uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Daniel Shapiro

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Design and Implementation of Instruction Set Extension Identification for a Multiprocessor System-on-chip Hardware/Software Co-design Toolchain

TITRE DE LA THÈSE / TITLE OF THESIS

Dr. M. Bolic

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

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

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Dr. T. Pearce

Dr. Y. Groza

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Design and Implementation of Instruction Set Extension Identification for a Multiprocessor System-on-chip Hardware/Software Co-design Toolchain

by

Daniel Shapiro

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the degree of Masters in Electrical Engineering

January 20th, 2009

School of Information Technology and Engineering

Faculty of Engineering

University of Ottawa

Your file Votre référence
ISBN: 978-0-494-51858-8
Our file Notre référence
ISBN: 978-0-494-51858-8

Abstract

Designers of embedded systems are faced with the challenge of creating systems containing customized processors. Efficient algorithms for the automated identification of extensions to processor instruction sets are needed, as well as flexible tools for the specification and generation of customized embedded systems.

This work on instruction set extension identification is part of a larger effort to design a multiprocessor system design tool. An algorithm for the identification of instruction set extensions is presented along with four approaches for improving the state of the art. In this research we addressed two important aspects of instruction set extension identification: reducing compiler execution time and identifying more instructions. Using the approaches described here, we achieved a compiler execution time speedup over the baseline compiler configuration as high as 3.8 times, and showed that compared to another popular approach our method identifies between 9.5 and 20 times the number of instruction set extensions.

Acknowledgments

B'Siyata D'shmaya. I thank my beautiful wife Liora for her love and support. We have just started on a long journey together through life, and already I can see that you will be a light unto me for all time. As well, I offer heartfelt thanks to my mother and father for everything. You opened my eyes to the world, read me my first book, and watched me take my first steps. It means a lot to me. I thank my supervisor Dr. Miodrag Bolic who has provided me with his trust and advice, and my colleagues and professors for their help on this project. Finally I thank my family for their encouragement, and specifically my brother Sidney who introduced me to the wonderful world of computers.

I would like to acknowledge CMC Microsystems (www.cmc.ca) for providing *workstations, software, support and FPGAs* as part of the System-on-Chip Research Network (SOCRN) infrastructure, enabled through grants from the Canada Foundation for Innovation (CFI) and provincial agencies. CMC is a non-profit corporation primarily funded by Natural Sciences and Engineering Research Canada (NSERC), with matching contributions from industry.

I would like also to acknowledge Natural Sciences and Engineering Research Canada (NSERC) for their support of this project.

Table of Contents

Table of Figures

Table of Tables

Glossary

- ADPCM – Adaptive Differential Pulse-Code Modulation.
- AES – Advanced Encryption Standard.
- AfterLateRewriting – A hook in the COINS compiler that describes one point at which a pass may be set to execute.
- ALU – Arithmetic Logic Unit.
- AMBA – Advanced Microcontroller Bus Architecture.
- ASIP – Application Specific Instruction-set Processor.
- BB - Basic Block.
- CFG – Control Flow Graph.
- CFU - Custom Functional Unit.
- COINS – COmpiler INfraStructure. A compiler infrastructure programmed in Java.
- CoVis – Coins Visualization. A package for viewing COINS compiler data graphically.
- CPLEX – A commercial math programming engine.
- CPX – A tool for finding common ISEs in a dataflow graph.
- CRC – Cyclic Redundancy Check. Contained in EEMBC benchmark.
- CSE – Common Subexpression Elimination
- Cygwin – A Linux emulator for Windows.
- DAG – Directed Acyclic Graph.
- DES – Data Encryption Standard.
- DFG – Dataflow Graph.
- DSP – Digital Signal Processing.
- DSPStone – A DSP benchmark available from [5]. It measures code size and execution time when compared to a human-generated reference sample.
- EasyCore – An early prototype of the SHIRA tool.
- Eclipse IDE – A free and open-source software development platform.
- EEMBC - Embedded Microprocessor Benchmark Consortium. A benchmark suite.
- ESL – Electronic System-Level.
- FFT – Fast Fourier Transform.
- FPGA – Field Programmable Gate Array.
- GCC – GNU Compiler Collection.
- GNU – GNU's Not Unix
- GPS – Global Positioning System.
- Graphviz – A graph visualization tool.
- Grappa – A Java graphing package.
- GSM – Global Standard Mobile. A benchmark that performs audio encoding and decoding according to the GSM standard.
- GUI – Graphical User Interface.
- HDL – Hardware Description Language.
- HIR – High Intermediate Representation.
- IDE – Integrated Development Environment.
- IDWT – A benchmark that runs the Inverse Discrete Wavelet Transform.
- ILP – Integer Linear Program.
- IP-core – Intellectual Property core. A standalone hardware component.
- IR – Intermediate Representation.
- ISA – Instruction Set Architecture.
- ISE – Instruction Set Extension.

- ISEGEN – A tool for generating Instruction Set Extensions based on the analysis of C input.
- ISS – Instruction Set Simulator.
- JNI – Java Native Interface.
- JUNG – Java Universal Network/Graph Framework.
- LCC – Little C Compiler.
- LEON3 – A 32-bit SPARC V8e processor [82].
- LINDO – A commercial IDE for mathematical modeling which includes a model solver.
- LIR – Low Intermediate Representation.
- LP – Linear Programming.
- MDES – The machine description model of the Trimaran compiler.
- MDMS – Modular Design of Multiprocessor Systems.
- MediaBench – A free benchmark suite.
- MiBench – A free benchmark suite described in [10].
- MILP – Mixed Integer Linear Program.
- MinGW - Minimalistic GNU for Windows. A free and open source project that facilitates the use of gcc in windows.
- miniMIPS – A processor implementing a subset of the MIPS ISA, available from OPENCORES.
- MIPS - Microprocessor without Interlocked Pipeline Stages. A RISC processor architecture.
- MPEG2 Encoder – A video processing benchmark.
- MPEG4 - A video processing benchmark.
- MPI – Message Passing Interface.
- MP-SOC – MutiProcessor System-On-Chip.
- Nauty – A graph isomorphism discovery tool, available from [117].
- NetBench – A benchmark for testing the response of a server to heavy traffic and disk access.
- OpenMP – A standard for shared memory communication.
- Quartus-II – An HDL design entry IDE from Altera.
- RISC – Reduced Instruction Set Computer.
- RT-OS – Real-Time Operating System.
- SDK – Software Development Kit.
- SHA – A benchmark that runs the SHA hash function.
- SHIRA - Software/Hardware Implementation and Research Architecture.
- SING – Simulator and Interconnection Network Generator.
- SNU-RT – A benchmark suite for the analysis of Worst-Case Execution Time (WCET) of a program.
- Spec2000int – A set of integer benchmarks.
- SSA – Static Single Assignment.
- TIE – Tensilica Instruction Extension.
- TLM – Transaction Level Model.
- TMD – Target Machine Description.
- TMD-MPE – A hardware engine that runs TMD-MPI.
- TMD-MPI – A hardware and software package which facilitates MPI communication in embedded multiprocessors.
- UTDSP – A benchmark suite for evaluating C code targeted at DSP architectures.
- Verilog – An HDL.
- VHDL - VHSIC Hardware Description Language. An HDL.
- VHSIC - Very High Speed Integrated Circuit.
- WISHBONE B.3 – A bus interconnect standard.

1 Introduction

Instruction Set Extension (ISE) identification is the process of analyzing a program to discover an instruction set which efficiently executes the program. New instructions can be implemented in hardware to speed up the execution of the program. Most processors have a defined and fixed instruction set. However, there is a trend towards the adoption of extensibility of instruction sets as is demonstrated by the instruction extension slot of the NIOS II processor [25], the Mimosys Clarity tool [110], the Xtensa architecture [106], and others.

ISE identification is applied when a processor which is known to perform a specific set of time-critical tasks does not satisfy execution time, power consumption, or real-time requirements on these tasks. There often is a constraint on the hardware size of the processor (or the multiprocessor embedded system that the processor is a part of). The goal in ISE identification is to find a "good" set of functional units to add to the processor to help it meet the real-time requirements. One good custom instruction will provide a speedup in the execution time of the tasks performed by the processor. A good set of custom instructions will improve the execution time of the time critical tasks in the program while satisfying the constraints on the system. These constraints may include metrics such as power consumption and clock frequency. From the perspective of the compiler an ISE is an instruction set extension, while from the perspective of the hardware it is the addition of functional units to the ALU and instruction decoding hardware in the instruction decode stage of the processor.

Now that the goal to identify instruction set extensions has been described, let us look at the tools and methods that were used for this work. ISE identification was implemented in the COINS compiler, a retargetable compiler written in Java. Four approaches to speed up the execution time

of the ISE identification algorithm are described and tested in this thesis. The contributions of this thesis are the implementation of an ISE identification algorithm as a pass in the COINS compiler, experiments evaluating four approaches to decrease the execution time or improve the results of the ISE identification algorithm, an example of a combination of these experiments, and analysis of the ISE identification problem.

In order to verify the correct behaviour of the compiler and ISE identification algorithm, a simple program was used as a test case. The parameters of the compiler were varied to further verify the implementation. The FFT, AES and SHA benchmarks from the MiBench benchmark suite were run through the modified COINS compiler to gauge the performance of the ISE algorithm.

Of the four approaches to improve the state of the art, one approach was a cutoff for the execution time of a part of the instruction set extension identification algorithm. Another approach tested was the pruning of low speedup ISEs, reducing the search space of the algorithm. An approach to parallelize the algorithm using multiple threads is described. Fast compilation time will be discussed in this thesis in terms of the speed of instruction set extension identification in the COINS compiler. Finally, our approach which identifies all valid ISEs is compared to an approach which identifies only ISEs which do not overlap in the dataflow graph of a basic block. The results and analysis presented herein include the execution time of the ISE identification algorithm, the number of the identified instruction set extensions, and the sum for all identified ISEs of the latency of the ISE minus the longest path through the ISE. We call this the latency gain.

The hardware latency of an ISE is defined as the longest path through the ISE in clock cycles. We count the longest path length as the largest sum of operation latencies through the ISE. The software latency is equal to the sum of the latencies of each operation in the ISE. The latency gain is

then the software latency minus the hardware latency. This represents the number of cycles difference between running a code segment with and without the ISE being added to the processor ISA. Another way to look at it is the number of cycles saved by one call to the ISE. A program is composed of several small sequential code segments called basic blocks that are described later on in this thesis. The latency gain of a basic block is equal to the sum of the latency gain of each ISE in the basic block. The saved latency for a program is then the sum for each basic block of the basic block latency gain times the number of calls to the basic block during the execution of the program. Execution statistics are often available from a profiling tool which counts the number of calls to each basic block in the program. We define all of these definitions more formally in the following way:

$$hardware\,latency(ISE_I) = longest\,path\,through\,nodes\,in\,ISE$$
$$software\,latency(ISE_I) = \sum latency(node_N) \forall N \in ISE_I$$
$$latency\,gain(ISE_I) = software\,latency(ISE_I) - hardware\,latency(ISE_I)$$
$$latency\,gain(BasicBlock_B) = \sum latency\,gain(ISE_I) \forall I \subset BasicBlock_B$$
$$latency\,gain(Program_P) = \sum (latency\,gain(BasicBlock_B) * number\,of\,calls\,to(BasicBlock_B)) \forall B$$

The design space in which the ISE identification algorithm is commonly applied is that of a program running on a customizable processor. Broadly speaking, prior work in this area can be separated into two groups. The first includes approaches that find ISEs quickly by enumerating maximized patterns in the graph representation of the application - as in [44]. The second includes work to maximize the speedup resulting from the selection of a set of ISEs under some simplifying constraints, such as the work in [65]. The method chosen for the work in this thesis is the latter maximization approach. We attempt to achieve low compiler execution time for the ISE identification algorithm while finding the maximum number of ISEs.

The ISE identification algorithm described in this work was implemented as part of a system design framework. Modern embedded systems from controllers to integrated multimedia cell phones are designed using a wide range of processor architectures, and bus standards. In recent years, the complexity of these embedded systems has increased while the time to market window has decreased. Under such competitive stresses and with such increases in complexity, Electronic System-Level (ESL) design tools have emerged which promise to enable fast implementation of complex systems based on high-level system descriptions and software.

The Software/Hardware Implementation and Research Architecture (SHIRA) project was proposed to address the challenges facing embedded system designers. Work on the SHIRA system design framework is ongoing at this time. From a high-level point of view, the framework is designed to perform the following three tasks: Partition software written in the C language into communicating tasks, customize processors to perform these tasks quickly, and then implement the system in VHDL. This is the basis for automated design of a heterogeneous multiprocessor system-on-chip.

There are several contributors to the SHIRA project. Contributors to the partitioning and communication modules include Michael Montcalm, Daniel Shapiro, Marc Branchaud, and others. The instruction set extension identification research and development is described here and was completed by Daniel Shapiro. Contributors to the hardware description generation modules include Daniel Shapiro, Vishal Thareja, Jonathan Parri, Marc Branchaud, Srivatsan Vijayakumar, Tony Lteif, Yvan Muheto, Ismaeel Al Ridhawiand, Saurabh Ratti, and others.

The goals of the SHIRA project are to speed up the execution time of a C program in an automated way, expose system-design options to the embedded system designer, generate hardware descriptions of multiprocessor embedded systems, provide fast compilation times for the tool user,

and provide the research community with an open-source platform for future investigation into this topic. The tool implementation is not yet complete, but the design of SHIRA as it stands now will be presented in later sections.

This work continues by presenting an introduction to the terms and concepts used in this thesis. This is followed by brief overview and history of the SHIRA project for designing heterogeneous multiprocessors. Next, several papers on algorithms for ISE identification will be reviewed and our own algorithm for instruction set extension identification will described. Several approaches to improve the execution time of the ISE identification algorithm are tested, followed by a description of the experiments to validate these approaches. The results of the experimentation are presented with analysis and then we conclude with a discussion of future work.

2 Background

This section explains some of the basic terminology and fundamental concepts used in the rest of the thesis. At a very high level this section will explain: FPGA, compiler, optimizing compiler, single static assignment, retargetable compiler, cross-compiler, Canadian cross-compiler, assembler, linker, debugger, loader, simulator, instruction set simulator, optimization, mathematical models, model constraints, design space exploration, instruction selection, and instruction set extension identification, generation of hardware description, speedup, and design constraints.

2.1 Hardware

According to [109] a Field Programmable Gate Array (FPGA) is "a logic device that contains a two-dimensional array of generic logic cells and programmable switches." FPGAs can be programmed to perform a specific task by configuring the programmable switches and the functions of the generic logic cells. A high-level programming language and description that is used as the source for the synthesis of hardware systems (possibly an FPGA configuration) is called a Hardware Description Language (HDL) [12]. VHDL and Verilog are two examples of HDLs. FPGAs are not programmed directly from the HDL description; instead the HDL must be compiled in a synthesis tool into a binary configuration bit stream file such as the ".bit" file format for Xilinx FPGAs. Computer prototyping can be achieved with an FPGA chip which is configured to simulate the behavior of a computer chip in a cycle accurate way. As the title "FPGA Prototyping by VHDL Examples" implies, FPGAs are often used to quickly prototype a chip design before it is fabricated as a silicon circuit [109].

2.2 Compilers

A C compiler is used to verify the syntactic and semantic correctness of the source code for a C program, and then convert the text of the program into assembly code for a target machine. In the

world of compilers, the "target" machine architecture refers to the computer architecture on which the compiled code will run. The "host" refers to the computer on which a compiler runs. And so the host and target may be different. For example, a PC with an x86 superscalar architecture can "cross-compile" code for a LEON3 SPARC V8 embedded processor. The host in that case is the PC, and the target is the SPARC. A retargetable compiler can be used to compile for multiple targets [8]. A compiler that is compiled on one machine, to execute on another machine, and produce assembly for a third machine is called a Canadian Cross-compiler [93]. Two good articles that introduce the topic of compilers for ASIPs are [18] and [38].

A compiler is "a program that accepts as input a program text in a certain language and produces as output a program text in another language, while preserving the meaning of that text" [5]. Compilers are most often used to convert from a high-level language such as C or C++ to a low level language such as assembly language for specific processor architecture. Some C compilers transform the output text of the compilation flow by rewriting and restructuring the program at various stages during the translation from input language to output language with the goal of improving code size and execution time. These are called optimizing compilers. Other compilers merely translate the input source code into successively simpler representations all the way until assembly code is exported into a file, without attempting to improve the code performance at compile time. One of the major optimization passes available in many optimizing compilers is called Static Single Assignment (SSA). The SSA pass ensures that every variable is assigned only once by duplicating variable names and thus forcing the input program into a representation called "three address code". SSA form is used in order to enable the application of several optimization passes as explained in [96].

Most compilers are divided into modules which operate at different levels of the compiler. Each

module performs a task on a standard type of program representation. Between modules in the compiler the representation of the source code is converted into a lower-level representation using increasingly simpler or hardware-specific notation. As the source code is expanded from C code into an executable binary the descriptions of parallelism and operations in the compiler representation become more concrete with successively larger and simpler representations. Each level in a compiler may have several passes. A hook in a compiler is a place where a pass can be added to the compiler in a standard and modular way. Compiler modules are composed of several passes which all work on the common program representation of that level.

In the COINS compiler, the representation in the high levels of the compiler (also called the frontend) is called HIR, or High Intermediate Representation. The lower levels of the compiler use a representation called LIR, or Low Intermediate Representation. Compiler code in the lower levels of a compiler that is used for only one target machine is called a backend. Retargetable compilers have many backends, one for each target machine. Even at the lower levels of the compiler, instructions are often still using an abstract representation of registers. Specifically, the compiler assumes that there are an infinite number of registers to read from and write to, but then near the final passes at the low levels of the compiler a pass exists which limits the number of simultaneously used registers to the number of registers that exist in the target machine. In effect, this pass inserts spill code, which moves data into and out of the register file of the target so that the program uses only the maximum number of registers available in the target, and not more. The result of this scheduling of the assembly in the program is the addition of load and store operations in several places in the code. The pass is called register allocation.

Assembly code output from the compiler must be converted into an executable binary in order to run, and this task is performed by an assembler. For most RISC architectures only one instruction

per line of assembly is allowed. The assembler converts the text for each line of the assembly into the corresponding binary machine code for a given instruction. However, this binary file is not in the final executable format required by the target. A loader can be used to move a binary file into the memory of a system. In the case of our toolchain, the loader processes executable files and turns them into memory files for use in an embedded system.

A C program is represented as a control flow graph in the low-level intermediate representation (IR) of a compiler, where each node in the graph is a block of code called a Basic Block (BB). A basic block is a list of sequentially executing instructions that do not contain jumps or branches. Because there is no change in control flow in a BB, it can be seen as a graph of functional units that will all be used to modify the input variables in the block and assign the result to the output variables. Unfortunately compilers do not generally calculate dataflow graphs, and so they must be calculated from the BB information, which is a list of annotated instructions.

A dataflow graph can be represented based on the information provided in a BB by following three steps. First ensure that each time a variable is assigned a value, its name is changed to a new name. Next ensure that all operations are in three address code format, or in other words that each operation involving more than three variables is simplified into intermediate instructions. This step is repeated recursively on the resulting instructions until only operations involving three variables or less remain in the BB. This operation grows the number of instructions in the BB while reducing the complexity of the instructions. Finally, the unique variables in the BB are called edges and the operations that are predecessors or successors to the edges are called nodes. The operation and variable identifiers for the nodes and edges must also be maintained so that the solution to the ISE identification can be applied back to the IR of the program.

It is common that only a small number of instruction set extensions can be added to each processor in a system due to constraints on the hardware size of an embedded system. As a result some design space exploration must be undertaken to decide what customizations (if any) to add into each processor.

Register pressure is "the number of hard registers needed to store values of the pseudo-registers at given program point" [26]. This is an important aspect of ISE identification because ISEs with multiple inputs and multiple outputs require that all of the inputs to the instruction be alive in the register bank and available at the same cycle when the instruction is called. These instructions also require that all of the outputs of the instruction have empty registers available to write into simultaneously. Effectively, ISEs can cause havoc during register allocation, because they may cause the need for many load and store operations that otherwise would not be necessary. These loads and stores can then lead to cache misses, and cost even more time to execute.

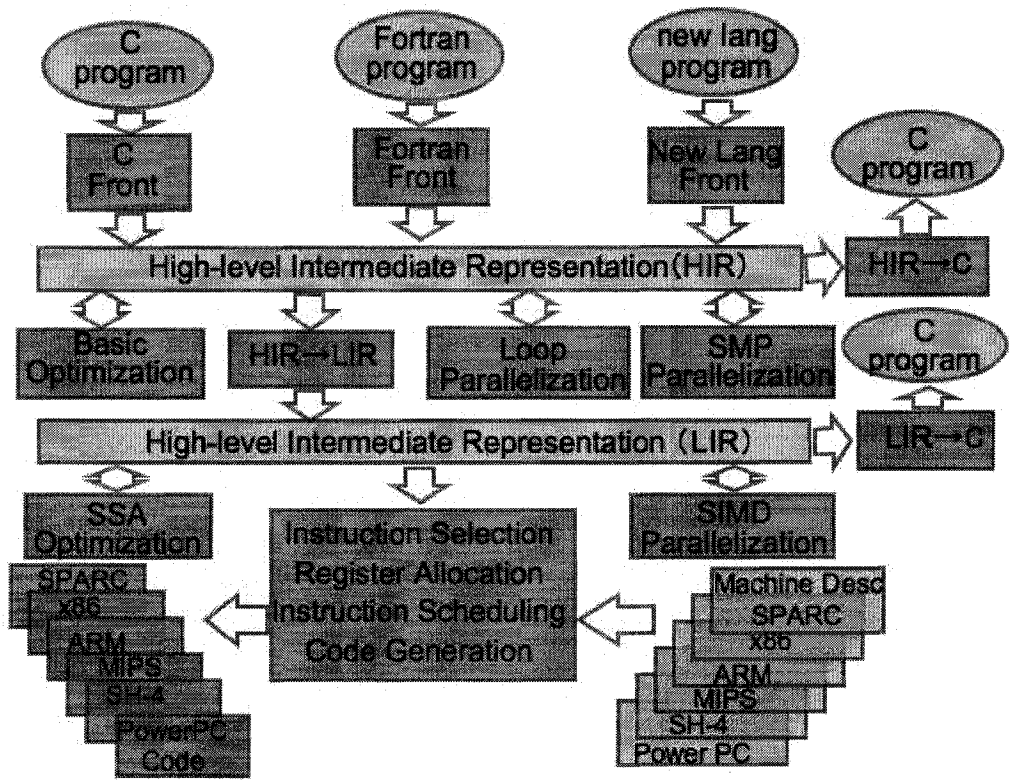

Figure 1: High-level view of the COINS compiler from [1].

The COINS compiler infrastructure shown in Figure 1 supports multiple input languages and target processor architectures [1]. It is written entirely in Java, and has modular language frontends, each with their own parsers which translate source code into the high level intermediate representation of the compiler called HIR. HIR trees can be manipulated by passes which are registered in the compiler registry. One example of an HIR pass is CommonSubexpElimHirE, a class which performs the Common Subexpression Elimination (CSE) optimization on an HIR tree. CSE is used to find subexpressions in an expression that are identical, and assign the result of the expression to the common locations where it is used. HIR is eventually converted into a lower level representation called LIR, upon which SSA optimization can be performed. For the experiments in this thesis ISE

identification was set up as a pass in the LIR part of the compiler. Instruction selection is the pass which converts LIR representation into lists of assembly instructions in the LIR format. Register allocation maps pseudo-registers to hard registers adding spill code as needed. Until this point in the compiler, the precedence of instructions is ambiguous, and instructions are represented as having multiple ancestors and descendants. Instruction scheduling is the process of turning the unscheduled IR list of instructions into lists in assembly while avoiding situations that will slow down the program, such as the need for stalls due to dependency between instructions. There are often many valid instruction schedules.

A good overview of instruction selection in general is available in [47]. Instruction selection in the COINS compiler is described in [72]. In the LIR of the COINS compiler, a dynamic programming approach is used to perform a pattern matching algorithm which chooses machine instructions to cover the LIR of the program. This pass is machine independent, insofar as the algorithm itself is target agnostic, although the target details are imported into the algorithm from the Target Machine Description (TMD). Each target of the COINS compiler has a corresponding description in the compiler backend called TMD. Each target "T" is described in the COINS package coins.backend.gen using three files: CodeGenerator_T which extends a generic code generator, MachineParams_T which extends the generic machine parameter file coins.backend.MachineParams, and T.tmd which describes the target machine operations.

TMD is used to map between IR and machine dependent representation in the same way as MD in GCC. Both representations also have constructs which expand IR patterns. However, there are some key differences. Most importantly, TMD uses a dynamic programming algorithm to match instructions to IR, and allows the user to script several lines of MD into a few lines of TMD using constructs such as "foreach". TMD enables the compiler to map between LIR code and the machine

code of the target. The register uses, bit lengths, and operation types are all defined in the TMD. The "defrule" construct is used to describe a pattern in LIR that matches a sequence of statements in the target assembly language. TMD common to all targets can be collected into the file "common.tmd", and can be included by reference in the TMD of a target. Interestingly, java statements can be called from inside the TMD to bypass the pattern matching approach to instruction selection in special circumstances.

Often compiler toolchains come with simulators. A simulator is a software package which emulates the functionality of a processor or system. An Instruction Set Simulator (ISS) simulates the behaviour of a microprocessor so that assembly code for the processor can be tested in a software environment. An ISS may be used to explore the design space of an embedded system by timing a code segment, monitoring the memory contents of the simulator, or otherwise observing the state of the simulator. Transaction Level Modeling (TLM) and simulation is a higher level of simulation which does not keep track of every instruction in the processor or system, but instead simulates high-level interactions in a model of the system components [103].

2.3 Design Space Exploration

Consider the design of a single application-specific processor. The designer can add or remove instructions to the processor and measure its performance. Many different micro-architectures could perform the same computations, but each different implementation has different metrics for power consumption, chip area, clock frequency, and execution time. The set of all possible implementations of microarchitectures can be graphed as a volume in a space where the axes are power consumption, chip area, clock frequency, and task execution time. The volume containing all of the points in the graph are the design space, and Design Space Exploration (DSE) is the process of searching the design space to find good tradeoffs between the various competing design

parameters. For example, if the frequency of a design is kept constant, as the chip area increases, so does the power consumption. Under the same assumption, as the chip area decreases the execution time increases.

When exploring the design space we may encounter equivalent custom instructions. Graph isomorphism is the problem of proving that two graphs are equivalent. In ISE identification it is important to know when two ISEs are functionally equivalent so that they are not both implemented as hardware, wasting area on the chip. Instead the two equivalent graphs should be merged into one instruction. This problem is known to be very hard to solve, but there are tools such as nauty which perform graph isomorphism testing [117].

A mathematical model of a finite element system can be composed of an objective, some constraints, and some variables. Optimization is the process of maximizing or minimizing the objective function in a mathematical model to some solution which satisfies the problem constraints, revealing the highest or lowest values of the variables. ISE identification in particular and DSE more generally are optimization problems. A design space can be described using a mathematical model and explored using well-defined matrix manipulation methods such as the simplex method to arrive at a global maximum for the objective function. Simplex can only handle linear or integer constraints, and we do not address nonlinear constraints in this thesis because they are solved using more complex mathematical methods than simple linear programming. Perhaps in the future we will be forced to use nonlinear models as the full scope of the system design problem is addressed by SHIRA.

A valid result obtained from a model is suboptimal if a better result can be obtained from the model (lower for minimization or higher for maximization). Obtaining the optimal solution from a model is

intractable if it cannot be obtained within reasonable time and space constraints [14]. According to [14] it is common for problems with polynomial time algorithms to be tractable. In this thesis we are most interested in the tractability of the ISE identification methods in the compiler which affect execution time. A solution is feasible if it satisfies all of the constraints in the model. The feasible region is the set of all feasible solutions in the solution space. Finally, a model is feasible if it has a feasible region which contains one or more solutions [68].

Combinatorial optimization is the process of finding the best discrete solution in a finite solution space. Solutions can be very problem specific, but there are general classifications of problems based upon the computational complexity of the model. Two good books on the subject of combinatorial optimization, and model tractability are [6] and [33]. The ISE identification problem requires optimization methods because the solver must search many different groupings of instructions in a C program in order to find the set of these groupings which provides the highest speedup to the program execution time.

The Linear Programming (LP) approach should be applied when a problem has real variables, a linear objective function, and linear constraints. Mixed-Integer Linear Programming (MILP) should be used instead when the problem has some integer or binary variables. Furthermore, MILP should be used only when the problem at hand does not have an extremely large number of unknown variables [77]. In [3] and [78], some basic optimization methods are categorized, and an explanation is given for when to apply each method. The following discussion on optimization uses concepts and definitions from [68].

2.3.1 Example LP

An LP is composed of an objective function and some constraints. For example: maximize the sum

of two salaries subject to the constraint that neither salary is less than zero, both salaries are less than or equal to $100,000.00, and neither salary is more than twice as much as the other salary. Assuming that the first salary is called x1 and the second salary is called x2, we can formulate this problem as follows:

$$\max x1 + x2$$

$$\text{s.t. (subject to the constraints) } x1 \geq 0$$

$$x2 \geq 0$$

$$x1 + x2 \leq 100000$$

$$0.5\, x1 \leq x2$$

$$0.5\, x2 \leq x1$$

The constraints on the non-negativity of the variables are unlike the other constraints, because they are telling us that the feasible region is in the top right quadrant of the x1, x2 graph of the solution space. In a standard form LP, the objective function is being maximized, all constraints are in the form $ax1 + bx2 \leq c$, all constraint constants (c in the previous expression) are non-negative, and all variables have a non-negativity constraint [68]. We now rewrite in standard form the salary maximization problem stated earlier, calling the objective function Z:

$$\max Z = x1 + x2$$

$$\text{s.t. } x1 + x2 \leq 100000$$

$$x1 - 2x2 \leq 0$$

$$x2 - 2x1 \leq 0$$

$$x1, x2 \geq 0$$

The simplex method can be used to solve a standard form LP. Because of the non-negativity constraints on a standard form LP, we know that the origin is a feasible solution, and so we use this point as the starting point in the simplex method. Simplex operates by moving from one feasible solution to a feasible solution at an adjacent intersection of two constraints. The direction of the traversal of these corner point solutions is towards a higher objective function value, as we will see. The problem can be seen graphically as the following figure:

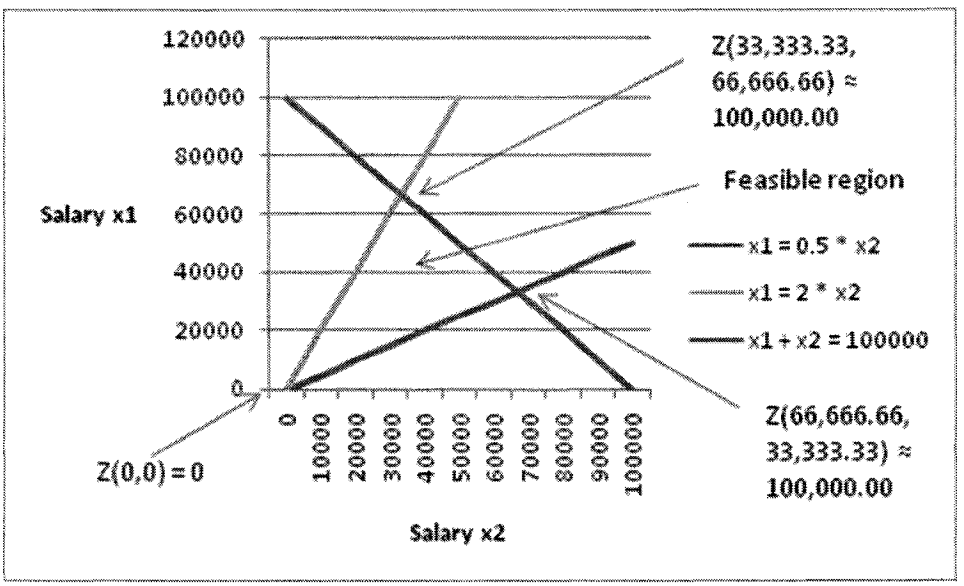

Figure 2: Graphical representation of a standard form LP

We can see in the figure above that there are three intersections of the constraints, one at the origin which is not helpful because it is already the starting point for the standard form LP. Two possible solutions are shown on the graph which maximize x1 + x2. For more complex problems we cannot see the solution easily, and so the simplex method is required to traverse the corner points in the model.

The standard form LP is now converted into a set of equations for the simplex method. The left hand side of each equation is appended with one non-negative slack variable in order to change the ≤ constraints into equality constraints. In our case the model is changed to:

$$
\begin{aligned}
\max Z = \quad & x1 \quad +x2 \\
& x1 \quad +x2 \quad +s1 \qquad\qquad\quad = 100000 \\
& x1 \quad -2x2 \qquad +s2 \qquad\quad = 0 \\
& -2x1 \quad 1x2 \qquad\qquad +s3 \quad = 0 \\
& x1, x2 \geq 0
\end{aligned}
$$

For this problem, the augmented solution is the solution which includes the slack variable values, and a basic solution is a corner point solution that may be feasible or infeasible. In our example the intersection of x2 = 0 and x1 + x2 = 100000 is a basic solution but not a feasible solution because it violates the other constraints on the solution. A basic feasible solution is a corner point solution that is feasible. During each iteration of the simplex method, some of the variables in the model are set to zero in order to force the solution onto a corner point. In each iteration the variables that have been set to zero change, moving the solution to a new corner point. A nonbasic variable is one that is currently set to 0, and a basic variable is one that is free. To find the corner point that is currently being investigated, simply look at the intersection off all of the nonbasic variables. The point of the simplex method is to move from the current basis (the set of basic variables corresponding to a corner point) to a basis with a higher objective function value.

The model is now put into a form called the proper form of the simplex tableau. In this form the objective function variables are moved to the left hand side of the equation, and are therefore negated. We can see the direction of improvement by looking at the objective function and finding

the variable that provides the highest contribution to the objective function. In other words, the variable in the objective function having the highest negative constant should be added to the list of basic variables and the first basic variable to approach a value of 0 is placed in the nonbasic variables list. The latter calculation is called the minimum ratio test. The minimum ratio test is the division of the right hand side of the equation tested, by the coefficient of the variable that will enter the list of basic variables. If the divisor is zero or negative in the minimum ratio test, then there is no limit on that row.

After replacing one basic variable with another, the tableau must be updated so that the column of the leaving basic variable contains a 1 in the row at which the switch occurs, and 0 everywhere else. This is done using a Gaussian operation where the each element in a row that is being modified has subtracted from it the value from the column of the entering basic variable in the row being modified multiplied by the corresponding element in the row of the leaving basic variable. Another operation that can help in updating the tableau is dividing or multiplying the whole row by a constant. This is legal because the row represents the left and right hand sides of an equation, and so multiplying both sides by a constant does not affect the result of the equation.

The first simplex operation is performed as follows on the tableau, choosing arbitrarily x1 as the direction of improvement we move from the first table to the second table shown below. s1 has the lowest value for the minimum ratio test, and so s1 is replaced by x1, and the tableau is updated.

Basic Variable	Eqn. #	Z	x1	x2	s1	s2	s3	RHS	MRT
Z	0	1	-1	-1	0	0	0	0	Never
s1	1	0	1	1	1	0	0	100000	100000
s2	2	0	1	-2	0	1	0	0	0
s3	3	0	-2	1	0	0	1	0	0

Table 1: Proper form tableau of example problem, step 1. x1 is the entering basic variable and s2 is the leaving basic variable.

Basic Variable	Eqn. #	Z	x1	x2	s1	s2	s3	RHS	MRT
Z	0	1	0	-3	0	1	0	0	Never
s1	1	0	0	3	1	-1	0	100000	33333
x1	2	0	1	-2	0	1	0	0	no limit
s3	3	0	0	-3	0	2	1	0	no limit

Table 2: Tableau step 2. x2 is the entering basic variable and s1 is the leaving basic variable.

Basic Variable	Eqn. #	Z	x1	x2	s1	s2	s3	RHS	MRT
Z	0	1	0	0	1	1	0	100000	never
x2	1	0	0	1	1/3	-1/3	0	33333.3	
x1	2	0	1	0	2/3	1/3	0	66666.6	
s3	3	0	0	0	1	1	1	100000	

Table 3: Tableau step 3. s2 is the entering basic variable, and x1 is the leaving basic variable.

We stop the solution now that there are no negative values in the objective function. The solution is seen in Table 3 above. Converting back to equations, x2=33,333.33 and x1=66,666.66. Following an alternate route we can reach x1=33,333.33 and x2=66,666.66. In fact all points on the line

segment between the two solutions Z(33333.33,66666.66) and Z(66666.66,33333.33) are feasible solutions to this maximization problem.

2.3.2 Branch and Bound

In the previous LP example, the variables found by the simplex method were real valued numbers. However in ISE identification some of the variables may be binary or integer. For example, the variable that tracks the inclusion of a node in an ISE is binary. That is to say, a node in a basic block cannot be partially included in an ISE. In order to deal with the binary and integer variables in the ISE identification problem, we need to use the branch and bound.

Branch and bound is a tree algorithm for resolving the binary or integer variables in a model. The objective in branch and bound is to enumerate as few solutions as possible while finding the optimal values for the integer and binary variables in the model. The root node in the branching tree is the starting point, and each generation in the tree corresponds to one binary or integer variable in the model. Each node branches into as many child nodes as there are possible values for the variable in the next generation. For example, the root node branches into 2 nodes if the first generation variable is binary, and k nodes if the variable corresponding to the first generation is integer and restricted to be in the range of 0 to k-1. Each node in the tree is bounded by the best possible solution that can result from setting the binary and integer values associated with that node, and nodes with all of the variables set to values are called leaf nodes. A node without all of the variables set (not in the lowest generation) is called a bud [68].

As the tree is grown out from the root node, the branch and bound algorithm stores the best valid solution found thus far as the incumbent solution. The trick to this method is to grow the tree in the direction of the most promising node, and to never grow the tree from a node where the incumbent

solution is better than the bounding function. In order to work properly, the bounding function must be optimistic about the outcome that results from enumerating child nodes. This is necessary so that we can conclusively decide to stop exploring in a direction where the bounding function is exceeded by the incumbent. When selecting which node to expand, there are several valid node selection policies available. For example, best-first search expands the node with the best bounding function value, depth first chooses only from the set of bud nodes that were just created, and breadth first search expands nodes level by level until leaf nodes appear. The advantage of best first search is that it follows the most promising path, while the advantage of depth first search is that an incumbent solution is found quickly. The variable selection policy is the association between generations in the tree and variables in the model. The search is terminated when there is no node in the tree with a bounding function high enough to merit its expansion [68]. For our ISE identification implementation, the branch and bound method was used by the LINDO 11 solver to find the solution to the MILP models passed to it from the COINS compiler.

2.3.3 Design Parameters and Quality

Some electrical engineering metrics which are important in this field are clock frequency, clock domains, and power consumption. Clock rate, or frequency, is the rate at which the system clock attached to the embedded system oscillates. The maximum clock frequency is reduced by long delay paths through combinatorial logic. A higher frequency will result in a faster implementation, but care must be taken not to set the clock frequency too high. If the latency of a register to register operation is longer than the clock period, then the wrong result will be loaded into the destination register. Multiple clock domains can be designed into a single chip, and messages can be passed back and forth between the domains using a message buffer. Finally, power consumption is a measure of the Watts dissipated by the computer chip which implements a design. The circuit consumes a certain amount of current between the source and sink of the power supply which drives

the chip, and a voltage is applied across the power and ground terminals. In real designs the power consumption varies greatly depending on the circuit switching rate and several other factors that are beyond the scope of this discussion.

In order to quantitatively inspect the benefit of various design decisions, a simple metric that is often used is speedup. Speedup is a measurement of the relative performance of two systems [73]. Specifically, it is the ratio of change in execution time of a task after some modification to the hardware and/or software has been applied to the system. Speedup is calculated by dividing the original execution time by the new execution time. Subtracting 1 and multiplying by 100 gives the percent difference in execution time. For example, if a program takes 50 seconds to execute on a microcontroller, and only 25 seconds to execute on 4-processor embedded system, then the speedup when switching to the multiprocessor system is 2.0, or a 100% improvement.

Software optimization tools within a compiler can modify a program to make it more efficient. This type of optimization is based upon formal methods. However, unlike the LP optimization described earlier, many optimization passes in a compiler are improvement transformations and cleanup transformations rather than maximization or minimization functions. For example, code that will never execute can be removed by a pass in the compiler called Dead Code Elimination (DCE). DCE is called many times during the compilation flow. There is a stronger focus in compilers on correctness than on optimality. Another example of an optimization that improves the situation rather than optimizing it is loop unrolling. In the simple case where the loop bound is a factor of the unrolling factor, a loop unrolling transformation replaces the body of the loop with several copies of itself in order to increase the basic block size inside the loop. Larger basic blocks are optimized more effectively than smaller ones in the passes following loop unrolling. Many compiler optimizations provide a speedup in the compiled program at the cost of compiler execution time. In [40], the

authors describe many such optimization techniques.

Speedup can also be achieved in the execution time of the designed system by changing the processor hardware. Increasing parallelism at the hardware level is one way to decrease the execution time of a program running on an embedded system, and as a result obtain a speedup. We will apply the technique of adding hardware to increase speedup on two levels: coarse grained and fine grained parallelization. Coarse grained parallelism will be achieved in our framework by partitioning programs into tasks and running the tasks on separate processors, while extending processors with pipelined custom instructions will be applied for fine grained parallelism.

It is important to keep in mind when designing an ASIP that a design choice about one system parameter affects other parameters. For example, the processor's clock frequency will slow down for all instructions in the ISA if only one instruction has a long combinatorial path. This can result in a large speedup for a specific task, but a large slowdown for the system as a whole.

The design space for an embedded system can be constrained by many project requirements, including but not limited to the clock frequency, clock domains, a maximum allowable chip area, real-time requirements on some critical sections of code, power consumption limits, and noise limits. The hardware size constraint leads to a limit on the number of processors in the system, and the number of instruction set extensions for each processor. There are some non-intuitive limits such as the limit on the execution time of the compiler. Because the compiler is used by people, it must complete its work in a reasonable amount of time. This restricts the degree to which the design space can be explored to a short window, unless otherwise specified by the user. In this thesis we account only for hardware size (area) and execution time constraints.

Some electronic system-level design tools increase the number of registers in a processor to increase the throughput of custom instructions as explained in [65], but for the sake of simplicity, we have opted not to follow this approach for the latest iteration of our framework. Finally, pipelining is a way of increasing the operating frequency and parallel processing of a sequence of hardware components. Specifically, adding pipelines reduces the time required for register to register signal propagation and so the clock frequency of the design can be increased. Furthermore, pipelining allows reuse of the pipelined hardware. When pipelined multi-cycle instructions are introduced into each processor that is customized, stalls must also be asserted to hold up the pipeline while a custom instruction is being executed in the ALU. At this time our ISE identification algorithm does not take into account the pipelining of ISEs and the hardware generation module is incomplete.

Real-time requirements are restrictions on the amount of time that a specific part of a program can execute without returning a result. Hard real-time requirements are requirements that must be met with absolute certainty by the deadline, and after the deadline the result will not be useful. On the other hand, soft real-time requirements should still return the result if the deadline was missed because the usefulness of the result degrades after the deadline instead of becoming invalid right away. The deadline for a hard real-time requirement cannot be missed and the data will not be useful after the deadline [17].

Real-time requirements should be taken into account in the ISE identification process. The implicit assumption made in [19], [20], [21], [25], [24], [33], [35], [43], [46], [44], [65], [66] and [67] is that the design goal is to speed up the program in an arbitrary way. Therefore their algorithms search for the highest speedups in the program that they can find, but not necessarily the most useful. Real life programs are not like benchmarks, and so some code segments are constrained by

real-time requirements and others are not. ISE identification could be performed when real-time requirements are added to a constraints file, which refers to the unmodified C program. Experiments on real-time requirements for the ISE identification algorithm were not completed as part of this work, and so it discussed and is left as future work. Constraints on programs could be added into the Eclipse IDE programming GUI, and linked into the constraints file.

2.4 Multiprocessor Systems

Many tools exist for the specification and implementation of multiprocessor systems, but few of them are open source to the scientific community, and fewer still are able to explore the design space of complex heterogeneous topologies in order to find cost-efficient designs. Some toolchains allow the user to specify an ISA based upon one program and then compile any program onto the resulting ASIP. This is helpful when the ASIP is designed to perform one task very efficiently, but also performs other tasks. One example of this situation is a cell phone, where phone calls are the primary use, but other uses such as playing games and navigating with a GPS unit may need to target the phone's custom processor. To achieve this feature, an instruction set description language such as TIE or MDES is needed [68], [102]. Some examples of ASIPs for cell phone technology are [10], [76] and [31].

An MP-SOC, or Multi-Processor System-on-Chip, is an embedded system composed of several processors, memory modules, and some kind of interconnection network. An interconnection network is a set of wires and state-aware switches. Interconnection networks may implement blocking or non-blocking communication, and are used to route messages throughout the MP-SOC. The designer of an MP-SOC can implement topologies using shared memory, message passing, or a hybrid approach. We decided that following standards for these approaches would make the toolchain more applicable, and so OpenMP for shared memory and MPI for message passing are

being developed to facilitate the design of complex topologies [37].

2.5 SHIRA Toolchain Overview

We now introduce a tool for designing multiprocessor systems, and explain the progression from user requirements and source code to instruction set extensions and processors. When completed, the SHIRA tool will be able to abstract complexity from the perspective of the system designer while exposing compilation options. We are investigating improvements in design automation and compile time of which will reduce the time-to-market of real products. We expect that reducing the design effort with automation, reducing the cost of embedded systems by simplifying the implementation where possible, and opening the source code to the public will contribute to the adoption of the tool. In this thesis we focus on the design and implementation of ISE identification. However, this section will inform the reader how the work in this thesis will be used in the context of the SHIRA project.

SHIRA is being designed to satisfy multiple requirements. Firstly, quick compilation time is being investigated in order to improve the debug time experienced by system designers. This will result in faster time-to-market performance. Another goal is to enable automated system design and hardware generation. This will allow system designers to make changes late in the design phase without complicated code rewriting. The tool will also support multiprocessor system design to promote design parallelism, and will execute algorithms for the minimization of cost per unit fabricated. Finally, SHIRA is being designed in a modular way. This promotes code readability, and enables integration with other projects.

We are attempting to keep the design flow experienced by the tool user flexible to changes in software or constraints of the user's project, in order to avoid the cost overruns associated with redesign. For example, one possible approach to the problem of electronic system-level design is to

implement the design using state machines that control arithmetic units and memories, or in other words translating directly from C into a hardware description as in [1] and [118]. However, such a solution is not flexible to changes late in the design process. Specifically it is not responsive to software changes after the product has been debugged, verified and produced as a silicon circuit.

We want to ensure that a design can be reprogrammed or updated after being shipped out as a product. For this reason we plan to include the default compliment of instruction set hardware in each processor in the design. Therefore regular programs compiled for the target processor which do not know about the ISEs will still work. We expect that reprogramming a generated multiprocessor system which includes instruction set customizations will have lower throughput than the original design. This is because the new software for the embedded system may not contain all of the same custom instruction patterns added to the processor, and will therefore end up being more sequential than the version of the program with ISEs which the processor was designed for.

Another goal in the design of the SHIRA framework was to keep the cost of the end-system as low as possible. Software bugs can be inexpensive to fix, but hardware bugs are often very expensive to fix. Also, it is expensive to verify a fully custom computer. It is therefore smart to first try simple designs with repeated use of verified logic blocks such as processors, and only to try complex designs if necessary. The SHIRA approach to designing embedded systems is therefore based upon an algorithm which attempts to minimize the complexity of the design. Initially the compiler attempts to meet the design constraints with a single processor design. If that approach fails to meet the design constraints, then a multiprocessor system design is attempted. In the event that the multiprocessor design still does not meet the design constraints, the multiprocessor design is improved by adding custom instructions to the processors in the design. The processors used in the SHIRA toolchain can be customized during the design flow, and are called "extensible processors"

[39]. These processors are also referred to as ASIPs, or Application Specific Instruction set Processors or customizable processors [39].

Another important goal for the SHIRA project is to expose formal optimization parameters and design decisions to the user in a user interface. This will allow the user to tune the compiler to the needs of the project. For example, the user can select one of several versions of the ISE identification algorithm to use during compilation. These versions will be presented later on in the thesis. Another example is the ability of the user to set the maximum hardware size of the end-system.

Multiprocessor system specification and simulation in SHIRA can be achieved in the Simulator and Interconnection Network Generator (SING) module. The user can specify a system topology in a user interface, or allow SHIRA to find an appropriate system topology. SING is an open-source tool implemented in Java for the design of multiprocessor embedded systems. The tool includes an API with simulation capabilities which can be used to explore the design space of an interconnection network, a GUI, and a wrapper for WISHBONE or AMBA compliant IP cores.

In our proposed SHIRA tool the input program is first partitioned into sections which do not need to share large amounts of data in shared memory in order to perform their tasks. Next each section is assigned to a shared memory cluster and examined to determine the benefit of sharing the workload of the task between multiple processors. The partitioning aspect of the compiler is based upon shared memory clusters communicating with each other via message passing, and to that end, OpenMP and MPI modules implemented at the high-level of the compiler are being investigated by Michael Montcalm. Modules for message passing communication called TMD-MPI and TMD-MPE were developed at the University of Toronto in 2006. We intend to use these TMD hardware modules or

develop our own along the same lines [51].

To facilitate interprocessor communication, the components in the embedded system are connected using an interconnection network. The standard that was selected for on-chip communication is WISHBONE B.3 because it is open-source, free to use, scalable, simple, and supports multiple topologies [3], [42]. The hardware description generation module in our system (SING) generates processors, busses, and interconnects components. When processors are generated, they may have instructions added to their instruction sets by the compiler. The way that the extended instruction set is generated will be described only briefly here, and will be discussed in more detail in the theses of Vishal Thareja and others who are working on that aspect of the SHIRA framework. Hardware description generation of instruction set extensions is not yet completed, and so the identification of the custom instructions is discussed here, but not the implementation in hardware.

2.6 Design Lessons Learned from the SHIRA Project

The following section looks at the challenges faced during the development of our toolchain so that others who may pursue this type of project will not make the same mistakes as we did. Developing a new piece of software is sometimes a learning process, and in this case it turned out to be a slow uphill battle. The main problems we faced were:

1. Getting bogged down in compiler customization details
2. Learning the details of compiler implementations
3. Adopting open industry standards
4. Mismatching the toolchain implementation with the capabilities of the development team

The design of our toolchain adapted to the increasing skill set and education level of the team. Our

group started off with six undergraduate students developing a capstone project, and grew into a well rounded team of graduate students.

In all, the design of our tool has gone through four major iterations:

1. A proof of concept developed entirely in-house. (2006)
2. A small prototype using simple tools. (2007)
3. A large prototype using industry standard tools. (2007-2008)
4. A prototype using tools and languages familiar to the research team. (2008-2009)

Version 1 - Compiler: none

The first iteration of the SHIRA project was a proof-of-concept prototype. It was called "EasyCore", and used to demonstrate that automated coprocessor generation would provide a speedup for a small C program. Over a four month period a custom MIPS processor was developed in VHDL. Using the processor description as a template, a C-to-hardware tool was created in Java based on C-parsers and templates. The pattern-based ISE identification tool looked for nested switch statements to implement as a hardware coprocessor. The toolchain included a custom assembler written in Java. We ensured that the generated processor would work on both Altera and Xilinx FPGAs. The user interface was a standalone Java applet that allowed the user to decide the trade-off between hardware size and speedup in the input application. Our implementation was only loosely bound to the Windows operating system, and the results of our testing on a program composed of nested switch statements showed between 1.4 and 1.5 times speedup over an unmodified uniprocessor implementation.

The drawbacks of our initial implementation were numerous. Most importantly, we could not add an off-the-shelf compiler and RTOS on top of our processor because we had not implemented the entire ISA of a MIPS processor. Also, our ISE identification algorithm was not based on a compiler's internal representation, instead it was based on a C code analyzer. The analyzer may have been

reading loops and structures that could have been optimized away with the use of an optimizing compiler, avoiding the need for adding hardware to the processor. As well, we did not adopt any standards for bus architectures. Finally, the application of the tool was very specifically tailored to one application, and a general solution to the embedded system design problem is needed.

Version 2 - Compiler: LCC

The second iteration of our tool included several improvements based on the lessons we had learned from the first prototype. We sought out a processor called miniMIPS [88] and added a floating point unit to it. As a compiler we selected LCC [3] because it is small, written in C, and free to use. A template-based hardware generator that conforms to the WISHBONE B.3 standard [15] was created, and a user interface for the design of multiprocessor systems was added to it. The compiler part of this second iteration of the framework was never completed because of several design and implementation shortfalls. Firstly, we came to realize that even miniMIPS was not enough of a standard processor to become widely adopted. Furthermore LCC is problematic because it is non-optimizing, and users expect the compiler to optimize the software before trying to optimize the hardware. Finally, the patterns we were using as custom instructions were very specific, and a more general approach was needed for finding fine grained parallelism.

Version 3 - Compiler: GCC

Because of the perceived shortcomings of the non-optimizing LCC compiler and miniMIPS processor used in the second version of the design, a third version of the tool was planned using the LEON3 processor from Gaisler Research, available for research and education for free under the GPL license [82]. The GCC compiler available from [83] was used instead of LCC, because it is widely accepted as a standard compiler, and it is an optimizing compiler. The very specific hardware templates for large patterns were kept in the toolchain, but instruction set extension identification code was added to find the more common fine grained parallelism. To run GCC from a windows platform, Cygwin and

MinGW were both tried as platforms for compiling GCC from the Windows operating system, and are available from [84] and [85] respectively. The compiler was developed and compiled in windows, to execute on Linux, and generate assembly for the LEON3.

At this time a group of four undergraduate students called MDMS including Marc Branchaud, Tony Lteif, Yvan Muheto, and Ismaeel Al Ridhawi were tasked with building a prototype for the hardware description generation of multiprocessor interconnection networks [59]. The code from their project was the basis for implementing multiprocessor systems in an automated way in future versions of the tool. The result of their work was a program written in java that could collect user requirements for a multiprocessor system, and then generate a modular and easy to read network in VHDL according to the specifications of those requirements. The tool was tested with function simulations and on FPGAs, and a GUI was implemented where a user could define nodes (representing processing elements or peripherals) and lines between nodes (representing bus lines). As well the program was able to generate documentation for all generated hardware. This version had one system-wide clock domain for the interconnection network.

Version 4 - Compiler: COINS

After several months of development it became clear that the approach of modifying the GCC compiler was very complicated, and compiling a Canadian cross-compiler with GCC was non-trivial. GCC is written in C, and as computer engineers with high expectations in terms of examples and documentation, the team found GCC to be fairly poorly documented. The learning curve was high for the team members primarily because the background of the team was computer engineering, and the code base required increasingly greater knowledge of software engineering and computer science. Also, GCC is an open project that often grows and changes, resulting in poor documentation on the part of the compiler writing team. Although the work on GCC was progressing, the decision was made to speed up progress by using a compiler written in a high level language like C++ or

Java. Incidentally, there have been discussions in the GCC community about rewriting the compiler in C++ [91]. We selected the COINS compiler for the fourth iteration of our design because the team members were all familiar with Java and the compiler was easy to understand, simple to use in both Linux and Windows, and showed performance metrics such as compile time, and optimization nearly as good as those for GCC [81]. Anecdotal evidence suggests that we experienced a fast turnaround time with java components compared to GCC. Each time we recompiled GCC it would take approximately 20 minutes running on a 3.0 GHz Intel Core(TM)2 Duo E8400 with 512 MB RAM assigned to the Linux OS. We used VMware Server which hosted Ubuntu Linux (Gutsy Gibbon) as the operating system. Recompilation in COINS takes approximately 1 second after the first compile. Incremental compilation introduced in GCC will probably invalidate this problem [94]. During this redesign period, the hardware generator was improved by including simulation capabilities, while the internal data structures were redesigned to be more object oriented, and the user interface was redesigned to be eclipse compliant. Work on the user interface is ongoing at this time.

The code for the MDMS project was reworked to include object oriented modeling, multiprocessor simulation, multiple clock domains, AMBA to WISHBONE B.3 bridges, processor and memory hardware description generation, and several other features. This new version was renamed to SING. At the same time, the Eclipse IDE was chosen as the platform for the user interface because like the rest of the project it is Java-based, and furthermore several system design tools such as Altera's NIOS-II IDE and Xilinx's Platform Studio SDK are built on the Eclipse IDE. Because the hardware description generation code changed so dramatically, the user interface had to be completely re-implemented in Eclipse.

Another change was the use of JUNG in our tool to visualize control flow and dataflow graphs. During the development of ISE identification, the results were difficult to read without some sort of

visual queue to show what instructions were being suggested. Images like Figure 8 provided insight into the structure of programs. Figure 9 comes from another visualization tool provided along with the COINS compiler which correlates line numbers in the IR or code with control flow graphs in graphical form. The instructions for installing the COINS Visualizer library called CoVis, Java graphing package called Grappa, and graph visualization tool called Graphviz are available from [92]. These libraries are needed in order to enable the visualization of data in COINS.

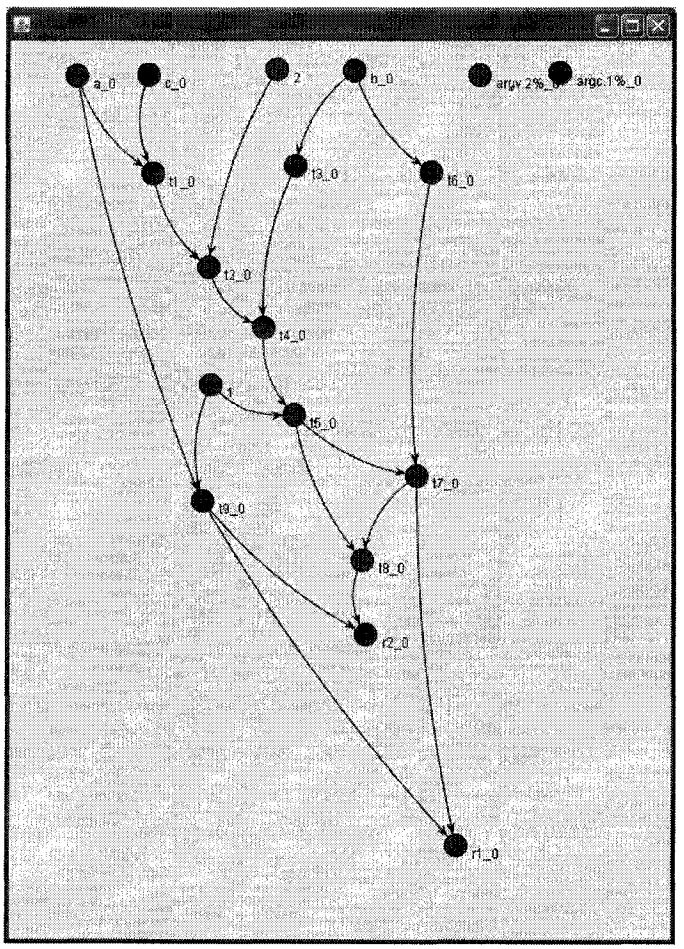

Figure 3: Datapath derived from a basic block and displayed in a JUNG popup window

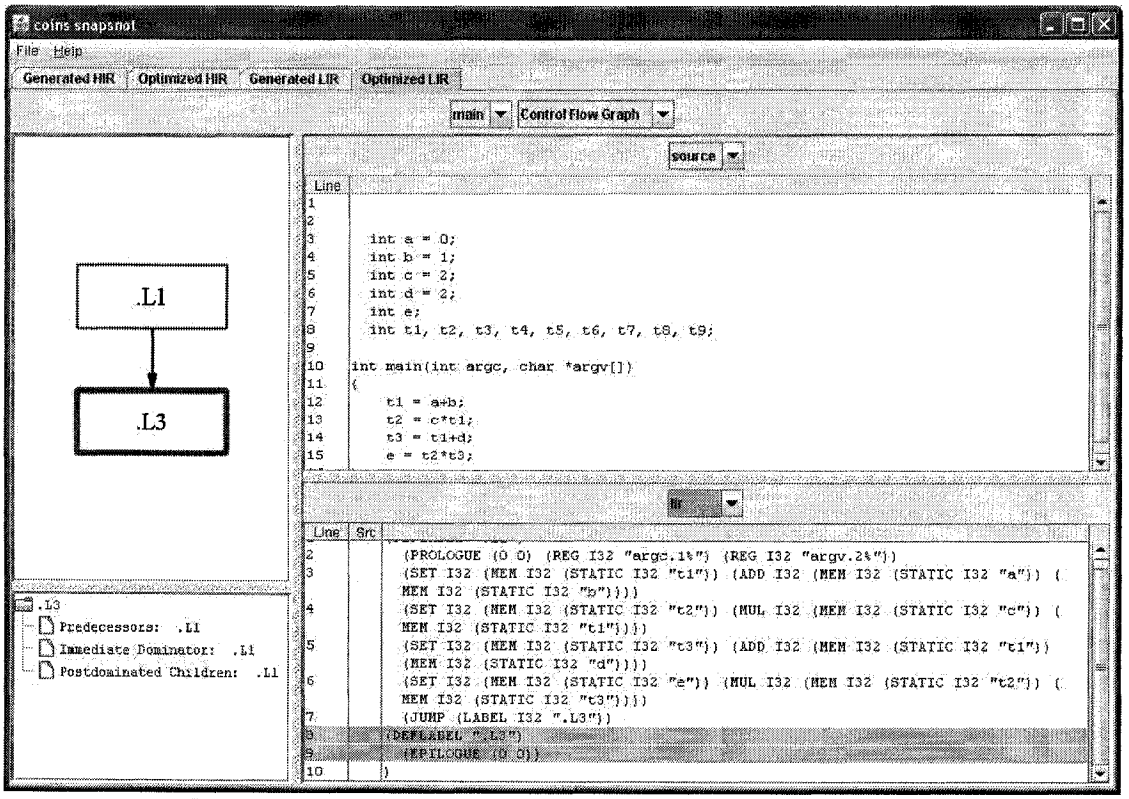

Figure 4: A control flow graph from the COINS compiler

Another area where the SHIRA framework was extended is coarse grain parallelism extraction. MPI and OpenMP were adopted as the standards for allowing the compiler to express communication patterns in the code [86], [87]. The COINS compiler was already designed to inject OpenMP code into C code. The future work required will be to pick up from there and mix MPI code into the program to express distributed memory communication. In the winter of 2008, COINS was successfully used to find a fuzzy logic pattern in a C program as part of the InnovateCanada competition [89]. The ISE identification algorithm itself was implemented with the help of the LINGO 10 interface with Java using JNI [90]. And finally, the gprof profiler can be used with the COINS compiler to get profiling data for C code. This profiling data is needed for the selection of custom instructions.

The memory files in our toolchain are created using a compiler, linker, and loader. A customized assembler based on the GNU as assembler and ld linker was developed in order to convert the assembly output of the COINS compiler into ".elf" executable format. Next a customized loader was created to convert ".elf" files into ".mif" memory files for use with Altera FPGAs. A similar approach will be taken for the Xilinx flow in the future.

Another change that will need to take place is a change from the LINDO 11 math module to the free and open source lp_solve package. We used LINDO because it was convenient in the short term to get running with results. However, it is not free to use, and therefore we should replace it so that SHIRA can be shared with other researchers at no cost.

This concludes the chronology of changes and design decisions that have been undertaken over the past three years of work on our framework. The main lessons we have learned from designing and implementing the framework were that compiler customization can be very time consuming and bug prone, and there is a sharp learning curve for computer engineers that move into the field of computer aided design of embedded systems. We also learned that it is important to adopt open industry standards in order to maximize the applications of the research and the usefulness of the framework. And finally, we learned that mismatching the toolchain implementation with the capabilities of the development team can cost a lot of time and effort.

3 Instruction Set Extension Identification

This section will first present the problem of ISE identification, and then several recent solutions will be presented and compared. Finally an algorithm for ISE identification will be presented. Not all published algorithms will be presented; instead the reader will be given an impression of the main issues that need to be addressed, and some of the approaches that have been tried by researchers in the field.

3.1 ISE Identification in Detail

The general description of the process of ISE identification algorithms will now be described. Recall that the ISE identification problem is the problem of finding a set of extensions to the instruction set that best speeds up a given C program, under some constraints such as a maximum available hardware size for ISE implementation. Each ISE identification approach is slightly different and so the following generalities will be elaborated in more detail later on in this section. Once a basic block has been represented as a dataflow graph, the graph must be cast as a mathematical model and passed to a math solving module.

There are generally two approaches to ISE identification: ILP oriented solver based approaches and heuristic methods. For ILP based approaches, a linear programming solver such as CPLEX available from [98] or LINDO available from [99] accepts a problem description containing a set of constraints on the problem, an objective function, and the data sets that represent the search space. The math program will then represent the search space and constraints in a matrix and perform optimization to find a local or global maximum for the objective function. The maximized objective will represent the nodes in the dataflow graph that should be used as a custom instruction in the processor. Alternatively, heuristic methods can be applied directly to the dataflow graph and the search space

can be pruned throughout the search process to reduce the search time. Generally, a set of the best possible solution(s) found thus far is maintained until the search space has been traversed by the solver. Tree based methods such as [19] can search of the design space quickly relative to other methods by sorting the solution space, forcing the number of inputs and outputs of the ISE to be small, and pruning invalid subtrees in the search space. The problem with this approach is the exponential relationship between the size of the problem and the solver execution time when the number of inputs and outputs is higher than 3 or 4. Population based methods such as [24] can reach a solution quickly as well, but they do not always find the same solution, and the solver can get stuck at a local maximum as explained in [20]. And so in conclusion, the design space for ISE identification is very large, and methods are needed to reduce the size of the search space. Reducing the ISE identification search space arbitrarily may remove the optimal solution in the search space. For example, if the ISE which provides the highest speedup for a program has 100 inputs and 100 outputs, restricting the number of inputs and outputs in the model will put the optimal solution out of reach. All algorithms presented here rely on the details of compiler implementations to partition the design space into models with tractable solution times, as we will see.

Once the ISE identification algorithm has completed its search of the dataflow graph for a single basic block, the control of execution is returned to the compiler which has to write down the selected instruction into a list of candidate instructions. Some ISE identification algorithms examine the same basic block several times, iteratively selecting the best undiscovered ISE. After all of the basic blocks in the program have been processed, the candidates can be ranked and a subset of the candidates is selected for implementation. Once again, the details of selecting ISEs from the population of identified valid ISEs varies widely between implementations, and so this is only a general overview and more detail will be provided later on for each ISE algorithm that we discuss.

After the ISEs that will be used in the implementation have been selected, there are two ways to implement the software aspect of the ISEs. Either the instructions corresponding to the ISEs can be replaced by inline assembly, or the machine description of the ISEs can be added to the compiler's machine description for the target processor. In the latter case the instruction selection phase of the compiler will replace the old sequences of instructions with the new ISEs.

The pass implemented in the COINS compiler which performs instruction set extension identification was scheduled as late as possible so that the software optimizers of the compiler do not ruin the careful selection of custom instructions. This can happen if the program code is changed by a compiler optimization pass that runs after ISE identification. This can also happen if the code for an embedded system changes but the identified ISEs do not. The question about when in the compilation flow to perform ISE identification has been addressed in several papers and books, including [20] and [46]. In the COINS compiler we execute the pass on the LIR after HIR and SSA optimizations have been completed. We do this because we always try to find a software solution to performance optimization before resorting to dealing with a problem at the hardware level, and because in the LIR optimizations have been completed but spill code has not yet been inserted for register allocation. If we perform the algorithm any later, then there will be mismatches between the identified ISEs and the original source code, but if we perform the algorithm any sooner then the identified ISEs may be unnecessary because they may have been invalidated by optimizations performed by the compiler. In [20] Section 3 explains that the best place to apply the ISE search in the compilation flow is just before register allocation. And so in conclusion, we schedule the pass before register targeting but after the major low level software optimization passes such as SSA.

The execution time of the ISE identification problem is intractable for programs with large sequences

of uninterrupted data processing [19]. The design space is therefore constrained by assumptions in order to make the solution time of the ISE identification problem tractable. We solve the global ISE identification problem just like compilers do: by using divide and conquer. For more detail on the application of divide and conquer in compilers, see [47]. In our case divide and conquer is used to split up the global ISE identification problem that looks at whole programs. Basic blocks are instead used as the search space in which to find ISEs, and the speedup information for each basic block is summarized into a decision making program. The decision making program must collect the results of many calls to an optimizer (one call per BB) and decide based upon all of the results which ISEs to include in the processor ISA.

The execution time of the compiler must be tractable. In the same way as C compilers could generate smaller code by exhaustively searching the possible instruction selection possibilities, a compiler could perform exhaustive search to find all possible ISEs. However, in both cases the exhaustive search may require hours of compile time as demonstrated in [19] and [28]. For a multi-million dollar project developers may accept such a compile time if the solution is the best one possible. However, this is not often the case. To get away from this problem, several solutions have been proposed which severely limit the optimality of the solution by requiring one output as in [25], or only very large ISEs as in [44]. The authors of [66] come to the same conclusion in their related work section about the tradeoff between optimality of the solution and solution time.

A pass was added to the COINS compiler by setting a trigger to run at the AfterLateRewriting hook. The ISE identification pass reads in the LIR of a C program and creates dataflow graphs from basic block information. Next the dataflow graphs are printed into a LINDO file and the LINDO 11 solver is called through its JNI interface. The solver identifies cost-effective instruction set extensions in the dataflow graphs and exports the results into a text file.

At this time the instruction selection pass has not been extended to include the new instructions. When the toolchain is complete, the custom instruction file will be parsed and each identified ISE is added to a list of candidate ISEs. The compiler will select a subset of the identified ISEs to be added to the instruction set of the processor. The compiler will emit assembly code containing the new instructions, and datapath descriptions will be passed to a hardware generation module which can realize the design in VHDL. Each designed processor will be programmed with the aforementioned custom assembly code. The reader can find additional information on ISE identification from [45], which surveys the topic.

3.2 ISE Identification Algorithm Implemented in COINS

As explained in the background section, we use MILP representation of a problem when it has a reasonable size, all variables are real, some variables are binary or integer, and the objective function and constraints are linear. In our model all variables are either real or binary. The two pieces of information taken together can be used to summarize the approach that we have followed, along the same lines as the algorithms in [65] and [33].

Formally, we define a graph G (V, E) which represents the dataflow graph of a single basic block. G is the graph, while V and E are the vertices and edges along the graph. Figure 5 in the next chapter shows an example of such a graph. The red dots are vertices and the lines are edges. We mark memory access vertices and vertices involved in non-ALU operations as "ILLEGAL" so that they cannot be included as part of an ISE. The number of inputs to the instruction must be less than or equal to the number of unreserved registers in the register bank, although that is not enforced in this model. Patterns of previously identified ISEs are disabled after they have already been identified.

Once the dataflow graph of a BB is profiled into ISEs, we can find the repeated patterns and perform instruction selection. To get to that step the problem consists of several subgraph enumeration steps. The ISE identification algorithm used in SHIRA borrows heavily from the algorithm in [66]. The ISE identification algorithm is cast as an ILP with constraints on the number of inputs, number of outputs, convexity of the ISE, requiring more than one node in an ISE, excluding loads and stores from an ISE, and disabling already discovered ISEs.

The model of the ISE problem was specified in the LINGO 11 math programming language. The output text of the solution was often truncated in the solution log file unless the width field is set higher than the default of 76. The maximum output setting of 200 characters per line was specified in the LINGO 11 user interface. The LINGO 11 model consists of a section for sets, a section for the compiler to lay down data, and a section with the constraints and objective function. The model assigns nodes to the best available ISE using a binary decision variable X which can be used to include a node "I" in the ISE (ex: X(I)=1) or include it in the software portion of the dataflow graph (ex: X(I)=0).

We define the following sets having an index for every node in the dataflow graph: LATENCY, X, HWCOST, SPEED, IS_INPUT, IS_OUTPUT, D, A, CONST, and ILLEGAL. LATENCY is a set used to figure out the longest path through the ISE. This technique was described in [65], and calculates the speedup of implementing a given ISE in hardware as we will see later on. HWCOST represents the cost of implementing the given node in gates, while SPEED represents the number of ALU clock cycles required to implement a node in software. IS_INPUT is a binary variable set to 1 when the successor of the current node is in the ISE, but the node itself is not in the ISE. IS_OUTPUT is another binary decision variable set to 1 when a node in the ISE has a successor node outside of the

ISE. D and A are integers used to ensure convexity of the ISE as we will see later on. CONST is a set that defines which variables are constant nodes, and which are not. ILLEGAL is used to prevent nodes from being included in an ISE. We also define the derived set ARC(NODE, NODE) to be a set of all arcs between nodes in the dataflow graph, with the source being the first node and the destination the second. Finally, NUM_INPUTS and NUM_OUTPUTS are used to represent the number of inputs and outputs in the ISE. These two variables are involved in limiting the search space to ISEs with the specified I/O constraints.

In the data section a pointer to the model status is maintained so that the solution status can be retrieved on command. In the data section of the model, the COINS compiler extension writes out the variables found in the basic block. For example: for simpler.c in Appendix A.1 we find that the compiler defines the NODE set in the form "NODE = c,t2,t1,t3,b,d,a,e;". These nodes may be decorated with compiler information, and so in actual fact the node names are more like "c_divexi32". The set "ARC" is instantiated next, showing all of the directed edges between nodes. The sets SPEED, HWCOST, and ILLEGAL are also populated in the data section of the model. The MAXINPUTS and MAXOUTPUTS variables are set in the data section by the compiler to impose constraints on the number of inputs and outputs that an ISE can have, which reduces the algorithm solution time. Finally, INCUMBENTSPEEDUP, a variable for eliminating uninteresting ISEs with low speedup is set in the data section of the ISE identification math model. In the default configuration, INCUMBENTSPEEDUP is set to 0, to ensure that it is nonbinding.

The constraints and objective are defined in the final section of the model. This section, as well as the header describing the sets is always the same when the compiler calls the LINGO 11 solver package. The first constraint defined is the restriction on the size of the hardware. The sum of HWCOST values in the ISE is piped into a variable called HWCOST_OF_ISE, which is the estimate

cost of implementing the ISE in hardware. This number can be used to rank ISEs by hardware size. Formally:

$$\sum_I X(I) * HWCOST(I) \quad = \quad HWCOST\_OF\_ISE$$

Another constraint in the model is on the number of nodes in the ISE. We do not want to enumerate all of the single instruction ISEs, and so we set a constraint that the number of nodes in an ISE is greater than or equal to 2. Formally:

$$\sum_I X(I) >= 2$$

As discussed earlier, we forbid some nodes from inclusion in the ISE. Both X(I) and ILLEGAL(I) are binary variables, and so we restrict their sum to 1, excluding some nodes from being included in the ISE. Or more formally:

$$X(I) + ILLEGAL(I) <= 1 \; \forall I$$

One trick taken directly from [33] is the ability to calculate the latency of the ISE without maximization in the objective function. Nodes in a directed acyclic graph can be seen as having ancestors and descendants, another way of saying predecessors and successors. In this case we choose to view the problem as successive generations of nodes. Each generation of the graph included in the ISE has a greater latency than the previous generation by the amount of latency that it adds to the solution. Although there is slack in the latency, it will be minimized by the objective function as it is pushed towards the minimum cost. More detail on the objective function will be presented later on in this section. The latency calculation can be expressed formally as follows:

$$LATENCY(I) \geq LATENCY(k) + SPEED(I) * X(I) \forall ARC(k, I)$$
$$LATENCY(I) < TOTALLATENCY \forall I$$

The first expression above finds the minimum latency for a node as a function of all of its ancestors. The next expression forces the total latency to be greater than the sum of the latencies of all of the included nodes in the ISE. This formulation finds the minimum path delay through the ISE when the solver pursues the objective function in the direction of the solution. The total latency is at least the critical path length, and no larger than the critical path length because under the objective function maximization the total latency of the ISE is minimized.

When constraining the number of inputs and outputs we must take special care not to miss the fact that the inputs to the basic block have inputs, although they are not represented in the dataflow graph. For this reason we treat inputs to the basic block as inputs to the ISE if they are included in the ISE. The same applies for outputs from the basic block. It should be noted that three arcs entering the ISE from the same source node count as only one input to the ISE. This is because only one register transfer is needed to get the data from the register bank into the ALU and down to the custom instruction. This is crucial to understanding the way that ISEs are enumerated. We must be careful to accurately model the physical implications of the embedded system.

Let us now look specifically at the input port constraints on the ISE. When both "k" and "I" are used, the letter "k" refers to the node that is outside the ISE and "I" refers to the node that is inside the ISE. The constraints on the number of inputs are designed to first control a binary decision variable which marks nodes as inputs to the ISE, and then second to limit the number of nodes that can be marked as inputs. Formally, the following four constraints are placed on the number of inputs to the candidate ISE:

$$IS\_INPUT(k) \geq X(I) - X(k) \forall ARC(k, I)$$
$$IS\_INPUT(k) \leq \sum X(I) \forall ARC(k, I)$$
$$\sum_I IS\_INPUT(I) = NUM\_INPUTS$$

$$NUM\_INPUTS \leq MAXINPUTS$$

The first expression above forces a node outside the ISE with any successor in the ISE to be counted as an input. It is not binding otherwise. The second constraint counts the number of nodes that are children of the current node, and if none of them is in the ISE then the node cannot be an input. If any child of the node is in the ISE then the constraint is nonbinding. The third constraint above counts up the number of inputs in the NUM_INPUTS variable, and the fourth constraint limits the number of inputs to the ISE to a user defined constant.

The output port constraints are very similar to the input port constraints. The first constraint in the listing below marks the nodes in the ISE with children outside the ISE as outputs. The next constraint marks nodes in the ISE with no descendants in the ISE as outputs. The following constraint prevents a node from being an output if all of its children are in the ISE. Finally, the number of outputs is counted and bound to a user defined value in the last two constraints below.

$$IS\_OUTPUT(I) \geq X(I) - X(k) \forall ARC(I, k)$$
$$IS\_OUTPUT(I) \geq X(I) - D(I) \forall I$$
$$IS\_OUTPUT(I) \leq \sum (1 - X(k)) \forall ARC(I, k)$$
$$NUM\_OUTPUTS = \sum_I IS\_OUTPUT(I)$$

$$NUM\_OUTPUTS \leq MAXOUTPUTS$$

The constraint which is by far most complicated in the ISE identification model is the convexity constraint. It requires that no path (formed by many instances of ARC(I, k)) between any nodes

with X(I)=1 contains a node with X(I)=0. In a directed acyclic graph such as a dataflow graph for a basic block, this can be explained simply as: No node that is an output has successors that lead to inputs. Our approach is to enumerate all paths from outputs to inputs and restrict the number of such paths to 0. The first challenge is to gather two sets of information on which nodes have ancestors in the ISE, and which have successors in the ISE. We call these sets A for ancestor and D for descendant. Formally:

$$A(I) \geq X(k) \forall ARC(k, I)$$
$$A(I) \geq A(k) \forall ARC(k, I)$$
$$D(I) \geq X(k) \forall ARC(I, k)$$
$$D(I) \geq D(k) \forall ARC(I, k)$$

Next we can use these sets in conjunction to find out when a node has both ancestors and descendants in the ISE but is not itself in the ISE. We constrain all such occurrences to never happen. The formal expression of this idea is the following: $D(I) + A(I) - X(I) \leq 1 \forall I$ The final part of the ISE identification model is the objective function. The speedup calculation is performed as a linear function, whereby the total software execution time of the ISE nodes is discounted by the longest path through the ISE. This allows us to represent the tradeoff between adding instructions and increasing parallelism. The number ISE_SPEEDUP is essentially how many clock cycles will be saved each time this basic block is executed, assuming that the ISE is added to the processor, and not the ratio of the software execution time to the hardware execution time, as is the standard speedup calculation. The ISE_SPEEDUP variable is subject to a constraint that it should save more time than INCUMBENTSPEEDUP time units. We now formally represent the objective function as the maximization of the speed of the selected nodes in software penalized by the speed of longest path of the ISE in hardware:

$$(\sum_I X(I) * SPEED(I)) - TOTALLATENCY = ISE\_SPEEDUP$$
$$ISE\_SPEEDUP > INCUMBENTSPEEDUP$$
$$MAX = ISE\_SPEEDUP$$

Once an ISE has been identified, it must be added to a list of patterns that will be handed off to the instruction selection phase for ranking and selection. However, in the more immediate term, the current ISE is used to get the next best ISE. For every already identified ISE, a set is created for it in the model. The set for the first ISE found will be called ISE0, the second will be ISE1, and so on. Each of these sets is instantiated alongside an uninstantiated set $SAME_N$ that acts as a binary counter of differences between the identified ISE and the current candidate ISE. "N" is used to refer to an ISE already identified. If 10 ISEs have been identified, then SAME0, SAME1, ..., SAME10 will be declared as sets in the model. As long as there is one difference in the X(I) values of a known ISE and a candidate ISE, the candidate ISE can be selected. Otherwise it is not a valid solution. The patterns of two ISEs can be exactly the same but the nodes forming the pattern cannot be the same exact nodes. We define this formally as follows:

$$X(I) - ISE_N(I) \leq SAME_N(I) \forall I, N$$
$$ISE_N(I) - X(I) \leq SAME_N(I) \forall I, N$$
$$ISE_N(I) + X(I) \geq SAME_N(I) \forall I, N$$
$$2 - ISE_N(I) - X(I) \geq SAME_N(I) \forall I, N$$
$$(\sum_I SAME_N(I)) \geq 1 \forall I$$

In the work of [65] and [33] there are many fewer ISEs enumerated because ISE patterns are collapsed to a node and then disabled. Our method enumerates far more ISEs. The five constraints above are each required in order to disable a node pattern in the graph. Each $SAME_N$ set is forced to contain a 1 of there is an index which differs from the X(I) set. In the first two constraints above an index of the $SAME_N$ set is forced to 1 if X(I) and ISEN(I) are not the same. In the third and fourth

constraint a $SAME_N$ element with index "I" is forced to 0 if X(I) and ISEN(I) are the same. In the final constraint, we force the number of differences between the ISE(I) and X(I) sets to be greater than 1 by forcing the sum of these differences represented by $SAME_N$ to be greater than or equal to 1.

One might assume that repeated patterns are not found by the solver, but identical ISEs all have the same speedup, and so they will all be enumerated next to each other. Successive iterations of the MILP solver will discover the repeated pattern. Also, we include for testing purposes a commented code segment in the model which can force various X values, causing some test scenarios to unfold.

Inside the COINS compiler there is no readily available dataflow graph calculated and ready to use. As mentioned in the correspondence in [100], a few processing steps on the LIR of the basic block can be used to build up an accurate representation of a BB as a dataflow graph. Because we had to debug the dataflow graph generation process, the GUI of Figure 8 was implemented. This GUI is informational only, and can show CFGs or DFGs.

LINDO 11 was selected as the mathematics engine for the project because the students were already familiar with the language of the solver from the optimization course SYSC 5004 [77]. The solver is not free and open-source, which goes against the driving principles of the SHIRA toolchain. In the future a tool such as lp_solve should replace the LINDO package as the mathematics engine [104].

In order to create the ILP for each execution of the algorithm in each basic block, a generic header and footer approach was taken, where the top and bottom of the ISE identification file always stays

the same. The compiler can parameterize the ILP problem by injecting text representing the current stage in the iterative solution process into the data section of the problem file.

3.3 Literature Review

Fourteen papers in the field of ISE identification were studied in this section to give the reader an impression of the state of the art. The various approaches to this problem can be gathered into groups. Some researchers have tried inexact simulation and heuristic methods such as genetic algorithms, while others have tried linear programming and integer linear programming methods to find exact solutions to the problem under some simplifying constraints. Adding bounds on the ISE identification problem reduces the runtime of the problem to a tractable length of time. For example, only searching for n-input 1-output ISEs reduces the design space dramatically, even for large basic blocks [25]. This is because the vast majority of valid ISEs have more than 1 output. There is no rule for categorizing basic blocks as large or small, but the general idea is that basic blocks with thousands of nodes are considered large (as noted in [66]), and those with only a few hundred nodes such as basic blocks in AES and FFT are considered smaller.

The ideas that will be discussed are analyzed chronologically to show trends over time, they are compared quantitatively by comparing the speedup obtained on benchmarks, and finally they are separated into categories based on the type of approach. The two categories that we will consider are algorithms for ISE identification, and approaches that improve the quality of existing algorithms. The trend over time has been towards more exact methods for ISE identification, and better speedup results. This trend can be seen in Table 4. For example, from 2003 to 2004 the papers presented here emphasize heuristics, while the emphasis of the work from 2005 and onwards is on ILP-based solutions. Also, interest in adding memory elements into ISEs has increased as the research has shown that memory loads and stores along with cache invalidation are a major

bottleneck in the design of a microarchitecture. In [19] an exact algorithm is used, but it is recursive and requires an unreasonable amount of time to execute on large basic blocks or loose constraints on the number of input and output ports on an ISE. For example, the algorithm in [19] had to consider approximately 100,000,000 ISEs for basic blocks with only 100 nodes. And to compound the trouble the relationship between number of nodes and number of ISEs is exponential, or perhaps polynomial as described in [66]. In the worst cases the runtime for the algorithm in [19] was measured in hours.

In reference to [77], [78], and Table 4 we can see that some researchers believe that the ISE identification problem should be solved as an integer linear program or network flow program. Others have opted for genetic algorithms or simulated annealing. Nearly every branch of the tree in [77] is being explored, but each approach sacrifices execution time for optimality.

The trend in speedup can also be seen in Table 4. Because some programs are easier to speed up than others, the benchmark that is used has a large effect on what the speedup will be. Therefore Table 4 shows which benchmarks were used to obtain the speedup claimed by the authors. One good example of progress in the area of ISE identification algorithms is the comparison of the 2003 result for encryption in [20], and the 2005 result of [35]. The results get approximately twice as good for the same benchmark.

ISE Identification Method	Benchmark	Publication Year	Reference	Speedup
Search Tree, Recursive, Heuristics and bounding	MediaBench	2003	[19]	1.3 to 1.6 for gsmdecode, 1.4 to 1.6 for g721encode
Autonomous search using guide function heuristic, followed by candidate selection	MiBench, NetBench, MediaBench	2003	[20]	1.33 to 1.87 for encryption. Low for branch intensive code.
VLIW-style fusion of instructions, loop vectorization, ISE data type sizing, heuristics	sad, fft, idwt, gsm, mpeg4	2003	[21]	Maximum of 4 for gsm, 3 for mpeg4, 20 for sad, 11.3 for fft, nearly 23 for idwt.
n-input 1-output cones evaluated by binate covering	DSPStone	2004	[25]	2.75 average
Genetic Algorithm	EEMBC and MediaBench (telecom domain)	2004	[24]	5 average, peak of 10 for AES
Identify unique schedulable patterns using sampling	MediaBench, Spec2000int	2004	[28]	Speedup was not the objective
Min-Cut Algorithm	EEMBC and MediaBench, cryptographic benchmarks	2005	[35]	From 1.3 to 2.3 for AES, depending on the input and output port constraints.
ISE enumeration via ILP, I/O and convexity constraints in ILP, nauty isomorphism testing, and the critical path length is a function of all intermediate nodes included in the ISE	AES Encryption and Decryption	2005	[33]	4.5 for AES encryption and 7.5 for AES decryption.
Lower memory transfer overhead between ALU and memory by including memory inside ALU, and penalizing data transfers, optimized by ILP from [50]	EEMBC and MediaBench, cryptographic benchmarks	2006	[43]	2.8 on average over sequential solution
Tune the optimizations in the compiler to the benefit of the ISE identification	MiBench, EEMBC CRC snippet	2006	[46]	1.55 on average over sequential solution
Enumerate MAXMISOs, merge into MIMO ISEs, select the best ones using an ILP	MediaBench (ADPCM Decoder/MPEG2 Encoder)	2006	[44]	1.2 to 3.7 over the sequential solution

ISE identification using ILP in CPLEX, maximize speedup based on good latency estimate, penalize data transfers of ISEs in the objective	MiBench	2007	[65]	approximately 1.3 to 6.5
The source-to-source transformations approach is combined with ISE identification to produce better speedup results than either technique alone	SNU-RT benchmarks, UTDSP benchmarks	2007	[67]	2.02
Complete subgraph enumeration using multiple-vertex dominators, proof of polynomial relationship between the number of BB nodes and the ISE algorithm execution time	MiBench	2007	[66]	Up to 6

Table 4: Select publications on ISE identification sorted by publication date

Some methods for ISE identification explicitly disallow the inclusion of some instructions in the ISE search. There are two reasons for doing so: an instruction may be impossible or unfit to implement inside an ISE as with the forbidden nodes in [43], or alternatively the compiler can tell the solver not to return a specific pattern of instructions because it has already been detected in a previous run of the solver as in [33]. In the latter case the approach is usually to reduce the instruction to a single node, removing the possibility of exposing attractive subgraphs common to the whole basic block. One of the underlying assumption of some of these approaches is that large ISEs are not a problem. Tensilica, however, seems to prefer to maintain the generality of the processor's instruction set by choosing numerous small and simple ISE extensions.

Now that we have seen the trends in the field, let us explore the references above more carefully to extract some key insights. In [19] a binary tree is presented with branches representing the inclusion or exclusion of a node in a dataflow graph in a candidate ISE. The binary search is implemented recursively, and areas of the search tree that violate a design constraint are pruned at

runtime. The constraints on the problem are a limit on the number of inputs to the ISE, a limit on the number of outputs from the ISE, and a check to ensure that the ISE is convex. The convexity constraint is a way of making sure that the subgraph selected can be scheduled in the processor, or in other words, that there is no path between two nodes in the ISE involving a node that is not in the ISE. The best solution is tracked as the algorithm executes, and is selected based upon an objective function. The objective function is called on candidate solutions to evaluate the speedup, which is calculated by subtracting the assumed number of clock cycles required to execute the nodes in the ISE minus the ceiling of the sum of the critical path latencies in the ISE hardware.

The binary tree approach in [19] is extended to a tree with n branches at each node, and used to find n ISEs simultaneously. To make the execution time tractable, identified ISEs are reduced to a single node in the graph and the ISE search algorithm is called repeatedly until the objective function cannot be satisfied (estimated speedup is 0). The algorithm was implemented using the MachSUIF framework [16].

The authors of [19] are the same as [50]. In [50] a broad overview of the ISE identification literature is presented, followed by a comparison of the most popular algorithms. The algorithm that the authors present is the optimal ISE identification algorithm from [19]. In [50] a broader perspective of ISE identification is given in 20 pages, whereas [19] was a shorter paper meant only to convey the key results and the algorithm.

In [20], a retargetable compiler toolchain is presented which finds ISEs and generates software that targets the designed ASIP. The section on related work provides a rich overview of the history of ISE identification until 2003. Approaches to ISE identification, and implementation of ISEs is discussed. In the ISE identification algorithm of [20], the number of input and output ports for the ISE is

constrained to reduce the search space, along with the estimated die size of the ISE. Several candidate subgraphs of a dataflow graph are produced based upon the design constraints and a guide function. Each node in the DFG is examined as the starting point for an ISE and the guide function ranks the most promising directions in which to grow the ISE. A hardware library was used to calculate the timing and area estimations for the design space exploration module. A stage for grouping candidate ISEs (called custom functional units in [20]) merges two ISEs if one ISE can be used to perform the computations of another ISE. The surviving ISEs are ranked and the top performing patterns are added to the design as a machine description in the MDES format [102]. Next a compiler toolchain is presented which used the MDES to compile a program onto the designed ASIP. Finally, the authors investigated how patterns found in one domain (such as DSP) apply to another domain (such as control systems). More detail can be found in [28].

In [21] a large set of candidate ISEs written in the TIE language is enumerated for an Xtensa processor. The ISE candidates are generated based upon profiling of the input program generated using real or estimated inputs for the program. The profiling is used identify computation intensive loops or "hot-spots" in the code. Also, the target processor is a VLIW-style processor that enables multiple instructions to be issued statically at the same time, allowing for synthesis of ISEs with numerous inputs and outputs. Instructions are also fused together to form composite instructions. Estimated speedup and hardware cost for each enumerated ISE are calculated. ISEs are selected based upon an objective function that measures area, number of inputs, and latency. The ISEs that are selected are already in the TIE language, and so any program can be compiled onto the resulting ASIP.

The authors of [25] present an automated method for adding custom instructions to the NIOS processor. Because of the restricted input and output port constraints on the NIOS custom

instruction slot, they assume a very tight input and output constraint of 2 inputs and one output for any valid ISE. The algorithm is scalable to any number of inputs with one output, and the ISEs are constructed by selecting a node in the dataflow graph and iteratively including all predecessor nodes that feed into the selected nodes. The method used also allows for the sharing of functional units between ISEs, reducing the hardware size of the two instructions. After enumerating the set of all candidate ISEs a selection process picks the best ones based upon the hardware size constraint and speedup. Speedup for each ISE is the number of cycles required to execute the ISE nodes unmodified, divided by the number of cycles required to execute the longest path in the ISE datapath. The authors simplify the problem by assuming that the added ISE does not slow down the clock frequency of the chip because of long combinatorial paths. The gain of each ISE is calculated as the speedup of the ISE multiplied by the number of occurrences of it. The selection process used is the binate covering algorithm, which attempts to cover a DAG with instruction patterns. The binate covering algorithm is explained in more detail in [25].

A genetic algorithms approach to ISE identification is presented in [24]. The algorithm can identify ISEs which include state registers and small lookup tables. There are two problems introduced when including memory access such as loads and stores into an ISE. One problem is the need for synchronization between the memory and the instruction, and another problem is the introduction of delays into the processor pipeline. To avoid these problems, the system in [24] does not access main memory directly, but instead includes memory in the datapath of the ISE. The authors note that there are many hot-spots in programs where the local variables can be reduced to a lookup table, and index counters on loops can be implemented as internal state registers. These two tricks are shown to reduce power consumption compared to keeping the state information in the registers and cache of the processor. Furthermore, this technique reduces the number of inputs and outputs of the enumerated ISEs, which means that further improvement in the design space exploration,

and power performance are gained or the previously mentioned methods. The internal state of a register inside an ISE can be read from or written to using two instructions added to the ISA which handle traffic between ISEs and the processor. The ISE identification algorithm contains an input port constraint, output port constraint, convexity constraint, an objective function, and a read-only constraint which allows read-only memory nodes to be included in the ISE. This approach is also interesting because it considers inter-basic-block communication directly in the ISE identification algorithm, and allows for memory modules in the ISEs. On a final note, the population used as the starting point in [24] is the same as the algorithm in [25]: 1-output n-input cones of maximum size.

In [28] a system called CPX is proposed which enumerates sub-graphs of a dataflow graph using with the nauty package for isomorphism testing [117]. The tool only searches in the wave-front of the dataflow graph that is scheduled into the processor's registers. The tool keeps or discards candidate ISEs in a buffer. Similar to best-first search, the algorithm required hours of compiler execution time to return a solution. Statistical sampling is possible to speed up execution by 100 times. One of the main results is that the top 10 most commonly identified ISEs in large benchmark suites such as Spec2000int do not change when 1% sampling, 10% sampling, or full search of the design space is undertaken. There are several other interesting results in the paper, but this result suggests that sampling the search space in a fair way can still return very good ISE candidates assuming that there are only a few ISEs that will be selected for implementation. Of course, taking the sampling approach makes the result highly suboptimal in specific cases, but reasonable on average. Also, the amount of sampling is expected to be inversely proportional to the quality of the identified ISEs. One can imagine that for large basic blocks, sampling the ISE search space by limiting the runtime of the algorithm should not heavily affect the design quality but will significantly affect the execution time of the compiler. This thought process leads into Proposal I in the next section, which reduces the size of the ISE search space in a smart way.

The Kernighan-Lin min-cut algorithm for partitioning graphs is described in [1]. In [35] an approach called ISEGEN is presented which uses the Kernighan-Lin min-cut algorithm to steer the toggling of nodes between inclusion in hardware or software. This approach is described in [35]. The algorithm is limited in search space by an input and output constraint on the ISE, and a convexity constraint. Also, there is a cap on the number of ISEs that can be added to the processor. The objective function is a gain function which penalizes completely ISEs which violate the convexity constraint, but only slightly those that violate the input and output constraints. The gain function rewards the discovery of large convex subgraphs and independent subgraphs. The results are compared to a genetic algorithms approach, and a 35% speedup is observed. ISEGEN finds the solution 20 times faster than the genetic algorithm approach.

The work in [43] is a continuation in thinking from the work in [24]. The idea of setting a constraint in the ISE identification algorithm that specifies a set of forbidden nodes was discussed earlier. The forbidden nodes approach is used in [43] to prevent the inclusion of pointer accesses in the candidate ISE. Vectors and loop-carrier scalars are allowed to be included in a candidate ISE. A merit function is used which considers the speedup resulting from the ISE and the overhead of the ISE in terms of the memory accesses. Direct Memory Access (DMA) is considered so that vectors can be very efficiently implemented into ISEs. DMA writes are inserted into the code by the compiler. Benchmark results make a convincing case that the bandwidth to the ISE helps to reduce power consumption and ISE size.

The approach in [33] is to cast the ISE identification problem as a Mixed Integer Linear Program, of the Integer Linear Program (ILP) class of problems. An ILP can be transformed into a matrix representation and efficiently and optimally solved by a solver package such as CPLEX from ILOG.

Finding an optimal solution means that there is no need to worry that the solver may have come up with a better solution if only it had been given more time to do so. The solution is exact and optimal to the expressed objective and constraints. In [33] a framework is presented based upon the Trimaran compiler framework. The generated candidate ISEs are ranked based upon a linear objective function and basic block statistics. ISEs are selected based upon the speedup (objective function value) of the ISE times the frequency of the basic block occurrence, summed across all basic blocks in which the ISE appears. Equivalent patterns are matched using the nauty package [117]. The ILP algorithm specifies input, output, and convexity constraints in a math programming language, and the objective function very elegantly finds the critical path length as a function of all intermediate nodes included in the ISE. The objective is set to maximize the difference between the execution time in software of the nodes covered by the ISE minus the calculated critical path length of the ISE. This method shows a very good speedup result, and an exponential relationship between the number of nodes and the solution time.

[46] and [67] deal with strategies for transforming source code to optimize the results of ISE identification. In [46] the authors combine source code rewriting passes with ISE identification. The rewriting includes loop unrolling, if conversion, algebraic simplification, value range propagation and other techniques. These techniques increase basic block size, and therefore suit ISE identification. Larger basic blocks allow the algorithm to see larger patterns in the dataflow graph. In [46] strategies are discussed for making the best decisions about which compiler optimizations to apply to the code before performing ISE identification. This approach is called ISE targeted transformation.

The results in [46] are confirmed in [67] by a separate team which implemented a source-to-source transformation framework based upon the CoSy compiler framework. The ISE algorithm used is described in [33]. They argue that communication is reduced between basic blocks when the

optimization passes in the compiler are tuned properly. This in turn allows for larger ISEs to be identified. Also, they claim that longer chains of bit level operations can be included in the ISE due to the smart optimization approach, which does not slow down the clock. A machine learning algorithm is employed to drive the source transformations in the compiler. lp_solve is used as the math package which solves the ILP problem. An average speedup of 1.47 was shown for the UTDSP and SNU-RT benchmarks. The authors claim that this is the best speedup to date for those benchmarks.

In [44] the ISE instruction selection problem is casted as an ILP problem, and the size of the solution space is not limited to the basic block level, resulting in larger dataflow graphs. Because even a single basic block can take days to return a solution, one might wonder why the authors of [44] tried to increase the size of the search space. As it turns out, the algorithm first finds the set of maximum sized Multiple-Input Single-Output, or MAXMISO. The MAXMISOs are combined to create Multiple-Input Multiple-Output (MIMO) instructions. The MIMOs based on MAXMISOs are then already convex by construction, and so the most complicated ISE identification constraint (the convexity constraint) need not be applied. The instruction selection algorithm is formulated as an ILP. A very interesting approach is used to avoid a maximization step in the objective function by listing each term in the problem as a separate constraint, forcing the largest constraint to be binding while the others are effectively ignored. Next, a toolchain for automating the ISE identification and selection problem is presented. The XPRESS Solver program was used to solve the ILP, and the SUIF2 compiler framework was used for the compiler. While the algorithm executes very quickly, it cannot see small instructions that may be more desirable in terms of power consumption and chip area. The approach taken in this paper is to focus on maximizing the size of the identified ISEs. The great result of [44] is that their method produces results for a whole benchmark in 7 or 8 minutes, instead of hours or days as did [19] and [28].

Continuing along the strain of thought expressed in [43] and [24], the algorithm in [65] presents the ISE identification problem in terms of the memory-to-ISE bandwidth bottleneck. Another interesting contribution is the formulation of this memory aware ISE identification algorithm as an ILP. The latency of ISEs is more accurately measured that in previous works by measuring the actual latency at the gate level with a hardware simulation tool from Synopsys. A full toolchain is presented with an automated design flow. The objective function contains terms which penalize data transfer to and from the core, allowing the solver to find and rank low bandwidth instructions higher than equivalent latency high-bandwidth instructions. The input and output constraints of previous approaches is augmented by this new bandwidth measuring constraint which shows up in the objective function. For a program having in the largest basic block 1155 nodes, the ISE identification algorithm required 21.5 hours to find the solution. In that trial 4-input and 4-output I/O constraints in the ILP were used.

In [66] a large improvement in ISE identification is shown by using pruning techniques in the subgraph enumeration. The main contribution is to prove that the execution time of the ISE identification problem is related polynomially to the number of nodes in the basic block. The algorithm for ISE identification presented in the paper uses recursive multiple-vertex dominator mathematics to identify convex cuts. Each ISE can be cast as a set of input and output nodes for the subgraph (the ISE). The input and output nodes can be changed until all ISEs have been enumerated. This sets an upper bound on the search time we can expect to see from an optimal ISE algorithm. The upper bound is shown to be related polynomially to the sum of the number of inputs and outputs allowed in the ISE.

Key Insights

The key insights that we can draw from these peer-reviewed papers are that there are several

criteria which define the design space of ASIPs, and the compilers that design them. Execution time, speedup, system power consumption, access to local memory elements in ISEs, chip area, and other key concepts clutter the design space of ASIPs, and compete for resources. Compilers are also drowning in complexity, with each layer affecting the results of the next, and multiple exact and inexact ways to compile. Some approaches execute quickly with modest speedup results, while others perform more complete searches of the design space to obtain better speedup results with longer execution times. The conclusion that we can draw is that we are forced in the case of large basic blocks to make a choice between optimality of the solution and compiler execution time. Often we cannot have both fast compiler execution time and optimal design parameters. Instead we must find a happy medium where the designed ASIP outperforms the requirements by some margin of safety, and the compiler executes in a reasonable amount of time. This conclusion is supported by the related work section of [50]. We now leverage these key insights and state four proposals for improving on the state of the art. Once the proposals have been defined we will proceed to experiment with these proposals and analyze the results of the experimentation.

3.4 Proposals to Improve Upon the State of the Art

After implementing ISE identification in the COINS compiler and studying the state of the art research in this field, several ideas for improving the state of the art came to mind, and they are proposed here:

Proposal I: The compiler should define an execution time cutoff per solver run.

We investigate the possibility that limiting the search time of the compiler in a smart way can still return good results. The larger basic blocks can reduce the execution time of the compiler by limiting the solver compile time, while the faster executing blocks would execute quickly and would not encounter the solver execution time cutoff. As shown in [28], most programs contain only a

small number of ISEs which cover a large number of locations in the code. We examine the possibility that limiting the solution time will similarly reveal the best performing ISEs.

For each basic block in a program an ILP model is passed to the LINDO 11 solver every time an ISE is searched for, and only when no ISE is found does the search move on to the next basic block. There are three options for implementing a limit on the compile time of the ISE identification algorithm. One option is to limit the execution time of the solver by limiting the total time that the solver executes during the compilation. In this scenario, we expect that the first basic blocks encountered will be fully searched for ISEs, while the last basic blocks will not be searched at all because the allotted solver execution time was already expended on the first basic blocks. A second option is to limit the execution time of the solver during the examination of each basic block. In this second scenario the large basic blocks will enumerate only the highest speedup ISEs and then run out of time, while smaller basic blocks will have enough time to enumerate all ISEs. A third approach is to limit the execution time of the solver each time it is called. In this scenario large and small basic blocks can be searched for ISEs with equal opportunity, except that larger basic blocks have more nodes and will therefore be more likely to reach the execution time limit than the small basic blocks.

The approach described in this thesis limits the execution time of each call to the optimization module that searches for ISEs, finding only the easiest to find ISEs until no solution is found. Because ISEs are enumerated in the order of their speedup, the best candidates will be enumerated first. There is a small margin of error for the time required to pause the solver, but it is not significant. The time required to pause the solver is on the order of milliseconds, not seconds. This slack in the execution time of the compiler is acceptable because it is not the milliseconds of compile time that we are trying to reduce; rather it is the hours and days of ISE identification time.

Proposal II: The compiler should be able to issue multiple threads, each searching one basic block at a time for ISEs.

As multi-core desktop computers become prevalent, it is important to scale out compiler technology to each of the available cores to get maximum throughput on all of the available cores. CPLEX is able to solve models on multi-core computers, but it is interesting to compare the single-core and multithreaded versions of ISE identification to judge the relative merits of threading the solver instances at the basic block level. Also, the LINDO 11 solver comes with an example of a Java program which calls the solver with a multithreading approach. We will attempt to parallelize the solver calls with multithreading, instead of processing each solver run as the LINDO 11 tool already can do.

The ISE identification algorithm was partitioned by creating a thread group of a user settable size and assigning basic blocks to any available worker thread until the basic blocks were all processed. The worker threads convert the basic block into three address code based heavily upon the COINS SSA optimization code. Each access to the LIR object by the threads is mutually exclusive, and so the access of all worker threads to the LIR and function reference objects, and by extension the partition, is thread-safe. A semaphore is used to synchronize the writing of identified ISEs to a common buffer where the ISEs can be ranked by the amount of saved latency estimated by the solver for that pattern.

We will investigate the advantages of performing this task at a low and high granularity in the compiler in order to decide where it belongs. The thread group operates at the program level, and can consume basic blocks from different functions at the same time. This is expected to reduce the wait time between functions in the cases where one basic block takes a long time to execute, and

the thread group is not fully utilized.

Proposal III: The ISE identification algorithm should enumerate all ISEs instead of disabling ISE nodes as the ISEs are identified.

Disabling nodes in the ILP model once an ISE has been found which contains the node cuts out from the solution space several other ISEs. In [34] this approach of removing nodes of identified ISEs from the solution is followed. We investigate the benefits and drawbacks of enumerating all valid ISEs in each basic block, instead of only finding ISEs in the nodes left over after each ISE is identified. We cannot compare our results (the number of identified ISEs) to the work in [34] because their ILP model is different. Instead we use our own model to test both approaches, and assess the value of each approach based upon the number of identified ISEs.

The thinking behind our approach is that the optimal set of ISEs in a program may be missed if we do not consider overlapping ISEs. One argument taken from [35] is that repeated ISEs provide a better speedup than non-repeated ISEs. The first figure in [35] shows one ISE repeated 6 times, and using some of the same nodes another ISE is repeated 3 times. If the ISE appearing less often saves more clock cycles than the ISE which appears more often, then the approach of [34] to disable ISE nodes after they are identified will miss the correct choice. It will in fact disable the nodes of each of the three instances of the higher-speedup pattern and then look for additional instructions. Unfortunately, at this point many of the nodes in the instruction which appears six times have already been disabled, and so that pattern is never found. Our approach of disabling node patterns will instead proceed to enumerate the six instances of the ISE, and during instruction selection we will have the opportunity to choose between the two mutually exclusive ISEs.

Proposal IV: Allow the user to specify that the compiler should not consider ISEs that provide a very small speedup.

One way of reducing the size of the design space is to look for instructions that add significant speedup capabilities to the processor architecture, and discard the ISEs with lower speedup values. Towards the end of the ISE search algorithm of [33], the "worst" speedup candidates are being enumerated. We know this because the algorithm in [33] enumerates ISEs in order starting from the ISEs with the highest speedup and moving down the list of possibilities all the way until no ISE can be found with a speedup greater than zero. These ISEs are the worst in the sense that they have the lowest objective function value in the model. Perhaps these candidates are found in many basic blocks and therefore merit inclusion in the architecture, but it is interesting to see what type of results are obtained with and without the heuristic.

We express the concept of removing ISEs with small speedups by adding a constraint to the ILP which acts as though it is an already identified instruction with a potentially higher objective function value than the current solution. We call this constant value the incumbent speedup, expressing the fact that an ISE needs to beat the stated speedup in order to be included as a candidate ISE. The hope is that this tactic will reduce search time by reducing search space.

We considered but did not implement the idea of a domain-specific incumbent speedup. This same incumbent speedup could be then known speedup resulting from a hardware library that is known to work well in a given domain. For example, we considered setting the incumbent speedup to the speedup incurred by a Multiply-and-ACcumulate (MAC) ISE. The problem with this approach is that ten ISEs with a speedup of 1.5 may be passed up because the speedup for a MAC turned out to be 1.53. Together the discarded solutions may have added value to the ASIP performance.

4 Results

In this section the results of experimentation with our ISE identification algorithm will be presented. First the execution time of ISE algorithm is measured. Examples are given which show the ability of the user to control the costs (hardware size) and the benefits (execution time) of the system being designed. Solver execution time cutoff, ISE identification multithreading, speedup cutoff, and a combination of these approaches were applied to the ISE identification algorithm.

4.1 Experiment Setup

The following experiments were performed on an Intel(R) Core(TM)2 CPU running at 1.87 GHz, with 2 GB RAM. Version 1.4.4.2 of the COINS compiler was used, and the operating system was Windows Vista(TM) Business.

The results from each run of the COINS compiler was passed into a file that was parsed by a small utility which counted several different events. Statistics on each basic block and each identified ISE were gathered. This summary data included the number of identified ISEs, the calculated number of cycles saved by all ISEs in a basic block (also called latency gain), and the number of milliseconds used by the compiler to compile the benchmark. The parser excluded any ISEs that were exact duplicates. In other words, if the solver returned two ISEs with the same pattern and involving the same nodes, then first ISE is counted and the second ISE is considered a glitch in the solver's traversal of the solution space.

4.2 Assumptions and Parameters

Our estimate of the cycles saved by implementing an ISE is calculated as the longest path delay through the ISE minus the execution time of each node in the ISE. This estimate does not take into

account the throughput of a pipelined ISE, for which an instruction can be called consecutively several times and each stage in the ISE can be on at the same time. For ISEs composed of chains of operations, which often occur in bit masking operations and other real situations, the latency gain is calculated as 0 cycles because the longest path of latencies through the ISE is equal to the sum of the latencies of each instruction in the ISE. This simple algorithm is an oversimplification of the cycle count problem, which must take into account savings due to parallelism in a chained instruction. For example, pipelines filled with instructions, may have several cycles of latency as the pipeline is filled, but the startup costs are often outweighed by the savings in throughput thereafter. We consider deep pipelines to be a good thing, but we chose an algorithm that would show the fastest speedup with the lowest latency. Our assumption about ISE latency is more of an indicator of the instruction level parallelism in a group of ISEs than it is an exact measure of the speedup resulting from the selection of a specific instruction.

We assume that the FFT, SHA and AES benchmarks are appropriate programs for measuring speedup of the compiler execution time and the number of identified instructions. We also assume that a speedup on the machine used to gather our data will translate into a similar speedup when the compiler is executed on other machines.

In the run configuration of each test parameters were passed to the Java Virtual Machine which specify the amount of memory needed to process the program. "-Xmx1000M" was used in all tests. The "-Xmx" virtual machine argument is used to let the compiler know how much physical memory to reserve for the compilation of the program. The "-S" argument was used because we use our own assembler and linker which has not yet been linked directly into the COINS compilation flow. For these experiments the SHA, AES and FFT benchmarks were taken from the MiBench benchmark version 1 and processed by the compiler.

In Appendix A.4 the table which defines the software latency of various instructions is presented. The values in the function were set to some reasonable latency estimates taken from [51] and [52]. ALU latencies were copied from [52], while the float conversion latencies for the LEON3 were taken from Table 23 of [51].

In Appendix A.5 the assumptions about the number of hardware resources required to implement each instruction in hardware is presented. We intentionally leave the exact units involved in hardware implementation vague. This is so that the user can input integer values for whatever units their synthesis tool provides. For example, a user can set the hardware size constraints to numbers corresponding to CLBs, LEs, number of gates, number of transistors, square nanometers of silicon, or any other hardware size metric. These parameters were set for our experiments to values selected based on our experience with ALU component sizes and compilation of IP cores in Quartus II. Our hardware size estimations are inexact compared to the work in [65] where the hardware size and software speed are evaluated from a synthesis tool and normalized to the latency of an adder.

We also assume that the basic block execution frequency for all basic blocks is the same. This assumption is used in the calculation of latency gain. The number of clock cycles saved in the execution of a program due to the addition of ISEs to the ISA should be calculated as the sum of number of cycles saved in a basic block multiplied by the number of times that the basic block is called. Our assumption is that every basic block is called only once, and every identified instruction is selected, even though many of them are mutually exclusive. This is an incorrect and very crude assumption, but it is the best guess we can make until we have developed our own basic block profiler for COINS. This assumption means that the numerical results from this thesis cannot be directly compared to other work using these benchmarks because the basic block execution

frequency is off. However, we assume that the trends in the data will hold true when correct basic block profiling is implemented, and the numerical results in this thesis can be compared to each other.

System designers need the ability to limit the size of custom instruction hardware, as well as the hardware size of the whole embedded system. The hardware size of the identified ISEs can be limited by the user by setting a parameter to constrain the design space. The assumptions made in this section about hardware size can be changed by the tool user in the function presented in Appendix A.5. During instruction selection, the hardware size function will be used to avoid exceeding the constraint on the size of the system hardware.

Although hardware sizes are set in a table inside the compiler, these values can be set by the user to whatever the implementation costs are for the target FPGA or ASIC technology. In our framework we set the cost of a 32-bit adder to 20 and the cost of a 32-bit multiplier to 100. These numbers do not resolve to real hardware size metrics such as gate count or logic elements in an FPGA. The available hardware size will be used to approximate the amount of hardware available for adding ISEs during instruction selection, which was not yet implemented, and so it should not affect the results of the tests in this chapter. As discussed earlier, some ISEs are equivalent and must be grouped by running as isomorphism test with a package such as nauty [117]. In these experiments no isomorphism testing was performed.

Several parameters were changed to various settings to obtain the data in this section of the thesis. These parameters are now described. The variable *private static final boolean GRAPH* is used to enable or disable graph printing. If enabled, the compiler will print a dataflow graph for each basic block encountered and a control flow graph for each control flow graph encountered. The variable

private static final boolean MULTITHREAD is used to enable or disable multithreaded solution of instruction set extension identification algorithm. If multithreading is enabled, then *private static final int NUMTHREADS* is used to set the number of threads in the thread pool which will consume tasks. The variable *private static final boolean TIMED_SOLVER* is used to enable or disable a limit on the execution time of a call to the LINDO 11 optimization software. If the limit is activated, then *private static final int remaining_exec_time* is used to set the number of seconds to wait before the solver is interrupted and forced to return the best feasible solution available so far.

The following variables were used to set the maximum number of inputs and outputs to and from an ISE, and the minimum speedup (improvement in latency in this context) that an ISE must provide in order to be valid, respectively:

private static final int max_inputs=6;

private static final int max_outputs=6;

private static final int incumbent_speedup=0;

4.3 Control Case

We first present a control case for the results sections which demonstrates that the ISE identification algorithm is functioning correctly. The program that was compiled is called "simpler.c". The code can be found in Appendix A.1. The results of the compilation are shown in Appendix B.1. Our debug output for the dataflow graph is shown in Figure 5.

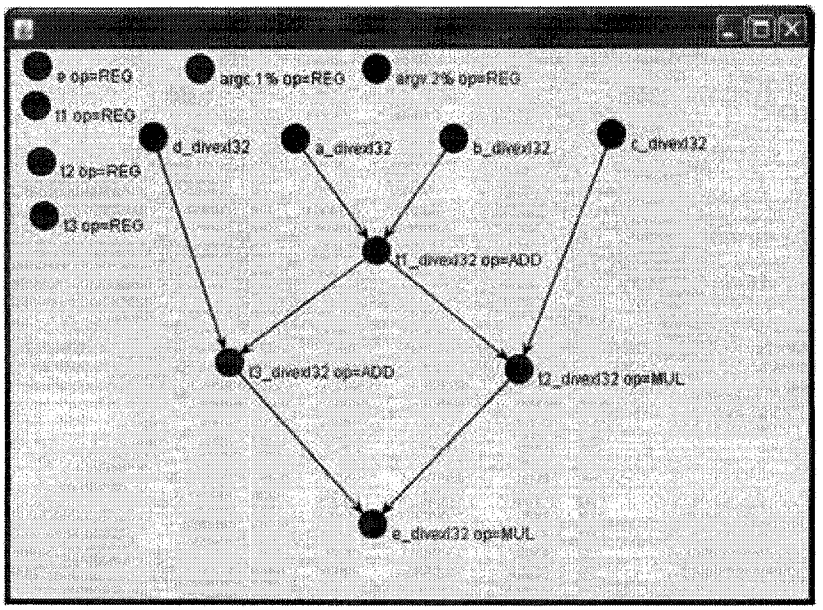

Figure 5: Dataflow graph for simpler.c

Let the notation 6C4 refer to 6 choose 4 combinatorial operation. In combinatorial mathematics, the result of such a choose operation is the maximum number of ways that 4 objects can be selected from a set of 6 objects while discounting the order of the objects taken from the set of 6 objects. Referring back to our example, there are 4 nodes in Figure 10 that process data. Only sets of 2 or more nodes can be selected as an ISE. We assume that the number of inputs and outputs to an ISE is 6 or less, and the hardware size allowed is set very high. Therefore there are 4C2 + 4C3 + 4C4 = 6+ 4 + 1 = 11 possible ISEs to find. We further reduce this set of 11 legal ISEs by three, since the three ISEs [t1, t3, e], [t1, t2, e], and [t1, e] are non-convex on the graph. Put another way, these three patterns have outputs that lead back to the inputs of the same subgraph through computation nodes that are not in the subgraph. And so we can say definitively that the ISE identification algorithm must find exactly 8 ISEs. As shown in the output in Appendix B.1, there are 8 identified ISEs as expected, and the ISEs do not include single-node cases or the three non-convex cases mentioned. The node disabling approach only finds one ISE: [t2, t3, e]. Testing has demonstrated that the ISE identification algorithm performs correctly.

4.4 Varying I/O Constraints

To obtain the data presented in Table 5, the COINS compiler with our ISE pass was used to compile the AES, FFT, and SHA benchmarks in the MiBench benchmark suite. Table 5 shows the change in the number of ISEs identified based on changes in the input and output port constraints. The baseline configuration for the tool is an ISE I/O constraint of 4 inputs and 4 outputs. We expect that reducing the number of inputs and outputs allowed for an ISE improves the compiler execution time, but at the cost reduced design space exploration and therefore reduced number of identified instructions, and reduced latency gain. As expected, more ISEs are found as the constraints on the ILP get looser for AES. This was not the case for FFT and SHA.

Experiment	**Benchmark**	**Compiler Execution Time 3 Samples and Mean (Seconds)**				**# Identified ISEs 3 Samples and Mean**			
		1	2	3	Mean	1	2	3	Mean
I/O constraint of (2,2)	AES	349	340	334	341	357	357	357	357
I/O constraint of (4,4), baseline	AES	424	424	424	424	620	619	619	619
I/O constraint of (6,6)	AES	416	421	437	425	659	658	658	658
I/O constraint of (2,2)	FFT	265	271	281	272	493	493	493	493
I/O constraint of (4,4), baseline	FFT	405	438	433	425	793	780	779	784
I/O constraint of (6,6)	FFT	383	376	377	379	816	816	816	816
I/O constraint of (2,2)	SHA	137	137	138	137	328	328	328	328
I/O constraint of (4,4), baseline	SHA	267	269	267	268	585	585	585	585
I/O constraint of (6,6)	SHA	265	266	265	265	599	599	599	599

Table 5: ISEs identified under various input and output constraints

We can now see in the following 3 figures which summarize the table above that the input and output port constraints in the model are initially very important and then the cost of increasing the

number of inputs and outputs decreases as the constraint is loosened. We hypothesize that the expected combinatorial explosion is likely to occur despite this apparent trend. As the ISEs are allowed to grow in size there are fewer and fewer convex ISEs, but the number of node combinations in the solution space grows to be very large. We did find that increasing the I/O constraint to (100, 100) caused the solver to execute endlessly. We conclude that there must be a bound on the ILP that branch and bound can apply, or else the solver will have to enumerate very large sections of the solution space. When the I/O constraint is small, there is a small bound on the solution to the model.

We can observe in the following figures that as we loosen the I/O constraint the compiler execution time increases, the number of ISEs identified increases, and the latency gain for all ISEs increases. The exception is with FFT when changing the I/O constraint from (4, 4) to (6, 6).

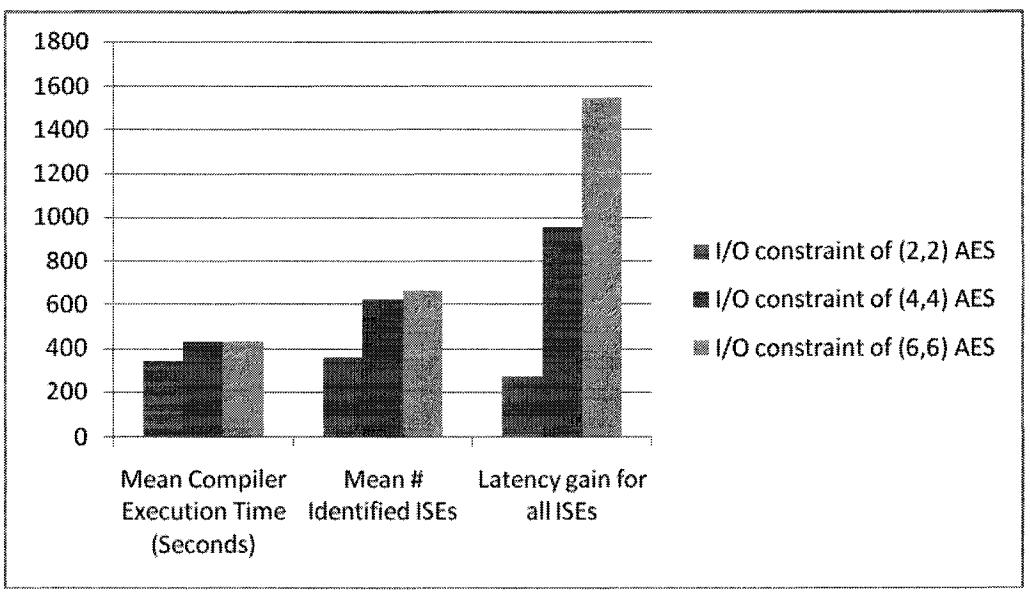

Figure 6: Mean compiler execution time, mean number of ISEs identified, and latency savings for all ISEs in the AES benchmark under various I/O constraints.

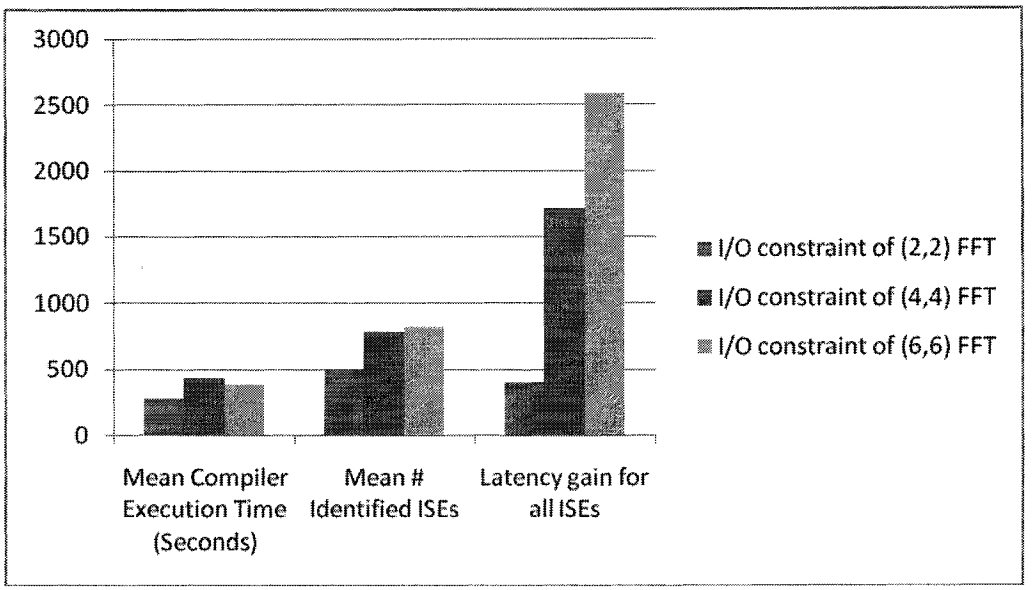

Figure 7: Mean compiler execution time, mean number of ISEs identified, and latency savings for all ISEs in the FFT benchmark under various I/O constraints.

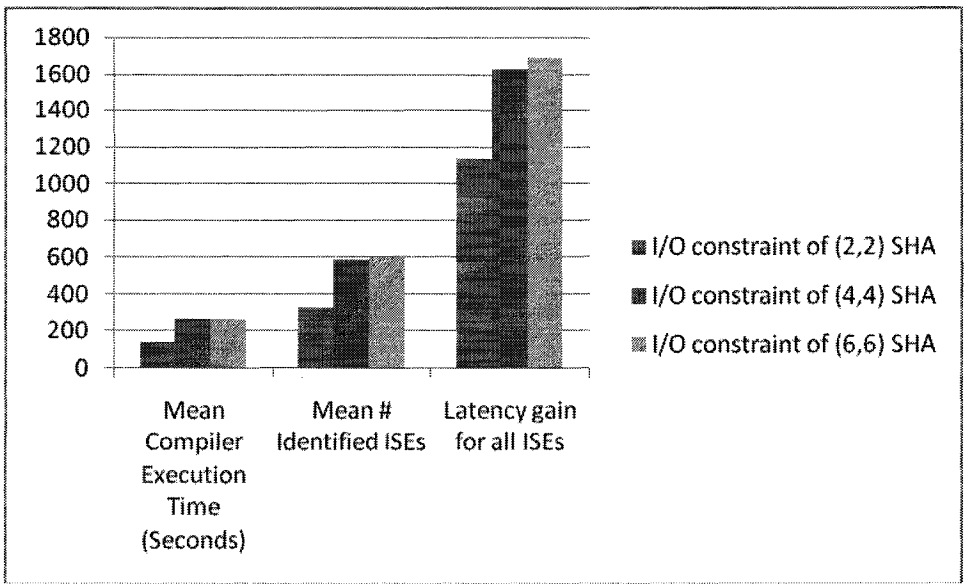

Figure 8: Mean compiler execution time, mean number of ISEs identified, and latency savings for all ISEs in the SHA benchmark under various I/O constraints.

4.5 ISEs Enumeration Approaches

We now test Proposal III. The approach followed in this thesis for selecting ISEs is different from

previous work. In the baseline configuration of our work all legal ISEs are enumerated. We implemented a second version of the ISE pass in the COINS compiler which sets the nodes associated with any already identified ISE to be disabled in the model. In Table 6 below, we compare the two approaches. The latency gain is presented in the figures for each benchmark, but not in Table 6.

Experiment	**Benchmark**	**Compiler Execution Time 3 Samples and Mean (Seconds)**				**# Identified ISEs 3 Samples and Mean**			
		1	2	3	Mean	1	2	3	Mean
I/O chonstraint of (4,4), enumerate all ISEs	AES	424	424	424	424	620	620	620	620
I/O constraint of (4,4), disable ISE nodes	AES	61	60	60	60	65	65	65	65
I/O constraint of (4,4), enumerate all ISEs	FFT	405	438	433	425	793	780	779	784
I/O constraint of (4,4), disable ISE nodes	FFT	25	25	26	25	39	39	39	39
I/O constraint of (4,4), enumerate all ISEs	SHA	267	269	267	268	585	585	585	585
I/O constraint of (4,4), disable ISE nodes	SHA	18	18	18	18	38	38	38	38

Table 6: ISE identification runtime for various benchmarks and settings

Enumerating all ISE nodes and disabling patterns of ISEs in the model slows down the compiler between 7 and 17 times the alternative of disabling nodes of identified ISEs in the ILP model. However, following the pattern disabling approach we see a 12 to 33 times improvement in the latency gain of the ISEs identified by the model, and a mean number of ISEs identified 9.5 to 20

times higher than the alternate approach. We can see the very large improvements in the figures below.

In the following three figures the data from Table 6 is graphed along with the latency gain observed for each benchmark. It is apparent that forcing nodes out of the solution does reduce the compiler execution time, but this comes at the cost of a severely limited view of the design space. Following this method for ISE enumeration only 60/424=14% of the solution space was enumerated for AES. For FFT, 6% of the solution space was enumerated, and 7% was enumerated for SHA. This represents in our view an unacceptably blunt heuristic. Consider that the second best ISE in all basic blocks in a program may be the same ISE, but in each basic block the ISE is not enumerated because it shares nodes with the best ISE. This is an example of a situation where our method will find the highest speedup ISEs for producing an embedded system and the method from [65] will not.

It should be noted that there is an advantage to disabling nodes in the model. The approach guarantees that all enumerated ISEs are mutually exclusive. This allows the tool to implement all identified ISEs during instruction selection without considering which nodes the ISE is involved with. Our method requires a list of node IDs to accompany each identified ISE so that during instruction selection the ISEs with nodes intersecting a selected ISE can be removed from the list of candidate ISEs.

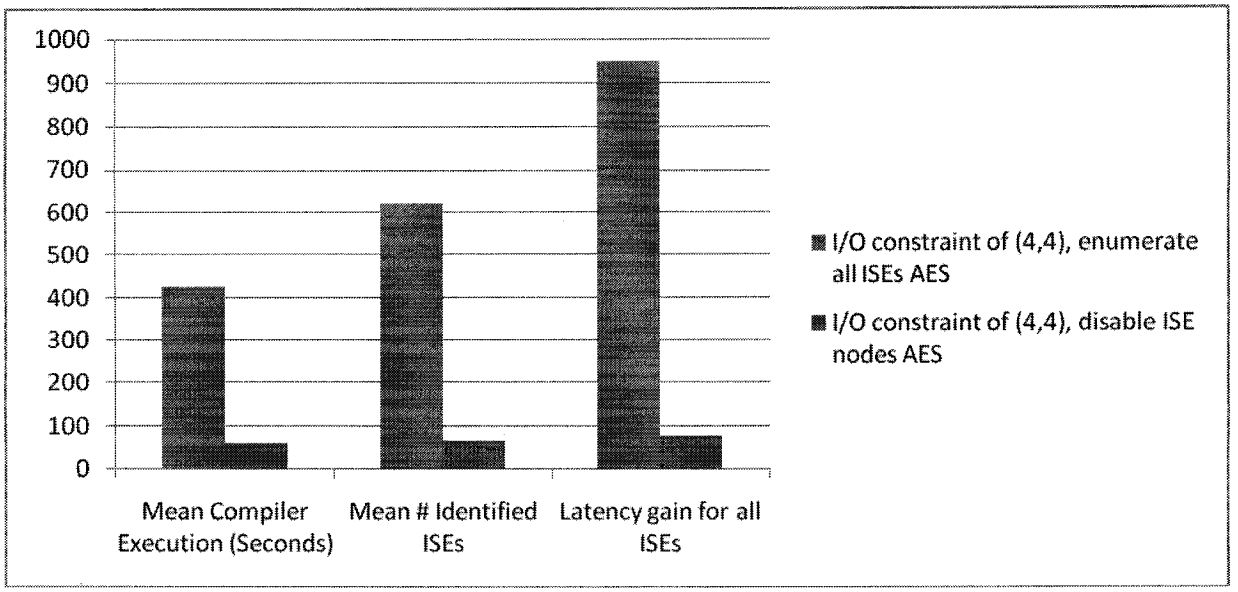

Figure 9: Mean compiler execution time, mean number of ISEs identified, and latency saved for all ISEs in the AES benchmark for node disabling and pattern disabling approaches.

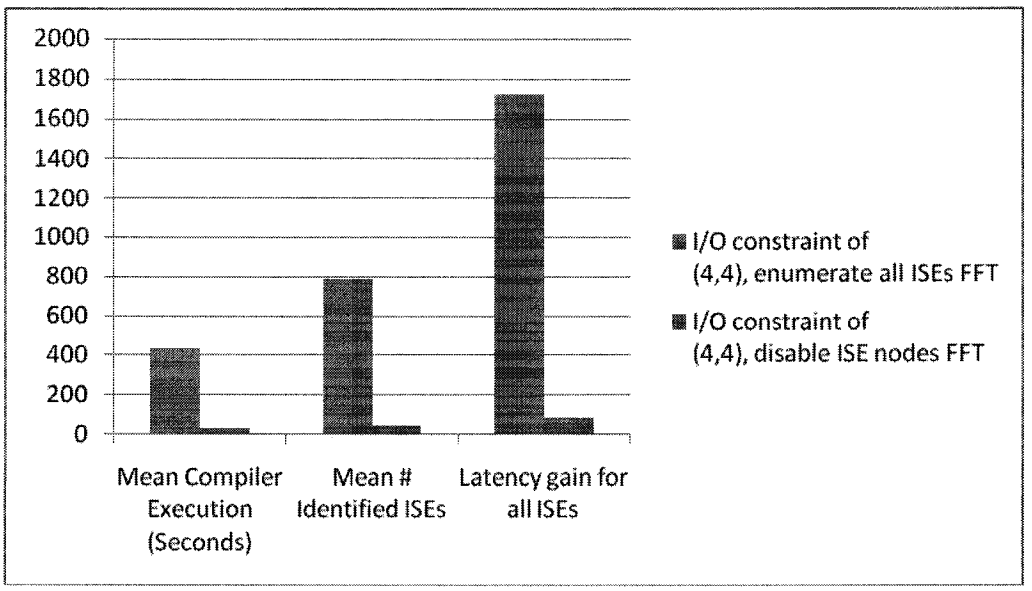

Figure 10: Mean compiler execution time, mean number of ISEs identified, and latency saved for all ISEs in the FFT benchmark for node disabling and pattern disabling approaches.

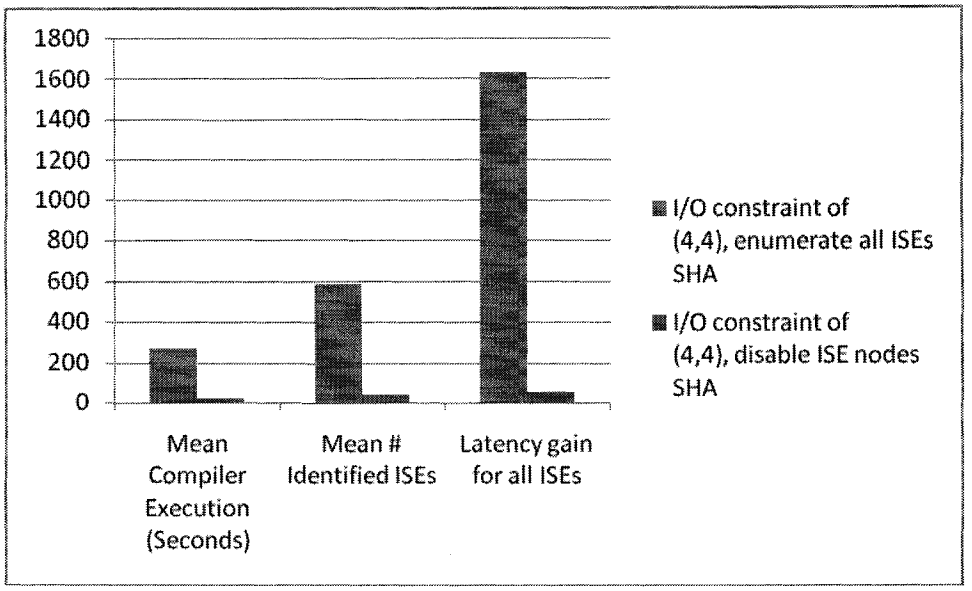

Figure 11: Mean compiler execution time, mean number of ISEs identified, and latency saved for all ISEs in the SHA benchmark for node disabling and pattern disabling approaches.

4.6 Limited Solver Execution Time

We now test Proposal I. To obtain the results presented in Table 7, we varied the maximum allowable execution time of the solver. We expect that this approach should reduce the size of the ISE design space and therefore reduce compile time. We can see that as expected the solver found more ISEs when it was given more time to execute. Since we already know that the solutions with the highest speedup in a basic block are found first, we conclude that this technique is beneficial when the designer is looking for a few maximum speedup ISEs in a kernel, but it may not help across a whole program because the least attractive solutions that come out of the solver last may be pruned in large basic blocks. This will cause some ISEs that appear in many basic blocks to be under-represented during selection of the custom instructions.

Experiment	Benchmark	Compiler Execution Time 3 Samples and Mean (Seconds)				# Identified ISEs 3 Samples and Mean			
		1	2	3	Mean	1	2	3	Mean
Solver execution time cutoff at 1 second	AES	448	428	426	434	667	667	667	667
Solver execution time cutoff at 0.5 seconds	AES	291	292	288	290	520	523	530	524
Solver execution time cutoff at 0.3 seconds	AES	138	144	145	142	333	344	350	342
Solver execution time cutoff at 1 second	FFT	418	396	396	403	343	342	343	343
Solver execution time cutoff at 0.5 seconds	FFT	392	402	395	396	791	780	779	783
Solver execution time cutoff at 0.3 seconds	FFT	293	294	279	289	694	692	663	683
Solver execution time cutoff at 1 second	SHA	286	279	272	279	699	699	699	699
Solver execution time cutoff at 0.5 seconds	SHA	264	248	281	264	561	564	585	570
Solver execution time cutoff at 0.3 seconds	SHA	192	187	195	191	506	498	513	506

Table 7: Solver execution time limit vs. number of identified ISEs

The data in Table 7 is simplified into the three graphs below. We can see right away that there are several trends in these experiments. Firstly, the number of ISEs identified decreases as the compiler

execution time decreases. Second, the compilation time required for a low execution time cutoff is lower than the execution time required by a large execution time cutoff.

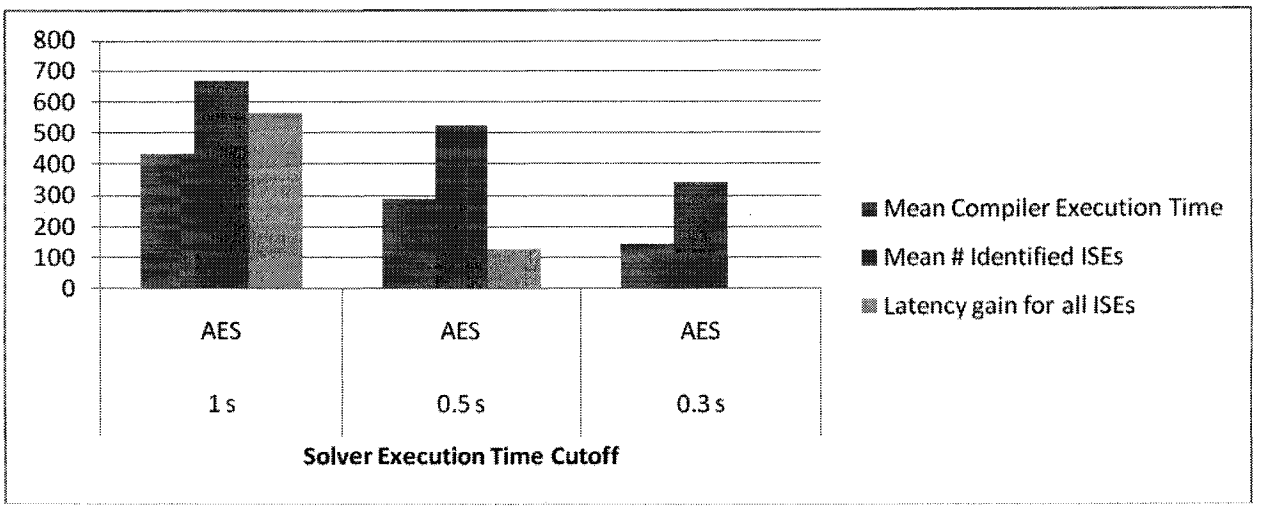

Figure 12: Mean number of identified ISEs, mean compiler execution time, and estimated latency gain for various solver execution time cutoff scenarios for the AES benchmark.

The results for FFT did not follow the trends seen for AES and SHA. This results is unexpected because the number of identified ISEs increases in this case as the compiler has less and less time to search for ISEs. It is entirely possible that the way the solver iterates the ILP solution is different under time constraints, but it is not apparent why that would happen.

An interesting trend to watch is the latency gain on the solution. Consistently, the number of cycles saved by all ISEs shrinks as the compiler cutoff shrinks. This indicates that the solution is getting worse as the compile time is decreased. Although the number of identified ISEs decreases as well, it does not decrease as quickly as the latency gain. In other words, we are paradoxically discarding the best ISEs. Our expectation was that the worst ISEs would be removed first. Perhaps large basic blocks are failing to search large portions of the design space due to time restrictions.

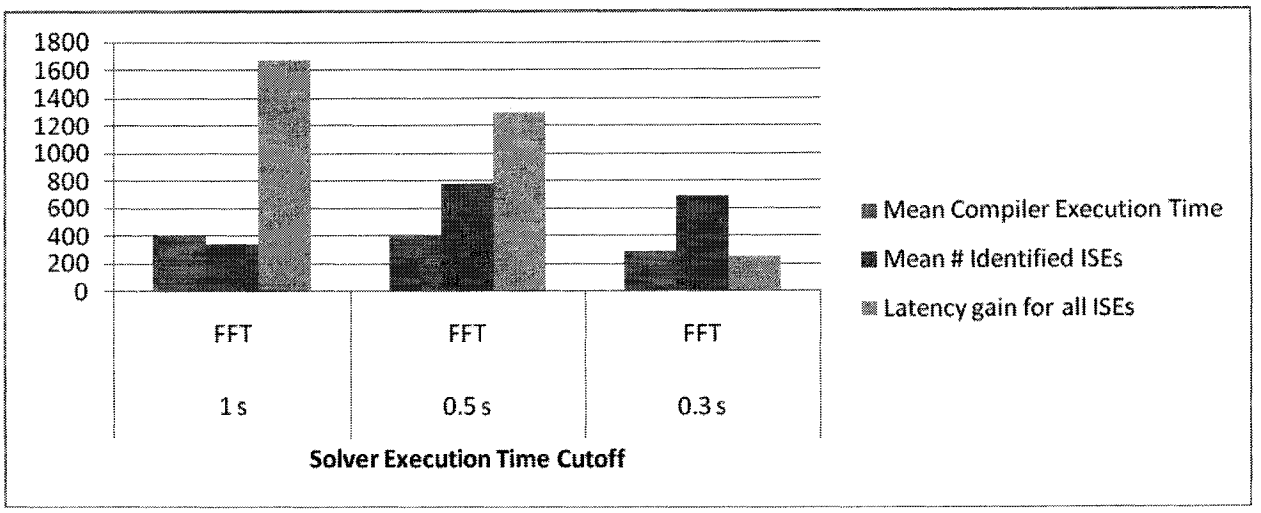

Figure 13: Mean number of identified ISEs, mean compiler execution time, and estimated latency gain for various solver execution time cutoff scenarios for the FFT benchmark.

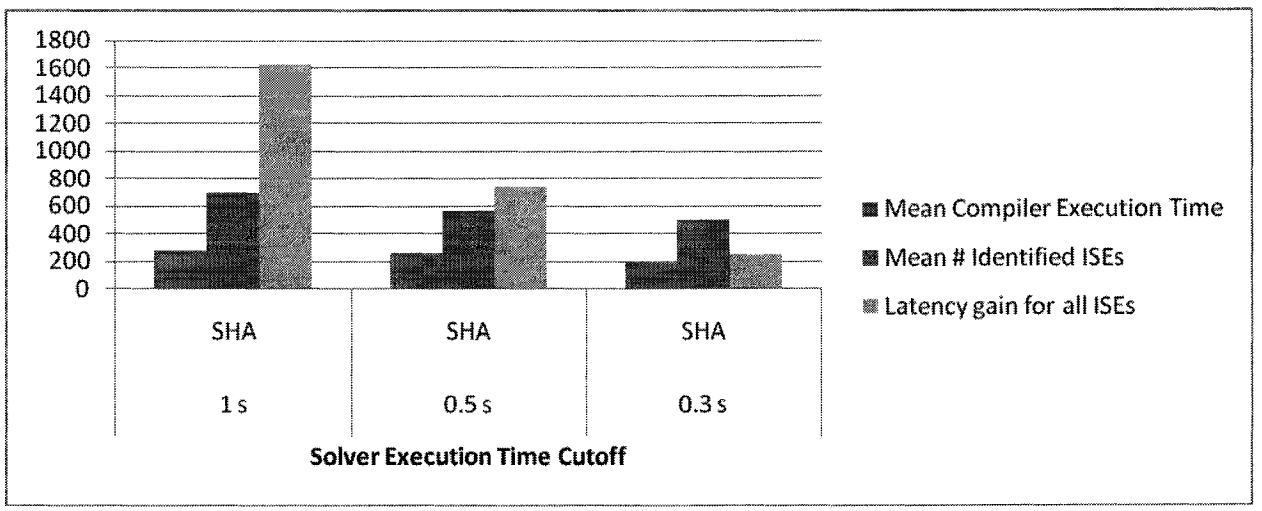

Figure 14: Mean number of identified ISEs, mean compiler execution time, and estimated latency gain for various solver execution time cutoff scenarios for the SHA benchmark.

The gathered data indicates that the compilation time is not always the same for identical trials when a compiler execution time cutoff is introduced. This randomness may be a result of the operating system handling the switch between the compiler and the LINDO solver. Compared to the baseline compilation from Table 5, FFT can be processed 1.47 times faster (425/289), at a cost of

ignoring 12% of the solution space ((779-683)/779). Similarly, SHA and AES can be processed faster at the cost of enumerating a smaller percentage of the solution space.

Although this approach was successful at speeding up the compilation time, we have not yet ensured that there are only low-speedup ISEs skipped by the execution time limit on the solver. There are situations where finding the first ISE in the basic block takes so long that the block is skipped altogether, removing both high value and low value ISEs from the solution. The strange results for the FFT benchmark are also a sign that limiting the execution time of the solver did not work as expected.

Since we already know that the solutions with the highest speedup in a basic block are found first, we conclude that this technique is good when looking for maximum speedup in a basic block, but it may not help across a whole program because the least attractive solutions that come out of the solver last may be pruned in large basic blocks. The incremental benefit of increasing the solver's time to think drops off as the cutoff gets larger.

4.7 Multithreading ISE Identification

We now test Proposal II. Several threading approaches were attempted in order to minimize the ISE identification execution time. First the baseline version of the compiler was executed. Next a multithread version of the compiler was used to distributed basic blocks to worker threads. At the program level in the compiler each function was dissected into basic blocks and each block was fed to any available worker thread. This multithread version of the compiler was executed with a thread group of 2 threads.

Experiment	Benchmark	Compiler Execution Time 3 Samples and Mean (Seconds)				# Identified ISEs 3 Samples and Mean			
		1	2	3	Mean	1	2	3	Mean
Baseline	AES	424	424	424	424	620	619	619	619
Multithreading at program level with 2 threads	AES	220	268	214	234	619	619	619	619
Baseline	FFT	405	438	433	425	793	780	779	784
Multithreading at program level with 2 threads	FFT	260	143	143	182	778	779	778	778
Baseline	SHA	267	269	267	268	585	585	585	585
Multithreading at program level with 2 threads	SHA	158	157	161	159	584	584	584	584

Table 8: Baseline vs. multithreading ISE identification algorithm.

The multithreading approach was very successful at speeding up the solution time and delivered the same results as the sequential solution but faster. The speedup of the compiler execution time ranges between 1.6 and 2.3 times. It is possible that the ISE identification algorithm will perform even better if executed in a cluster of computers that distributes the calculation. In Table 8 above we can see that the results from the multithreaded version of the ISE identification algorithm were very stable. Although the execution time of the compiler varied slightly, the solution returned by the compiler was always consistent. This indicates that there is a benefit to multithreading the ISE identification problem, but no cost in terms of reduced solution quality.

We can see this fact more easily in the figures below where the mean compiler execution time is

higher for the baseline, but the mean number of ISEs returned by the compiler is the same as the baseline.

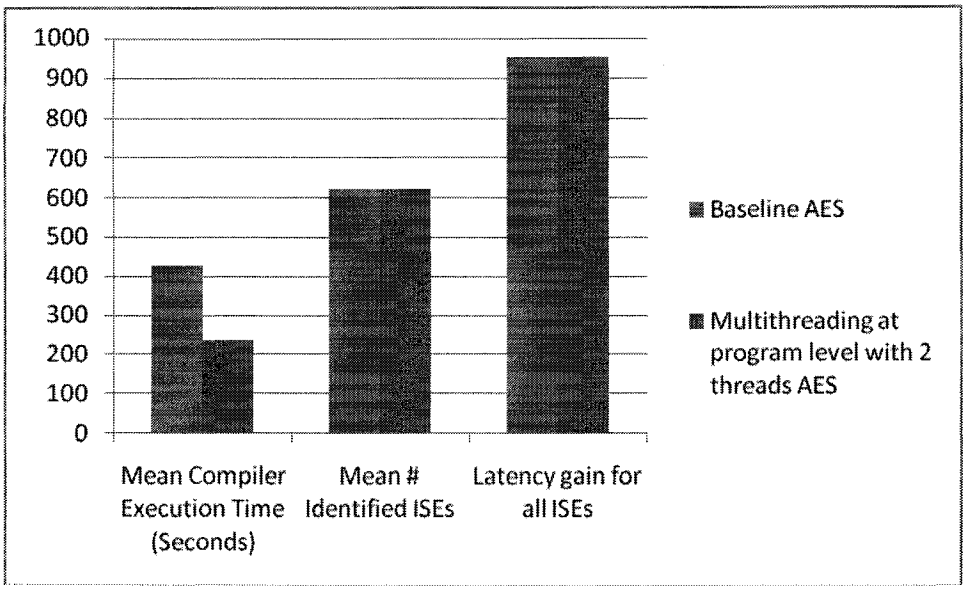

Figure 15: Mean number of identified ISEs for AES, mean compiler execution time, and saved latency of all identified ISEs for the baseline configuration and the multithreading version of the ISE identification algorithm.

One anomaly in the data is the latency saved by multithreading the FFT benchmark. We can see in Figure 16 that the total latency slightly exceeded the baseline, which is odd because the baseline identifies all valid possible ISE. This result was checked twice and persisted.

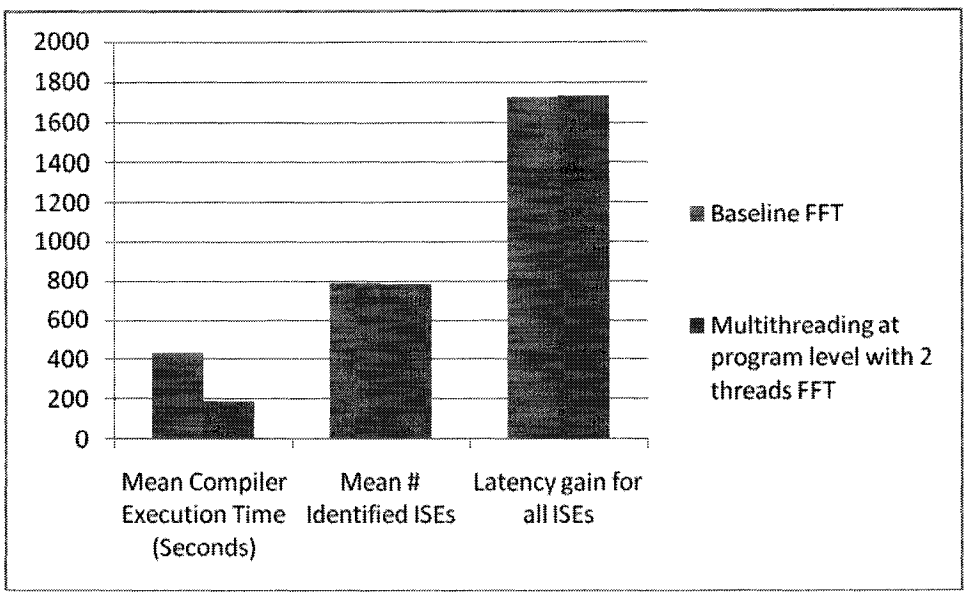

Figure 16: Mean number of identified ISEs for FFT, mean compiler execution time, and saved latency of all identified ISEs for the baseline configuration and the multithreading version of the ISE identification algorithm.

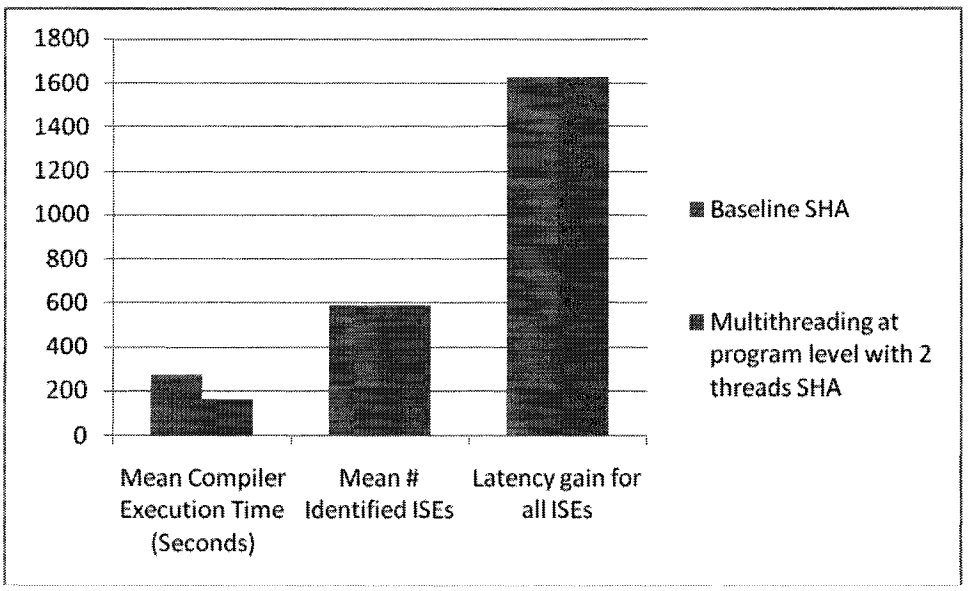

Figure 17: Mean number of identified ISEs for SHA, mean compiler execution time, and saved latency of all identified ISEs for the baseline configuration and the multithreading version of the ISE identification algorithm.

Running the multithreading algorithm inside each function instead of at the program level resulted in

a slower execution time. It appears that at the function level, there is often one large basic block per function, which leaves little room for parallelization because all threads must complete work before the next function can be processed, and most of the time one thread is available and waiting, while the other processes the large basic block. This has the effect of causing the basic blocks to be processed more sequentially than in the program level implementation.

In the "Windows Task Manager", the performance tab showed that for multithreading the processing was occurring in both processors. The CPU usage climbed to 100% and stayed there until the solution was completed. This indicates that the results will be even better for a faster processor.

4.8 Ignoring Small Speedup Patterns in ISE Identification

We now test Proposal IV. In this section we evaluate the effect of pruning ISEs that provide a very small speedup from the ISE identification algorithm. The results for both the pruning and baseline compiler configurations were tested with several benchmarks, and the results are presented in Table 9 below.

Experiment	**Benchmark**	**Compiler Execution Time 3 Samples and Mean (Seconds)**				**# Identified ISEs 3 Samples and Mean**			
		1	2	3	Mean	1	2	3	Mean
cutoff = 0	AES	424	424	424	424	620	619	619	619
cutoff = 1	AES	207	205	212	208	238	238	238	238
cutoff = 0	FFT	405	438	433	425	793	780	779	784
cutoff = 1	FFT	302	307	323	311	633	633	633	633
cutoff = 0	SHA	267	269	267	268	585	585	585	585
cutoff = 1	SHA	83	83	83	83	167	167	167	167

Table 9: Baseline v.s. ignoring ISEs with a latency gain of less than 1 clock cycle

We calculate that the speedup in compiler execution time resulting from the "cutoff=1" approach is between 1.3 and 3.2 times. This approach also decreased the number of enumerated ISEs between 28.5% and 80.7%.

We can see that increasing the threshold for ISE speedup results in the compiler execution time being cut very significantly along with the number of identified ISEs. However, unlike the solver execution time cutoff, we know that the discarded solutions are only the low speedup ISEs and never the high speedup ISEs. However because of the way that the number of saved cycles is calculated, ISEs composed of chains of operations will not be enumerated, while wide bandwidth instructions are enumerated. That said, we can conclude that the ISEs enumerated are the high speedup ISEs. As mentioned earlier, a small set of high speedup ISEs is better suited to speeding up a loop kernel or program hotspot than a general program. These ISEs are generally not very flexible, and contain several nodes. However, since we set the requirement for selection so low (ISE must save more than 1 clock cycle) we can assume that the ISEs will not be completely unique to the basic block where they were found. In the following figures we see that AES and SHA contained a high percentage of low value ISEs, while FFT contained a low percentage of low value ISEs.

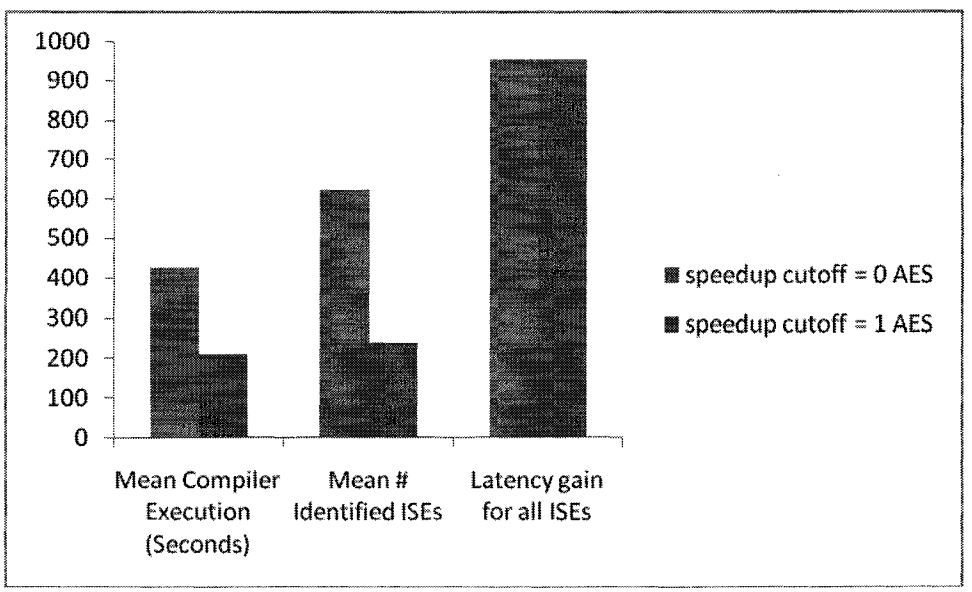

Figure 18: Mean number of identified ISEs for AES, mean compiler execution time, and total latency saved by all ISEs. These results are for a model which ignores low speedup ISEs and the baseline model.

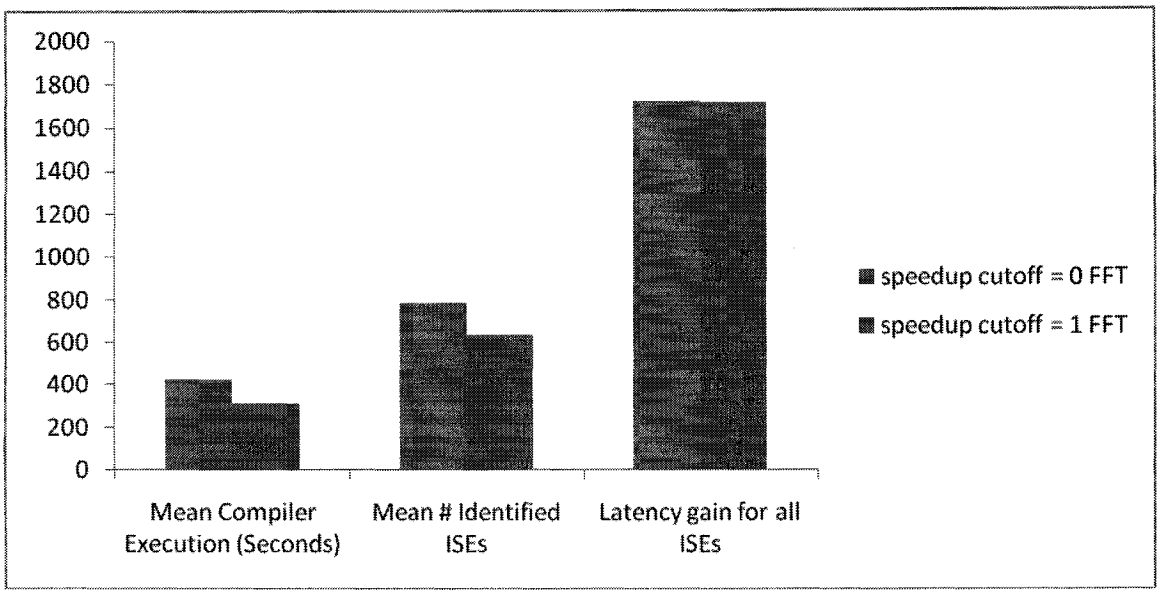

Figure 19: Mean number of identified ISEs for FFT, mean compiler execution time, and total latency saved by all ISEs. These results are for a model which ignores low speedup ISEs and the baseline model.

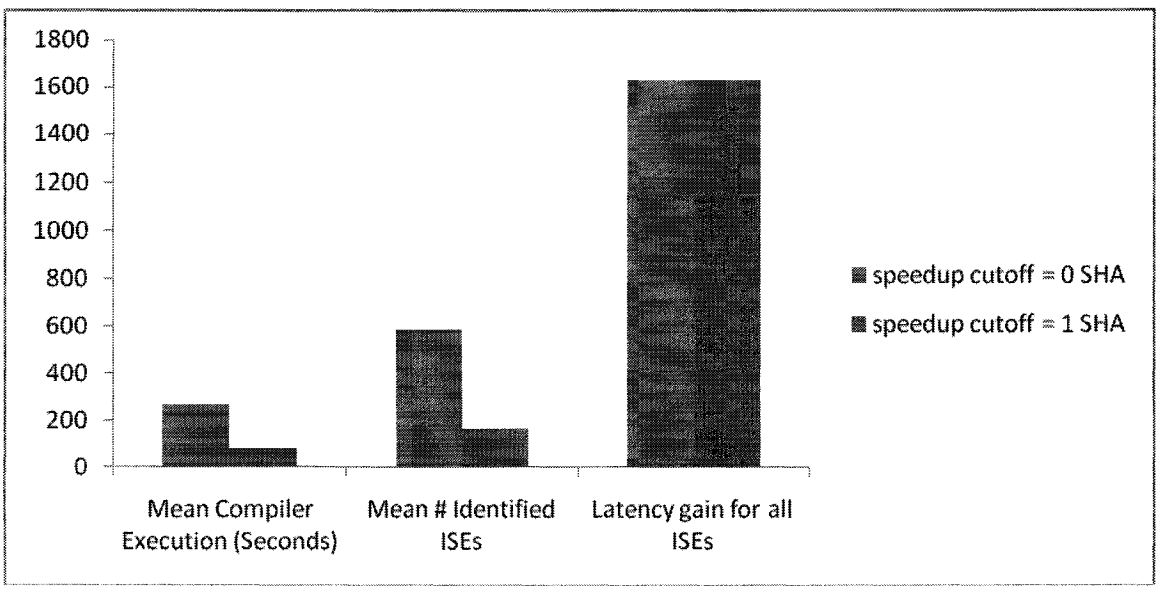

Figure 20: Mean number of identified ISEs for SHA, mean compiler execution time, and total latency saved by all ISEs. These results are for a model which ignores low speedup ISEs and the baseline model.

We can also see in these results that the number of ISEs in the form of a sequential path was high for all three benchmarks. It is those completely sequential datapaths that have a calculated speedup

of 0 clock cycles. We showed that pruning the lowest performing ISEs directly in the ISE identification algorithm can be beneficial. Although it may be beneficial in many situations to set the minimum objective function value to a small number instead of 0, the attempt to speedup the ISE identification algorithm using a higher number will probably severely reduce the optimality of the solution.

4.9 Several ISE Identification Approaches Combined

In order to see if better results could be obtained by combining approaches, the AES, FFT, and SHA benchmarks were compiled with the settings for 2 threads, an I/O constraint of (10, 10), no limit on the solver execution time, and a minimum latency gain of 1 for ISEs. We chose to use multithreading because the benefits were high (approximately 2x speedup) and there was no apparent cost in terms of solution quality (# of identified ISEs and speedup potential of the ISEs). The I/O constraint was loosened so that ISEs with high I/O could be detected as we expect will happen in accordance with the trend in I/O constraints in the test of section 4.4. An I/O constraint of (10, 10) was selected because the previous tests showed that for small I/O constraints the solution time is tractable. We chose to disable in the model ISEs having a speedup of less than 1 clock cycle so that the gain of the solution space will remain high while the compile time searching for low speedup ISEs will remain low.

Forcing the latency gain of instructions to at least 1 reduces the number of instructions by discarding many low latency gain instructions. We can see this happen as the number of instructions identified by this experiment is less than the baseline. However, the increased I/O constraint finds us much better ISEs that the baseline cannot see, drastically improving the saved latency. Also, we use multithreading to run the solution faster than the baseline.

Experiment	Benchmark	Compiler Execution Time 3 Samples and Mean (Seconds)				# Identified ISEs			
						3 Samples and Mean			
		1	2	3	Mean	1	2	3	Mean
Baseline	AES	424	424	424	424	620	619	619	619
Multithreading at program level with 2 threads, incumbent speedup (cutoff) = 1, I/O constraint of (10,10)	AES	300	334	314	317	330	330	330	330
Baseline	FFT	405	438	433	425	793	780	779	784
Multithreading at program level with 2 threads, incumbent speedup (cutoff) = 1, I/O constraint of (10,10)	FFT	384	485	369	413	675	675	675	675
Baseline	SHA	267	269	267	268	585	585	585	585
Multithreading at program level with 2 threads, incumbent speedup (cutoff) = 1, I/O constraint of (10,10)	SHA	67	72	71	70	154	154	154	154

Table 10: Baseline configuration compared to a configuration with 2 threads, an I/O constraint of (10, 10) and a minimum ISE speedup of 1 cycle.

The results for these experiments are presented in Table 10, and they are quite good. We observe that the saved latency of ISEs jumps by 1.1 to 2.2 times, while the compiler runs between 1.0 and 3.8 times faster than the baseline. These gains were accompanied by a drop in the number of enumerated ISEs by 1.1 to 3.8 times.

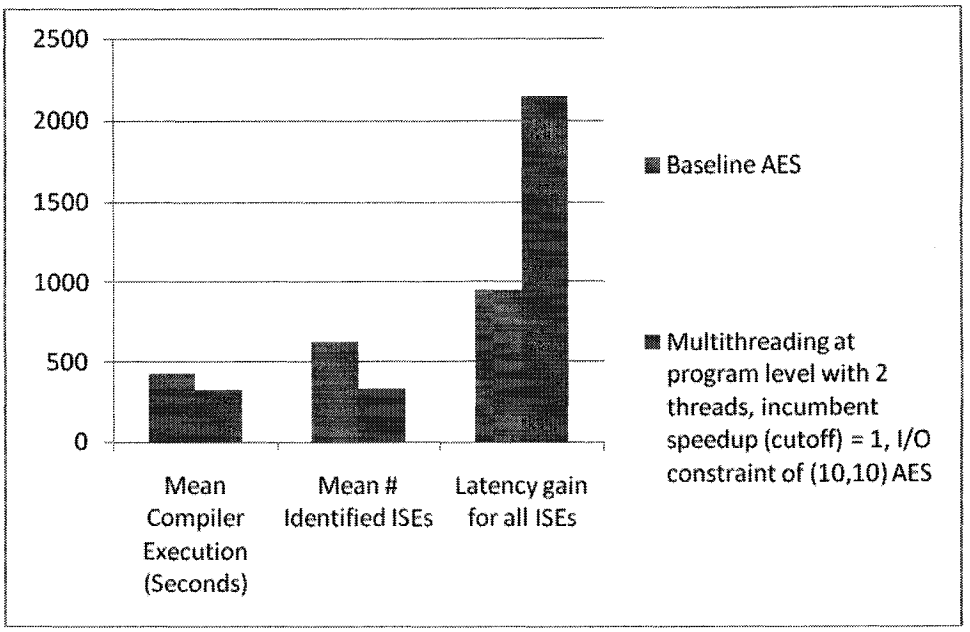

Figure 21: Mean number of identified ISEs for AES, mean compiler execution time, and total latency saved by all ISEs. These results are for the baseline model, and a model with a (10, 10) I/O constraint, 2 threads for finding ISEs in basic blocks, and which ignores low speedup ISEs.

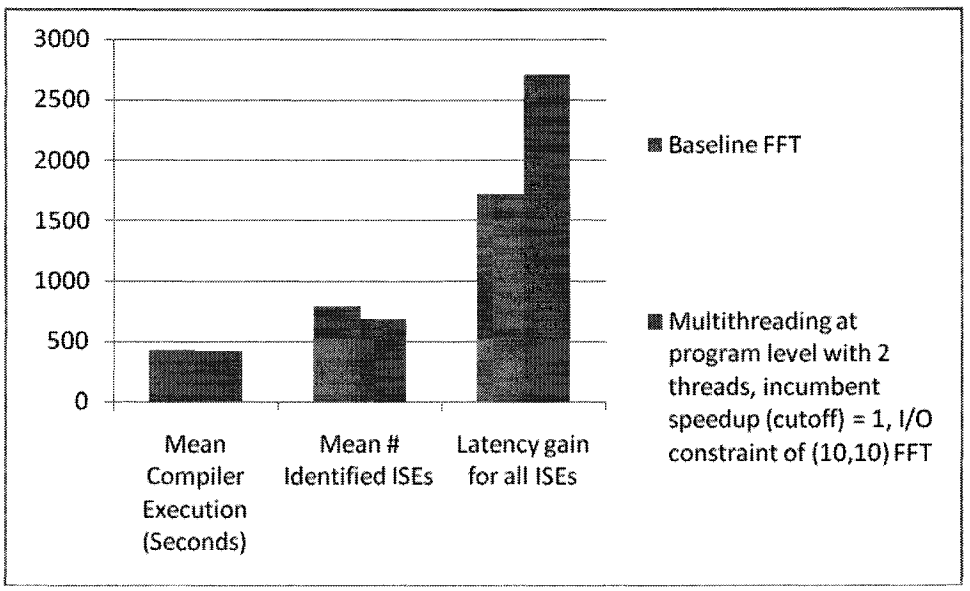

Figure 22: Mean number of identified ISEs for FFT, mean compiler execution time, and total latency saved by all ISEs. These results are for the baseline model, and a model with a (10, 10) I/O constraint, 2 threads for finding ISEs in basic blocks, and which ignores low speedup ISEs.

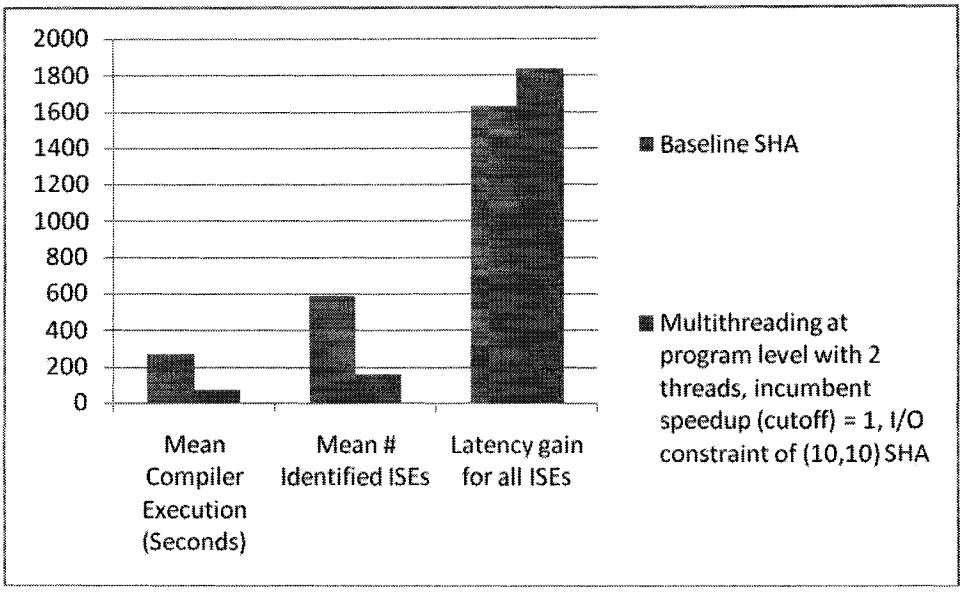

Figure 23: Mean number of identified ISEs for SHA, mean compiler execution time, and total latency saved by all ISEs. These results are for the baseline model, and a model with a (10, 10) I/O constraint, 2 threads for finding ISEs in basic blocks, and which ignores low speedup ISEs.

4.10 Analysis of Results

We have shown that an ISE identification algorithm was correctly implemented into the COINS compiler. The execution time, number of ISEs identified, and total cycles saved for the ISEs identified by the algorithm were improved using several techniques. The table below contains a summary of the results. There is an experiment label next to each experiment description in the table so that each experiment can be referred to more easily in the following figures and text.

There were thirty experiments overall, each with slightly different input parameters to our ISE identification algorithm. All of the data from the graphs and table in this section can be found in earlier sections of the thesis. In this section we look at the whole data set to search for trends about the ISE identification design space as a whole. We can see in Figure 24 that there is a weak but visible correlation between the latency gain for all ISEs and the mean number of ISEs

identified. This does make sense because the more ISEs there are to choose from the higher the possibility will be that the ISE is parallel. This observation supports our contention that enumerating all ISEs in the basic block is a better approach than disabling the nodes in the ILP model corresponding to each identified ISE.

It is also interesting to note in Table 11 that 5 of the top 12 latency saving experiments actually discarded ISEs from the solution by forcing the saved latency to be greater than 1. Keep in mind that the saved latency formula is biased towards wide ISEs rather than ISEs with chained operations. In our experiments the low speedup ISEs discarded did not sacrifice any potential latency, but the long chained ISEs may have provided high speedup. A speedup calculation or a function that accounts for latency and throughput would most likely perform even better, because the long thin ISEs would not be discarded.

Experiment Label	Disable patterns or nodes	Multithreading with 2 threads	Solver execution time cutoff	I/O constraint	Speedup cutoff
E1	Patterns	2 threads	No solver time cutoff	(10,10)	Minimum speedup=1
E2	Patterns	No Multithreading	No solver time cutoff	(6,6)	No speedup cutoff
E3	Patterns	2 threads	No solver time cutoff	(10,10)	Minimum speedup=1
E4	Patterns	2 threads	No solver time cutoff	(10,10)	Minimum speedup=1
E5	Patterns	2 threads	No solver time cutoff	(4,4)	No speedup cutoff
E6	Patterns	No Multithreading	No solver time cutoff	(4,4)	No speedup cutoff
E7	Patterns	No Multithreading	No solver time cutoff	(4,4)	Minimum speedup=1
E8	Patterns	No Multithreading	No solver time cutoff	(6,6)	No speedup cutoff
E9	Patterns	No Multithreading	1 second solver time cutoff	(4,4)	No speedup cutoff
E10	Patterns	No Multithreading	No solver time cutoff	(4,4)	No speedup cutoff
E11	Patterns	2 threads	No solver time cutoff	(4,4)	No speedup cutoff
E12	Patterns	No Multithreading	No solver time cutoff	(4,4)	Minimum speedup=1
E13	Patterns	No Multithreading	1 second solver time cutoff	(4,4)	No speedup cutoff
E14	Patterns	No Multithreading	No solver time cutoff	(6,6)	No speedup cutoff
E15	Patterns	No Multithreading	0.5 second solver time cutoff	(4,4)	No speedup cutoff
E16	Patterns	No Multithreading	No solver time cutoff	(2,2)	No speedup cutoff
E17	Patterns	No Multithreading	No solver time cutoff	(4,4)	No speedup cutoff
E18	Patterns	2 threads	No solver time cutoff	(4,4)	No speedup cutoff
E19	Patterns	No Multithreading	No solver time cutoff	(4,4)	Minimum speedup=1
E20	Patterns	No Multithreading	0.5 second solver time cutoff	(4,4)	No speedup cutoff
E21	Patterns	No Multithreading	1 second solver time cutoff	(4,4)	No speedup cutoff
E22	Patterns	No Multithreading	No solver time cutoff	(2,2)	No speedup cutoff
E23	Patterns	No Multithreading	No solver time cutoff	(2,2)	No speedup cutoff
E24	Patterns	No Multithreading	0.3 second solver time cutoff	(4,4)	No speedup cutoff
E25	Patterns	No Multithreading	0.3 second solver time cutoff	(4,4)	No speedup cutoff
E26	Patterns	No Multithreading	0.5 second solver time cutoff	(4,4)	No speedup cutoff
E27	Patterns	No Multithreading	0.3 second solver time cutoff	(4,4)	No speedup cutoff
E28	Nodes	No Multithreading	No solver time cutoff	(4,4)	No speedup cutoff
E29	Nodes	No Multithreading	No solver time cutoff	(4,4)	No speedup cutoff
E30	Nodes	No Multithreading	No solver time cutoff	(4,4)	No speedup cutoff

Table 11: A table to associate experiment labels to experiment details, sorted by latency gain of the experiment.

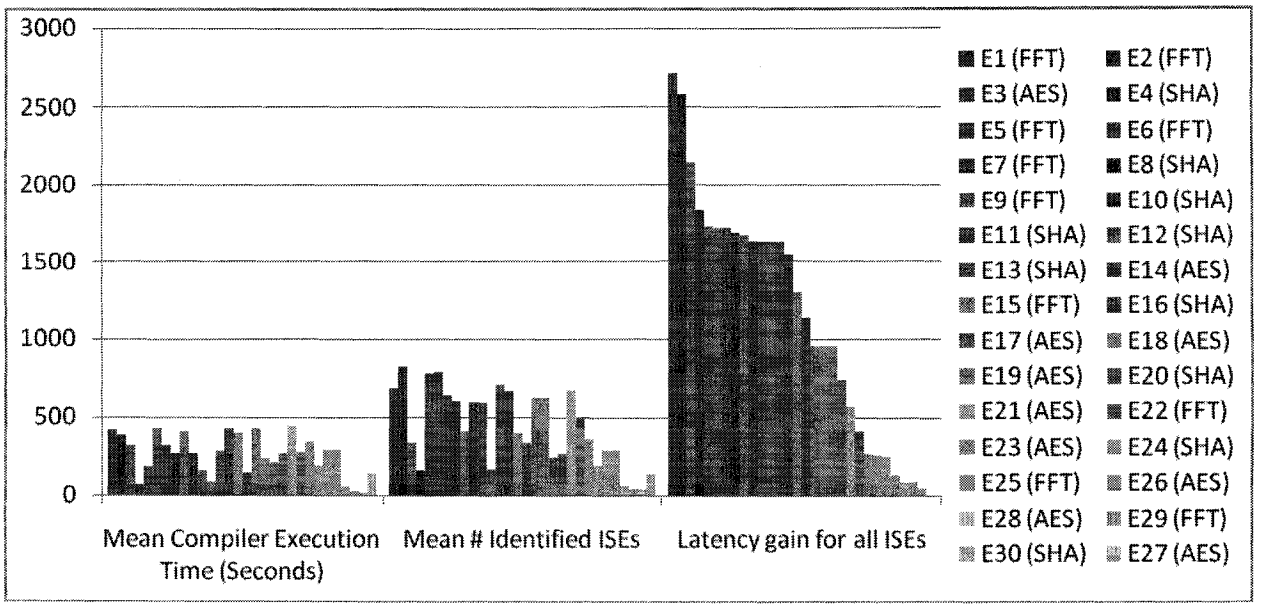

Figure 24: Experiments sorted from highest to lowest according to the sum of the latency saved by the identified ISEs.

In Figure 25 we see the experiments from Table 11 sorted by mean number of ISEs identified. Observe that FFT and AES dominate the top of the list for mean number of ISEs identified. This indicates that the large basic blocks of FFT and AES are simply more suited to ISE identification than SHA. However, when we sort the data according to fastest compile time in Figure 26, we see that AES and SHA were more often processed quickly than FFT. It is logical that as the compiler found fewer ISEs in SHA it would compile faster. There are 10 experiments with each benchmark, and so the distribution of the benchmarks can tell us information about the relative benefit of ISE identification for these benchmarks.

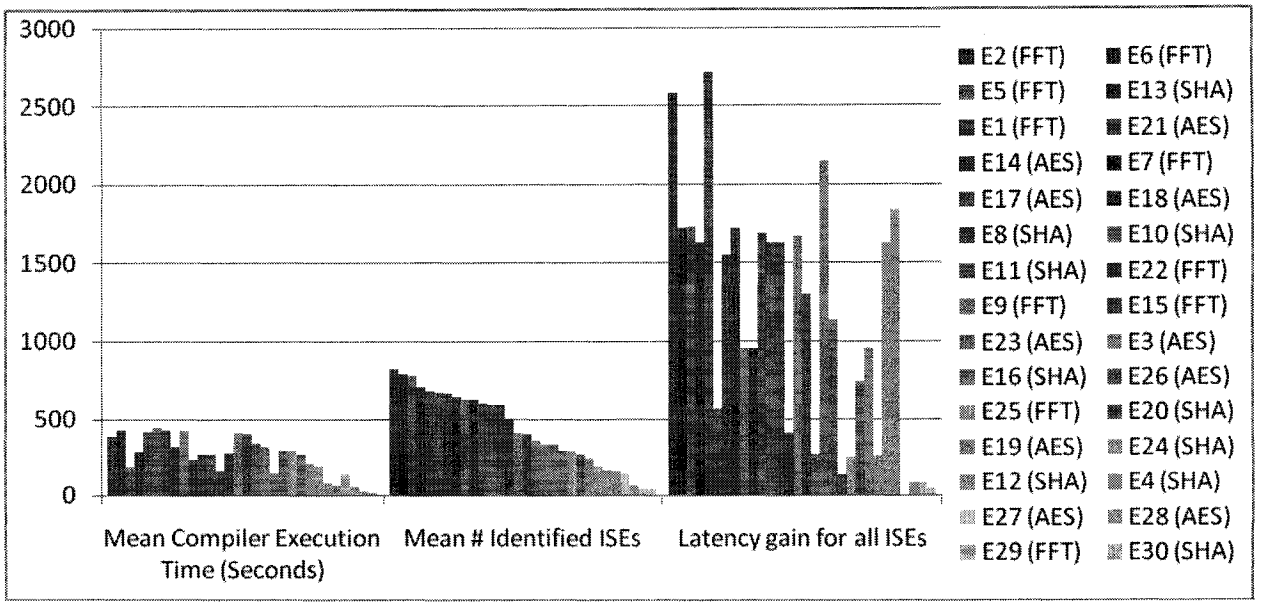

Figure 25: Experiments sorted from highest to lowest according to the mean number of identified ISEs.

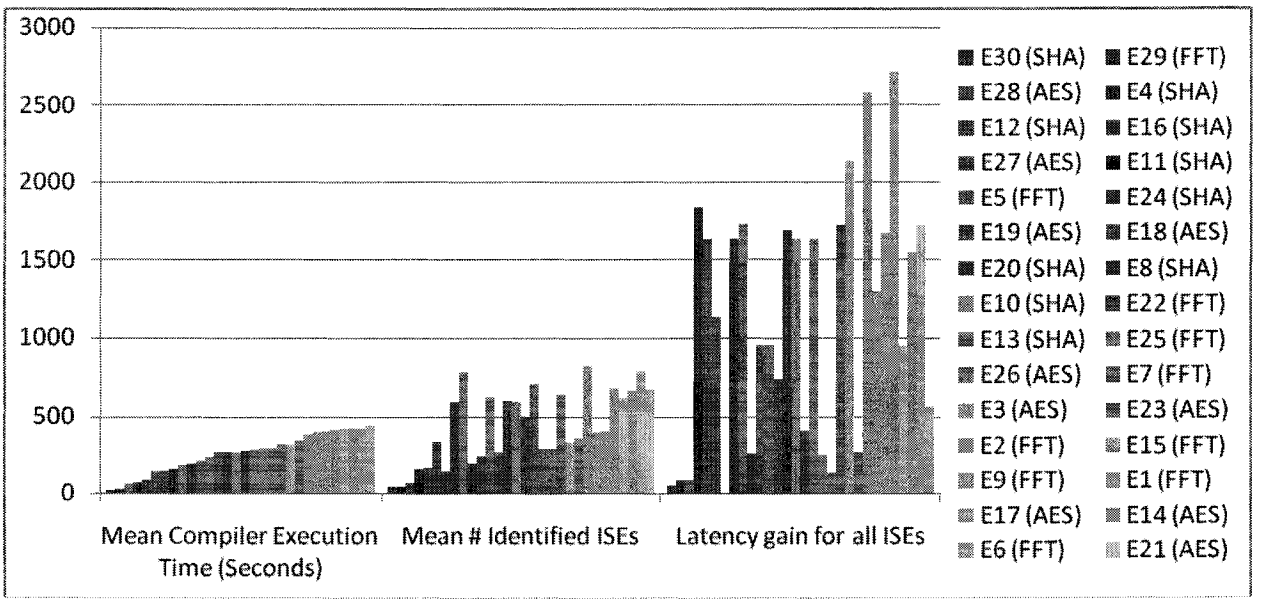

Figure 26: Experiments sorted from lowest to highest according to mean compiler execution time of the experiment.

We can see a trend in Figure 26 above, where the solver finds more ISEs as it takes more time to compile a benchmark. However, the trend does not carry over into the latency gained metric. This

result is an indication that saved latency is much more correlated to the number of identified ISEs than the time required to compile the benchmark.

5 Future Work

The future work to be done in this field is extensive. In this chapter we will talk about possible improvements in our ISE identification model and toolchain, as well as directions which we may pursue in our hardware/software codesign research.

5.1 ISE Identification

We should use nauty isomorphism testing to merge equivalent identified ISEs into a single instruction, as is done by so many other groups studying ISE identification [117]. Another area which we would like to improve is to generate ISE datapaths for each processor in a multiprocessor system, and to pipeline the added instructions to maximize the clock frequency of the design. We should consider pipelining as part of the ISE identification process, with reference to Chapter 7.5.3 of [39].

More work is needed to integrate target processor details into the ISE identification algorithm. It is interesting to note that in our approach all inputs to the datapath must get their values from the register bank or from the immediate field of the instruction. The number of outputs of the datapath should be less than or equal to the number unavailable registers minus the number of inputs to the datapath. This rule is somewhat conservative, but it does ensure that the custom instruction is always schedulable. Another area that needs attention in our model is the relationship between ISE selection and register pressure. We must study the techniques described in [65], [24], and [43], learning from their approach and perhaps contributing insight in the direction of memory usage and power consumption. It seems clear that we must get higher throughput on the custom instructions, or else the ISE identification described here will not be of much use inside the LEON3 processor.

It stands to reason that developers of ASIP multiprocessor systems have resorted to such a full-custom architecture because of a real-time constraint on the performance of the system. Perhaps we can use this restriction to our advantage. We propose that real-time requirements be specified graphically in Eclipse GUI, and be stored in a data structure in the compiler backend. At compile-time, the requirements on code segments can be propagated down to the basic block level using a worst-case scheduling algorithm. During compilation, the compiler can enforce the requirement in the ILP model, and drive the optimization of the architecture towards the goal of meeting all of the real-time requirements. In other words we will select a minimum set of ISEs which passes the real-time requirement for all basic blocks, plus a little bit of extra speedup to guard against faults in real-time systems. This instruction selection problem can be formulated as an ILP problem. This optimization should occur at the instruction selection stage after the ISEs have been enumerated.

An adaptive execution time cutoff was considered but not implemented in this work. The idea is that the compiler would be able to control the amount of searching in each basic block to try and spread the search time evenly throughout the basic blocks. This time distribution should account for the fact that large basic blocks need more time than small basic blocks in order to cover a large number of the available ISEs. Perhaps this approach would perform better than the model of Proposal I did.

5.2 SHIRA Toolchain

We wish to use a tool such as lp_solve to replace the LINDO 11 package as the mathematics engine. The generated instructions should be based on TMD specifications, so that entire instruction sets can be generated at the same time as the code for the target. Hardware generation capabilities are key to the application of instruction set extension identification, and work is ongoing to reach this goal.

Some work is needed to facilitate the use of closed-source third party code in our ASIP designs.

Another important objective to improve the usefulness of the SHIRA framework is the ability to mount an operating system onto the generated custom processor. The key thing to remember about these library and OS issues is that the base processor may be compatible with the closed source binaries because it includes the entire base instruction set, and so compiled closed-source libraries can be used by simply ignoring the ISEs in the processor.

Configuration of buffers and caches is another area where our tools should be improved. Message buffer size configuration is discussed in [107] and we do not currently have the option to set the buffer size. Accurate simulation and generation of buffers is important when power consumption is of concern to the system designer. For example, if a designer is implementing an ISE to save power in an embedded system, but the ISE results in more cache misses, then the ISE may actually do more harm than good. The cache analysis should be used during instruction selection.

It will be very interesting to try and implement real-time requirement handling in our toolchain. This alternate direction in thinking moves in a direction that has been studied extensively in the context of embedded systems, but not to our knowledge for ISE identification. The compiler should fail if it encounters a basic block for which the real-time requirement cannot be met. It is useful to have the compiler fail as early as possible. This is because many programmers use the failure messages in the compiler to figure out where the problems in the system lie. Therefore, time and by extension money will be saved in the development process if users of the compiler are directed to the error as fast as possible. In this case the error would be the real-time requirement violation.

The current version of the SHIRA user interface is being ported to the Eclipse IDE, and the side panel will be re-implemented to be more compact. Also, in future versions of SING the low-level hardware simulation capabilities will be improved. Specifically, TCL scripts will be used to call a

simulator such as ModelSim. A suite of test cases must be designed to ensure that the exported VHDL can be tested quickly by the user. Finally, SING will be fully integrated into SHIRA in order to explore the trade-off between adding one more processor and adding another custom instruction. We wish to formalize the balance between the number of processors and processor specialization in MPSoC designs.

5.3 Instruction Selection

The instruction selection for ISE identification has not yet been implemented into the COINS compiler, but the necessary steps are explained below. Credit goes to the COINS compiler group and Ikuo Nakata for aiding in the understanding of the TMD and the backend, and proposing the solution to our custom instruction specification problem.

The TMD for the x86 backend of COINS contains highly complex and parallel LIR patterns because of the complexity of the architecture. Ikuo Nakata of the COINS research and development team has suggested that we can specify ISEs in the TMD of a target such as SPARC by rewriting instructions in the LIR in a "PARALLEL" construct. Next we will write an instruction definition and add it to the COINS compiler target machine description for the target processor. This approach will allow us to specify multiple-input multiple-output custom instructions in the COINS compiler. We will ensure that selected ISEs will always be used wherever they can be used. Even at compile time we can add the desired definitions and rules to the TMD file of the target. However, the TMD must be modified before it is used to generate the code generator.

Our goal for the future is to generate hardware descriptions based upon TMD, so that not only do we generate customized processors, but we can generate entire instruction sets from the TMD description.

6 Conclusions

The COINS compiler was extended by adding a pass which performs instruction set extension identification. The pass was tested using a small program to verify the functionality of the tool, and then was tested with the SHA, FFT and AES benchmarks from the MiBench benchmark suite. The model for ISE identification contained only real and binary variables, and only linear constraints.

We found that there is only a small difference in the execution time of our model when the input and output port constraints are loosened slightly. It appears that there is only a small number of ISEs that have high I/O in the benchmarks that we tested, and therefore increasing the constraints on the I/O of ISEs eventually did not improve the solution.

Next, several approaches that can speed up the identification of instruction set extensions or improve the solution quality were examined by experimentation. The results of these experiments showed that enumerating all ISEs in a basic block can result in a better solution than state of the art algorithms, but the model runs relatively slowly. We also found that limiting the execution time of the model solver failed to improve the latency gain in the candidate ISE design space. This result is probably a product of the differences in basic block size. Large basic blocks run out of time when searching for valuable ISEs, while small basic blocks do not run out of time and enumerate less important ISEs.

We discovered that dropping ISEs with high latency and low parallelism (low or 0 saved latency) from the solution space sped up the compilation time as much as 3.2 times while the potential in the solution space for saving clock cycles with ISEs remained nearly the same.

This research has presented a new model for ISE identification which can be massively multithreaded on a multicore computer. This approach to parallize the ISE identification algorithm is not specific to our algorithm and may contribute to the practical application of ISE identification in other tools as well as our own. We found that parallelizing the processing of basic blocks is thread safe except for the returning results from the threads, which can be controlled via a semaphore.

When multithreading ISE identification was combined with a cutoff in the saved latency of ISEs and the I/O constraint was loosened, we observed an increase in the number of high parallelism ISEs, a decrease in compiler execution time, and a decrease in the number of identified ISEs.

By observing the results of many compiler configurations, we found that there is a correlation between the number of identified ISEs and the saved latency. We also found that there is a correlation between the compiler execution time and the number of identified instructions.

On a final note, there are several possible methods which may reduce the compiler execution time from hundreds of seconds to tens of seconds which we should pursue in the near future, but the focus on reducing the compile time should not blind us from the ultimate reason for developing custom processors: The embedded system designer wants to save time and money when developing a product. Users of an embedded system compilation tool are used to waiting for several minutes for the compiler to finish processing, and perhaps the embedded system designer wants to wait a little bit longer to find the best solution to the ISE identification problem, instead of rushing quickly to the tool with the fastest compilation time.

7 References

[1] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291-308, 1970.

[2] M. F. Parkinson, P. M. Taylor and S. Parameswaran, "C to VHDL converter in a codesign environment," *VHDL International Users Forum. Spring Conference, 1994. Proceedings of,* pp. 100-109, 1994.

[3] D. R. Hanson and C. W. Fraser, *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley Professional, 1995.

[4] M. Gasteier, M. Glesner, "Bus-based communication synthesis on system-level," *System Synthesis, 1996. Proceedings. , 9th International Symposium on,* pp. 65-70, 1996.

[5] Stefan Pees A, "DSP Compiler and Processor Evaluation - DSPstone" [online], Aachen, Germany: Aachen University, Jan. 1997 [cited Jan. 11, 2009], available from World Wide Web: <http://www.iss.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html>.

[6] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, A. Schrijver, *Combinatorial Optimization*. John Wiley & Sons, 1998.

[7] D. Grune, H. Bal, C. Jacobs and K. Langendoen, *Modern Compiler Design.*, 1st ed.Wiley, 2000, pp. 1.

[8] R. Leupers and P. Marwedel, *Retargetable Compiler Technology for Embedded Systems - Tools and Applications.*, 1st ed.Springer, 2001.

[9] A. Bakshi, V. K. Prasanna and A. Ledeczi, "MILAN: A model based integrated simulation framework for design of embedded systems," in *LCTES '01: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems,* 2001, pp. 82-93.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[11] J-Horng Jeng, F. Lai, E. Naroska and U. Schwiegelshohn, "Multimedia ASIP SoC codesign based on multicriteria optimization," *Consumer Electronics, 2001. ICCE. International Conference on,* pp. 342-343, 2001.

[12] P. Arató, T. Visegrády and I. Jankovits, *High Level Synthesis of Pipelined Datapaths*, 1st ed. Wiley, 2001, pp. 270.

[13] S. Mohanty, V. K. Prasanna, S. Neema and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation," *SIGPLAN Not.,* vol. 37, pp. 18-27, 2002.

[14] J. Martin, *Introduction to Languages and the Theory of Computation.*, 3rd ed.McGraw-Hill Science/Engineering/Math, 2002, pp. 500.

[15] OpenCores Organization, "Wishbone system-on-chip (soc) interconnection architecture for portable ip cores" [online], Sept. 2002 [cited Oct. 13, 2008], available from World Wide Web: <http://www.opencores.com/projects.cgi/ web/wishbone/wbspec b3.pdf>.

[16] M. D. Smith and G. Holloway, "An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization" [online]: Harvard University: Division of Engineering and Applied Sciences, Jul. 16, 2002 [cited Oct. 13, 2008], available from World Wide Web: <http://www.eecs.harvard.edu/hube/software/nci/overview.pdf>.

[17] F. Cottet, J. Delacroix, C. Kaiser and Z. Mammeri, *Scheduling in Real-Time Systems.* Wiley, 2002, pp. 1.

[18] R. Leupers, "Compiler Design Issues for Embedded Processors," *IEEE Design & Test of Computers*, vol. 19, no. 4, Jul. 2002, pp. 51-58.
[19] K. Atasu, L. Pozzi and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," *Design Automation Conference, 2003. Proceedings,* pp. 256-261, 2003.
[20] N. Clark, H. Zhong and S. Mahlke, "Processor acceleration through automated instruction set customization," *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on,* pp. 129-140, 2003.
[21] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems,* 2003, pp. 137-147.
[22] D. Novillo, "Tree SSA A New Optimization Infrastructure for GCC," in *Proc. GCC Developers Summit*, Ottawa, Ont., May 25–27, 2003, pp. 181–193.
[23] A. Grasset, F. Rousseau and A. A. Jerraya, "Network interface generation for MPSOC: from communication service requirements to RTL implementation," *Rapid System Prototyping, 2004. Proceedings. 15th IEEE International Workshop on,* pp. 66-69, 2004.
[24] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne and Nikil Dutt, "Introduction of local memory elements in instruction set extensions," *Design Automation Conference, 2004. Proceedings. 41st,* pp. 729-734, 2004.
[25] G. D. Micheli, *Synthesis and Optimization of Digital Circuits.*, 1st ed. McGraw-Hill, 1994, pp. 89.
[26] V. N. Makarov, "Fighting register pressure in GCC," in *Proc. GCC Developers Summit*, Ottawa, Ont., June 2nd-4th, 2004, pp. 85–103.
[27] J. Cong, Y. Fan, G. Han and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays,* 2004, pp. 183-189.
[28] P. G. Sassone and D. S. Wills, "On the extraction and analysis of prevalent dataflow patterns," *Workload Characterization, 2004. WWC-7. 2004 IEEE International Workshop on,* pp. 11-18, 2004.
[29] W. Worstell, S. Adler, P. Domigan, O. Johnson, H. Kudrolli, D. Lazuka, P. Monteverde, J. Nevin, R. Rohatgi, L. Romanov and S. Starsja, "Design and performance of a prototype whole-body PET/CT scanner with fiber optic readout," *Nuclear Science Symposium Conference Record, 2004 IEEE,* vol. 5, pp. 3280-3284 Vol. 5, 2004.
[30] T. Kogel and H. Meyr, "Heterogeneous MP-SoC - the solution to energy-efficient signal processing," *Design Automation Conference, 2004. Proceedings. 41st,* pp. 686-691, 2004.
[31] M. Jiang, B. Yang, X. Wang and T. Zhang, "SW/HW co-design of a Java-based ASIP for pervasive computing in mobile applications," *Electronic Design, Test and Applications, 2004. DELTA 2004. Second IEEE International Workshop on,* pp. 369-371, 2004.
[32] Wieferink et al. System level processor/communication co-exploration methodology for multiprocessor system-on-chip platforms. In IEEE Proceeding - Computers and Digital Techniques (January 2005), vol. 152, pp. 3-11.
[33] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms.*, 3rd ed. Springer, 2005, pp. 597.
[34] C. Ozturan, G. Dundar and K. Atasu, "An integer linear programming approach for identifying instruction-set extensions," *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on,* pp. 172-177, 2005.
[35] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi and P. Ienne, "ISEGEN: generation of high-quality instruction set extensions by iterative improvement," *Design, Automation and*

Test in Europe, 2005. Proceedings, pp. 1246-1251 Vol. 2, 2005.

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

[37] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing).*, 1st ed. Wiley-Interscience, 2005, pp. 288.

[38] R. Leupers, M. Hohenauer, J. Ceng, H. Scharwaechter, H. Meyr, G. Ascheid and G. Braun, "Retargetable compilers and architecture exploration for embedded processors," *Computers and Digital Techniques, IEE Proceedings -,* vol. 152, pp. 209-223, 2005.

[39] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon).* 1st ed. Morgan Kaufmann, 2006, pp. 6.

[40] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition).* 2nd ed. Addison Wesley, 2006.

[41] CMPware Inc., "Cmpware product brief" [online], May 2008 [cited Jan. 18, 2009], available from World Wide Web: <http://www.cmpware.com/Docs/CmpwareProductBrief.pdf>.

[42] Chafic Jaber, Fabio Forciniti, Thierry Fischer, and Renaud Pacalet . "Overview on System-on-Chip BUSSES, NoC, Crossbar" [online], Nov. 2006 [cited Jan. 18, 2009], available from World Wide Web: <http://soc.eurecom.fr/SoC/l3/presentations/jaber-fischer-forciniti.pdf>.

[43] P. Biswas, N. Dutt, P. Ienne and L. Pozzi, "Automatic Identification of Application-Specific Functional Units with Architecturally Visible Storage," *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings,* vol. 1, pp. 1-6, 2006.

[44] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels and S. Vassiliadis, "Automatic selection of application-specific instruction-set extensions," *Hardware/software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference,* pp. 160-165, 2006.

[45] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon).*, 1st ed. Morgan Kaufmann, 2006, pp. 145-183.

[46] P. Bonzini and L. Pozzi, "Code transformation strategies for extensible embedded processors," in *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems,* 2006, pp. 242-252.

[47] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon).*, 1st ed. Morgan Kaufmann, 2006, pp. 83-91.

[48] O. Muller, A. Baghdadi and M. Jezequel, "ASIP-based multiprocessor SoC design for simple and double binary turbo decoding," in *DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe,* 2006, pp. 1330-1335.

[49] H. Nikolov, T. Stefanov and E. Deprettere, "Multi-processor system design with ESPAM," in *CODES+ISSS '06: Proceedings of the 4th International Conference on Hardware/software Codesign and System Synthesis,* 2006, pp. 211-216.

[50] 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, pp. 1209-1229, 2006.

[51] J. Gaisler, "LEON3 GR-XC3S-1500 Template Design" [online], Goteborg, Sweden: Aeroflex Gaisler AB, Feb. 2006 [cited Jan. 12, 2009], available from World Wide Web: <http://gaisler.com/doc/leon3-gr-xc3s-1500.pdf>.

[52] V. S. Pai, "The RSIM Simulation Environment" [online], Houston, Texas: Rice University, Feb. 2006 [cited Jan. 13, 2009], available from World Wide Web: <http://www.owlnet.rice.edu/~elec526/handouts/rsim.pdf>.

[53] M. Saldaña, *A Parallel Programming Model for a Multi-FPGA Multiprocessor Machine*, M.A.Sc. dissertation, Graduate Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada, May 2006.

[54] J. Wawrzynek, M. Oskin, C. Kozyrakis, D. Chiou, D. A. Patterson, S. Lu, J. C. Hoe, K. Asanovic, "RAMP: A Research Accelerator for Multiple Processors," Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, Report No. UCB/EECS-2006-158, Nov. 2006.

[55] G. Gibeling, A. Schultz, K. Asanovic, "The RAMP Architecture & Description Language," presented at *WARFP 2006,* Austin, TX, March, 2006.

[56] R. Leupers, K. Karuri, S. Kraemer and M. Pandey, "A design flow for configurable embedded processors based on optimized instruction set extension synthesis," *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings,* vol. 1, pp. 6 pp., 2006.

[57] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon).* 1st ed. Morgan Kaufmann, 2006, pp. 59-76.

[58] Free Software Foundation Inc., "GNU binutils" [online], Aug. 2007 [cited Oct. 26, 2008], available from World Wide Web: <http://www.gnu.org/software/binutils/>.

[59] T. Lteif, Y. Muheto, and I. Ridhawi, M. Branchaud. Modular design of multiprocessor systems. Tech. rep., School of Information Technology and Engineering, University of Ottawa, 2007.

[60] Altera Corporation, "Quartus ii version 7.2 handbook - volume 4: Sopc builder" [online], Oct. 2007 [cited Jan. 18, 2009], available from World Wide Web: <http://www.altera.com/literature/hb/qts/qts qii5v4.pdf>.

[61] P. J. Ashenden, *Digital Design (VHDL): An Embedded Systems Approach using VHDL.* Morgan Kaufmann, 2007, pp. 352-388.

[62] Xilinx Inc., "Embedded system tools reference manual - embedded development kit, edk 9.2i." [online], Sept. 2007 [cited Jan. 18, 2009], available from World Wide Web: <http://www.xilinx.com/support/documentation/sw_manuals/edk92i_est_rm.pdf>.

[63] Chen Le, Feng Hui-juan, Yu Yong-li, Zhang Liu and Nie Cheng-long, "Workflow Model of Equipment Maintenance Support Based on Petri-Net," *IEEE International Conference on Control and Automation, 2007. ICCA 2007.,* 2007, pp. 2819-2822.

[64] Yawen Niu, Jinian Bian, Haili Wang, Kun Tong, "An Efficient Cooperative Design Framework for SOC On-Chip Communication Architecture System-Level Design". in *Proc. of the 10th International Conference on CSCW in Design, CSCWD 2006,* Nanjing, China, May 3-5, pp. 118-127, 2006.

[65] K. Atasu, R. G. Dimond, O. Mencer, W. Luk, C. Ozturan and G. Dundar, "Optimizing Instruction-set Extensible Processors under Data Bandwidth Constraints," *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07,* pp. 1-6, 2007.

[66] P. Bonzini and L. Pozzi, "Polynomial-Time Subgraph Enumeration for Automated Instruction Set Extension," *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07,* pp. 1-6, 2007.

[67] R. V. Bennett, A. C. Murray, B. Franke and N. Topham, "Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems," in *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems,* 2007, pp. 83-92.

[68] J. W. Chinneck, "Practical Optimization: A Gentle Introduction" [online], Ottawa, On.: Carleton University, Nov. 2007 [cited Jan. 11, 2009], available from World Wide Web: <http://www.sce.carleton.ca/faculty/chinneck/po.html>.

[69] D. Goodwin, C. Rowen and G. Martin, "Configurable Multi-Processor Platforms for Next Generation Embedded Systems," *Design Automation Conference, 2007. ASP-DAC '07.*

Asia and South Pacific, pp. 744-746, 2007.

[70] S. Xu and H. Pollitt-Smith, "Optimization of HW/SW Co-Design: Relevance to Configurable Processor and FPGA Technology," *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on,* pp. 1691-1696, 2007.

[71] Fei Sun, S. Ravi, A. Raghunathan and N. K. Jha, "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, pp. 2035-2045, 2007.

[72] T. Watanabe, T. Fujise, K. Mori, K. Iwasawa and I. Nakata, "Design Assists for Embedded Systems in the COINS Compiler Infrastructure," *Innovative Architecture for Future Generation High-Performance Processors and Systems, 2007. Iwia 2007. International Workshop on,* 2007, pp. 60-69.

[73] John L. Hennessy, David A. Patterson, *Computer Organization and Design: The Hardware/Software Interface. Third Edition, Revised*, Morgan Kaufmann, 2007, pp. 248-249.

[74] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra and E. F. Deprettere, "A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs," in *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis,* 2007, pp. 9-14.

[75] Eric Zeman, "Symbian and ARM go multicore, future smartphones to be faster than superman - mobile blog - InformationWeek" [online], Oct. 2007 [cited Nov. 24, 2008], available from World Wide Web: <http://www.informationweek.com/blog/main/archives/2007/10/symbian_and_arm.htm>.

[76] T. Jing, D. Han and Qi. Wang, "Implementation of a ASIP for SELP Vocoder At Low Bit Rate of 600bps," *Industrial Electronics and Applications, 2007. ICIEA 2007. 2nd IEEE Conference on,* pp. 305-308, 2007.

[77] J. W. Chinneck, "Decision Tree for Methods Taught in SYSC-5004" [online], Ottawa, Ont.: Carleton University [cited Oct. 15, 2008], available from World Wide Web: <http://www.sce.carleton.ca/courses/sysc-5004/DecisionTree.pdf>.

[78] J. W. Chinneck, "Classification of Optimization Methods" [online], Ottawa, Ont.: Carleton University [cited Oct. 15, 2008], available from World Wide Web: <http://www.sce.carleton.ca/courses/sysc-5004/ClassificationOfMethods.pdf>.

[79] G. Gibeling, "RDLC2: The RAMP Model, Compiler & Description Language," M.A.Sc. thesis, Dept. of Electrical and Computer Sciences, University of California at Berkeley, CA, May 2008.

[80] The Eclipse Foundation, "Eclipse.org home" [online], [cited Oct. 6, 2008], available from World Wide Web: <http://www.eclipse.org/>.

[81] COINS-project, "12. performance of COINS compilers" [online], [cited Oct. 6, 2008], available from World Wide Web: <http://www.coins-project.org/international/COINSdoc.en/perf/perf-frame.html>.

[82] Gaisler Research AB, "Leon3 processor - gaisler research" [online], [cited Oct 6, 2008], available from World Wide Web: <http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53>.

[83] Free Software Foundation Inc., "GCC, the GNU compiler collection - GNU project - free software foundation (FSF)" [online], [cited Oct. 6, 2008], available from World Wide Web: <http://gcc.gnu.org/>.

[84] Red Hat Inc., "Cygwin information and installation" [online], [cited Oct. 13, 2008],

available from World Wide Web: <http://www.cygwin.com/>.
[85] Anonymous. "MinGW | minimalist GNU for windows" [online], [cited Oct. 13, 2008], available from World Wide Web: <http://www.mingw.org/>.
[86] Anonymous. "Message passing interface (MPI) forum home page" [online], [cited Oct. 13, 2008], available from World Wide Web: <http://www.mpi-forum.org/>.
[87] Anonymous, "OpenMP.org" [online], [cited Oct. 13, 2008], available from World Wide Web: <http://openmp.org/wp/>.
[88] OpenCores Organization, "OPENCORES.ORG - miniMIPS: Overview" [online], [cited Oct. 6, 2008], available from World Wide Web: <http://www.opencores.org/projects.cgi/web/minimips/overview>.
[89] D. Shapiro, V. Thareja, S. Ratti, S. Vijayakumar, M. Yang, "Fuzzy Logic Co-processor for Leon 3,"Innovate Canada 2007-2008 Canadian FPGA Programming Contest, Ottawa, Ont., May 2008.
[90] LINDO Systems Inc., "LINDO systems inc :: Newsletters :: News features of LINGO 10" [online], [cited Oct. 6, 2008], available from World Wide Web: <http://www.mpassociates.gr/software/distrib/science/lindo/index_lingo.html>.
[91] I. L. Taylor, "Gcc in cxx branch created" [online], June 2008 [cited Oct. 18, 2008], available from World Wide Web: <http://article.gmane.org/gmane.comp.gcc.patches/165063>.
[92] I. Nakata, "[Coins-compiler-users] CoVis: COINS visualizer tool available?" [online], May 2008 [cited Oct. 7, 2008], available from World Wide Web: <http://lists.sourceforge.jp/mailman/archives/coins-compiler-users/2008-May/000023.html>.
[93] Gary V. Vaughan. "Autoconf, automake, and libtool: Canadian cross concepts" [online], Feb. 2006 [cited Oct. 13, 2008], available from World Wide Web: <http://sources.redhat.com/autobook/autobook/autobook_264.html>.
[94] T. Tromey, "IncrementalCompiler - GCC wiki" [online], Apr. 2004 [cited Sept. 19, 2008], available from World Wide Web: <http://gcc.gnu.org/wiki/IncrementalCompiler>.
[95] Gaisler Research AB, "Home - gaisler research" [online], [cited Apr. 22, 2008], available from World Wide Web: <http://www.gaisler.com/cms/index.php>.
[96] ARM Ltd., "AMBA overview" [online], [cited Sept. 18, 2008], available from World Wide Web: <http://www.arm.com/products/solutions/AMBAHomePage.html>.
[97] Anonymous, "COINS: COmpiler INfra-Structure" [online], Apr. 2007 [cited Oct. 29, 2008], available from World Wide Web: <http://www.coins-project.org/international/overview-en/overview.pdf>.
[98] ILOG Inc., "ILOG CPLEX: High-performance software for mathematical programming and optimization" [online], [cited Oct. 6, 2008], available from World Wide Web: <http://www.ilog.com/products/cplex/>.
[99] LINDO Systems, "LINDO systems - optimization software: Integer programming, linear programming, nonlinear programming, global optimization" [online], [cited Oct. 6, 2008], available from World Wide Web: <http://www.lindo.com/>.
[100] I. Nakata, D. Shapiro, T. Watanabe. "[Coins-compiler-users] CoVis: COINS visualizer tool available?" [online], May 2008 [cited Oct. 6, 2008], available from World Wide Web: <http://lists.sourceforge.jp/mailman/archives/coins-compiler-users/2008-May/000027.html>.
[101] B. Chapman, G. Jost and R. V. D. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming.* Cambridge, Mass., MIT Press, 2008.
[102] Anonymous, "The MDES User Manual" [online], Mar. 2000 [cited Oct. 26, 2008], available from World Wide Web: <http://www.trimaran.org/docs/mdes_manual.pdf>.
[103] W. Klingauf, M. Burton, R. Günzel and U. Golze, "Why we need standards for transaction-

level modeling (SOCcentral 22293)" [online], Apr. 2007 [cited Oct. 19, 2008], available from World Wide Web: <http://www.soccentral.com/results.asp?EntryID=22293>.
[104] Anonymous, "lp_solve reference guide" [online], Aug. 2008, [cited Oct. 20, 2008], available from World Wide Web: <http://lpsolve.sourceforge.net/5.5/>.
[105] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu and E. Deprettere, "Daedalus: Toward composable multimedia MP-SoC design," *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE,* pp. 574-579, 2008.
[106] Tensilica Inc., "Design process" [online], [cited Oct. 20, 2008], available from World Wide Web: <http://www.tensilica.com/methodology/design_process.htm>.
[107] H. Nikolov, T. Stefanov and E. Deprettere, "Systematic and Automated Multiprocessor System Design, Programming, and Implementation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 27, pp. 542-555, 2008.
[108] CoWare, Inc., "CoWare® Platform Architect" [online], San Jose, Calif.: CoWare, Inc., [cited Oct. 26, 2008], available from World Wide Web: <http://www.coware.com/PDF/products/PlatformArchitect.pdf>.
[109] P. P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version.* Wiley-Interscience, 2008, pp. 11.
[110] Xilinx Inc., "Xilinx : Mimosys" [online], [cited Nov. 11, 2008], available from World Wide Web: <http://www.xilinx.com/products/design_tools/logic_design/advanced/esl/mimosys.htm#lang>.
[111] MultiCellPhone.com, "Ericsson U500 mobile platform with multi-processor at MultiCellPhone.com" [online], Feb. 2008 [cited Nov. 24, 2008], available from World Wide Web: <http://www.multicellphone.com/ericsson-u500-mobile-platform-with-multi-processor/>.
[112] Tensilica Inc., "Mobile handset processors and DSPs" [online], [cited Nov. 24, 2008], available from World Wide Web: <http://www.tensilica.com/markets/mobile_handset/index.htm>.
[113] NVIDIA Corporation, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," Santa Clara, CA, Version 2.0, Jun. 2008.
[114] Intel Corporation, "Intel® core™ i7 processor overview" [online], [cited Nov. 24, 2008], available from World Wide Web: <http://www.intel.com/products/processor/corei7/index.htm>.
[115] Intel Corporation, "Intel® Core™2 duo processor overview" [online], [cited Nov. 24, 2008], available from World Wide Web: <http://www.intel.com/products/processor/core2duo/>.
[116] Advanced Micro Devices Inc., "AMD phenom™ processors: Multi-core performance for life-like gaming, crystal-clear digital media and serious megatasking" [online], [cited Nov. 24, 2008], available from World Wide Web: <http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_15331,00.html>.
[117] Brendan D. McKay, "The nauty page" [online], [cited Nov. 26, 2008], available from World Wide Web: <http://cs.anu.edu.au/~bdm/nauty/>.
[118] Agility Design Solutions Inc., "DK design suite download :: Product downloads :: Support :: Agility design solutions :: Algorithm to implementation. fast." [online], Sept. 2008 [cited Oct 6, 2008], available from World Wide Web: <http://agilityds.com/support/download_dk.aspx>.
[119] COINS-project, "1. How to use COINS compiler infrastructure" [online], May 2008 [cited Jan. 11, 2009], available from World Wide Web: <http://www.coins-project.org/international/COINSdoc.en/infra/infra-frame.html>.

Appendix A: Select Source Code

A.1: simpler.c

```
int a = 0;
int b = 1;
int c = 2;
int d = 2;
int e, t1, t2, t3;

int main(int argc, char *argv[])
{
  t1 = a+b;
  t2 = c*t1;
  t3 = t1+d;
  e = t2*t3;
}
```

A.2: SING Specification and Simulation Source Code

```
package userRequirements;

import java.util.*;
import userRequirements.Component.ArbiterType;
import userRequirements.Component.BusWidth;
import userRequirements.Component.ClkDivideType;
import userRequirements.MasterPort.ControlBits;

public class Runme {

      public static void main(String[] args) {
        long begin = System.currentTimeMillis();
            NetworkCreator netCreator = new NetworkCreator();
            InterconnectionNetwork interconn = netCreator.getIn();

            int addysize = 32;
            int sel = 0;
            int memSize = 128;
            BusWidth busSize = BusWidth.THIRTY_TWO;

            int masGroups = 5;
            int slvGroups = 5;

            int numMasInGroup = 4;
            int numSlvInGroup = 4;

            Crossbar cb =
interconn.addCrossbar(ArbiterType.ROUND_ROBIN,busSize,ClkDivideType.ONE
,slvGroups,masGroups,addysize,sel);

            Connector[] masters = new Connector[masGroups * numMasInGroup];

            for(int i = 0; i < (masGroups * numMasInGroup); i++)
            {
                  masters[i] = interconn.addProcessor(busSize,
ClkDivideType.ONE,addysize, sel,1);
            }

            Connector[] slaves = new Connector[slvGroups * numSlvInGroup];

            for(int i = 0; i < (slvGroups * numSlvInGroup); i++)
            {
                  slaves[i] = interconn.addMemory(memSize, busSize,
ClkDivideType.ONE,addysize, sel,new int[]{i*128}, new int[]{(i+1)*128 - 1});
            }

            SharedBus[] sbMas = new SharedBus[masGroups];

            for(int i = 0; i < masGroups; i++)
            {
```

```
                sbMas[i] = interconn.addSharedBus(ArbiterType.ROUND_ROBIN, busSize,
ClkDivideType.ONE, 1, numMasInGroup, addysize, sel);
            }

            SharedBus[] sbSlv = new SharedBus[slvGroups];

            for(int i = 0; i < slvGroups; i++)
            {
                sbSlv[i] = interconn.addSharedBus(ArbiterType.ROUND_ROBIN, busSize,
ClkDivideType.ONE, numSlvInGroup, 1, addysize, sel);
            }

            //Connection time
            //Masters to SB masers

            for(int i = 0; i < (masGroups * numMasInGroup); i ++)
            {
                sbMas[(i / numMasInGroup)].connectToMasterPortOn(masters[i]);
            }

            //SB slaves to Crossbar
            for(int i = 0; i < slvGroups; i++)
            {
                sbSlv[i].connectToMasterPortOn(cb);
                //cb.connectToMasterPortOn(sbMas[i]);
            }

            //slaves to SB slaves
            for(int i = 0; i< (slvGroups * numSlvInGroup); i++)
            {
                slaves[i].connectToMasterPortOn(sbSlv[i/numSlvInGroup]);
            }

            //Crossbar to SB masters
            for(int i = 0; i < masGroups; i++)
            {
                //cb.connectToMasterPortOn(sbMas[i]);
                sbMas[i].connectToSlavePortOn(cb);
            }
            if(true)
            {
            //write
            Random generator = new Random();
            int gotNumber = 0;
            int randy = 0;
            int addressWR = 0;
            for (int i = 0; i < (masGroups*numMasInGroup - 1); i++)
            {
                randy = generator.nextInt();
                addressWR = generator.nextInt( slvGroups*numSlvInGroup*memSize );
                //    addressWR = 16;
                System.out.println("****PREPARING TO WRITE NUMBER "+randy+" IN
ADDRESS "+ addressWR);
```

```
                BitSet write = new BitSet();
                write.set(ControlBits.WE_O.getIndex());
                masters[i].sendMasterMessage(addressWR, randy, write);
                //    mas1.release();
                System.out.println("\n****PREPARING TO READ");

                //read back
                BitSet read = new BitSet();
                read.clear(ControlBits.WE_O.getIndex());
                masters[i+1].sendMasterMessage(addressWR, i, read);
                System.out.println("\n\n***\n\n");
            }
        }
        netCreator.generateHardware();
        System.out.println((System.currentTimeMillis()-begin));
        System.exit(0);
}
}
```

A.3: A Small SING Specification

```
package userRequirements;

import java.util.*;

import userRequirements.Component.ArbiterType;
import userRequirements.Component.BusWidth;
import userRequirements.Component.ClkDivideType;
import userRequirements.MasterPort.ControlBits;

public class RunSimple {

      public static void main(String[] args) {

            NetworkCreator netCreator = new NetworkCreator();
            InterconnectionNetwork interconn = netCreator.getIn();

            int addysize = 32;
            int sel = 0;
            int memSize = 128;
            BusWidth busSize = BusWidth.THIRTY_TWO;

            Connector master = interconn.addProcessor(busSize,
ClkDivideType.ONE,addysize, sel,1);
            Connector slave = interconn.addMemory(memSize, busSize,
ClkDivideType.ONE,addysize, sel,new int[]{0}, new int[]{127});

            //Connect Processor to Memory
            master.connectToSlavePortOn(slave);

            //synthesize system
            netCreator.generateHardware();

            //exit
            System.exit(0);
      }
}
```

A.4: A Function to Hold the Settable Assumptions About the Execution Time of Instructions

```
public static int softwareSpeed(String string)
{
      switch(Op.toCode(string))
      {
      case Op.ADD: return 1;
      case Op.ASM: return 1;
      case Op.ASMCONST: return 1;
      case Op.BAND: return 1;
      case Op.BNOT: return 1;
      case Op.BOR: return 1;
      case Op.BXOR: return 1;
      case Op.CONVFI: return 3;
      case Op.CONVFS:return 3;
      case Op.CONVFT:return 3;
      case Op.CONVFU:return 3;
      case Op.CONVFX:return 3;
      case Op.CONVIT:return 3;
      case Op.CONVSF:return 3;
      case Op.CONVSX:return 3;
      case Op.CONVUF:return 3;
      case Op.CONVZX:return 3;
      case Op.DIVS: return 9;
      case Op.DIVU: return 9;
      case Op.FLOATCONST: return 1;
      case Op.INTCONST: return 1;
      case Op.LSHS: return 1;
      case Op.LSHU: return 1;
      case Op.MODS: return 1;
      case Op.MODU: return 1;
      case Op.MUL: return 3;
      case Op.NEG: return 1;
      case Op.RSHS: return 1;
      case Op.RSHU:return 1;
      case Op.SUB:return 1;
      case Op.SUBREG: return 1;
      case Op.TSTEQ: return 1;
      case Op.TSTGES: return 1;
      case Op.TSTGEU:return 1;
      case Op.TSTGTS:return 1;
      case Op.TSTGTU:return 1;
      case Op.TSTLES:return 1;
      case Op.TSTLEU:return 1;
      case Op.TSTLTS:return 1;
      case Op.TSTLTU :return 1;
      case Op.TSTNE: return 1;
      case Op.ZEROS: return 1;
      default:

      }return 10;
}
```

A.5: A Function to Hold the Settable Assumptions About the Hardware Size of Instructions

```
public static int hardwareSize(String string)
{
      switch(Op.toCode(string))
      {
      case Op.ADD: return 20;
      case Op.ASM: return 20;
      case Op.ASMCONST: return 20;
      case Op.BAND: return 10;
      case Op.BNOT: return 10;
      case Op.BOR: return 10;
      case Op.BXOR: return 10;
      case Op.CONVFI: return 20;
      case Op.CONVFS:return 20;
      case Op.CONVFT:return 20;
      case Op.CONVFU:return 20;
      case Op.CONVFX:return 20;
      case Op.CONVIT:return 20;
      case Op.CONVSF:return 20;
      case Op.CONVSX:return 20;
      case Op.CONVUF:return 20;
      case Op.CONVZX:return 20;
      case Op.DIVS: return 100;
      case Op.DIVU: return 100;
      case Op.FLOATCONST: return 10;
      case Op.INTCONST: return 10;
      case Op.LSHS: return 20;
      case Op.LSHU: return 20;
      case Op.MODS: return 20;
      case Op.MODU: return 20;
      case Op.MUL: return 100;
      case Op.NEG: return 20;
      case Op.RSHS: return 20;
      case Op.RSHU:return 20;
      case Op.SUB:return 20;
      case Op.SUBREG: return 20;
      case Op.TSTEQ: return 10;
      case Op.TSTGES: return 10;
      case Op.TSTGEU:return 10;
      case Op.TSTGTS:return 10;
      case Op.TSTGTU:return 10;
      case Op.TSTLES:return 10;
      case Op.TSTLEU:return 10;
      case Op.TSTLTS:return 10;
      case Op.TSTLTU :return 10;
      case Op.TSTNE: return 10;
      case Op.ZEROS: return 10;
      default:
      }
      return 10;
}
```

Appendix B: Selected Compiler Output

The output of the compiler very simple. The name of the function that the compiler is entering is printed as "<Function functionName>". Next the compiler prints "`Solving...`" as it passes a processed basic block into the LINDO 11 solver. The ISE found by the solver is printed as it is returned to the compiler. The format of the ISEs that are printed is as follows: Each node included by the ISE is defined in terms of its inputs and outputs. This allows us to build up a custom instruction datapath. For example, the instruction

```
t2_divexI32=MUL(c_divexI32,t1_divexI32)
t1_divexI32=ADD(a_divexI32,b_divexI32)
t3_divexI32=ADD(t1_divexI32,d_divexI32)
e_divexI32=MUL(t2_divexI32,t3_divexI32)
```

can be rewritten replacing all `t2_divexI32` as follows:

```
t1_divexI32=ADD(a_divexI32,b_divexI32)
t3_divexI32=ADD(t1_divexI32,d_divexI32)
e_divexI32=MUL(MUL(c_divexI32,t1_divexI32),t3_divexI32)
```

Next the `t1_divexI32` can be replaced:

```
t3_divexI32=ADD(ADD(a_divexI32,b_divexI32),d_divexI32)
e_divexI32=MUL(MUL(c_divexI32,ADD(a_divexI32,b_divexI32)),t3_divexI32)
```

And finally, we replace t3_divexI32 to form the custom instruction:

```
e_divexI32=MUL(MUL(c_divexI32,ADD(a_divexI32,b_divexI32)),ADD(ADD(a_divexI32,b_divex
I32), d_divexI32))
```

There is one such line for each output of the custom instruction written in terms of the inputs to the custom instruction.

B.1: Compilation of simpler.c with non-binding constraints

In this section the output from the modified COINS compiler is presented for the compilation of the simpler.c program found in Appendix A.1. The output represents the sequence of identified candidate instruction set extensions. First the compiler outputs the string "Solving..." when it calls the optimization module, and then it outputs a set of strings representing one identified ISE. This process is repeated until the solver does not find a feasible solution, at which time the compiler outputs the duration for which the current function was analyzed. The previous section explains how to interpret the printed ISE identification information.

```
<Function main>

Solving...
t2_divexI32=MUL(c_divexI32,t1_divexI32)
t1_divexI32=ADD(a_divexI32,b_divexI32)
t3_divexI32=ADD(t1_divexI32,d_divexI32)

Solving...
t2_divexI32=MUL(c_divexI32,t1_divexI32)
t3_divexI32=ADD(t1_divexI32,d_divexI32)

Solving...
t2_divexI32=MUL(c_divexI32,t1_divexI32)
t3_divexI32=ADD(t1_divexI32,d_divexI32)
e_divexI32=MUL(t2_divexI32,t3_divexI32)

Solving...
t2_divexI32=MUL(c_divexI32,t1_divexI32)
t1_divexI32=ADD(a_divexI32,b_divexI32)

Solving...
t1_divexI32=ADD(a_divexI32,b_divexI32)
t3_divexI32=ADD(t1_divexI32,d_divexI32)

Solving...
t2_divexI32=MUL(c_divexI32,t1_divexI32)
e_divexI32=MUL(t2_divexI32,t3_divexI32)

Solving...
t2_divexI32=MUL(c_divexI32,t1_divexI32)
t1_divexI32=ADD(a_divexI32,b_divexI32)
t3_divexI32=ADD(t1_divexI32,d_divexI32)
e_divexI32=MUL(t2_divexI32,t3_divexI32)

Solving...
t3_divexI32=ADD(t1_divexI32,d_divexI32)
e_divexI32=MUL(t2_divexI32,t3_divexI32)

Solving...

RUNTIME = 2523 MILLISECONDS
```