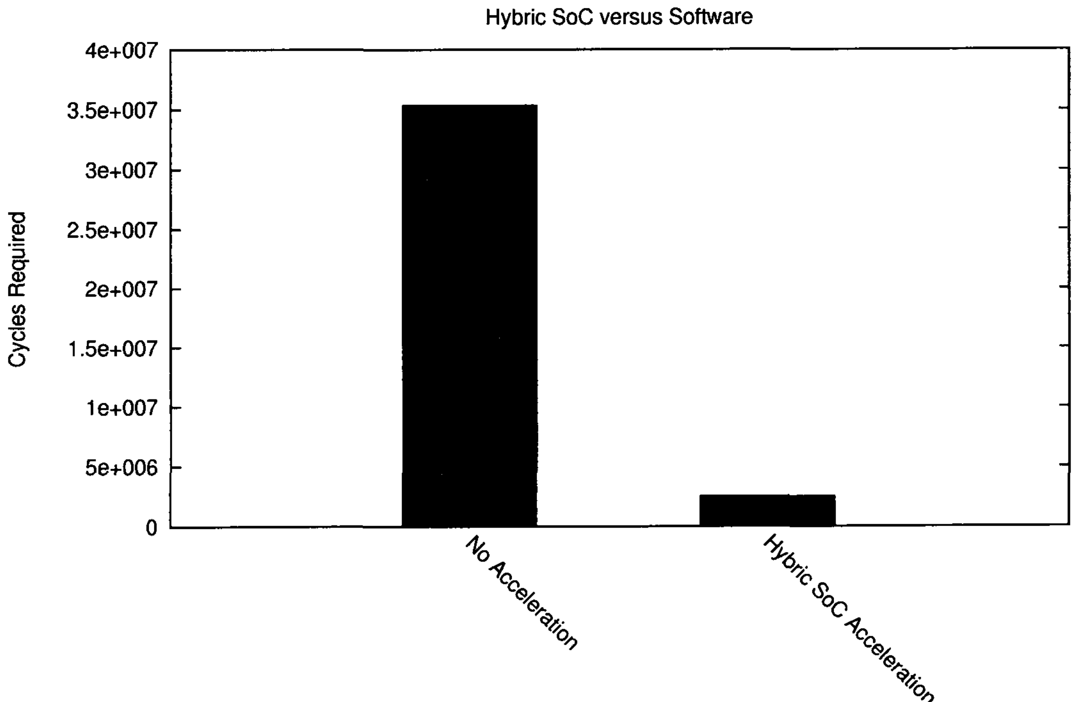


Figure 8.3: Profiled Required Cycles for Hybrid SoC Design versus Software.

8.6 Final Integration

The ultimate test remains of physically running the code on the FPGA and verifying the performance gain. After physically integrating our custom instructions we expect a similar performance to that based off of our profiling information.

The custom logic from each of the selected instructions has been integrated into its corresponding wrapper. The final design consumes 6,140 LEs which is only a difference of 2% from our independently generated separate synthesis values. Our maximum obtainable frequency on this FPGA with the design in its current configuration is 63.2MHz.

The application's source code has been modified to reflect the new hardware calls to both internal and external custom instructions. Generic 2-1 instructions are already available as the wrappers are situated within SOPC builder. During compilation **gcc** adds these two new instructions. We are then able to call them using either in-line assembly which can be tedious or C function constructs provided by NIOS for integer based functions int `__builtin_custom_ini (int n, int dataa)`. The function constructs were used. The constructs did a basic variable mapping which is easily seen in the generated assembly.

Since the coprocessors are not connected to the ISA architecture, but are memory mapped on the Avalon bus, simply reading and writing from the assigned accelerator address will initiate and configure the coprocessor. The coprocessor sits in an idle state when not active and is triggered by a bit flag at that address as well. The coprocessor lowers the flag when the operation is complete so the data can be read back into the register file.

The final results were rather surprising as the number of cycles actually required with the custom instructions were within <6% of those obtained by in depth profiling. Using the Performance Counter IP[47] provided by Altera the number of cycles required to run the program was physically counted as 2600935. Fig. 8.4 shows the results of the sample program physically running on the FPGA. The verifiable speedup obtained is 13.59x.

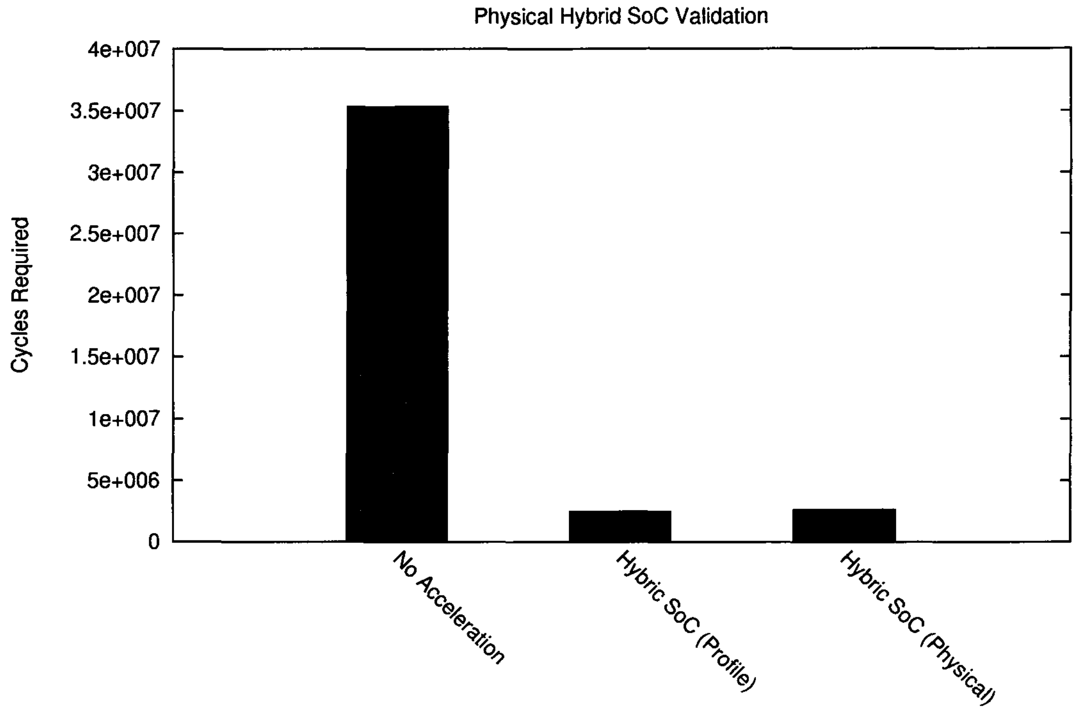


Figure 8.4: Hybrid SoC Validation on FPGA.

8.7 Garnering Further Speedup

It has already been mentioned that a variable clock compared to the system clock is in fact an option for coprocessor based accelerators. For the sake of curiosity we will examine this idea here as it has already been discussed and can be incorporated into developing CS metrics.

Consider the current system configuration with one additional component. We will add a PLL and change the clock rate of the **factorial()** coprocessor. A PLL is a control system that will attempt to generate an output signal based off a relation of the phase of an input or reference signal. Based off our reference signal of 50MHz, we can create an output signal with a phase shift or a different frequency. We will change the frequency of the **factorial()** coprocessor to 60MHz. The FIFO between the Avalon bus and coprocessor will stop any synchronization or timing problems from occurring. The difference is presented in Fig. 8.5. The speedup from the 50MHz Hybrid SoC design is 1.08x and 13.59x from the original one. These results were obtained by physically

running the variably clocked design on the reference board.

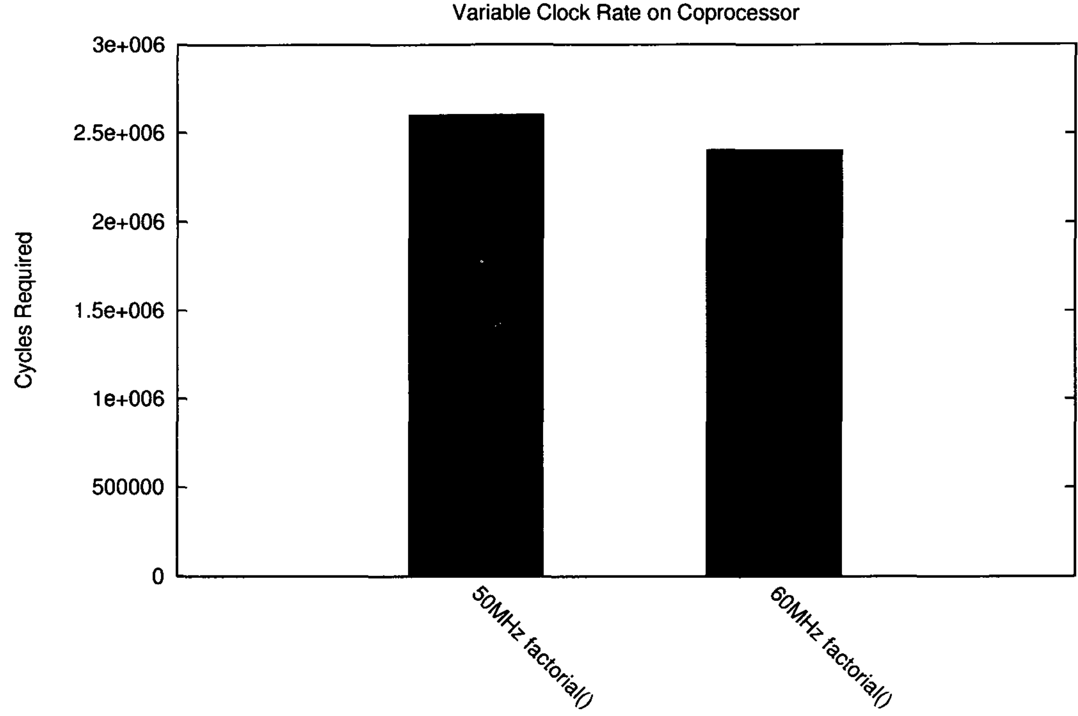


Figure 8.5: Comparison of 50 and 60MHz **factorial()**.

8.8 Concluding Remarks

The performance gains seen in this walk-through have been enormous. Despite providing such results for the sample program, other programs may not fare as well. We will see this in the next section using automated candidate instruction selection algorithms instead of manually selecting candidate hardware functions.

The most difficult portion of this chapter was the effective profiling of the hardware and software with such high precision. The profiling required an understanding of the underlying architecture as simply using **gprof** does not give cycle accurate results. Cycle accurate profiling for this application was extremely difficult as it involved looking at each type of instruction in the program executing at a software and hardware level with ModelSim. Good estimates were obtained by physically generating test systems

and obtaining the metrics through the Performance Counter for common instruction dispatches. The cycle accuracy study looked at the SRAM controller, SDRAM controller, bus pipeline interaction for atomic instructions and worst-case branch fails. The cache was left on during the analysis. Such great effort required takes away from the tools and framework developed for this thesis. The need for a cycle-accurate NIOS II simulator is clear if this is to become a viable work flow and direction to alleviate manual monotony.

The custom instruction wrappers have been created along with an effective flow for all hybrid SoC trade-off needs. With an additional cycle accurate simulator or profiler such tools and design directions may be given more consideration. The development of the custom instruction wrappers makes integrating hardware into a hybrid SoC simple. Knowing this, the future possibly lies in more estimate oriented cycles saved values where this is represented as a best-case/worst-case range in the selection model. This could be very beneficial in situations where the cycles saved of one custom instruction addition may be related to the inclusion of another piece of custom hardware.

Chapter 9

Further Evaluation

9.1 Overview

We have already examined the hybrid SoC idea in the Altera NIOS II design flow and will now attempt to further evaluate both the hybrid SoC algorithm and the underlying framework. The hurdle with evaluating such a work is that the problem is not always bounded and constrained. This is clear as most of the ideas presented thus far have been a generalization under the direction of providing a framework for similar future work. There exist many target architectures with many inherent features and quirks.

We have gone through a complete case study example but to fully test each of these concepts would be an exhaustive task, thus this section has been broken down into three sub-sections. These three tests should offer adequate insight into performance possibilities. By no means are these tests a guarantee of performance as the resulting speedup in reality is based off of the custom instruction candidates provided by the user. The better candidates provided to the framework, the better the hybrid selection.

There are many benchmarks available for the testing of various platforms from embedded to desktop targets. With respect to verifiable comparison, a problem exists where the benchmark comparisons themselves rely heavily on the custom instruction candidates that are given by the user and the target system constraints. A few benchmark applications will be run through automated instruction identification tools to generate custom instruction candidates. This method of evaluation is not a guaranteed best assessment

but can provide a real-world application scenario. The benchmarks used in the application based evaluation come from MiBench[48]. MiBench is a free, open-source and commercially representative embedded benchmark suite that focuses on benchmarks in the industrial areas of automotive, consumer, network, office, security and telecom applications. SPEC2006[49] is another common benchmark suite used throughout the research community, not necessarily for embedded target systems. For example, the included combinatorial optimization benchmark 429.mcf requires 860MB of memory for the 32-bit data model. This is out of the range of most embedded systems. A second problem arises as all SPEC2006 benchmarks obtain their large input data sets at runtime as an argument through **main()**. The input data is not integrated into the executable at compilation. This is not so much a problem in a simulation environment, but a physical implementation without a file system, wrapper or debug UART connection makes the execution very difficult. For these reasons SPEC has been omitted. The benefit of MiBench can be seen as two-fold, realistic industrial applications and certain benchmarks with input data integrated into the executables or generated within the benchmark.

The first evaluation examines the execution time of the solver given ILP scripts of growing complexity. The objective is to show that even larger sets of custom instruction candidates can still be sorted through at a quick pace. The second evaluation looks at two automated custom instruction identification algorithms used for a theoretical target system. It is shown that the number of found custom instructions is small for the given MiBench benchmarks and is easily analyzed by our tool at a fast pace. Thirdly, the Tensilica XPRES[8] compiler will attempt to generate custom instruction candidates for MiBench benchmarks. The target environment is an Xtensa LX based processor supporting both internal processor datapath augmentations and external coprocessors. Tensilica provides a custom C/C++ cross-compiler that can easily accommodate new instructions and the benchmarks being compiled. This is the reason for the selection, as it is infeasible to manually generate a NIOS II system for each benchmark as done in the previous case study. Basic TIE support is available in the hybrid developed SoC tool to be able to obtain area and timing estimates from the identified custom instructions.

We cannot make comparisons between other implementations or hybrid methods in good faith. The only true comparison is that of the hybrid SoC against the base system. Similar custom instruction identification, metrics, compilation and target specifications cannot be guaranteed with the other implementations and are not designed to. Furthermore, other works do not take the approach of accepting agnostic custom candidates

from a user. The user plays a pivotal role in the potential performance gain. Instead the methodology and framework are presented from many angles leaving the reader to see the benefit that may be present by taking such an approach in future SoC designs. We have shown a verifiable improvement on the sample program with a NIOS II target and show in the next subsection a slight improvement on a variety of benchmark applications using current automated instruction identification algorithms and tools. Improvements are based on the comparison of the developed hybrid SoC from the previously defined framework and the base implementation.

9.2 Examining Solver Times

Here we look at the execution time of the solver for scenarios of varying complexity with a global hardware constraint. The number of custom instruction candidates was tested from 10-2000 with the pertinent data shown in Table 9.1. The maximum global hardware constraint for each scenario is shown in the table as well. Each candidate custom instruction has been given a random hardware weight using the specified range with coprocessors receiving an overhead of 200 units. The extra coprocessor latency is 5 clock cycles.

Similarly, the objective values were also assigned using a normalized random function with a constant range of [10-100000] cycles saved. The percent of available instructions that can be modeled both internally and externally is also varied. A percentage is given for each test scenario. This percentage indicates what percentage of total custom instruction candidates can be implemented in the datapath simulating architectural constraints. By doing this there exists the assumption that all custom instruction candidates can be implemented as coprocessors. With regards to ISA based constraints, a maximum of 50 internal and 50 external hardware blocks can be chosen. The results of the trials are shown in Fig. 9.1.

It is generally expected that the number of candidate custom instructions will be less than 100 when either done manually or through an automated mechanism. Considering this, it is safe to say that the solver times are quite fast as each trial ran until an optimal solution was found < 1s. Due to the nature of the branch-and-bound algorithm such results are not guaranteed. With this data it is interesting to show that much larger sets may in fact be readily supported.

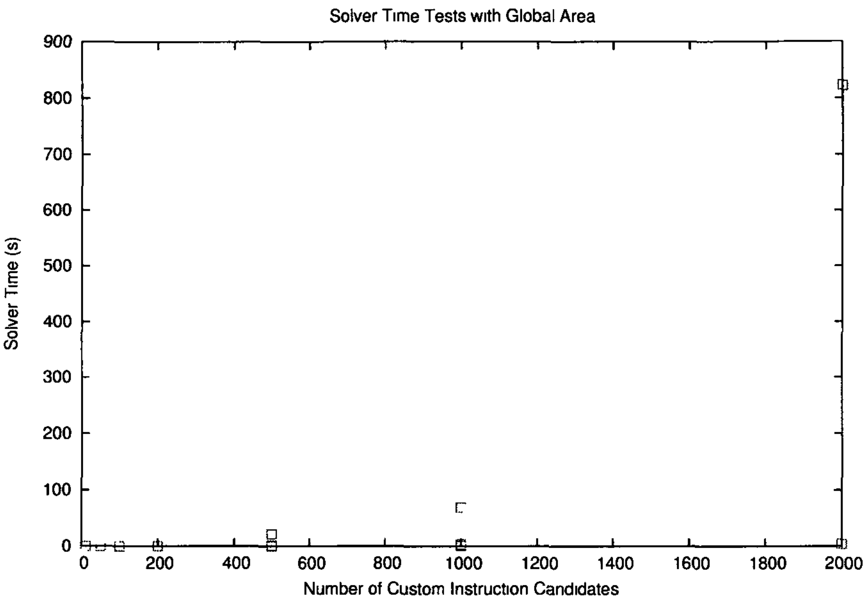


Figure 9.1: Solver Time Trials.

9.3 Theoretical Performance Assessment

In this section we look at two theoretical instruction identification algorithms to provide custom instruction candidates for select MiBench benchmarks. These algorithms were originally designed solely for ISEs. In this work we consider ISEs as possible coprocessor implementations and the algorithms are therefore valid and worthy of analysis as shown in Chapter 3.

Referring back to Fig. 1.4, we are placing this instruction identification algorithm in the box marked *Identification of Custom Instruction Candidates* and feeding the results to the hybrid SoC framework application. These algorithms and their implemented form do not generate VHDL or constraint files. For this test the generation has been done manually from the selected data flow graph sections.

Both CRC32 and FFT from MiBench are analyzed next in a hypothetical scenario with large datasets. These two were selected as the runtime for each custom instruction identification algorithm took less than an hour. Many other benchmarks proved too lengthy having a runtime in excess of 2 days for the identification algorithm passes to complete.

Our hypothetical target system goes as follows. The NIOS II/f processor uses 1,600

Table 9.1: Solver Times with Global Area.

Custom Instruction Candidates	Time (s)	Maximum Hardware	Hardware Range	% of Instructions Both Internal and External
10	0.016	1000	[1-1000]	25
10	0.013	10000	[1-1000]	25
50	0.02	1000	[1-1000]	25
50	0.093	10000	[1-1000]	25
100	0.036	1000	[1-1000]	25
100	0.186	10000	[1-1000]	25
100	0.281	10000	[1-1000]	50
100	0.257	10000	[1-1000]	100
200	0.117	1000	[1-1000]	25
200	0.252	10000	[1-1000]	25
500	0.296	1000	[1-1000]	25
500	1.085	10000	[1-1000]	25
500	1.795	10000	[1-1000]	5
500	0.29	1000	[1-10000]	25
500	0.428	10000	[1-10000]	25
500	21.867	100000	[1-10000]	25
500	1.13	100000	[1-10000]	25
500	0.351	10000	[1000-10000]	25
1000	3.284	10000	[1-1000]	25
1000	68.77	10000	[1-1000]	25
1000	1.216	1000	[1-1000]	25
2000	4.644	10000	[1-1000]	25
2000	823.1	100000	[1-1000]	25

LEs in Cyclone II targets and is included in a FPGA SoC that contains 3,000 LEs. The ISE identification algorithms used are both [11, 12], denoted as Shapiro and Atasu respectively. There is a maximum of 2 coprocessors permitted in this system and 3 augmented internal instructions. The cycle savings metric generated from these algorithms is done on a per instruction basis. The matching hardware implementations are assumed to be one-to-one mappings of the selected DFG segments. These nodes are mapped directly to Altera arithmetic library components. The hardware sizes used can be found in Appendix A with the corresponding node instruction and matching hardware component.

The maximum instruction IO constraint for this system is 2-1 due to hypothetical ISA constraints on both coprocessors and internal instructions. Also shown in the table is the maximum hardware size of a considered instruction within the algorithms.

The results of the FFT benchmark are shown in Table 9.2 and CRC32 in Table 9.3. The two ISE identification algorithms are used to create a list of candidate instructions. The number of instructions found by each algorithm is indicated along with how many were actually used in the internal/coprocessor placement as *Instructions Selected.* The final system speedup is also listed based on the cycles that implementing these instructions will save.

In both cases the most instructions that could be selected were selected. In the first test the number of found instructions in both algorithms was different but the instructions that were selected were found in both, $Atasu \subset Shapiro$. Using the available metrics the best speedup that can be achieved with a hybrid SoC for this FFT problem is 1.0188x. CRC32 had a better outcome with regards to speedup, 1.257x, but was not particularly demonstrative as only two custom instruction candidates were even found and both were selected to be placed internally. These tests are not groundbreaking but show that even these simple cases are easily examined. Larger benchmarks failed in these identification algorithms so these results serve as a proof-of-concept for larger problems with the hope of a greater performance gain as seen previously with manual intervention.

Table 9.2: FFT Hardware Selection for Hybrid SoC.

ISE Algorithm		Maximum HW Size (LE)			
		100	1000	10000	15000
Atasu	*Instructions Found*	8	17	17	17
	Instructions Selected	5	5	5	5
	Best Speedup	1.0061	1.0188	1.0188	1.0188
Shapiro	*Instructions Found*	9	21	23	20
	Instructions Selected	5	5	5	5
	Best Speedup	1.0061	1.0188	1.0188	1.0188

Table 9.3: CRC32 Hardware Selection for Hybrid SoC.

ISE Algorithm		Maximum HW Size (LE)			
		100	1000	10000	15000
Atasu	*Instructions Found*	2	2	2	2
	Instructions Selected	2	2	2	2
	Best Speedup	1.2857	1.2857	1.2857	1.2857
Shapiro	*Instructions Found*	2	2	2	2
	Instructions Selected	2	2	2	2
	Best Speedup	1.2857	1.2857	1.2857	1.2857

9.4 Xtensa Hybrid SoC with XPRES Identification

Tensilica is seen as a leader in custom instruction support. The Xtensa processor easily facilitates the addition of custom instructions through its automated tools and HDL format TIE. Custom hardware designed with TIE can be integrated internally into the datapath or through a specialized coprocessor interface.

The XPRES Compiler[8] is designed to automatically identify parts of a user application that would benefit from hardware acceleration and generate matching TIE. Here we look to use XPRES to automatically identify potential candidate instructions for a hybrid SoC where the set of candidates will be extracted and analyzed by the hybrid SoC framework application and then selected for the Xtensa core.

In particular we are looking for automated fusions. Fusions are the Tensilica term for the merge of multiple instructions into one. Xtensa permits up to 8 coprocessors specified in TIE. XPRES does not generate coprocessor TIE, however, the generated internal TIE can be modified with coprocessor statements[50]. Each identified fusion was profiled using Tensilica's cycle accurate profiler. Both the original internal TIE and the modified coprocessor variant were profiled for each candidate. Both the cycles saved and area required were accurately generated from this profile. Area was returned as a value of gates. Unfortunately, the area is given as the total area of the entire system. The difference of the augmented system and the base system was used as the area constraint for custom instructions in the framework.

Note that FLIX identification was disabled during the running of the XPRES Compiler. FLIX is used to generate parallel VLIW instructions and we wish to emulate a basic RISC architecture. The IO constraints given to XPRES are 2-1. The maximum

number of operations that can be fused was set to 12.

Coprocessors on the Xtensa platform are state based requiring a context switch during execution and a separate register file leaving a large overhead. To deal with this issue and avoid complexity, operands are transferred to the coprocessor register files before coprocessor execution. Since our operations are not state based the context switch of save and restore are not completed on the main processor during coprocessor execution.

The steps of this small evaluation can be summarized as:

1. Profile base program.
2. Configure and execute XPRES.
3. Generate matching coprocessor statements for TIE.
4. Analyze and re-profile system by removing all but one TIE candidate.
5. Repeat Step 4 until all custom instruction candidates as both internal and external implementations have been profiled and examined. Cycles saved, number of gates used and timing data are returned after each profile.
6. Run results of all custom instruction candidates through hybrid SoC framework negating synthesis as we already have our area and timing information.
7. Reactivate selected TIE blocks and profile again for final hybrid SoC speedup.

For this analysis a maximum of 8 coprocessors can be selected and 10 internal instructions with a global area of 100,000 gates. Four benchmarks were examined with the XPRES Compiler. Results in Table 9.4 are based off a final profile of the hybrid SoC using the Tensilica profiler. Those speedups indicated as negligible either had equivalent performance or worse with the candidate hardware available.

By examining the outcome of **stringsearch** it can be easily seen that the communication overhead required for the coprocessor implementation on the Xtensa is too great forcing as many instructions as possible into internal implementations with the remainder falling into coprocessor slots. It seems as though the overhead is almost not worth the execution savings of the coprocessor implementations for these small fusion instructions. Despite the fact that the instruction fusion limit was set at 12 the largest grouping of base instructions was 7 with most consisting of 3 or 4 operations.

Table 9.4: Hybrid SoC with Tensilica.

Benchmarks	Instructions Found	Instructions Selected	Internal Selection	External Selection	Speedup
`fft`	1	0	-	-	Negligible
`crc32`	1	0	-	-	Negligible
`bitcount`	27	18	10	8	1.13
`stringsearch`	11	11	10	1	1.003
`basicmath`	3	0	-	-	Negligible

After running through these benchmarks the XPRES Compiler's performance was not as high as expected. Two of the programs had worse performance and one was on par. `bitcount` was the only benchmark to achieve a good level of candidate instructions and final performance boost for the hybrid SoC. `stringsearch`'s results only offered a minor improvement.

Chapter 10

Extensions

10.1 Overview

This chapter looks at ideas to further extend the model and framework that have been developed thus far. The concepts presented here are considered potential future work directions. Topics of interest include: DMA, modifying overall objectives, introduction of scheduling and the use of an effective PTAS as a solver backup. Some of these ideas address assumptions that have been made so far. Assumptions have included: single-issue instruction processor (RISC), deterministic user application runtime, relatively accurate profiling data, minimal or accountable hazards, known maximum hardware, no upper limit on performance/speedup, cycle accurate processor architecture including stalls, mutually exclusive execution of processor and coprocessors, no memory access, limited IO, known target architecture, available generic custom instruction wrappers, FPGA target and the expectation of optimality from the user with respect to the given solution. The most significant assumption made throughout this work is that of beneficial candidate custom instruction candidates along with correct constraint metrics being provided to the framework by a user.

10.2 Exploiting DMA within Coprocessors

Direct Memory Access (DMA) has been mentioned several times in this document as a potential venue for performance gain. Coprocessors come in different architectures as we have seen. One of the common architectures is that of the bus connection. This was in fact the case for the previous NIOS II case study in Chapter 8. In this configuration it is possible for the coprocessor to access memory units also located on the bus through DMA.

DMA is a feature found in many computers and microprocessors. DMA allows hardware subsystems or peripherals access to system memory while being completely independent of the main processor. Such a feature can come into play for custom instruction candidate selection. Consider the scenario of an array that is stored in memory. Currently this data would have to be loaded into the register file and transferred to the coprocessor a few operands at a time. If this linear pattern is picked up from a data-flow graph or at a functional pointer level we can bypass the processor entirely and solely give the coprocessor a start and end address for its processing. If this is done it is imperative that the data cache writes back all its data along with flushing out the entire processor pipeline to ensure that any store instructions are completed.

Where this is envisioned as most useful is through automated analysis of custom instructions. Most automated ISE methods exclude load and store operations from selection including [11, 12, 20]. If these memory operations are removed from the forbidden node list, those instructions identified with them can be tested as potential coprocessor instructions. Only those with many linear loads or stores will benefit from such an implementation. Consider the example shown next in Fig. 10.2 where 32 bit rows are loaded in a sequential format. These three DAG segments occur one after the other as indicated with the bottom arrows.

For this to be a viable option, DMA controller logic would need to be added to the coprocessor wrapper along with configuration of the primary processor. An adequate method for determining the potential performance increase of using this method or not should be devised.

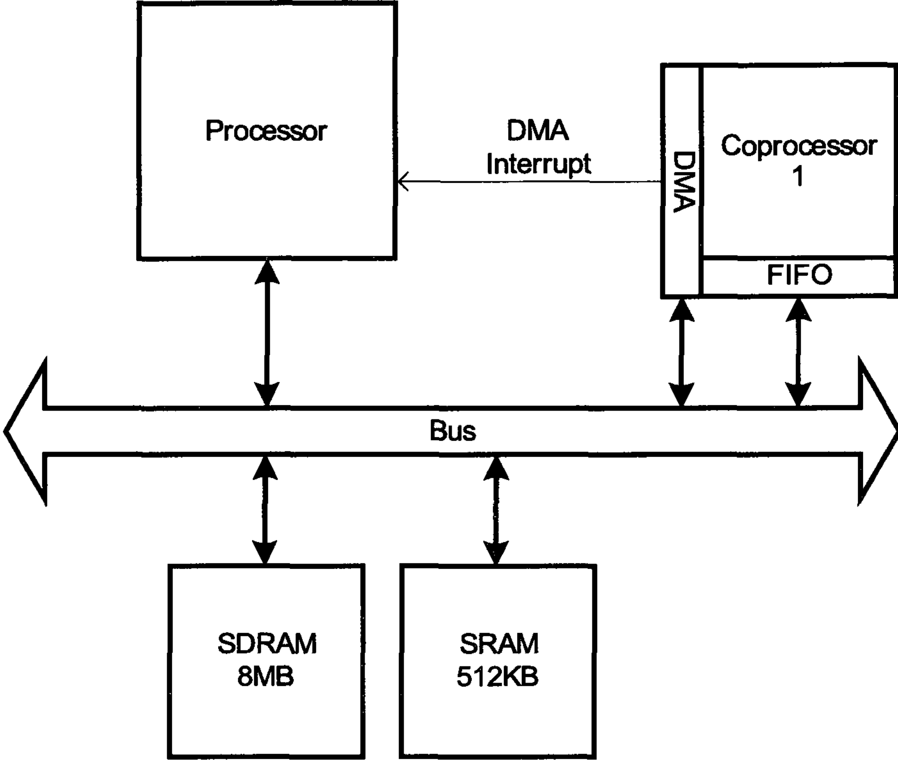


Figure 10.1: DMA Coprocessor.

10.3 Changing Objectives

So far we have been optimizing one objective in particular, performance. Hooks have been provided to alter the optimization ILP model that is generated. The objective can be easily changed to any other consideration. The provided hooks must be used to generate a new model which includes parsing different constraint information from the matching constraint files. Other objectives may include minimizing power, minimizing area or maximizing throughput. If the required metrics can be collected and a proper model generated, the existing framework will make a hybrid SoC selection.

10.4 Multi-Objective Directions

Carrying on with objective functions, it is clear that this entire work has focused around a single objective function. Multi-objective optimization as the name implies is an opti-

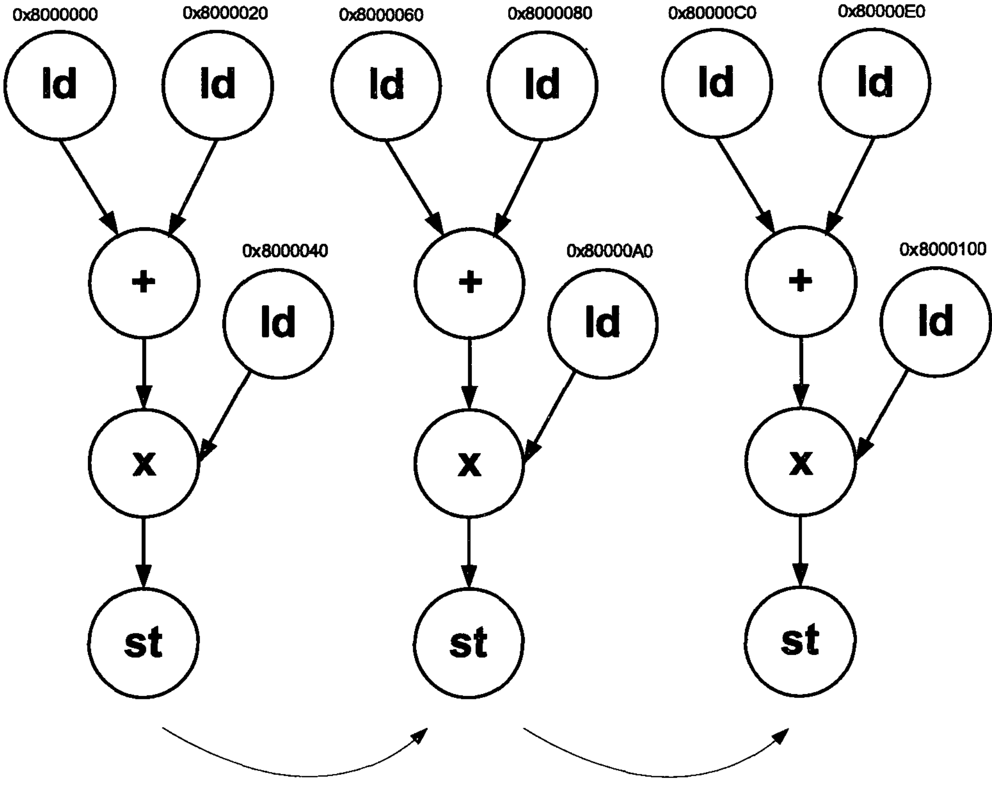


Figure 10.2: DAGs with Linear Memory Load Access.

mization problem where multiple objective functions are being maximized or minimized. An example of extending this work from a multi-objective perspective is the maximization of performance while minimizing area. There are a variety of ways to tackle this problem. Exact algorithms exist for a variety of multi-objective problems including the basic knapsack[25]. A recent trend in multi-objective optimization has been multi-objective Evolutionary Algorithms (MOEA) which offer a fast execution time with an often adequate result[51]. The downside is that there is no optimality guarantee. There are a variety of metaheuristic algorithms for multi-objective problems[52]. Again, metaheuristics cannot guarantee an optimal solution or even a solution at all.

There are many different approaches to adding in this functionality from the solver itself to the underlying solver algorithm: heuristic, exact, guaranteed range, evolutionary or genetic. The support exists already for custom model generation and the calling of a custom solver. This may facilitate integration of such a feature in the future.

10.5 Scheduling

There can be clear parallelism when separate hardware processing units are available. This is the case with the hybrid SoC. The hybrid SoC involves internal custom instructions intertwined into the existing processor datapath and the coprocessor which is in fact a separate processing entity.

By using scheduling information both the processor and coprocessor(s) could execute concurrently. For example, once the coprocessor is called, if no dependencies are upcoming the processor may continue along with the application. Currently the processor stalls if a coprocessor is executing and waits until a result is returned. Investigations into compiler scheduling algorithms may allow for more parallelism in hybrid SoC systems by scheduling concurrent execution.

10.6 Proper Polynomial Time Approximation Scheme

Originally it was planned to use a Polynomial Time Approximation Scheme as a backup if LPSolve were to fail at finding an optimal solution. Failure would occur because of a timeout or infeasible solution. Unfortunately those PTASs mentioned in Chapter 2 for a variety of knapsack problems do not apply to the variability of the current models generated. The current approximation method due to the possible failure conditions is a brute force random assignment within a given time period. The best brute force solution is then compared to a sub-optimal solution if available. This occurrence is highly unlikely as custom instruction candidate data sets are often small as seen by the automated identification and manual identification techniques currently employed. Unfortunately, this may also be due to the smaller benchmarks or test programs that are examined for such works. This was the reasoning behind the inclusion of random hybrid SoC models of various sizes in Chapter 9.

It is of interest to further examine these guaranteed gap approximation schemes for a potential solution that meets all conditions that the current models may contain. Evolutionary algorithms have also been used for similar problems. This is out of the scope of the current work, but is of interest for future analysis.

Chapter 11

Conclusion

11.1 Overview

The hybrid SoC is a potential design space exploration avenue for any SoC that supports both internal processor datapath and coprocessor custom instructions. In this work a user has a list of potential candidate custom instructions in an HDL format. These custom hardware blocks can be obtained through automated identification solutions or manual intervention. Each HDL file's characteristics and limitations are specified in a series of matching constraint files. The group of candidate HDL and constraint files are fed into the hybrid SoC framework tool. A series of target FPGA platforms are selected by the user. Each candidate is automatically synthesized for the given FPGA make to obtain area and timing constraints. For each target FPGA selected an ILP model is generated and dispatched where multiple instantiated solvers attempt to select the best set of hybrid SoC custom instructions to offer the highest performance gain.

The final automated solution indicates which custom instructions should be included in the final design and whether they should be placed within the datapath or outside as a coprocessor.

The key tenets shown in this work have been:

- Development of a hybrid SoC framework to facilitate candidate selection while integrating into existing industrial tools.
- Mathematical trade-off given known or obtainable metrics for hybrid solutions.

- Development and integration of a customizable ILP hybrid SoC model into the development framework.
- Consideration of both internal and external custom instruction solutions in the same solution space.

11.2 Discussion

11.2.1 Results

It has been shown that if good candidates are given to the hybrid SoC framework a large speedup can be obtained. Unfortunately, in an attempt to use existing automated candidate identification tools and algorithms the results were not as great as expected yielding speedups of well less than 2x. By manually identifying custom instructions of the sample case study program a speedup approaching 14x was verifiably achieved. The high dependence on good candidate custom instructions is key to achieving a large performance gain.

In all cases the developed hybrid SoC framework looked at all candidates along with target constraints and was able to make a proven optimal decision on where and how the custom hardware blocks should be integrated. This functionality was the main goal of the described work. The effectiveness of the work is hard to gauge but is clearly demonstrated in the proof-of-concepts as a viable option for achieving a performance gain.

11.2.2 Concluding Remarks

The hybrid SoC offers a wide breadth of design possibilities especially considering the variety of available instruction candidates and target architectures. The problem was tackled from a general perspective while offering individual insights especially through the many examples given.

If a user is using such a tool to select which custom instruction hardware to use and how to best integrate the selection, the outcome is expected to be accurate with respect to the used metrics. This can be guaranteed by the selection algorithm and solver. However, if the metrics provided are not accurate then the results obtained may not reflect the actual outcome expected by the user. As the tool relies on an accurate

cycle level model of the system, better and more accurate simulation and profiling tools should be considered part of the flow in profiling custom instructions. If to obtain these metrics exhaustive manual intervention is required, then the overall benefit to the user begins to diminish.

A large feature of the developed framework application is its ability for customization. The hooks provided allow a multitude of extensions not even considered for this individual piece of work. Such customizations include new or altered hybrid SoC model generation initiatives to a completely different solver. The support for existing user flows is there, allowing the user to focus solely on hybrid SoC algorithms without having to focus on the remainder of the toolchain.

With the given case study it was shown how a basic well known target SoC architecture could be extended to allow for automated hybrid SoC consideration. Hopefully in the future such generic wrappers become a part of more industrial systems facilitating the use of coprocessors and internal augmentations in a non-mutually exclusive manner.

Appendix A

Altera Cyclone II Hardware Sizes

Operation	Datatype	HW Size(LE)	Clock Cycles	Altera Component Name
ADD	float	975	14	altfp_add_sub
CONVFI	float	457	1	altfp_convert
CONVFS	float	457	1	altfp_convert
CONVFT	float	689	1	altfp_convert
CONVFX	float	84	1	altfp_convert
CONVSF	float	342	1	altfp_convert
CONVUF	float	342	1	altfp_convert
DIVS	float	5941	33	alt_fp_div
DIVU	float	5941	33	alt_fp_div
FLOATCONST	float	0	0	zeros changed to hold value
INTCONST	float	0	0	zeros changed to hold value
MUL	float	926	5	alt_fp_mul
NEG	float	2897	20	altfp_inv
SUB	float	975	14	altfp_add_sub
TSTEQ	float	58	3	altfp_compare
TSTGES	float	97	3	altfp_compare
TSTGEU	float	97	3	altfp_compare
TSTGTS	float	98	3	altfp_compare
TSTGTU	float	98	3	altfp_compare
TSTLES	float	97	3	altfp_compare
TSTLEU	float	97	3	altfp_compare

TSTLTS	float	97	3	altfp_compare
TSTLTU	float	97	3	altfp_compare
TSTNE	float	58	3	altfp_compare
ZEROS	float	0	0	zeros
ADD	int	49	1	lpm_add_sub
BAND	int	32	0	band
BNOT	int	0	0	-
BOR	int	32	0	lpm_or
BXOR	int	32	0	lpm_xor
CONVFI	int	457	1	altfp_convert
CONVFS	int	457	1	altfp_convert
CONVIT	int	342	1	altfp_convert
CONVSF	int	342	1	altfp_convert
CONVSX	int	0	0	-
CONVUF	int	342	1	altfp_convert
CONVZX	int	0	0	-
DIVS	int	1343	7	lpm_divide
DIVU	int	1550	7	lpm_divide
INTCONST	int	0	0	-
LSHS	int	384	1	-
LSHU	int	384	1	barrel_shifter
MODS	int	1426	0	MODS
MODU	int	1114	0	MODU
MUL	int	1598	7	lpm_mult
NEG	int	50	1	-
RSHS	int	384	1	-
RSHU	int	384	1	barrel_shifter
SUB	int	53	1	lpm_add_sub
SUBREG	int	0	0	-
TSTEQ	int	21	0	lpm_compare
TSTGES	int	54	0	lpm_compare
TSTGEU	int	54	0	lpm_compare
TSTGTS	int	32	0	lpm_compare
TSTGTU	int	32	0	lpm_compare

TSTLES	int	54	0	lpm_compare
TSTLEU	int	54	0	lpm_compare
TSTLTS	int	32	0	lpm_compare
TSTLTU	int	32	0	lpm_compare
TSTNE	int	21	0	lpm_compare
ZEROS	int	0	0	-

Appendix B

NIOS II Case Study Test Program

```
#include <stdio.h>
#include "system.h"
#include "nios2.h"
#include "altera_avalon_performance_counter.h"

/*Function Declarations*/
int power3(int x);
int addUpto(int x);
int factorial(int x);
int power4(int x);
int mac(int a, int b, int c);
int limit(int a);
int func1(int a, int b);
int func2(int a, int b);
int func3(int a, int b);
int func4(int a, int b);

int main()
{
  int temp,temp0;
  int i,r;

  PERF_RESET (PERFORMANCE_COUNTER_BASE);              //Reset Performance Counters to 0
  PERF_START_MEASURING (PERFORMANCE_COUNTER_BASE);    //Start the Counter
  PERF_BEGIN (PERFORMANCE_COUNTER_BASE,2);            //Start the overhead counter
  PERF_BEGIN (PERFORMANCE_COUNTER_BASE,1);            //Start Function Counter
  PERF_END (PERFORMANCE_COUNTER_BASE,2);              //Stop the overhead counter

  srand(1),
  printf("Functional Profile Testing\n");

  for(i=0; i<1000;i++)
  {
    r=rand() % (10 - 1 + 1) + 1;
    temp=power3(r);
    temp0=addUpto(temp);
    temp0=factorial(temp);
    temp0=power4(r);
    temp0=mac(r, temp, temp0);
    temp0=limit(r);
    temp0=func1(temp,r);
```

```
    temp0=func2(temp,r);
    temp0=func3(temp,r);
    temp0=func4(temp,r);

  }
  PERF_END (PERFORMANCE_COUNTER_BASE,1);           //Stop the Function Counter
  PERF_STOP_MEASURING (PERFORMANCE_COUNTER_BASE);  //Stop all counters

  perf_print_formatted_report((void *)PERFORMANCE_COUNTER_BASE, ALT_CPU_FREQ, 2,
  "Function Calls","PC overhead");

  return 0;
}


int power3(int x)
{
    int y;
    y=x*x*x;

   return y;
}

int addUpto(int x)
{
    int y=0;
    int count;
    for(count=1;count<=x;count++)
    {
        y=y+count;
    }
   return y;
}

int factorial(int x)
{
    int y=1;
    int count;
    for(count=1;count<=x;count++)
    {
        y=y*count;
    }
   return y;
}

int power4(int x)
{
    int y;
    y=x*x*x*x;

   return y;
}

int mac(int a, int b, int c)
{
    return(a+(b*c));
}

int limit(int a)
{
    if(a>5){
     return 10;
    }else{
```

```
     return 0;
    }
}

int func1(int a, int b)
{
    return(3*a+4*b);
}

int func2(int a, int b)
{
    return(a*a+4*b);
}

int func3(int a, int b)
{
    return((a+b)*2);
}

int func4(int a, int b)
{
    return((a-b)*2);
}
```

Appendix C

Generated ILP For NIOS II Case Study

```
Maximize
48418power3_P + 46218power3_C + 15895245addUpto_P + 15893015addUpto_C + 16854941factorial_P + 16852748factorial_C
+ 61161power4_P + 58961power4_C + 2048mac_C + 27108limit_P + 24908limit_C + 1114func1_P + 1794func2_P + 11func2_C
+ 2086func3_P + 226func3_C + 1869func4_P + 206func4_C

Subject to

\Global Hardware Constraint
193power3_P + 267power3_C + 352addUpto_P + 606addUpto_C + 355factorial_P + 609factorial_C + 235power4_P
+ 488power4_C + 521mac_C + 87limit_P + 341limit_C + 235func1_P + 212func2_P + 466func2_C+ 221func3_P
+ 475func3_C + 230func4_P + 484func4_C <=14381

\Non-Duplication Constraints
power3_P + power3_C <=1
addUpto_P + addUpto_C <=1
factorial_P + factorial_C <=1
power4_P + power4_C <=1
limit_P + limit_C <=1
func2_P + func2_C <=1
func3_P + func3_C <=1
func4_P + func4_C <=1

\ISA Limitations
power3_P + addUpto_P + factorial_P  + power4_P  + limit_P + func1_P + func2_P + func3_P + func4_P  <=2
power3_C + addUpto_C + factorial_C  + power4_C + mac_C + limit_C + func2_C + func3_C + func4_C  <=2

Binary
power3_P
power3_C
addUpto_P
addUpto_C
factorial_P
factorial_C
power4_P
power4_C
mac_C
limit_P
limit_C
func1_P
```

```
func2_P
func2_C
func3_P
func3_C
func4_P
func4_C
```

Appendix D

Glossary of Terms

ASIP Application-Specific Instruction Set Processor - A processor tailored towards a particular application. ASIPs typically involve the use of ISEs forming custom instructions that are specific to the target application.

BIP Binary Integer Program - An ILP with only integer values of 0 or 1.

Basic Blocks - A code portion that has only one exit and one entry point. Jump instructions cannot be located within a basic block.

CFG Control Flow Graphs - Representation of all paths that might be traversed by a program during execution.

CISC Complex Instruction Set Computer - An ISA in which instructions may contain many low-level operations. RISC instructions on the other hand only contain one operation.

Coprocessor - Also known as a hardware accelerator. External processing unit used to supplement the features of the primary processor.

DAG Directed Acyclic graphs - A directed graph with no cycles.

DFG Data Flow Graphs - Graphical representation of data dependencies that occur within a program during execution.

DMA Direct Memory Access - Hardware feature giving primary or secondary hardware units direct access to system memory through a customized DMA controller.

DSE Design Space Exploration - The concept of exploring multiple design decisions within hardware software co-design.

FPGA Field Programmable Gate Array - Configurable hardware device.

HDL Hardware Description Language - Formal description of digital logic.

Hot Spots - Areas of an application where high computational complexity or a large portion of execution time occur. Hot spots can be determined through profiling or low-level analysis. Hot spots may indicate which areas of a program may benefit from customization.

Hybrid SoC - A SoC where the defined processor has an ISA that has been augmented with coprocessor and customized internal datapath instructions. The new hardware is found within the SoC.

ILP Integer Linear Program - A mathematical optimization problem with variables restricted to the integer domain.

IO Input Output - A constraint defining how many inputs and outputs a given hardware block can support.

ISA Instruction Set Architecture - Definition of a platforms architecture especially from the programming side by specifying registers, instructions, data types, memory and addressing.

ISE Instruction Set Extension - The addition of custom instructions to an ISA. Usually done to garner a performance gain using matching custom hardware.

LE Logic Element - Main building block of Altera FPGAs forming the fabric that a hardware design is converted into. Logic elements provide a general metric of how much hardware is being used on a Altera chip.

LP Linear Program - A mathematical optimization problem with unrestricted variables.

NP-Hard Non-Deterministic Polynomial-Time Hard - Colloquially a problem that is at least as hard as the hardest problems in the domain of not having a solution that can execute in polynomial time.

PTAS Polynomial-Time Approximation Scheme - An approximation algorithm for optimization problems, notably those which are NP-hard.

RISC Reduced Instruction Set Computer - An ISA with a simple instruction base particularly known for not allowing memory access outside the realm of load-store instructions.

RTL Register Transfer Level - Description of synchronous digital logic in a circuit based on the flow of signals.

SIMD Single Instruction Multiple Data - A categorization of parallel computing within Flynn's taxonomy where multiple processing elements perform the same operation on multiple data.

SoC System-on-Chip - Integration of many computing components onto a single chip. This is easily facilitated on the FPGA platform.

VHDL VHSIC Hardware Description Language - A hardware description language abstraction of RTL.

VLIW Very Long Instruction Word - An ISA which takes advantage of instruction level parallelism by scheduling operations directly into the instructions. Instructions are of variable width to accommodate this feat.

References

[1] J. Parri and S. Ratti, "Trigonometric function approximation neural network based coprocessor," in *Microsystems and Nanoelectronics Research Conference, 2009. MNRC 2009. 2nd*, oct. 2009, pp. 148 –151.

[2] V. Thareja, M. Bolic, and V. Groza, "Design of a Fuzzy Logic Coprocessor using Handel-C," in *Soft Computing Applications, 2007. SOFA 2007. 2nd International Workshop on*, aug. 2007, pp. 83 –88.

[3] J. Parri, J. Desmarais, D. Shapiro, M. Bolic, and V. Groza, "Design of a Custom Vector Operation API Exploiting SIMD within Java," in *Canadian Conference on Electrical and Computer Engineering*, may. 2010.

[4] R. Gonzalez, "Xtensa: a configurable and extensible processor," *Micro, IEEE*, vol. 20, no. 2, pp. 60 –70, mar/apr 2000.

[5] Altera. (2008) Nios II Custom Instruction User Guide. [Online]. Available: http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf

[6] M. Bass and C. Christensen, "The future of the microprocessor business," *Spectrum, IEEE*, vol. 39, no. 4, pp. 34 –39, apr 2002.

[7] X. Guan, H. Lin, and Y. Fei, "Design of an application-specific instruction set processor for high-throughput and scalable fft," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, april 2009, pp. 1302 –1307

[8] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, ser. CASES '03. New York, NY, USA: ACM, 2003, pp. 137–147.

[9] D. Goodwin, S. Leibson, and G. Martin, *Customizable Embedded Processors.* Morgan Kaufmann Publishers, 2007, ch. Automated Processor Configuration and Instruction Extension, pp. 123–124.

[10] W. Amme and E. Zehendner, "Experiences in analyzing data dependences for programs with pointers and structures," in *Euro-Par'97 Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 1300. Springer Berlin / Heidelberg, 1997, pp. 342–346. [Online]. Available: http://www.springerlink.com/content/w441h24627725838/

[11] D. Shapiro, "Design and Implementation of Instruction Set Extension Identification for a Multiprocessor System-on-chip Hardware/Software Co-design Toolchain," Master's thesis, University of Ottawa, 2009.

[12] K. Atasu, O. Mencer, W. Luk, C. Ozturan, and G. Dundar, "Fast Custom Instruction Identification by Convex Subgraph Enumeration," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, July 2008, pp. 1–6.

[13] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 7, pp. 1209 –1229, july 2006.

[14] R. Tervo. (1998) Exploring the Intel 8087. [Online]. Available: http://www.ee.unb.ca/tervo/ee6373/dos8087.htm

[15] Xilinx, "MicroBlaze Soft Processor v7.20 Frequently Asked Questions," Tech. Rep., 2009. [Online]. Available: http://www.xilinx.com/products/design_resources/proc_central/microblaze_faq.pdf

[16] H. Rosinger. (2004) Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link(FSL) Channel. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp529.pdf

[17] A. Gaisler, "Grlib ip core users manual," Tech. Rep., 2010. [Online]. Available: http://www.gaisler.com/products/grlib/grip.pdf

[18] *Cascade: Programmable Application Coprocessor Generation*, Critical Blue, 2007. [Online]. Available: http://www.criticalblue.com/criticalblue_products/pdf/CascadeProductOverviewMay2007.pdf

[19] S. Sang, X. Li, and Y. Ye, "Automatic Instruction Generation for Application Specific Co-Processor," in *ASIC, 2005. ASICON 2005. 6th International Conference On*, vol. 2, oct. 2005, pp. 934 –938.

[20] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Design Automation Conference, 2003. Proceedings*, june 2003, pp. 256 – 261.

[21] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr, "Architecture implementation using the machine description language lisa," in *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.*, 2002, pp. 239 –244.

[22] C. Price, *MIPS IV Instruction Set*, 1995.

[23] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver, *Combinatorial Optimization.* John Wiley & Sons, 1998.

[24] G. Dantzig and M. Thapa, *Linear programming 1: introduction.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.

[25] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems.* Springer, 2004.

[26] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, vol. 4, no. 4, pp. 373–395, December 1984. [Online]. Available: http://www.springerlink.com/content/h78g4755w1441080/

[27] D. Spielman and S. Teng, "Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time," *J. ACM*, vol. 51, no. 3, pp. 385–463, 2004.

[28] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver, *Combinatorial Optimization.* John Wiley & Sons, 1998.

[29] R. M. Karp, *50 Years of Integer Programming 1958-2008.* Springer Berlin Heidelberg, 2010, ch. Reducibility Among Combinatorial Problems, pp. 219–241. [Online]. Available: http://www.springerlink.com/content/l45h7011865p0257/

[30] J. Wu, J Williams, and N. Bergmann, "An ILP formulation for architectural synthesis and application mapping on FPGA-based hybrid multi-processor SOC," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, sept. 2008, pp. 451 –454.

[31] F. Sun, N. Jha, S. Ravi, and A. Raghunathan, "Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors," in *VLSI Design, 2005. 18th International Conference on*, jan. 2005, pp. 551 – 556.

[32] F. Sun, S. Ravi, A. Raghunathan, and N. Jha, "Custom-instruction synthesis for extensible-processor platforms," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 23, no. 2, pp. 216 – 228, feb. 2004.

[33] ——, "Hybrid custom instruction and co-processor synthesis methodology for extensible processors," in *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, jan. 2006, p. 4 pp.

[34] ——, "A Synthesis Methodology for Hybrid Custom Instruction and Coprocessor Generation for Extensible Processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 11, pp. 2035 –2045, nov. 2007.

[35] S. Martello and P. Toth, "A Branch and Bound algorithm for the zero-one multiple knapsack problem," *Discrete Applied Mathematics*, vol. 3, no. 4, pp. 275 – 288, 1981.

[36] D. Pisinger, "An exact algorithm for large multiple knapsack problems," *European Journal of Operational Research*, vol. 114, no. 3, pp. 528 – 541, 1999.

[37] C. Chekuri and S. Khanna, "A PTAS for the multiple knapsack problem," in *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '00. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000, pp. 213–222. [Online]. Available: http://portal.acm.org.proxy.bib.uottawa.ca/citation.cfm?id=338219.338254

[38] A. Jerraya and M. Rahmouni, *Behavioral Synthesis and Component Reuse with VHDL*. Springer, 1997.

[39] Xilinx, *Multi-Port Memory Controller (MPMC)(v6.00.a)*, 2009. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf

[40] L. Pozzi and P. Ienne, "Exploiting Pipelining to Relax Register-File Port Constraints of Instruction-Set Extensions," in *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, 2005, pp. 2–10.

[41] (2010) Lp solve reference guide. [Online]. Available: http://lpsolve.sourceforge.net/5.0/index.htm

[42] Altera. (1996) LPM Quick Reference Guide. [Online]. Available: http://www.altera.com/literature/catalogs/lpm.pdf

[43] Xilinx. (2007) ISE 9.2i TCL Quick Reference Guide. [Online]. Available: http://www.xilinx.com/products/design_tools/logic_design/implementation/ise9_qrefguide.pdf

[44] Altera, *Quartus II Handbook*, 2010. [Online]. Available: http://www.altera.com/literature/hb/qts/qts_qii52003.pdf

[45] (2010) Catapult C Synthesis. Mentor Graphics. [Online]. Available: http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/

[46] Altera. (2008) NIOS II Custom Instruction User Guide. [Online]. Available: http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf

[47] ——. (2010) Embedded Peripherals IP User Guide. [Online]. Available: http://www.altera.com/literature/ug/ug_embedded_ip.pdf

[48] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, 2 2001, pp. 3 – 14.

[49] (2010) SPEC CPU2006. Standard Performance Evaluation Corporation. [Online]. Available: http://www.spec.org/cpu2006/

[50] Tensilica, *Xtensa Processor Extensions Synthesis (XPRES) Compiler*, 2007.

[51] K. Tan, E. Khor, and T. Lee, *Multiobjective Evolutionary Algorithms and Applications.* London, , GBR: Springer-Verlag, 200501 2005, iD: 10133709.

[52] M. G. Resende, J. P. de Sousa, and A. Viana, *Metaheuristics : Computer Decision-Making.* Kluwer Academic Publishers, 2004.