Scheduling Algorithms for Instruction Set Extended Symmetrical Homogeneous Multiprocessor Systems-on-Chip

Michael Montcalm

Thesis Submitted to the Faculty of Graduate and Postdoctoral Studies
In partial fulfilment of the requirements for the degree of Masters of Applied Science, Electrical and Computer Engineering [1]

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

© Michael Montcalm, Ottawa, Canada, 2011

[1]The M.A.Sc. program is a joint program with Carleton University, administered by OCIECE

Abstract

Embedded system designers face multiple challenges in fulfilling the runtime requirements of programs. Effective scheduling of programs is required to extract as much parallelism as possible. These scheduling algorithms must also improve speedup after instruction-set extensions have occurred. Scheduling of dynamic code at run time is made more difficult when the static components of the program are scheduled inefficiently. This research aims to optimize a program's static code at compile time. This is achieved with four algorithms designed to schedule code at the task and instruction level. Additionally, the algorithms improve scheduling using instruction set extended code on symmetrical homogeneous multiprocessor systems. Using these algorithms, we achieve speedups up to 3.86X over sequential execution for a 4-issue 2-processor system, and show better performance than recent heuristic techniques for small programs. Finally, the algorithms generate speedup values for a 64-point FFT that are similar to the test runs.

Acknowledgements

I would like to thank the University of Ottawa for allowing me to continue my studies here at the School of Information Technology and Engineering. To my supervisors, Dr. Miodrag Bolić and Dr. Voicu Groza, I thank you for your support and guidance as well as for the courses each of you taught me at both the graduate and undergraduate levels.

To my colleagues in and associated with the Computer Architecture Research Group, I would like to say thanks for helping me with course work, as well as helping in research related to my thesis. A special thanks goes to Daniel Shapiro for working with me on what would become the basis for my thesis.

I would also like to thank my parents, my brother, my extended family and my friends for giving me their support while I did my studies; there are too many of you to list.

Finally, and most importantly, I would like to thank Erin. Thank you for supporting me and pushing me to finish this work. You put up with the long hours of writing, the research and coding, and my techno-babble and frustrations. I probably would not have finished this if you hadn't been there to help. I love you.

Contents

Abstract **ii**

Acknowledgements **iii**

List of Figures **viii**

List of Tables **x**

List of Terms **xi**

1 Introduction **1**

1.1 Background 1

1.2 Motivation 3

1.3 Objectives 4

1.4 Contributions 4

1.5 Conditions and Assumptions 5

1.6 Organization of Content 6

2 Background **8**

2.1 Intermediate Representation 8

2.1.1 High Intermediate Representation 9

2.1.2 Low Intermediate Representation 10

2.2 Instruction Scheduling Methods 10

2.2.1 Local Instruction Scheduling Algorithms 11
2.2.2 List Scheduling 11
2.2.3 Global ISAs 13
2.3 Clustering Methods 15
2.4 Linear Programming 15
2.5 Transitive Closure 17
2.6 Branch and Bound 18
2.7 SHIRA 21
2.8 Compilers 22
2.9 FPGAs 22
2.10 NIOS II 23
2.11 Fourier Transform 24
2.12 Chunking 24
2.13 Instruction Set Extensions 25

3 State of the Art 27

3.1 Scheduling Approaches 27
3.1.1 MAX-MIN Ant System Optimization 27
3.1.2 Statistical Optimization Approach 29
3.1.3 Modified Mixed Integer Linear Program Approach 30
3.1.4 Decomposition-based Constraint Optimization Approach . . 30
3.2 VLIW Compilers and Compilation Methods 32
3.2.1 Tetris 32
3.2.2 Adaptive Compiler Method 32
3.3 Instruction Set Extension Approaches 33
3.3.1 The Instruction-Set Extension Problem: A Survey 33
3.3.2 Algorithms for the Automatic Extension of an Instruction-Set 35
3.4 Summary 36

4 Scheduling Algorithm Development 37

4.1 Design of the MPS Algorithm 38
4.2 Design of the IMP algorithm 45
4.3 Design of the ITS Algorithm 49
4.3.1 Primary Problem 49
4.3.2 Secondary Problem 50
4.3.3 Model Definition 50
4.4 Development of the CSA Algorithm 54
4.5 Summary 56

5 Results 57

5.1 Creation of Test Programs 57
5.2 Comparison with results in Literature 61
5.3 Variation in Model Processing Time 70
5.4 Scheduling of Fast Fourier Transform Benchmark 74
5.5 Summary 77

6 Conclusions and Future Work 78

6.1 Conclusion 78
6.2 Future Work 80
6.2.1 Combination with Dynamic Scheduling 80
6.2.2 Heterogeneous Processors 81
6.2.3 Asymmetrical Network Topologies 82
6.2.4 Balancing the Tradeoff between ISEs speedup and Schedule length 82
6.2.5 Coprocessor Scheduling 85

A Example Calculations 91

B Algorithm Coverage **93**

List of Figures

2.1 Example LP 17

2.2 Sample Dataflow Graph 17

2.3 Sample Dataflow Graph with arcs showing transitive closure 18

2.4 Complete Enumeration Tree 19

2.5 SHIRA Framework 21

4.1 Dependency graph for the example described in [27]. 39

5.1 Different methods for summing an array in three address code . . . 60

5.2 Comparison of the results in Table 5.1. 64

5.3 Comparison of the results in Table 5.2. 66

5.4 Comparison of the results in Table 5.3. 68

5.5 Comparisons of speedup for MPS, IMP, ITS and CSA. 69

5.6 Comparison of Speedup Results for MPS with Varying Cutoff Times 71

5.7 Comparison of Speedup Results for IMP with Varying Cutoff Times 72

5.8 Comparison of Speedup Results for ITS with Varying Cutoff Times . 73

5.9 Comparison of Speedup Results for CSA with Varying Cutoff Times 74

6.1 Motivating Example Schedule (no ISEs) 84

6.2 Motivating Example Schedule with ISEs 84

A.1 Calculation of W, C, and WCET for example problem 91

A.2 Calculation of COMM, XC and S for dependent nodes 92
A.3 Calculation of T, R, XD and XDPossible for independent nodes . . . 92

B.1 Steps covered by each algorithm 94

List of Tables

4.1 Truth table for xc given the constraints 4.1.7 through 4.1.10 42

4.2 Truth table for $xdPossible(V_i, V_j)$ given the constraints 4.1.12 through 4.1.15 44

5.1 Comparison of the algorithms presented in [7] compared to the optimal solution. The numbers represent the length of the critical path in terms of clock cycles. 63

5.2 Results of both the MPS and IMP algorithms, as compared to the optimal or theoretical optimal solution, represented in the column "OPT." Results marked with an $*$ were found to be optimal. 65

5.3 Normalized comparison of heuristic results from [7] and the task level algorithms from this thesis 67

5.4 Average speedup calculated for MPS, IMP and ITS up to and including 100 nodes. This calculation is with respect to a uniprocessor approach. Note that for ITS and CSA the model assumes two issue slots per processor, whereas all other results are assumed to have one issue slot per processor. 70

5.5 List of hand optimized ISEs used in [24] 75

5.6 Speedup of FFT scheduling with ISEs in [24] 75

5.7 Speedup of FFT scheduling with and without ISEs in this thesis . . 76

List of Terms

- Arc - A directed or undirected connection between nodes
- ASIP - Application Specific Instruction-set Processor
- B&B - Branch and Bound
- BB - Basic Block
- CFG - Control Flow Graph
- CSA - Combined Scheduling Algorithm
- DC - DeClustering
- Eclipse IDE - A free and open-source software development platform
- FPGA - Field Programmable Gate Array
- GCC - GNU Compiler Collection
- GNU - GNU's Not Unix
- GUI - Graphical User Interface
- HDL - Hardware Description Language
- HIR - High Intermediate Representation

- IDE - Integrated Development Environment
- ILP - Integer Linear Program, also Instruction Level Parallelism
- IMP - Instruction set extended MultiProcessor scheduling algorithm
- Incumbent - Current best solution to a branch and bound problem
- ISE - Instruction Set Extension
- ISM - Instruction Scheduling Model
- ITS - Instruction and Task Scheduling algorithm
- LIR - Low Intermediate Representation
- LMBC - Load Balancing with Minimized Communications
- LP - Linear Programming
- LPTDC - Largest Processing Time variant of DeClustering
- MILP - Mixed Integer Linear Program
- MIMO - Multiple Input Multiple Output
- MISO - Multiple Input Single Output
- MPS - MultiProcessor static Scheduling algorithm
- MP-SOC - MutiProcessor System-On-Chip
- Node/Vertex - The base unit of a graph
- PPS - Preferred Path Selection
- Quartus-II - An HDL design entry IDE from Altera

- RISC - Reduced Instruction Set Computer
- SHIRA - Software/Hardware Implementation and Research Architecture
- SING - Simulator and Interconnection Network Generator
- SSA - Static Single Assignment
- Task - Another term for Basic Block
- Verilog - An HDL
- VHDL - VHSIC Hardware Description Language
- VHSIC - Very High Speed Integrated Circuit

Chapter 1

Introduction

1.1 Background

Scheduling in the field of computing is the method of ordering portions of a program (called processes) to a set of resources (CPUs for example) in order to maximize efficiency and minimize processor utilization. The goal of scheduling is to meet a timing or power requirement as specified by the developers. Scheduling can operate at multiple levels. When scheduling threads or data flows at the operating system level, methods such as round robin can be used to assign resources to threads or flows for a certain length of time. At the task level, sequential sequences of assembly code between branching operations are assigned to the resources on a single processor or core. The lowest level, instructions, are assigned individually to the resources in each processor or core. For task and instruction scheduling, algorithms are applied as a compiler optimization step in order to reduce pipeline stalls, avoid illegal execution orders, and to exploit as much low-level parallelism as possible.

Scheduling algorithms are composed of two major stages. The first, sequencing, defines the execution order, and is accomplished by building a set of dependencies for each node and reordering them based on their dependencies [30]. This creates a

Directed Acyclic Graph (DAG) which can then be used by the second stage. The second stage, scheduling, removes ordered nodes from the dependency list once they are no longer dependent on a prior node, and places them into available slots/cycles for execution [30]. The longest sequential series of nodes representing tasks or instructions in the DAG is called the *Critical Path.* The overall goal of scheduling is to minimize the runtime of a program by minimizing the critical path. This is accomplished by running independent tasks or instructions on different processors.

Current CPUs, GPUs and Systems-on-Chip (SoCs) offer more Instruction Level Parallelism (ILP) than can be currently exploited by the majority of programs being run on them. This is because the compilers compiling these programs are unable to exploit all of the available ILP. Concurrency mapping for multiprocessor systems has been a topic of study since the 1960's [21, 7]. Despite this, it was shown in [22] that only about 10% - 33% of instructions are dual scheduled in a single cycle. This level of parallelism amounts to small speedup, especially when the theoretical speedup is much higher. Following Amdahl's Law, a program that can be fully parallelized (that is, 100% of the program can benefit from parallelization) will have a speedup that is equal to the number of processors that it can be scheduled across. In the case of a two processor system, the maximum speedup would be 2, except in a case of super-linear speedup.

Scheduling can be categorized into two major methods; those performed before the register allocation step, and those performed after register allocation. Each method has its own advantages and disadvantages. Scheduling before register allocation allows for maximum parallelism. However, if the number of virtual registers used in creating this parallelism exceed the number of registers actually available during that clock cycle, spill code will need to be inserted to avoid deadlocks, resulting in a slower execution time. Scheduling after register allocation is more difficult because of false dependencies arising from the register allocation. This limits the amount of parallelism available for use. However, as all registers have been allocated, no spill

code is generated.

In the case of static scheduling, the critical path of a portion of static code can be calculated at compile time. When dealing with dynamic scheduling, the critical path may not be possible to calculate at compile time, and is better suited to heuristic approaches. However, as "scheduling methods in the presence of variations typically rely on worst-case timing estimates for hard real-time applications"[21], heuristic scheduling dynamic portions of the program at runtime can be aided by optimizing the static portions as much as possible at compile time, thereby minimizing the worst case execution time.

1.2 Motivation

The literature currently dominating the field of scheduling at the instruction and task level is based on heuristic techniques [21, 29, 34], which all state that due to its NP-hard quality [6], scheduling problems cannot be optimally solved in a reasonable amount of time. However, as heuristics (including genetic algorithms and interior points methods) lack the ability to provide optimal and deterministic results, they were used in this thesis only as a comparison to the ILP methods implemented.

The motivation for using an ILP based approach was due to the fact that these algorithms were aimed at solving sections of a large program, and not the program as a whole. These algorithms were to be used as a compilation pass that would optimize the static code (which would then be scheduled dynamically at runtime by a heuristic algorithm). They would not need to be performed in real time and the longer compilation times were considered an acceptable trade-off for the greater benefit to processing time that the ILP based algorithms would provide.

1.3 Objectives

The objectives set for these algorithms were to provide a scalable method for scheduling static code across multiple functional units (FUs) or processors. The code will be scheduled at both the basic block and instruction level, and would be required to schedule instruction sets that had been extended from previous work [25]. The FUs or processors should be seen as generic, so that the algorithms can be used to generate valid schedules for multiple processor types. Additionally, the number and arrangement of processors or FUs should also be generic; the algorithms will be able to schedule for an N-functional unit processor. Finally, the communication time between the processors or FUs should be generic. Different communication times are to be taken into account, and the algorithms will be able to schedule code with varying communication costs between FUs or processors.

1.4 Contributions

The four scheduling algorithms modified and created for this thesis were based on an Integer Linear Program (ILP) which provides deterministic results, and would provide provably optimal solutions in cases where the ILP finishes. The novelty of this work lies in the fact that the final algorithm, (discussed in Section 4.4) combines multiple processors, multiple levels of scheduling, and instruction set extensions. A visual representation of the algorithms and their coverage can be seen in Appendix B. In the Appendix image the algorithms are clearly separated with MPS performing only task level scheduling, IMP performing both task level scheduling and instruction set extension identification, ITS performing task and instruction level scheduling, and CSA performing all three functions. Other research studies, examined in Section 3, perform their scheduling on some of the components, but to the best of our knowledge, none perform all of the steps done in this work.

In order to keep the problem size reasonable, chunking [35, 8] of the problem was used. This creates a set of small problems that can each be fully optimized, before the set of optimized chunks is itself optimized. This is expected to provide more exact and repeatable results than a heuristic, and provide better results than attempting to optimally schedule the entire scheduling problem in a single piece. Additionally, constraints on the system are used to prune the search space to the point where the problem becomes tractable. This makes the static scheduling algorithm useful in rapid design space exploration for micro-architectures and systems [13].

The compiler that we used to generate the control data flow graph (CDFG) that the ILP model uses is the COINS compiler, a retargetable compiler written in Java and based on GCC [5]. The modeling language used was the LINGO math programming language [23]. The Instruction Set Extensions (ISEs) that this work schedules in addition to regular instructions and tasks are generated from prior work [25] that has been integrated into COINS. This thesis adds to a toolchain detailed in Section 2.7 by adding a scheduling pass in the COINS compiler after the ISE identification pass developed in [25]. The algorithm allows for the scheduling of both regular instructions and ISEs in individual tasks as well as tasks in a function call or program.

1.5 Conditions and Assumptions

The algorithms presented herein perform the majority of their scheduling on a set of randomly generated programs of varying sizes. The algorithms are run assuming a two processor system connected through a symmetrical communication medium (i.e. the delay from Processor 1 to Processor 2 is the same as the delay from Processor 2 to Processor 1) for task level scheduling. The two processor system is selected as it is both a commonly used setup, and because the literature used for comparison also implements a two processor system. When scheduling instructions, the algo-

rithms work within a single processor, which itself has two functional units (with no communication delays between them).

The number of processors, the processing time for each instruction type, and the communication delays must all be known in advance, and are specified in definitions in the compiler. The type of processor can be changed by specifying a new set of instruction execution times in the definition of the processor. The processor is assumed to have only one instruction pipeline; multiple pipelines are not taken into consideration. Additionally, communication issues such as bus contention or dynamic delay are not taken into consideration. It is also assumed that any instruction set extensions found will be applied to all functional units. As the algorithms presented in this thesis are used to prove the ability of an ILP-based method to create viable schedules for data flow graphs composed of static code, the results are limited to the schedule of instructions or tasks, and no compiled code is output from the model.

The results of the pass run with and without ISEs, and with and without both instruction and task level scheduling show the gradual improvements in runtime as more portions of the pass are utilized. The work presented in this thesis shows that an ILP-based approach to scheduling both tasks and instructions to multiple processors produces high levels of parallelism for sparsely connected data flow graphs, can produce deterministic results in all cases, and very quickly reach the optimal solution for smaller problems. The speedup measured for the algorithms is in comparison to the sequential case, where 1.00 is the time taken for the instructions to execute sequentially, with no slack between instructions, on a single processor

1.6 Organization of Content

Chapters 2 and 3 detail various scheduling algorithms and technical information, as well as detailing recent publications in the field. Chapter 4 of the thesis is broken down to give a step by step explanation of each of the algorithms. Chapter 5 gives

an explanation of the programs that were tested as well as the results of these tests, and compares them to current approaches. The results are concluded in Chapter 6 and discussion of possible extensions to this work are also included. Appendices will include example programs, as well as diagrams to aid in the explanation of the algorithms presented.

Chapter 2

Background

The following chapter is an overview of the technologies, terminology, and methods used in the implementation of the algorithms used in this thesis. The chapter will first cover what Instruction Scheduling Algorithms (ISA) are, as well as the types of ISAs currently in use in industry. This section will not include ISAs created or modified for work in this thesis, only those which served as an inspiration. The chapter will then cover Integer Linear Programs (ILP), as well as branch and bound techniques. The chapter will then give an overview of Field Programmable Gate Arrays (FPGA), compilers, and the SHIRA toolchain.

2.1 Intermediate Representation

Intermediate representations (IR) in compilers are languages used to more easily bridge the gap between source language and machine language. They are useful in that they allow a compiler to use a small set of IRs to allow multiple source and machine languages, rather than require a direct translation for every supported source language to every supported machine language [15].

An IR will generalize certain constructs in the source language in order to allow

multiple constructs in multiple source languages to be represented by their more basic components. If multiple transformations are required, multiple stages of IRs may be needed in order to successfully translate the source code to the machine language. This also simplifies the process of translation, as only a small portion of the translation needs to be performed at any given step [15].

In the COINS compiler [5], there are two major types of IR; High Intermediate Representation (HIR), and Low Intermediate Representation (LIR). The details of each will be expanded upon in the following sections.

2.1.1 High Intermediate Representation

HIR is composed of structures common to most source languages, namely C, FORTRAN, Pascal and Java. These include if-else structures, loops, variable assignment etc. After the source languages are transformed into HIR, several actions can be performed to limit the amount of code required to be translated into LIR and then machine code.

It is at this high level that code optimizations are usually performed (reducing the need to schedule portions of the program that will never be run). High level parallelization can also be performed, as can dependency analysis. It is at this stage that loop transformation and in-line expansion are also performed [5].

It is important that the translation be done at as high a level as possible (retaining as many high level concepts as possible that are common to most languages) as too much decomposition into a state closer to machine language would make for ineffective high level optimizations at this stage. The same is true for LIR. If the code is not close to being machine level code, low level optimizations would be ineffective.

2.1.2 Low Intermediate Representation

Whereas HIR contains structures common to multiple source languages, LIR contains compiler components used in various processors [5]. However, it does not contain anything specific to a single processor type. Thus it is easier to create general fine grained optimizations without requiring a set of optimizations for each architecture. Fine grained approaches such as register allocation and optimization of memory accesses are best performed at the LIR level.

In addition to those optimizations, it is as the LIR level that instruction level scheduling should take place. It is the level chosen for both the task and instruction level algorithms developed in this thesis to perform their work. As registers have been specified, no spill code is created, preventing the schedule from becoming longer from inserted code.

2.2 Instruction Scheduling Methods

As defined earlier, instruction scheduling (and scheduling in general) is composed of two major steps. The first step is to sequence the instructions in a DAG based on their dependencies. The second step is to placed these ordered instructions into the processors or ALUs in order of their place in the DAG and within the bound of the system constraints.

There are several types of dependencies that must be considered when ordering instructions into the DAG.

1. Flow Dependence: Instruction i_2 is flow dependent on instruction i_1 if the output of i_1 is used as an input of i_2.

2. Anti-dependence: Instruction i_2 is anti-dependent on instruction i_1 if i_2 writes to a value that must first be read by i_1.

3. Output Dependence: Instruction i_2 is output dependent on instruction i_1 if both i_1 and i_2 write to the same location, and i_1 must execute before i_2.

Once the instructions have been sequenced into the DAG based on dependencies, the Data Dependence DAG (DDD) can be passed onto the second step for instruction scheduling. The two main system constraints for the scheduling step include the number of available registers and the number of functional units in the processor. However, the system can still be constrained by the number of reads and writes available to the system, as well as the instruction encoding.

2.2.1 Local Instruction Scheduling Algorithms

Local ISAs schedule instructions within basic blocks, where a basic block is defined as the set of sequential instructions between branch or jump instructions [30]. Instructions must be scheduled within the sequential runtime of the basic block, and cannot be moved past these boundaries. The most common type of local scheduling algorithm is List Scheduling (LS).

2.2.2 List Scheduling

List scheduling algorithms, such as the DeClustering (DC) and Largest Processing Time DeClustering (LPTDC) algorithms from [7], function by creating three different sets of data: the directed graph which contains sequenced instructions, the data ready set (DRS), which contains schedulable instructions (i.e. instructions with no dependencies) and the schedule itself. The basic steps are to find instructions in the graph that have no dependencies/predecessors and add them to the data ready set. From there, the next step is to take an instruction from the data ready set and attempt to schedule it. This is repeated until both the graph and data ready set are emptied.

Central to LS is the priority function. This function picks which operations go into the schedule, and has two main goals. The first is to generate the shortest schedule possible, and the second is to only generate valid schedules. The ability to generate valid schedules is taken care of by the DRS, while the ability to create short schedules are governed by several heuristics. The four main heuristics as stated in [30] are:

1. Height: The number of cycles that must pass between the execution of an operation and that of the sink in the DDD.
2. Urgency: Defined by the size of the absolute timing interval of an operation. The smaller the interval, the higher the urgency that it must be placed within that cycle.
3. Resource Requirements: The more operations requiring a given resource, the more likely it will be that a given operation requiring this resource will be scheduled early in order to avoid resource conflicts.
4. Successors: The number of successors to a given operation.

The priority function gives a priority to every operation based on the above heuristics. As certain values (such as resource requirements) are machine dependent, the priority function itself is machine dependent.

Avoidance methods

Restrictions on timing can result in no valid schedule being found. Some heuristics can be used to increase the probability of a valid solution being found, but tend to lengthen the schedule length in places where it is not needed [30]. There are, however, other methods that can be utilized to generate short, valid schedules.

Advantages and Disadvantages of local scheduling algorithms

Local scheduling has the benefit of a simplified view of the problem of scheduling. As it operates within a basic block, local scheduling algorithms never need consider jumps or branches when attempting to schedule a piece of code. This allows for a more rapid assignment of sequential statements to processors, giving local scheduling techniques the benefit of faster compiler execution times.

This speed and simplicity come at the cost of almost never being globally optimal. As instructions or tasks are scheduled, space between basic blocks, or within basic blocks may be wasted. As the local scheduling algorithms cannot use code from outside the basic block to fill in these spaces, the processors go idle more than may be necessary. To become globally optimal, the task or program being scheduled must be viewed as a single problem, with all the jumps and branches included. This can only be accomplished with Global scheduling, as will be discussed in the next section.

2.2.3 Global ISAs

While local scheduling techniques do exploit a decent amount of ILP, they can never be optimal, as they can never fully exploit all of the ILP available from modern processors [30]. In order to completely utilize this available ILP, independent instructions must be moved between basic blocks. This requires a global approach to scheduling.

Trace

Trace scheduling is the most basic of the global scheduling algorithms [30], though it works well on multiple architectures. The objective of trace is to generate a schedule optimized for a selected trace. The trace algorithm can be broken down into three major steps. First, the trace is selected from a control flow graph. The trace contains in it basic blocks which have yet to be scheduled, and is a unique set of operations. The second step is to schedule the selected trace as though it were a basic block.

The third step adds compensation code to the list of unscheduled instructions, if the previously scheduled trace had code motions that required extra code to be inserted into the program.

Tree

Tree scheduling is a variant of Trace scheduling that attempts to deal with the potential exponential growth of code that can occur in trace [30]. Tree accomplishes this by working on disallowing moves in Trace that would result in block duplication. Tree first partitions the CDFGs into top trees, and a varied form of Trace scheduling is applied to them. Next, the CDFGs are partitioned into bottom trees and the same Trace variant is applied. While not all blocks will be present in both trees, they will be present in one and will therefore be handled in either the first or second scheduling pass. This method limits the potential exponential growth of code to one that follows an O(n) growth. While Tree does result in slightly longer schedules, [30] shows that Tree outperforms Trace due to better memory allocation.

Overall disadvantages of global scheduling algorithms

While a global scheduling method has the possibility of being globally optimal (whereas a local scheduling method can only be locally optimal) the main issue with it is that the NP-hard characteristic of task and instruction scheduling leads to the use of heuristics, or will result in an algorithm that will never complete. As it is the main focus of this research to find an optimal, deterministic method of scheduling custom instructions and tasks, the heuristic method is unusable for our research due to its lack of determinism and provable optimality. With the only other option being to use ILP on the entire problem being unfeasible, we opt instead for breaking the global problem into multiple sub problems, and achieving local optimality using the ILP. From there, the locally optimal pieces can then be scheduled by the ILP in an op-

timal fashion. While this does not provide global optimality, it does allow for local optimality at every stage.

2.3 Clustering Methods

Clustering heuristics such as the Load Balancing with Minimum Communication (LBMC) and Preferred Path Selection (PPS) algorithms from [7] are applicable to multiple types of algorithms, and are used to decrease the number of combinations by placing closely related nodes into a single node. This is accomplished by defining an amount of linking between nodes that will be used to cluster them into a single node. This single node is then given a set of values corresponding to computation time and priority. This reduces the number of nodes needed to be scheduled, at the cost of speedup, as nodes within the cluster are not optimized.

2.4 Linear Programming

A Linear Program (LP) is a mathematical approach to determine the best outcome in a given model given a list of requirements and constraints. The best outcome may be the maximum or minimum value for one of the variables in the model, and the requirements and constraints to the model are given as a set of linear equations. These outcomes and variables can be broken down into four categories [4].

- Decision Variables: A set of values which are not known when the LP is started, but which can be controlled in order to provide the best outcome.
- Objective Function: A variable which is set to be either maximized or minimized as the goal.
- Constraints : Linear equations that set limits on individual variables or combinations of variables. Used to limit the solution space to valid combinations.

- Variable Bounds: A constraint on a variable that prevents it from being any value. Example: production from a machine must be greater than 0, and less than 1000 per day.

The set of linear constraints and variable bounds will create a feasible region of solutions. A maximized objective will be found along the largest edge of the feasible region. For example, given that product x_1 can only be manufactured twice per day, product x_2 can only be manufactured thrice per day, and only four items can be packaged and ready for shipment per day give the following set of constraints.

$$x_1 \leq 2 \tag{2.4.1}$$

$$x_2 \leq 3 \tag{2.4.2}$$

$$x_1 + x_2 \leq 4 \tag{2.4.3}$$

The valid numbers of product that can be made in a given day are within the feasible area as shown in Figure 2.1. The best feasible solutions, which are the maximized objective function are all solutions along the line $x_1 + x_2 \leq 4$. When an LP has searched all of the feasible area, it will have completed, and will have found the maximized objective function. In the case of scheduling, this amounts to the shortest possible schedule. When the models used for the algorithms in Chapter 4 complete, the model finishes and outputs the completed schedule, and states than an optimal makespan has been found for the given set of constraints. As the size of the feasible area increases, the LP will take progressively longer to complete its search. This reduces the likelihood that the LP will find a valid solution within a given amount of time, but provides no ability to predict whether or not the optimal will have been found by a certain time.

As it is possible to half finish a product in a given day, all answers along the line $x_1 + x_2 \leq 4$ are valid. However, in the case of Integer Linear Programs (ILP), the only valid solutions are those that are composed only of variables that are integers.

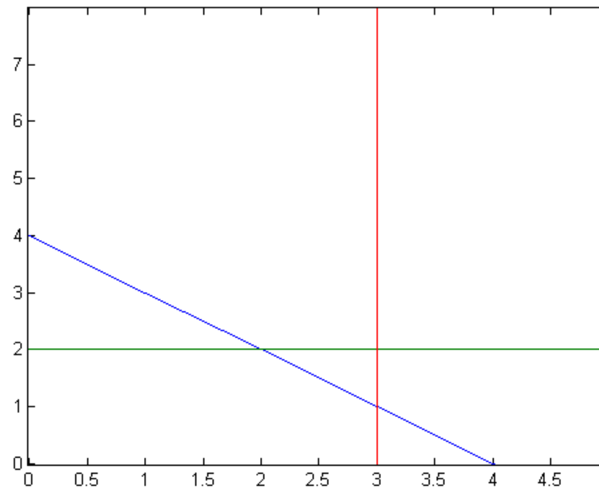

Figure 2.1: Example LP

An example of this would be assigning people to the machinery; it is not possible to assign half a of a person to a machine. In this case, there are only two valid solutions, the point (1,3) and the point (2,2).

2.5 Transitive Closure

In a directed graph G, the transitive closure of G is the set of all nodes reachable from any node V. It establishes a matrix that allows for easy solutions to the question of reachability [28]. Construction of the transitive closure of a graph answers the reachability question for any node in the graph with $O(1)$ complexity.

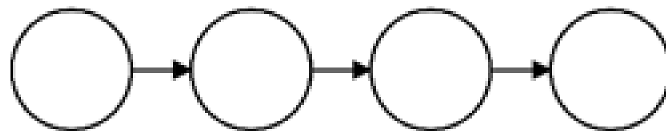

Figure 2.2: Sample Dataflow Graph

For the example graph in Fig 2.2, the matrix M_1 showing direct connections can

be created. It is assumed that any node can reach itself. From this, the transitive closure can be built. For any node V_i that can directly reach node V_j, node V_i updates with the nodes that V_j can reach.

$$M_1 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1) \end{pmatrix}$$

Updating M_1 with this information creates the transitive closure T.

$$T = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1) \end{pmatrix}$$

The transitive closure T can be represented by the graph in Fig. 2.3, where all nodes V_i that can reach nodes V_j through one or more hops are shown with direct connections.

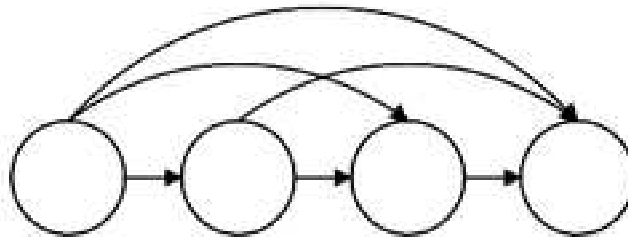

Figure 2.3: Sample Dataflow Graph with arcs showing transitive closure

2.6 Branch and Bound

The branch and bound (B&B) approach is a technique for solving integer programming problems in situations where the standard approach to LP cannot be used. B&B

is based on the fact that the result of enumeration of integer solutions results in a tree. Each branch of the tree represents a different value for each of the variables in the problem, and the tree will have an equal number of branching levels as it does variables (plus the root of the tree). An example of this for a three variable system with the constraints

$$x_1 \leq 3, 0 \leq x_2 \leq 1, 0 \leq x_3 \leq 1 \quad (2.6.1)$$

can be seen in Fig. 2.4. Each leaf node represents one of the solutions to the entire system. The other nodes represent sets of possible solutions. Each node has the same constraints as its parent node (plus additional constraints).

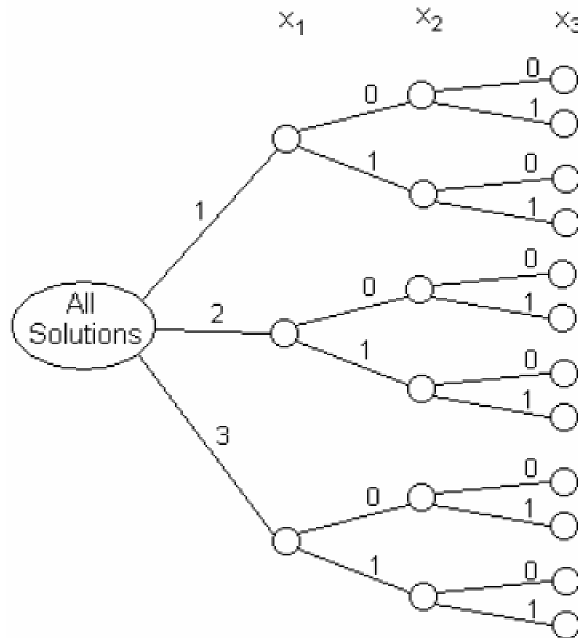

Figure 2.4: Complete Enumeration Tree

The purpose behind branch and bound is to avoid creating a large tree, which would be too time consuming to traverse [4]. To accomplish this, the tree is grown in stages, starting from the node that is the most promising. This node is determined

at each stage in the tree's growth by estimating a bound on the objective function. This bound will represent the best obtainable value of the objective function. If the bound on the objective function is estimated well, then only a small fraction of the tree is ever grown.

In addition to branching and bounding, the algorithm also prunes sections of the tree it deems unusable. This occurs when it can be shown that the node and its descendants will never produce a feasible or optimal solution. This pruning also prevents the tree from growing too large.

In B&B, there are three main ways to select nodes [4]. Best-first selection chooses the bud node that has the best value of the bounding function throughout the entire B&B tree (that has been grown so far). Depth-first selection also selects the bud node with the best value of the bounding function, but chooses only from the bud nodes that have just been created. As depth-first selection goes one level deeper for each iteration of the program, it can lead to an incumbent in as few as n steps (where n is the number of nodes). Breadth-first selection is the opposite of depth-first, and simply selects bud nodes in the order that they were created.

Once the bud node has been selected, it and its descendants must be analyzed to see if they can be proven infeasible or sub-optimal. Proving that a node will be suboptimal is a simple comparison of the node's bounding function value with the one selected. If the node's value is worse than the bounding function value, the node and all its descendants will never be optimal. Proving a node to be infeasible is more complex. Given the binary problem

$$-10x_1 - 5x_2 + 6x_3 + 4x_4 \geq 0 \tag{2.6.2}$$

where x_1,x_2,x_3,x_4 must be 0 or 1, and solution $(1, 1, ?, ?)$ will result in

$$-15 + 6x_3 + 4x_4 \geq 0 \tag{2.6.3}$$

No possible solution for this can exist, and the solution $(1, 1, ?, ?)$ and all of its descendants can be pruned from the search space as shown in[4].

2.7 SHIRA

The work of this thesis is a part of an ongoing effort to design a toolchain for the automated design of a customizable embedded system. The Software Hardware Interconnect Research Application (SHIRA) toolchain is the current realization of this effort. The framework for SHIRA is shown in Fig 2.5, and is made of several pre-existing tools, including the Eclipse IDE, the *gprof* profiler, and the COINS compiler [32] [9] [5]. SHIRA is designed to be both user friendly and to abstract the complexity of designing an embedded system from the UI while still exposing the functionality of the underlying components.

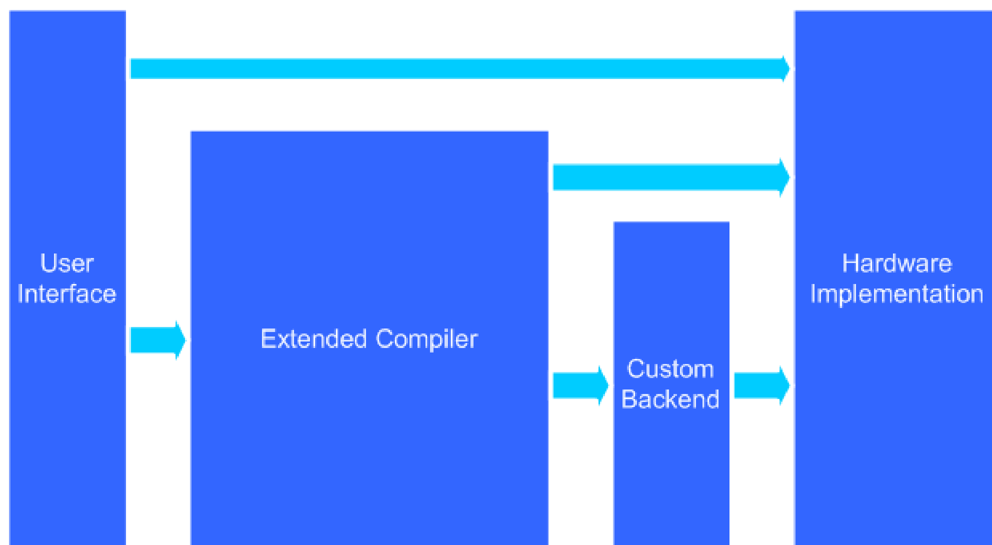

Figure 2.5: SHIRA Framework

At present the user interface block is still in progress, and several pieces in the compiler and custom back end are complete. The hardware implementation block is partially complete allowing a set of hardware blocks to be placed into an interconnection network. The work of this thesis lies between the compiler and back end, as it takes the code just before it has been fully compiled (working at the intermediate representation level) and performs scheduling just before the code would be turned

into architecture specific instructions in the custom back end. The instruction set extension used in this thesis and taken from [25] also exists at this level.

2.8 Compilers

A compiler is a program used to translate source code (such as C/C++ or Java) into machine code to be used by the processor that will run the program [1]. During this translation the code goes through multiple changes on its way to becoming machine code. Advanced statements and constructors are replaced with their operational equivalents. Class structures are replaced with pointers, complex calculations are broken into three address code and variables used many times are put in single static assignment form.

Central to the SHIRA toolchain is the COINS compiler infrastructure, which is a multi-pass compiler written in Java [5]. COINS is also an optimizing compiler, and allows for different optimizations to be used on the different passes. Passes can also be implemented in the different intermediate representations that the code passes through on its way to becoming machine code. The fact that COINS is so modular makes for faster execution for compilations requiring less than the total number of passes and optimizations to be used. Additionally, new compilation techniques and optimizations can be added to COINS without the need to rewrite the rest of the compiler.

2.9 FPGAs

A Field Programmable Gate Array (FPGA) is an integrated circuit made up of multiple logic elements connected by a switching fabric [16]. This allows the user to route signals through different logic elements to create a multitude of circuits, state machines and processors onto the FPGA. This programmability and reusability make

FPGAs an inexpensive testing platform for designs heading to the ASIC market.

As the clock speeds of FPGAs continue to increase [33] [16], their usefulness as a reprogrammable final product also increases. Until recently, the speed of FPGAs limited them to a testing platform, as results comparable to that of a high speed ASIC were not possible. With the increase in clock speeds, FPGAs have become a suitable production platform.

2.10 NIOS II

The NIOS II is a 32-bit RISC soft-core architecture designed for development on Altera's FPGA development kits [2]. The soft core nature of the NIOS II allows programmers and developers to customize nearly every aspect of the processor. The main reason it was chosen as the architecture for this thesis was its ability to add custom instructions. Like its native instructions, the custom user-defined instructions for the NIOS II can use up to 2 32-bit registers as inputs and can write back to one 32-bit register. While this does not allow the flexibility to use the multiple-input multiple-output instructions that can be specified for a VLIW architecture, the 2-input 1-output instructions it supports are more than adequate for our testing purposes.

In addition to custom instructions, the NIOS II allows custom user defined peripherals, such as the fuzzy logic co-processor being developed by other members of the CARG team [31]. This feature is useful for testing a fully integrated system including custom instructions, scheduling, and co-processors, which is a future goal of the CARG team. The NIOS II also includes a Memory Management Unit (MMU) and a Memory Protection Unit (MPU) allowing it to run operating systems such as uClinux and FreeRTOS [2].

Future development into the scheduling algorithms developed for this thesis would most likely be tested on the NIOS II, given its flexibility and the CARG

team's familiarity with the device.

2.11 Fourier Transform

The Fourier Transform (FT) is a mathematical conversion from the time domain into the frequency domain. In the case of computer processing, a set of sample points it used in place of a continuous function, called a Discrete Fourier Transform (DFT). The decomposition of the original signal into its component frequencies is accomplished by creating a set of sine waves of varying amplitudes that will mimic the original signal when summed. This makes the decomposed signal (in the frequency domain) practical for use in analysis of the signal's strength [3].

For a given sample size N for a DFT, the equation generating the frequency domain is

$$X(k) = \sum_{n=1}^{N-1} x(n) W^n k_N, k = 0, 1, ..., N-1 \tag{2.11.1}$$

where

$$W_N = e^{-j(\frac{2\pi}{N})} \tag{2.11.2}$$

The DFT of a signal can be broken down into a set of additions and multiplications for the real and imaginary portions of the equation. Its ability to be easily parallelized makes it an ideal candidate for the set of ILP algorithms in this thesis.

2.12 Chunking

Chunking is a method used by mathematicians for breaking down complex problems into manageable portions [35]. It is also employed by teachers to break down problems, allowing students to grasp that the solution to a problem does not necessarily have to be done in a single step [8]. In the case of the ILP algorithms in this thesis,

and the models they work upon, chunking can be employed to break the program into smaller pieces. In the event that a program or basic block is too large to be processed, artificial branches in the program can be used to break the code up into manageable sized portions. In the work in this thesis, IF statements (forced to be true) are used to create artificial branches, with the next chunk of code within the selected branch. This is especially useful for Fourier Transforms, as a 64-point transform contains close to 16384 lines of code once broken down into three address code in Single Static Assignment (SSA) format.

2.13 Instruction Set Extensions

The instruction set extension identification research from [25] is implemented in the SHIRA toolchain as a compiler pass. Instruction set extension identification is performed by searching the set of nodes in the program. The research in [25] focuses on finding instruction set extensions (ISEs) within a basic block/task. Upon completing the search of a task's instructions for convex (i.e. implementable and schedulable) patterns, a list of candidate ISEs is created.

Once the list of candidate ISEs has been created either in full or to within a time constraint, a knapsack problem bound by a hardware size variable specified in SHIRA fits the ISEs to the hardware attempting to maximize the total speedup. This speedup is calculated by comparing the speedup of the candidate ISEs to their corresponding hardware cost. For example, a single ISE candidate may provide a speedup to the instructions it replaces, reducing the critical path by 300 clock cycles. However, if this single instruction requires all available hardware, it would be more beneficial to implement two ISEs which each decrease the critical path by 200 clock cycles, but which both will fit into the given hardware. Implementing the pair of smaller ISEs would result in a total reduction of 400 clock cycles.

The task of finding ISEs is performed on each program before it is scheduled.

Differences in the instruction mix specific to the application may result in a different set of ISEs being the best performers. While a arithmetic intensive program such as FFT may benefit from a multiply and accumulate (MAC) ISE, one that contains mostly logical operations will not benefit from the addition of MAC to the instruction mix. The instruction set extension identification highlights a new set of candidates suited to each application.

Chapter 3

State of the Art

Specific examples of previous and state of the art research and development for task and instruction-level scheduling and instruction set extension will be summarized in this chapter. This is to provide specific examples of both ILP and heuristic based approaches, as opposed to the more general description of ISAs given in Chapter 2. Section 3.1 describes recent approaches to task and instruction scheduling, Section 3.2 covers types of VLIW compilers and their approaches, and Section 3.3 covers recent approaches to Instruction Set Extensions.

3.1 Scheduling Approaches

This section will discuss recent research in the field of task and instruction level scheduling. Each method will be examined and compared to the ILP methods created in this thesis.

3.1.1 MAX-MIN Ant System Optimization

The Max-Min Ant System (MMAS) algorithm was originally introduced in [34] and is a cooperative heuristic search based on the behaviour of ants. The authors of [34]

observed that ants are able to create an optimal path between a food source and their colony, despite their poor vision. Not only were the ants able to accomplish this goal, but they were able to create this path in a very short amount of time. This is because while a single ant may take a random path, it will decide which direction to move based on the strength of a pheromone trail. As each successive ant covers similar ground, the portions of the near optimal path will be covered with more pheromones, resulting in a stronger signal to each successive ant.

The Ant System (AS) algorithm is based on this method. It consists of a number of agents that cooperate to explore the search space. this method has been successfully modified for applications to a number of NP-hard problems [12]. As mentioned in Section 2, list scheduling algorithms depend heavily on their priority list. AS uses an evolutionary approach to this problem, using iteration in searching for an optimized schedule. Like with the methods presented in this thesis, AS uses two stages to generate the schedule. First, a set of agents traverse the data flow graph, each generating an instruction list. As with the description of real ants, the agents will each be creating a pheromone trail in the form of the instruction list, many of which overlap. These lists are given to the second stage which finds the parts of the schedules that have the highest quality. This quality is established by overlapping all of the instruction lists to find those which are most repeatedly covered (i.e have the strongest pheromone trail).

One of the main drawbacks of the AS approach is the possibility of ending up at a local minimum schedule length, while not being able to find the global minimum. The Max-Min Ant System (MMAS) builds upon traditional AS in order to address this problem. MMAS does this by expanding the bounds on pheromone trails left by the agents in the first step of AS. This results in a non-zero chance that any path will be chosen, thereby preventing AS from becoming stuck at a local minimum.

MMAS was shown to outperform several variants of List Scheduling, as well as Force Directed Scheduling, with speedup as high as 23% with an average speedup of

6.2%. However, for filter benchmarks such as FIR, MMAS fails to find near optimal results, and has only slightly better numbers than the list Scheduling and Force Directed methods. The work in [34] focuses on resource constrained static instruction scheduling, and shows that over 99.3% of instructions have fewer than 90 nodes. As such, the inability to provide either determinism or provable optimality shows that there is room for better, more exact methods, provided that the results can be obtained in a reasonable amount of time.

3.1.2 Statistical Optimization Approach

The work in [21] presents a pair of algorithms based on statistical makespan analysis. They use a heuristic and simulated annealing, respectively, for improving the makespan for soft real-time applications that do not have exact or strict timing requirements. The method used is designed to be implemented on heterogeneous multiprocessor systems, and was motivated by the variations found in applications such as H.264 high definition video decoding.

The method used in [21] implements Monte Carlo analysis by focusing on problems limited to the order of a few hundreds nodes, as opposed to thousands or millions. Additionally, the Monte Carlo analysis performed is done in an incremental fashion to decrease runtime.

Both the Monte Carlo base heuristic and the simulated annealing in [21] were successful in improving the makespan of H.264 and MPEG-2 by 6.4-30% for statistical guarantees of 99%, 95% and 90%, compared to approaches using worst-case estimates. However, as the focus of this thesis is on real-time systems with hard real-time deadlines, a statistical approach cannot be utilized. As there is always a chance that the methods presented in [21] may fail to meet their deadlines, they are unsuitable for the applications focused on in this work.

3.1.3 Modified Mixed Integer Linear Program Approach

The authors of [29] focus their approach towards optimally scheduling instructions to lower power requirements while still providing an acceptable runtime. To do so, a MILP formulation was integrated with the Dynamic Voltage Scaling (DVS) and the Dynamic Power Management (DPM) that are used to reduce power consumption through exploitation of idle times and tradeoff between noncritical runtimes and power consumption.

As with the approaches taken in this work, the work in [29] required modifications to their MILP in order to make runtimes more acceptable. As with our work, the authors of [29] use linearization techniques to reduce the runtimes. The work in [29] also employs a run time heuristic to integrate the DPM and DVS techniques. As the work of this thesis is aimed at real time systems with hard real time constraints, a heuristic such as this is not useful, as it cannot give the guarantees required for the system.

Another modification to the MILP that cannot be utilized in this work is a set of changes to the MILP that "trade-off design quality for reduced run times" [29].

The results of the basic MILP (without relaxed constraints) provides a 34.26% improvement over the existing techniques, when tested with MPEG-1, JPEG and MP3 benchmarks. While the heuristic techniques ran much faster than the MILP, they came to within 6% of the optimal. However, the lack of determinism in several of the approaches prevents these techniques from being used in the real time applications that this thesis is geared towards.

3.1.4 Decomposition-based Constraint Optimization Approach

The authors of [20] solve relaxed versions of the problems in Benders decomposition, using constraints and a two part program to prune the search space to reach their solution in a reasonable amount of time. While the work of this thesis does break

the solution up into parts, there are several differences in our approaches. Firstly, the problems solved in this thesis cover a large range of basic block sizes. While the problem sizes in [20] reach up to 100 tasks, the basic blocks in our set of programs reach well over 400 nodes, and have varying levels of communication, as well as a varying set of instructions in their makeup. This is far beyond the limitation of 30 tasks that the authors of [20] assert is the maximum size for linear programming methods to solve within a reasonable amount of time.

These basic blocks are cases where the amount of communication is limited, resulting in a much harder set of test cases (as less communication leads to more possible permutations for task placement).

As stated in [20], static scheduling is relevant for rapid design space exploration and signal processing applications. These fields are also the target of the work in this thesis; to quickly assess the runtime of a program on the system to be built before it is actually constructed. While [20] shows that ILP based approaches are extensible, they fail to use them in cases when the code itself has undergone instruction set extension. They perform only task scheduling (no mention of task-instruction, or other levels of hierarchy) and no mention of ISEs.

As the work contained in this thesis would be applied to a larger system (possibly used in conjunction with a higher level dynamic scheduler) more constraints (memory size, latencies, etc...) may need to be added to the model to prune the solution space. It was for this reason that an ILP approach was selected.

The authors of [20] break their algorithm up into master and sub-portions. Their approach jumps into the sub-problem solver after the master problem solver has reached the satisfiability criteria. The ILP based solutions presented in this thesis were inspired by this two stage approach to the scheduling problem, but approach it from a bottom-up direction, as opposed to the top-down implementation that the work in [20] used. Instead of running a master portion first to reach a set of satisfiability criteria, and then running the subproblem, we opt to run a sub-algorithm on the

instruction in each basic block first, and then hand the optimized blocks off to the master algorithm. This master algorithm then creates the final schedule with the optimized basic blocks. In the case of instruction set extensions, the ISEs are selected in a pass before the scheduling of instructions and basic blocks commences.

3.2 VLIW Compilers and Compilation Methods

This section will briefly explore methods for compilations geared specifically at VLIW processors, and their comparisons to the work done in this thesis.

3.2.1 Tetris

The register pressure reduction technique developed in [36] uses a set of complex partitioning methods to create an algorithm capable of reducing the quantity of spill code generated by prior development in VLIW compilers. This register pressure alleviation method is used in the step just prior to instruction scheduling and register allocation. This differs significantly from the methods used in this thesis, which operate after register allocation, and thus do not cause the creation of spill code. Results from the Tetris algorithm shows a 6% improvement over the previous approaches at reducing spill code for an 8 FU system with 32 registers. It should also be noted that the Tetris algorithm performs best with tight restrictions, as it gave less than a 1% improvement in a system with 8 FUs and 64 registers.

3.2.2 Adaptive Compiler Method

The authors of [14] developed and implemented a VLIW scheduling technique aimed at scheduling code onto dynamic reconfigurable VLIW processors. As their focus is different from the static focus of the work in this thesis, their choice of a heuristic method to solve this NP-hard problem was correct. By first mapping out the resources

used by each chunk of code and then scheduling it considering the available hardware, the authors of [14] achieve a 22.6% decrease in the amount of configuration data (for configuring the VLIW processor) necessary over previous methods.

This heuristic approach would be beneficial if combined with the compile-time approach used in this thesis. The optimization of each small static piece may be beneficial to the heuristic algorithm working at a more abstract level.

3.3 Instruction Set Extension Approaches

This section will discuss recent approaches in the field of instruction set extension, and will be compared to the method used in this thesis, which is the same as that used in [19, 26, 25].

3.3.1 The Instruction-Set Extension Problem: A Survey

The work in [10] is an overview of the methods used in the solution of the problem of Instruction Set Extension identification. The approach to solving this problem can be broken into two main categories: "complete customization [which] involves the whole instruction-set which is tuned towards the requirements of an application, [and] partial customization [which] involves the extension of an existing instruction-set by means of a limited number of instructions."

The approach used in this thesis, also used in [19, 26, 25] is of the latter. Certain instructions are not considered as candidates for the instruction set extension, including memory-to-register and register-to-memory operations.

The study in [10] shows that another division in the approach to instruction set extensions is the level of abstraction of the problem, generally referred to as the granularity. A fine grained approach works at the level of assembly operations and generally implements smaller clusters of instructions. A coarse grained approach

works at a higher level, identifying loops, and is able to implement a beneficial loop as a whole in hardware.

A fine grained approach (such as the one used in this thesis) achieves good speedup by benefiting from proper exploitation of isomorphism; cutting hardware costs by implementing small groups of instructions that appear many times within the program. A coarse grained approach achieves good speedup by selecting the largest and most beneficial portions to be turned into hardware, getting its benefits in a few large pieces, rather than dozens of smaller ones.

As each graph of n nodes contains 2^n possible graphs, this presents a large problem to the solution of finding an optimal solution. While some approaches go for a fast solution using inexact heuristics, the approach in [25] uses proper pruning of the design space to prevent the branch and bound algorithm from searching impossible instruction combinations.

Another difficulty noted in [10] is the problem of cyclic versus acyclic graphs. As the approach used in this thesis works in a fine grained manner, it unrolls all cycles to create an acyclic graph to work on.

Once all of the possible instruction set extension candidates have been found, an optimal set of these instructions must be found to maximize the gain for the problem at hand. While exact methods often require far too much time to run, while heuristics can not only not provide optimality, but even feasibility. The authors of [10] consider ILPs as a good solution to the problem, which is implemented in [25] as a knapsack problem.

Another division found in [10] is that of instruction I/O. One method for trimming the number of candidates is to consider only the subgroup containing multiple input single output (MISO) instructions, rather than the multiple input multiple output instruction (MIMO), which make up the remainder, and contain within them the MISO instructions. The method used in this thesis considers MIMO instructions, as MIMO allows for more ISE candidates to be identified.

The combination of exact methods, considering MIMO ISE candidates, and a fine grained approach make the method of finding ISEs implemented in this work a computation intensive, but highly effective algorithm.

3.3.2 Algorithms for the Automatic Extension of an Instruction-Set

The work in [11] showcases a pair of methods for the generation of instruction set extensions. The authors aim to break the task of finding convex multiple-input multiple-output instructions (MIMO) into two main steps. The first step finds a suboptimal set of multiple-input, single-output (MISO) instruction extension candidates using a heuristic. These MISO instructions are then given to the second portion of the algorithm, which combines different combinations of MISO instructions to create MIMO instruction set extension candidates.

While the authors of [11] do find decent speedup values, their main focus is simply on the speed with which the algorithm can be run at compilation. As stated before, the ISE selection method used in this thesis - as well as the overall goal of the scheduling algorithm developed in this thesis - aims for the opposite goal, which is to achieve the maximum possible program execution time speedup.

Instead of generating a set of MISO instructions that may or may not be beneficial when combined to form MIMO instructions (a process that may result in MIMO instructions that are simply a pair of independent MISO instructions, with no added benefit from the combination) the ISE selection algorithm from [25] searches for MIMO instructions directly; finding beneficial MISO instructions may occur, but is simply a byproduct of the search for MIMO instructions.

3.4 Summary

From the state of the art- several similarities and differences were highlighted in comparison to the methods used in this thesis. The algorithms in this thesis most closely resemble the constraint based approach of [20]. The ISE identification algorithm, originally from [25] and modified to be a pass in the compiler that is used in two of the algorithms uses several of the methods surveyed in [10], and takes into account more ISE types than [11]. When compared to the two state of the art techniques in VLIW compilers, the work in this thesis resembled neither method.

Chapter 4

Scheduling Algorithm Development

This chapter will detail the design and development of four different algorithms used in this thesis to schedule tasks and instructions. The algorithms are each run in the Lingo math tool. The models that are generated for each algorithm are created during compilation in Java, and are outputted in the Lingo modelling language to be run in the Lingo tool. To facilitate ease of execution of multiple programs, the Lingo tool is called automatically from the compiler after the model is created. The results of each instruction level execution are saved by the compiler to be used in creating the task level model.

These algorithms are used to generate speedup results, and will be compared to the prior art covered in Chapters 2 and 3. Sections 4.1 through 4.4 detail the creation of the MPS, IMP, ITS (originally developed as a part of other papers written in CARG [26, 19]) and CSA algorithms respectively. These algorithms were modified for use in this thesis, and the CSA algorithm contains elements from MPS, IMP and ITS. Some sets of equations used in the models for each algorithm will be repeated in other sections as needed.

4.1 Design of the MPS Algorithm

The Multiprocessor Static scheduling ILP model (MPS) addresses only the scheduling of basic blocks into processors but does not schedule within a basic block, and is described in this section. The section will begin by introducing the variables and constants used in the model, and will then describe the objective function and constraints. The example used in this section is the same one used in [20, 26, 27]. Many of the variable names in [20] are used in the MPS model for the same purpose in order to ease the comparison between approaches.

We begin by introducing the model variables and constants. Each task V_i has an execution time $w(V_i)$, and a start time $S(V_i)$. Each task is associated with a communication delay *comm* which represents the time that the task must wait to receive data from a predecessor task. The overall algorithm for MPS can be seen in Algorithm 1. For our example the tasks will be called V_1 through V_8. The dependencies among tasks is shown in Fig. 4.1.

Algorithm 1 MPS_TASK_SCHEDULE_PASS

1: Create Transitive Closure T and Identity Matrix ID and write them into model MPS
2: **for** $(basicBlock \in blocks)$ **do**
3: Find dependencies of $basicBlock$
4: Associate a value $comm$ to each dependency
5: Write $w(basicBlock)$ and dependencies into model MPS
6: **end for**
7: $RETURNmodel(MPS)$

Each dependency arc is also associated with a communication delay, named c, representing the time required to transmit data between tasks that are not assigned to the same processor. A detailed description of the relationship between *comm* and c is given below in Equation 4.1.3. The binary variable xc indicates when two tasks are assigned to the same processor.

All pairs of tasks (V_i, V_j) are associated with three numbers:

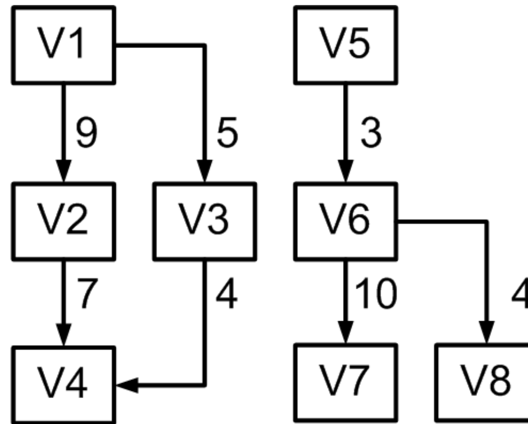

Figure 4.1: Dependency graph for the example described in [27].

1. The transitive closure $T(V_i, V_j)$ which is a constant
2. The binary decision variable $xd(V_i, V_j)$ which orders independent tasks running in the same processor.
3. The integer $xdPossible(V_i, V_j)$ (due to the constraints placed on it will always result in a 1 or a 0). $xdPossible(V_i, V_j)$ allows us to track the possibility that two tasks are independent.

The transitive closure T for every pair of tasks is calculated before the model is initiated, and represents the reachability of some task V_i by some task V_j by traversing task dependencies. We have assumed that the transitive closure of a task and itself is 1. The complete transitive closure array for the example in Fig. 4.1 is presented in the matrix below.

$$T = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

We assume a fixed number of processors and for this example we set the number of processors to 2. Each processor is associated with an execution time $Ptime$ required to complete all of the tasks assigned to it. For every task and every processor there is a binary decision variable $X(V_i, P)$ which represents the assignment of the task to the processor. X is 1 if the task is assigned to the processor, and 0 otherwise.

The execution time of tasks V_1 to V_8 in this example are represented by the constants $[3, 3, 3, 7, 2, 3, 5, 6]$. The variable w indexes these constants such that $w(V1) = 3$, $w(V2) = 3$, and so on. The communication delays are modeled similarly using c and the constant array $[9, 5, 7, 4, 3, 10, 4]$, with each number representing one of the arcs in Fig. 4.1. Effectively whenever c or w are used in the model, they are really representing a constant.

The objective of the model is to minimize the execution time of the application on the MP-SoC, called the *makespan* [26].

$$Objective = MIN(makespan) \tag{4.1.1}$$

An example of the following calculations for w, c, and $WCET$ can be seen in Fig. A.1 of Appendix A. We define the constant $WCET$ which represents the worst case execution time of the model. $WCET$ is the sum of all communication and computation times and is used later on in the model when we must force certain

constraints to be non-binding.

$$WCET = \sum c(V_i, V_j) + \sum w(V), \forall V, V_i, V_j \tag{4.1.2}$$

We now begin to introduce the model constraints, starting with the processing time, $Ptime$, for each processor P in Eq. 4.1.4. An example of the following calculations for $comm$, xc and S can be seen in Fig. A.2 of Appendix A.The communication cost $comm$ is the product of the communication time c and binary decision xc. This results in a communication cost of 0 when V_i and V_j are in the same processor.

$$comm(V_j) = (\sum c(V_i, V_j) * xc(V_i, V_j))\forall V_i), \forall V_j \tag{4.1.3}$$

$$Ptime(P) + WCET * X(V, P) \geq w(V) + S(V) + comm(V), \forall V, P \tag{4.1.4}$$

The models in this thesis do not allow task duplication. Eq. 4.1.5 is used to ensure that each task is assigned to exactly 1 processor. When task duplication is eventually included in SHIRA's design (to reduce communication costs) then this constraint can be removed.

$$(\sum X(V, P) = 1\forall V)\forall P \tag{4.1.5}$$

In each processor, the total time the assigned tasks can execute for must be less than the maximum cycle time (the makespan). Put another way, the makespan is equal to the time taken by the processor executing the longest piece of sequential code. More accurately, it is greater than or equal to the longest execution time of a processor, but the objective function minimizes the makespan, putting the makespan exactly on the constraint and equal to the longest execution time. The constraint for this relationship is presented in Eq. 4.1.6.

$$(makespan \geq Ptime(P))\forall P \tag{4.1.6}$$

We must choose an ordering of all tasks in the system, and for that reason the binary variable $xc(V_i, V_j)$ is set to 1 when two tasks A and B are in different processors, and

otherwise is 0. Also, ID takes on a large number when A and B are the same task in order to prevent an ordering relationship between a task and itself. In the following constraints ID is an identity matrix with the same dimensions as T.

$$(X(V_i, P) - X(V_j, P) \leq xc(V_i, V_j) + WCET * ID(V_i, V_j), \forall V_i, V_j)\forall P \tag{4.1.7}$$

$$(X(V_j, P) - X(V_i, P) \leq xc(V_i, V_j) + WCET * ID(V_i, V_j), \forall V_i, V_j)\forall P \tag{4.1.8}$$

$$(X(V_i, P) + X(V_j, P) + WCET * ID(V_i, V_j) \geq xc(V_i, V_j), \forall V_i, V_j)\forall P \tag{4.1.9}$$

$$(2 - X(V_i, P) - X(V_j, P) + WCET * ID(V_i, V_j) \geq xc(V_i, V_j), \forall V_i, V_j)\forall P \tag{4.1.10}$$

Eq. 4.1.7 through 4.1.10 compute which of two different, independent tasks on the same processor will be scheduled first, and result in the truth table shown in Table 4.1 for the variable xc. We can see that these constraints result in an exclusive-or operation on the processor assignments of any two tasks V_i and V_j.

Table 4.1: Truth table for xc given the constraints 4.1.7 through 4.1.10

$X(V_i, P)$	$X(V_j, P)$	$xc(V_i, V_j)$
0	0	0
0	1	1
1	0	1
1	1	0

Looking at the data dependency in the CDFG of a program, some tasks are known to be dependent on the data from other tasks. All successor tasks must begin after their predecessor task(s), and therefore we define a constraint for each successor with respect to its predecessors as shown in Eq. 4.1.11. Because the tasks with no successor will have no upper bound, we minimize the start times of the tasks in the

objective function defined in Eq. 4.3.1. This brings the execution times of tasks without successors to their minimum as required.

$$S(V_j) \geq S(V_i) + w(V_i) + comm(V_j) \forall V_j \rightarrow V_i \tag{4.1.11}$$

We must also define the precedence among non-dependent tasks so that the model does not allow them to execute concurrently on the same processor, and this is achieved using the constraints of Eq. 4.1.12 through 4.1.16. An example of the following calculations for R, T, xd and $xdPossible$ can be seen in Fig. A.3 of Appendix A. Eq. 4.1.16 ensures that given two tasks, one task is first and the other is second if they are assigned to the same processor. It creates a situation where $xd(V_i, V_j) = 1$ or $xd(V_j, V_i) = 1$, but never both at the same time. Implicit in Eq. 4.1.12 is 4.1.13 but it is stated for clarity (the values of i and j in T are reversed between the equations). In the case of independent tasks, Eq. 4.1.12 through 4.1.15 force $xdPossible$ to 1; when dependent, they force $xdPossible$ to 0.

$$xdPossible(V_i, V_j) \leq 1 - T(V_i, V_j), \forall V_i, V_j \tag{4.1.12}$$

$$xdPossible(V_i, V_j) \leq 1 - T(V_j, V_i), \forall V_i, V_j \tag{4.1.13}$$

$$xdPossible(V_i, V_j) \leq 2 - T(V_i, V_j) - T(V_j, V_i), \forall V_i, V_j \tag{4.1.14}$$

$$xdPossible(V_i, V_j) \geq 1 - T(V_i, V_j) - T(V_j, V_i), \forall V_i, V_j \tag{4.1.15}$$

$$xdPossible(V_i, V_j) = xd(V_i, V_j) + xd(V_j, V_i), \forall V_i, V_j \tag{4.1.16}$$

We clarify the implications of the constraints of Eq. 4.1.12 through 4.1.15 in the truth table shown in Table 4.2. We see that the constraints perform a logical-nor operation on $xdPossible$. And so in Eq. 4.1.16 when $xdPossible(V_i, V_j)$ is 0, both $xd(V_i, V_j)$ and $xd(V_j, V_i)$ must be 0. On the other hand, when $xdPossible(V_i, V_j)$ is 1, then either $xd(V_i, V_j)$ or $xd(V_j, V_i)$ must be 1, and the other will be 0.

Tasks that reach no tasks in the transitive closure, and are reached by no tasks in the transitive closure are called independent tasks, as they have no dependent tasks

Table 4.2: Truth table for $xdPossible(V_i, V_j)$ given the constraints 4.1.12 through 4.1.15

$T(V_i, V_j$	$T(V_j, V_i)$	$xdPossible(V_i, V_j)$	$xdPossible(V_j, V_i)$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

and are not dependent on any tasks. These tasks can execute in any order. This causes an issue for the model, as either of two tasks can be executed first. The model must be constrained so as to determine an order between independent tasks.

We now set constraints which order all independent tasks in the model. We find each pair of tasks that are independent using the data from the transitive closure array T, and the result is stored in the variable R as shown in Eq. 4.1.17. Any pair of tasks V_i, V_j where $T(V_i, V_j) = 0$ and $T(V_j, V_i) = 0$ are independent as long as $V_i \neq V_j$.

$$R(V_i, V_j) = 1 - (1 - T(V_i, V_j)) * (1 - T(V_j, V_i)) \forall V_i, V_j \tag{4.1.17}$$

For each pair of independent tasks we assure that they do not execute at the same time if they are in the same processor by using the constraints of Eq. 4.1.18. As one will execute before the other, one task will be named the predecessor and the other the successor. Put another way, for two independent tasks the variable xd loosens the constraint of the "predecessor" task which would have forced it to execute before its "successor," but it does not loosen the constraint on the "successor" task with regards to the "predecessor." Furthermore, xc loosens the precedence constraints when the independent tasks are in different processors because their ordering no longer matters.

The R variable is used to unbind this constraint for tasks that are not independent.

$$\begin{aligned} S(V_i) + WCET * (xd(V_i, V_j) + R(V_i, V_j) + xc(V_i, V_j)) \\ \geq S(V_j) + w(V_j) + comm(V_j) \forall V_i, V_j \end{aligned} \tag{4.1.18}$$

The results from the model assuming 2 processors is the set of start times [0,3,6,9,0,2,5,10], task-to-processor assignments [1,1,1,1,2,2,2,2], and makespan 16. The starting point for the MPS model was [20], and our contribution was to formulate a BLP to solve the same type of problem.

4.2 Design of the IMP algorithm

The MPS model from the previous section can schedule tasks into homogeneous processors, but to configure those processors to best reduce the makespan of the scheduled code, several constraints were added from the Instruction-Set Extension (ISE) domain [25]. Specifically, the knapsack problem constraints and data from the instruction selection problem were added to MPS as seen in Algorithm 2.

Algorithm 2 IMP_TASK_SCHEDULE_PASS

1: Find ISEs in program
2: **for** ($basicBlock \in blocks$) **do**
3: Select ISEs in program to modify $w(basicBlock)$
4: **end for**
5: Create Transitive Closure T and Identity Matrix ID and write them into model IMP
6: **for** ($basicBlock \in blocks$) **do**
7: Find dependencies of $basicBlock$
8: Associate a value $comm$ to each dependency
9: Write $w(basicBlock)$ and dependencies into model IMP
10: **end for**
11: $RETURNmodel(IMP)$

This ISE addition was performed before the creation of the instruction list to be scheduled, as creating the ISEs after instruction scheduling had occurred could result in spill code, or in wasted cycles. This expanded model is called the Instruction set

extended Multiprocessor scheduling algorithm (IMP). Algorithm 2, modified from Algorithm 1 in Section 4.1 shows the addition of ISE creation and selection pass. The ISE loop is run before any scheduling takes place (lines 2-4) and all of the best ISEs are selected. This shortens the value w in the basic blocks. The scheduling pass runs as it did in the MPS algorithm.

We begin by presenting the data added to the model and then proceed to explain the new variables along with the added and changed constraints. When the scheduling model is called we assume that the candidate instruction set extensions have been enumerated using the instruction enumeration algorithm from [25] with an I/O constraint of (4,4) so as to generate more ISE candidates (as limiting the I/O to (2,1) or (2,2) would reduce the number of viable ISEs found). TU for Times Used, is the number of times a node is used by (i.e. implemented in) any ISE. The $INC(ISE, TASK)$ variable is a binary decision variable which specifies when an ISE should be included in a specific task.

$$TU(Node) \geq \sum INC(ISE_i, V) \forall V, ISE_i \rightarrow V, Node \rightarrow ISE_i \tag{4.2.1}$$

As a node can only be included in a single ISE, a constraint is needed to prevent the node from being used in several ISEs (Eq. 4.2.2).

$$TU(Node) \leq 1 \forall Node \tag{4.2.2}$$

ISEs are represented in the model as having associations to any instructions which other ISEs may also want to use. Therefore we are able to perform conflict resolution between mutually exclusive ISEs at the same time as the ISE selection. Each conflict-prone instruction in the model is represented with a name (k_i), and the list of names is included as data in the model. Each conflicting use of a node by an ISE is marked (e.g. $ISE_USES = [ISE_7 \rightarrow k_3, ISE_8 \rightarrow k_3]$). Equations 4.2.1 and 4.2.2 are implemented by the work from [25] which is used in step 3 of Algorithm 2.

ISEs themselves are represented with the labels ISE_1 to ISE_N, and the hardware cost (HC) of implementing each ISE is represented in the model by constants. Equivalent ISEs are marked as being part of an equivalence class:
$EQUIV = [ISE_3 \rightarrow L_1, ISE_4 \rightarrow L_1]$. $EQIV_HC$ defines the cost of implementing any ISE which is part of an equivalence class. The saved time resulting from each use of an ISE is represented by clock cycles in the array SC. The available chip area for implementing ISEs in the design is represented by a knapsack capacity constant (e.g. $CAP = 30$).

$CU(PROCESSOR, CLASS)$ is used to calculate when a class of equivalent ISEs has been used already in a given processor, and therefore all subsequent uses of that class do not require additional hardware (Eq. 4.2.3).

$$\begin{aligned} CU(P, Class) \geq \\ \sum((INC(ISE_i, V) * (1 - UL(ISE_j))) - (1 - X(V, P))) \\ \forall ISE_i \in Class, \forall ISE_j \rightarrow V, \forall P \end{aligned} \tag{4.2.3}$$

$ST(TASK)$, calculated with the constraint of Eq. 4.2.4, is a measure of the time actually saved by a task due to the addition of ISEs to the processor that the task was assigned to. Eq. 4.2.5, 4.2.6 and 4.2.7 are modified versions of Eq. 4.1.4, 4.1.11, and 4.1.18 respectively.

$$ST(V) = \sum(SC(ISE_i) * INC(ISE_i, V)) \forall ISE_i \rightarrow V \forall V \tag{4.2.4}$$

In Eq. 4.2.5 $Ptime$ represents the processing time for each processor P as it did for the MPS algorithm. As with the MPS algorithm, all successor tasks must begin after their predecessor task(s), and is constrained by Eq. 4.2.6. To keep independent tasks from executing at the same time in the same processor we constrain the model with Eq. 4.2.7.

$$Ptime(P) + X(V,P) * WCET \geq S(V) + w(V) + comm(V) - ST(V) \quad (4.2.5)$$

$$S(V_i) \geq S(V_j) + w(V_j) + comm(V_j) - ST(V_j) \forall V_j \to V_i \quad (4.2.6)$$

$$\begin{aligned} S(V_i) + WCET * (xd(V_i, V_j) + R(V_i, V_j) + xc(V_i, V_j)) \\ \geq S(V_j) + w(V_j) + comm(V_j) - ST(V_j) \forall V_i, V_j \end{aligned} \quad (4.2.7)$$

For each ISE, a constant UL is 1 when the ISE is not equivalent to other ISEs, and 0 when it is part of any equivalence class.

$$ULC + \sum(LIC(P)) \leq CAP, \forall P \quad (4.2.8)$$

As shown in Eq. 4.2.8, the cost of implementing equivalent ISEs and non-equivalent ISEs is computed separately using the cost of unlisted ISEs ULC (Eq. 4.2.9) and the cost for listed ISEs in each processor $LIC(PROCESSOR)$ (Eq. 4.2.10).

$$ULC = \sum INC(ISE_i, V) * HC(ISE_i) * UL(ISE_i) \forall ISE_i \to V \quad (4.2.9)$$

$$LIC(P) = \sum CU(P, Class) * EQIV_HC(Class) \forall Class \quad (4.2.10)$$

In the case when two equivalent ISEs are implemented in different processors, they both contribute to the hardware size. However, when two equivalent ISEs are implemented in the same processor, then the hardware cost of implementation is only the cost of one instance of the ISE. Research into implementing a candidate ISE in a single processor (which would affect the scheduling model) is not within the scope of this work.

4.3 Design of the ITS Algorithm

The primary problem model for Instruction and Task Scheduling algorithm (ITS) schedules instructions in a basic block to a set number of functional units (ALUs) in a processor. Our assumptions are that the primary model does not see the other basic blocks, nor their communication. It is also assumed that all communication between basic blocks occurs at the beginning and end of each block. After finding a valid assignment of instructions to functional units given the constraints of computation time and dependencies, the primary problem hands this list of optimized basic blocks to the secondary problem model to have the basic blocks assigned to processors.

The secondary model receives the set of tasks (where each task is a basic block) from the primary problem. The secondary model then schedules the tasks to each processor in the system, given the constraints of computation time, communication time, and dependencies. The high level algorithm for our approach is presented in Algorithm 3. We first discover the execution time of each basic block (w) on a single configurable VLIW processor. We then find the best case schedule of basic blocks across multiple configurable VLIW processors in a symmetric multiprocessor system.

Algorithm 3 High Level Algorithm

1: Generate list of basic blocks *blocks* from CDFG
2: **for** ($basicBlock \in blocks$) **do**
3: $w(basicBlock) = ITS_INSN_SCHEDULE_PASS(basicBlock)$
4: **end for**
5: $makespan = ITS_TASK_SCHEDULE_PASS(blocks, w)$

4.3.1 Primary Problem

For each basic block of application source code, Algorithm 4 is used to discover an assignment of instructions to homogeneous functional units of a VLIW processor. The objective is to minimize the schedule length.

Algorithm 4 ITS_INSN_SCHEDULE_PASS

1: Create Transitive Closure T and Identity Matrix ID and write them into primary model
2: **for** ($instruction \in basicBlock$) **do**
3: Find execution time of $instruction$
4: Find dependencies of $instruction$
5: Write execution time and dependencies into primary model
6: **end for**
7: $w(basicBlock) = primarymodel$
8: $RETURN w(basicBlock)$

4.3.2 Secondary Problem

We find an assignment of tasks to processors in Algorithm 5 using the instruction-scheduled tasks received from the models of the primary problem. The objective is again to minimize the schedule length, this time using tasks instead of instructions.

Algorithm 5 ITS_TASK_SCHEDULE_PASS

1: Create Transitive Closure T and Identity Matrix ID and write them into secondary model
2: **for** ($basicBlock \in blocks$) **do**
3: Find dependencies of $basicBlock$
4: Write $w(basicBlock)$ and dependencies into secondary model
5: **end for**
6: $RETURN secondarymodel$

4.3.3 Model Definition

For the Primary and Secondary models of ITS we use variations of the same constants and variables. The objective of each model is the same, to minimize the schedule length (also called the makespan) as shown in Eq. 4.3.1. For the Primary model this objective translates as the minimum schedule length of a sequence of instructions inside a VLIW processor. Whereas for the Secondary model this translates as the minimum schedule length when basic blocks are assigned into a multiprocessor system.

$$Objective = MIN(makespan) \tag{4.3.1}$$

The constant $WCET$ represents the worst case execution time as shown in Eq. 4.3.2. It is the sum of all communication and computation times. T is the transitive closure of a graph. For the primary model the graph is the dataflow graph (DFG), whereas for the secondary model the graph is the CDFG. ID is an identity matrix with the same size as T.

$$WCET = \sum c(V_i, V_j) + \sum w(V), \forall V, V_i, V_j \tag{4.3.2}$$

Having presented the objective of the model, we now turn our attention to the constraints. As we can see in Eq. 4.3.3 the processing time for each processor P is equal to the latest start time S of a task assigned to P plus the execution time of the task, plus the delay in the task due to communication. Whereas c is the communication delay between two tasks (V_i, V_j) when they are assigned to different processors, *comm* captures the choice to communicate that results from a particular task schedule. The communication penalty is therefore *comm* as defined in Eq. 4.3.10, and for the primary model *comm* is always 0.

$$\begin{aligned} Ptime(P) + WCET * X(V, P) \geq \\ w(V) + S(V) + comm(V), \forall V, P \end{aligned} \tag{4.3.3}$$

Eq. 4.3.4 ensures that each task is assigned to only one processor, and that each instruction is assigned to only one ALU in a processor. If task duplication for multiprocessors is allowed (to reduce communication costs) then this constraint can be removed in the secondary model.

$$(\sum X(V, P) = 1 \forall V) \forall P \tag{4.3.4}$$

The makespan for the secondary model is the longest execution time of the processors in the system, and for the primary model it is the critical path length. The objective function minimizes the makespan in Eq. 4.3.5 and therefore it will be exactly on the constraint and equal to the longest execution time.

$$(makespan \geq Ptime(P)) \forall P \tag{4.3.5}$$

In the secondary model, the communication between all processors in the system must be defined in order to find a valid schedule for the multiprocessor. Therefore the binary variable $xc(V_i, V_j)$ is set to 1 when two tasks A and B are in different processors, and otherwise is 0. The four constraints of Eq. 4.3.6 to 4.3.9 are only included in the secondary model, and these constraints perform an exclusive-or operation on two tasks and only allows xc to be 1 when the tasks are in different processors.

$$(X(V_i, P) - X(V_j, P) \leq xc(V_i, V_j) + WCET * ID(V_i, V_j), \forall V_i, V_j) \forall P \tag{4.3.6}$$

$$(X(V_j, P) - X(V_i, P) \leq xc(V_i, V_j) + WCET * ID(V_i, V_j), \forall V_i, V_j) \forall P \tag{4.3.7}$$

$$(X(V_i, P) + X(V_j, P) + WCET * ID(V_i, V_j) \geq xc(V_i, V_j), \forall V_i, V_j) \forall P \tag{4.3.8}$$

$$(2 - X(V_i, P) - X(V_j, P) + WCET * ID(V_i, V_j) \geq xc(V_i, V_j), \forall V_i, V_j) \forall P \tag{4.3.9}$$

The constraint in Eq. 4.3.10 calculates the communication cost between two tasks. For the primary model we set *comm* to 0 for all instructions.

$$(comm(V_j) = (\sum c(V_i, V_j) * xc(V_i, V_j)) \forall V_i), \forall V_j \tag{4.3.10}$$

Task/instruction dependencies are defined in a constraint for each successor with respect to each of its predecessors as shown in Eq. 4.3.11. $S(i)$ is the start time of task/instruction i.

$$S(V_i) \geq S(V_j) + w(V_j) + comm(V_j) \forall V_j \rightarrow V_i \tag{4.3.11}$$

Precedence among non-dependent tasks is achieved using the constraints of Eq. 4.3.12 through 4.3.16. Eq. 4.3.16 ensures that given two tasks/instructions they do no execute concurrently in the same processor/pipeline. It creates a situation where

$xd(V_i, V_j) = 1$ or $xd(V_j, V_i) = 1$, but never both at the same time. These constraints perform a logical-nor operation on $xdPossible$, such that neither task/instruction is dependent for $xdPossible$ to be 1.

We find each pair of tasks that are independent using the data from the transitive closure array T, and the result is stored in the variable R as shown in Eq. 4.3.17. Any pair of tasks V_i, V_j where $T(V_i, V_j) = 0$ and $T(V_j, V_i) = 0$ are independent as long as $V_i \neq V_j$.

$$xdPossible(V_i, V_j) \leq 1 - T(V_i, V_j), \forall V_i, V_j \tag{4.3.12}$$

$$xdPossible(V_i, V_j) \leq 1 - T(V_j, V_i), \forall V_i, V_j \tag{4.3.13}$$

$$xdPossible(V_i, V_j) \leq 2 - T(V_i, V_j) - T(V_j, V_i), \forall V_i, V_j \tag{4.3.14}$$

$$xdPossible(V_i, V_j) \geq 1 - T(V_i, V_j) - T(V_j, V_i), \forall V_i, V_j \tag{4.3.15}$$

$$xdPossible(V_i, V_j) = xd(V_i, V_j) + xd(V_j, V_i), \forall V_i, V_j \tag{4.3.16}$$

$$R(V_i, V_j) = 1 - (1 - T(V_i, V_j)) * (1 - T(V_j, V_i)) \forall V_i, V_j \tag{4.3.17}$$

For two independent tasks the variable xd forces the task precedence indicated by $xdPossible$, as shown in Eq. 4.3.18. The R variable is used to unbind this constraint for tasks that are not independent.

$$\begin{aligned} S(V_i) + WCET * (xd(V_i, V_j) + R(V_i, V_j) + xc(V_i, V_j)) \\ \geq S(V_j) + w(V_j) + comm(V_j) \forall V_i, V_j startTime(V_i) + WCET * xd(V_i, V_j) \geq \\ startTime(V_j) + w(V_j) + comm(V_j) \\ \forall V_i independent from V_j \end{aligned} \tag{4.3.18}$$

4.4 Development of the CSA Algorithm

The task of combining the IMP and ITS algorithms to create the Combined Scheduling Algorithm (CSA) involves application of the ISE portion of the IMP algorithm to the instruction level (primary) model used in the ITS algorithm. The generation of ISEs themselves is unchanged, and still uses the knapsack problem to select which ISEs will best fit the given program. The difference is that the output from COINS of the given program will show the ISEs at the instruction level, as opposed to showing only the reduced runtime at the task level as is done in the IMP algorithm. An example of the modified algorithms can be seen below. At the top level, Algorithm 6 remains essentially unchanged.

Algorithm 6 High Level Algorithm

1: Generate list of basic blocks *blocks* from CDFG
2: **for** $(basicBlock \in blocks)$ **do**
3: $w(basicBlock) = CSA_INSN_SCHEDULE_PASS(basicBlock)$
4: **end for**
5: $makespan = CSA_TASK_SCHEDULE_PASS(blocks, w)$

For each basic block, Algorithm 7 is used first to find and select ISEs that will modify both the list of w for the instructions, as well as the size and shape of the Transitive Closure T before the instruction level model is written.

Algorithm 7 CSA_INSN_SCHEDULE_PASS

1: Find ISEs in *basicBlock*
2: Select ISEs in program to modify $w(basicBlock)$ and change the size of T
3: Create Transitive Closure T and Identity Matrix ID and write them into the primary model
4: **for** $(instruction \in basicBlock)$ **do**
5: Find execution time of *instruction*
6: Find dependencies of *instruction*
7: Write execution time and dependencies into primary model
8: **end for**
9: $w(basicBlock) = primarymodel$
10: $RETURNw(basicBlock)$

After the primary model is run, a result of a task level w is found. This new w is to be used in Algorithm 8. The objective is again to minimize the schedule length, this time using tasks instead of instructions.

Algorithm 8 CSA_TASK_SCHEDULE_PASS

1: Create Transitive Closure T and Identity Matrix ID and write them into secondary model
2: **for** $(basicBlock \in blocks)$ **do**
3: Find dependencies of $basicBlock$
4: Write $w(basicBlock)$ found in primary model and dependencies into secondary model
5: **end for**
6: $RETURN secondarymodel$

During the development of CSA, several additional constraints were discovered to further prune the search space. These constraints were directly applied to the *makespan* variable, and were retroactively applied to all of the previous algorithms before testing began.

$$2 * makespan \geq \sum W \tag{4.4.1}$$

The optimal arrangement of instructions or tasks to processing lanes or processors can never achieve a speedup of more than two, without the addition of ISEs (and in the case of super-linear speedup). This fact was translated into a form usable by the model in Eq. 4.4.1. The makespan can never be less than the sum of the sequential runtime of all instructions/tasks in the model. To eliminate the division, the equation was formated to state that twice the makespan is greater than or equal to the sum of all instructions/tasks.

In development of this constraint, another constraint on the system was needed. In cases where there are few nodes in the model, a single large node can be greater than the total of all other nodes together. This means that the difference of this large w and the value of half the sum of all w in the model will never be used, and should be pruned from the search space. Eq. 4.4.2 states that the makespan must be greater

than or equal to the largest w in the model.

$$makespan \geq W(V), \forall V \tag{4.4.2}$$

During initial testing, it was also realized that the model required additional pruning to the upper bound of the makespan to decrease the time to find an initial solution. In order to prevent the slack in the model from allowing schedules of impractical length, the upper bound of the system was given as the sequential runtime of the program.

This output conforms to the requirements necessary to run Algorithm 6, and assign the extended instructions to the functional units in the VLIW processor. It is assumed that each functional unit will contain the necessary extensions to run the ISEs. Further development will include the ability to constrain the problem to allow for differing functional units, and differing processors. Allowances for differing functional units and processors will potentially decrease the size of the final design (in terms of chip area), but will increase the complexity of the solver, as all permutations for the multiprocessor, multi-functional unit system will need to be attempted.

4.5 Summary

The above four sections describe the creation of the four algorithms used in this thesis. The simplest, MPS, is created to schedule task level data-flow graphs onto processors. MPS is then modified to create two new algorithms: IMP which has ISEs added into the tasks to shorten them, and ITS, which exposes the instruction level models and allows them to be scheduled to shorten the tasks. Finally, the IMP and ITS algorithms are combined to create CSA, which encompasses ISEs visible at the instruction level, instruction level scheduling, and task level scheduling.

Chapter 5

Results

This chapter begins by detailing the creation of the test programs used to gather results from all four algorithms, as well as the benchmark used. It will then proceed to compare the results of our algorithms against those in the literature. Finally it will show more in depth testing with both the randomly generated programs as well as the FFT benchmark.

For the measurements in this section, the term "speedup" is always measured in comparison to the sequential case (all instructions executed on a single functional unit, one after the other, with no delays). When referring to an "efficient" program, this means that the portion of the program that will benefit from parallelization is very high. Finally, when referring to the execution time of the algorithms, this time is measured at compilation. The static code output by the algorithms is already scheduled; no run-time scheduling of the static code will occur.

5.1 Creation of Test Programs

Differences in test platforms and test programs in the literature concerning scheduling [20, 34, 37] make it difficult to compare the results from multiple sources. The

work of [7] uses a simulated environment and a set of randomly generated programs containing a specified number of basic blocks to compare multiple scheduling techniques. This simulated environment, which assumes no cache misses, bus contention or other sources of delay is simple to design, and provides the same platform for all scheduling techniques. Unlike with specified test benches, the randomly generated programs used in [7] allow for in house generation of similar programs. As long as the programs contain a random number of instructions for the specified basic block size, the results of each benchmark are comparable. It was this setup that was used to test the four versions of the scheduling algorithm (MPS, IMP, ITS, and CSA). For the purpose of this chapter, pseudo-random numbers generated by $java.Random$ will be referred to as random numbers.

The original random programs were generated for the work in [26, 19]. To generate the random programs, a generator program was developed in Java in order to write the test programs. The generator writes these test programs in C. This generator would randomly create both integer and floating point variables to be used in the generated program. Variables were assigned names of i_x (for integers) and f_x (for floating point) where the value of x is a number that is incremented to produce a set of variables (such as $i_1 = 25$, $i_2 = 10$, $i_3 = 4$, etc...). Each would then be given a random integer or floating point value. From there, a predetermined set of $if - else$ statements would be created, using a randomly generated comparative operator and the first available variable (randomly selected between integer and floating point). No advanced operations, such as string operations, or matrices were used.

Each if statement would then be filled with a random sized set of randomly generated statements in three address code. Originally, the operations allowed were limited to simple comparisons (such as XOR), and statements were segregated so that only one type of variable was used; for example, only integer in one statement, and only floating point in another.

These programs were created to showcase benefits of ISEs, but were deemed not

general enough to be considered realistic. Casting, arithmetic operations, shifting and other operations were added to the program generator in order for the programs to become less uniform, the results from ISE identification to be more realistic, and to make the programs harder to solve than an average real-life case.

In addition to additional operators, a method for allowing for random amounts of communication was added to the program. As the variables were used in the program, a counter would be incremented to select a new variable (as each variable had a number associated with it). To facilitate communication between basic blocks, a random sized rollback of the counter was used to reuse some of the variables from each basic block. This resulted in a random number of basic blocks having communication. Again, this was done for a more realistic result, as fully disconnected programs, (programs with zero communication) are not realistic.

In addition to generating the programs, the generation code also ensured that each program was viable. This was accomplished by compiling each program as it was generated, and ensuring no errors occurred. If anything prevented the test program from being compiled, it was deleted and the generator created another program of the same size. Upon completion, the generator had created a set of randomly generated test programs range in size from 5 to 100 basic blocks, each containing a random number of operations.

For the FFT benchmark, we generated a 64-point FFT program in C. Chunking of the C program was added in order to split the single basic block (with approximately 8192 nodes) into a set of smaller problems. The program was divided (or chunked) in two separate stages. The first divided the program into its 64 points; while the second divided the calculations for each point into a smaller set of sub-tasks. This second division would make each sub-task's size small enough for the algorithms to process at the instruction level.

The FFT program was completely unrolled in order to create the three address code used for scheduling. Otherwise, a loop would be treated as a single node, leading

to a program with a few gigantic nodes; easy to schedule, but provides poor speedup. In unrolling the loops completely, the compiler will not reuse registers as it would if the loop were simply treated as a single node. This will result in more register-memory interactions. For loops that, due to dependencies, execute in a completely linear fashion it would be simpler to treat them as a single node. This would reduce the amount of register to memory operations. Additionally, the number of registers used would be static, as the loop would operate with the set of registers needed, only allowing access to them once it had completed.

Finding the most efficient method of compiling three address code for parallelization is beyond the scope of this thesis, as that would be based in the higher levels of the compiler, whereas this work focuses on the code once it has been turned into IR after three address code has been created. An example of the current output compared to an arrangement of variables that would be better suited to parallelism is shown in Fig. 5.1, separated by a vertical divider. The example shows two different methods for computing the sum of an array. The variables $VAR[0]...VAR[N]$ are the array to be summed, ANS is the final answer, and the various $TMP[x]$ variables are used as temporary variables.

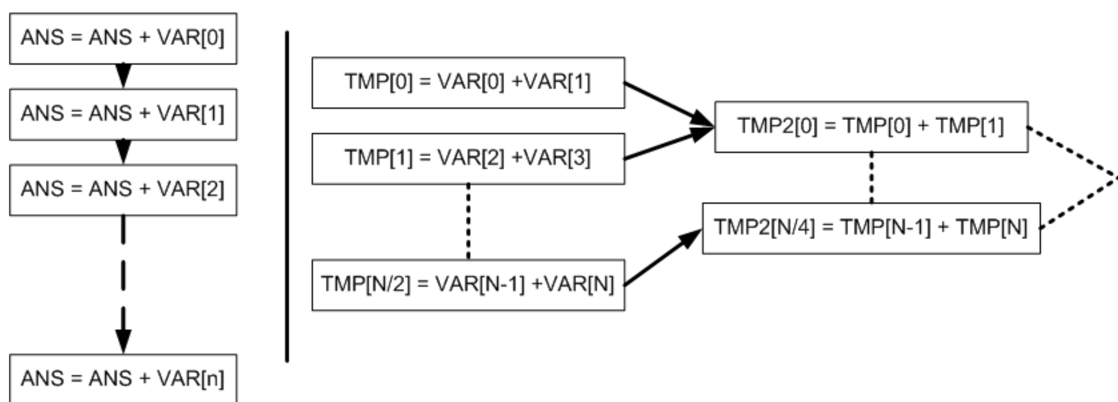

Figure 5.1: Different methods for summing an array in three address code

The first method, shown on the left hand side of the vertical break in Fig. 5.1,

is the method currently output by the compiler when the addition of variables is written in a loop. As it reuses the variable ANS in each equation, it creates a series of dependencies between all of the equations; dependencies that do not necessarily have to exist. When turned into the transitive closure, this set of dependencies creates a matrix where the area above the identity line is completely filled with 1's, leading to a schedule with no speedup.

The second method (on the right hand side) results in the creation of more temporary variables, but is more easily split between processors or FUs as the variables at each stage (such as $TMP[0]...TMP[N/2]$ and $TEMP2[0]...TMP2[N4/]$) have no dependencies with one another.

5.2 Comparison with results in Literature

The results from this section were derived from a set of randomly generated DAGs ranging in size from 5 to 100 tasks, each of which generated a random amount of communication. We compared our results from MPS and IMP [26] to results presented in [7]. The program sizes tested with MPS and IMP were restricted to sizes tested in [7]. The results on the randomly generated programs for the algorithms compared in [7] were generated in that article; this thesis simply compares our work to those answers.

The algorithms used to generate the results in [7] are two list scheduling algorithms as similar to the description in Section 2.2.2 called DeClustering (DC), and a variant of DeClustering using the Largest Processing Time rule (LPTDC). Also, two clustering heuristics similar to the description in Section 2.3 called Load Balancing with Minimized Communications (LBMC), and Preferred Path Selection (PPS) were used. All of the results from the work in [7] and the results of the MPS and IMP algorithms are obtained from testing on a two processor system, with each processor having only one execution lane.

The results from ITS [19] and CSA cannot be directly compared to the work in [7], as the algorithms used in [7] do not use multiple execution lanes for scheduling, and do not exploit instruction level parallelism. A comparison of the speedup results from all four of the algorithms developed in this thesis is presented at the end of this chapter, as they were all run on the same set of programs. This highlights the large difference when the two processor system has two execution lanes per processor, which are exploited by the instruction level parallelism.

As the results presented in [7] gave no indication of the sequential execution time of the programs, nor did they give the overall speedup provided by each algorithm, a method for comparing our results to that of [7] was required. As the programs compared in [7] are shown with their optimal solution to compare to the various heuristics, a calculation based on the optimal solution and the solution found can be made to indicate how close to the optimum the results were. We use the calculation in Eq. 5.2.1 to determine how close the solution found was to the optimal.

$$(ResultantTime - OptTime)/OptTime \qquad (5.2.1)$$

As the $ResultantTime$ approaches the optimal, the number shrinks until it reaches zero, at which point the solution found is the optimal. In cases where our solution could not prove its solution to be optimal, the theoretical optimum of $SequentialTime/2$ was used. Optimality is proven when the algorithm completes its search, and proves that no other arrangement provides a better speedup. In the results of IMP, the addition of ISEs to the mix enables the results of some of the test to perform better than the optimal, as the new ISEs increase the speedup beyond 2 for the 2 processor system.

The following comparisons are with respect to the heuristics presented in [7], not to the pre-solved optimal solutions used for their comparisons. Table 5.1 shows the original optimal and heuristic schedule lengths for randomly generated programs of

sizes 10-32 in increments of 2, while Table 5.2 shows the results of MPS and IMP in relation to the optimal or theoretical optimal. The heuristic methods describes in [7] were: Load Balancing with Minimized Communications (LBMC), regular DeClustering (DC) as well as a Largest Processing Time variant (LPTDC), and Preferred Path Selection (PPS).

Table 5.1: Comparison of the algorithms presented in [7] compared to the optimal solution. The numbers represent the length of the critical path in terms of clock cycles.

# Task Size	Opt	LBMC	DC	LPTDC	PPS
10	207	281	279	283	289
12	246	366	320	356	352
14	271	629	370	368	373
16	326	620	358	428	489
18	428	892	476	732	526
20	378	783	469	521	596
22	488	905	488	605	996
24	618	1276	800	817	961
26	614	1097	687	687	1150
28	671	1194	875	812	1279
30	811	1842	859	1144	1530
32	886	1806	886	1353	1698

Converting the results of the comparison in Table 5.1 into a graph in Fig. 5.2 we see that the DC algorithm is by far the best candidate.

When represented as a graph in Fig. 5.3, MPS is shown to be extremely close to the optimal (in many cases, the result is likely optimal but cannot be proven so

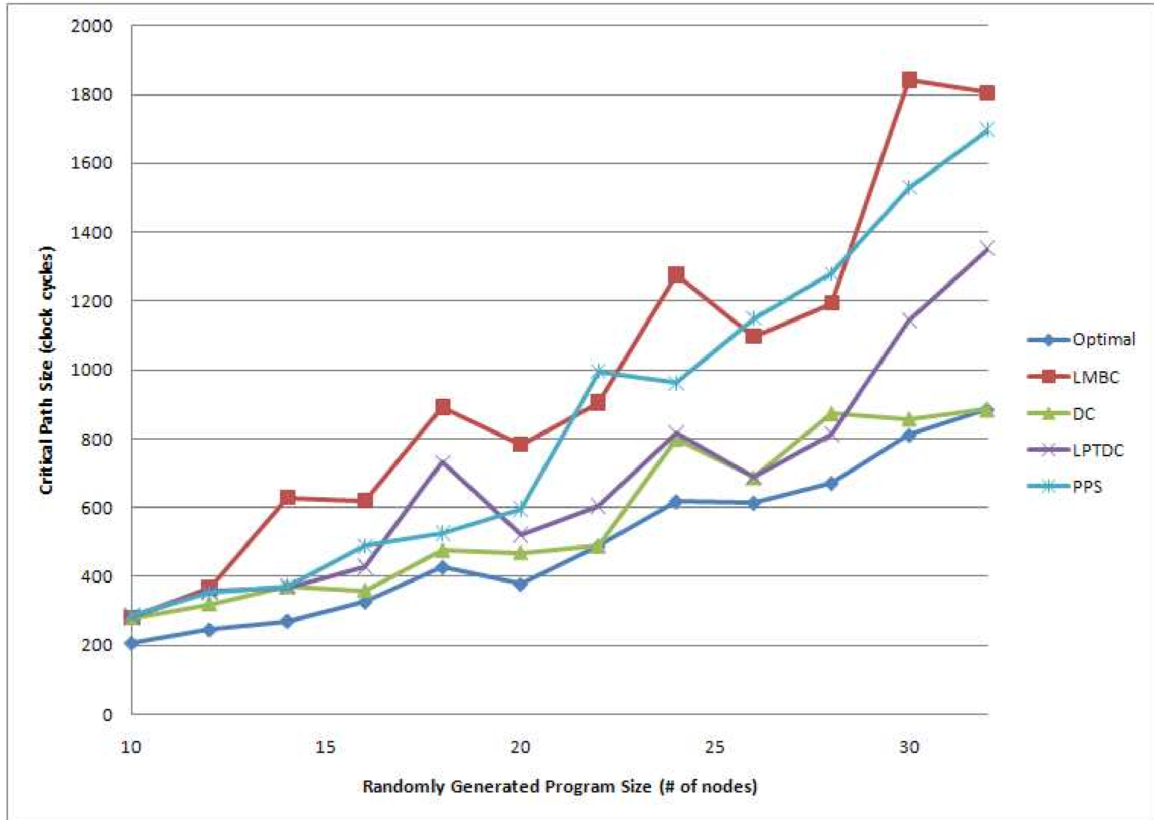

Figure 5.2: Comparison of the results in Table 5.1.

before the timer completes), and IMP shows that the addition if ISEs gives a result better than that of the optimal (without ISEs), in most cases. The difference between the optimal and the result of IMP is speedup gained by the addition of ISEs into the instruction mix.

Performing the calculations in Eq. 5.2.1 on each of the schedule lengths in Tables 5.1 and 5.2 generates the results displayed in Table 5.3 in percentage. As stated earlier, cases where ISEs have decreased the schedule length of IMP to less than the optimal are shown as negatives. The smaller the result in Table 5.3 the shorter the critical path is.

Table 5.2 also shows how the addition if ISEs can make the model easier to solve. By reducing the lengths of certain nodes, the model may become easier to solve. This

Table 5.2: Results of both the MPS and IMP algorithms, as compared to the optimal or theoretical optimal solution, represented in the column "OPT." Results marked with an $*$ were found to be optimal.

# Task Size	Opt	MPS	Speedup	IMP	Speedup
10	595	595*	2.00	555	2.14
12	689	690	2.00	652	2.11
14	769	769*	1.73	624*	2.14
16	889	891	2.00	823	2.16
18	606	619	1.96	558	2.17
20	621	623	1.99	572	2.17
22	453	454	1.99	450	2.01
24	498	510	1.95	481*	2.07
26	1469	1470	2.00	1333	2.20
28	1529	1529	2.00	1397	2.19
30	1481	1486	1.99	1346	2.20
32	1146	1499	1.53	1384	1.66

is the case for the program of size 24. The MPS algorithm fails to prove optimality, while the IMP algorithm finds the optimal solution.

Figure 5.4 shows how close each algorithm gets to the optimal. The optimal case is represented as the line across the X-axis. As can be easily seen, the MPS and IMP algorithms far outperform the heuristics listed in [7].

While the comparison between the heuristics in [7] and both MPS and IMP shows that the ILP based algorithms perform remarkably well for task sizes 10-32, a larger sample is required to find the point at which the ILP begins to fail. As the calculation of each static portion is unrelated to the calculation of its neighbours, the process can

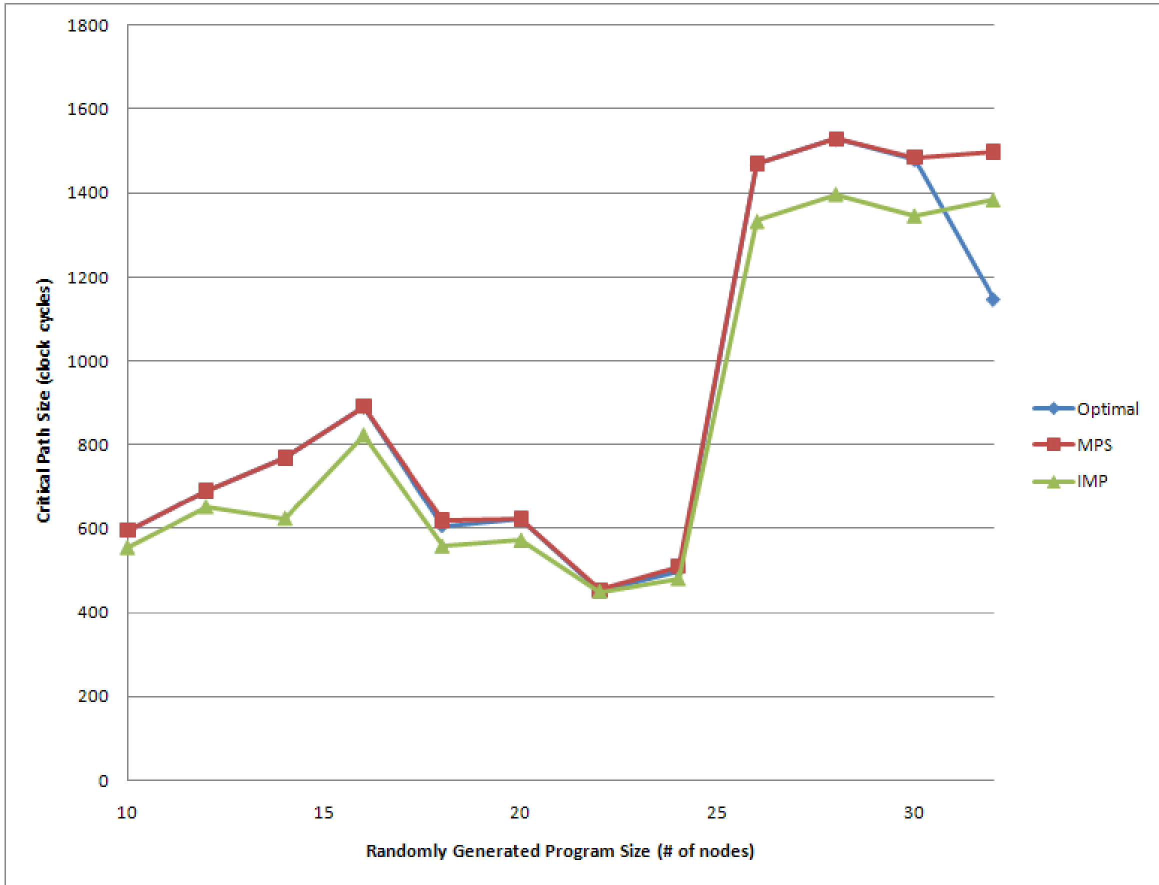

Figure 5.3: Comparison of the results in Table 5.2.

be completely parallelized. Additionally, as each stage (ISE identification, instruction level scheduling and task level scheduling) is completely independent from each other stage, these too can be performed in parallel.

Each stage of the algorithm was given a maximum of 60 minutes in which to find a solution. Therefore, for MPS the worst case execution would be one hour, for IMP and ITS, the worst case execution time would be 120 minutes, and the worst case time for CSA would be 180 minutes. These longer values were chosen as this research aims to make a scheduling algorithm for embedded systems. Requiring a longer compile time in order to achieve the best results is acceptable, as the ASIC or ASIP produced for an embedded system is not changed frequently, if ever, once it

Table 5.3: Normalized comparison of heuristic results from [7] and the task level algorithms from this thesis

# Task Size	LBMC	DC	LPTDC	PPS	MPS	IMP
10	35.75	34.78	36.71	39.61	0.00	-6.72
12	48.78	30.08	44.72	43.09	0.15	-5.37
14	132.10	36.53	35.79	37.64	0.00	-18.86
16	90.18	9.82	31.29	50.00	0.22	-7.42
18	108.41	11.21	71.03	22.90	2.15	-7.92
20	107.14	24.07	37.83	57.67	0.32	-7.89
22	85.45	0.00	23.98	104.10	0.22	-0.66
24	106.47	29.45	32.20	55.50	2.41	-3.41
26	78.66	11.89	11.89	87.30	0.07	-9.26
28	77.94	30.40	21.01	90.61	0.00	-8.63
30	127.13	5.92	41.06	88.66	0.34	-9.12
32	103.84	0.00	52.71	91.65	30.80	20.77

has been finalized.

Figure 5.5 shows a comparison of all four scheduling algorithms, ranging in size from 5 to 100 task level nodes. As the average size of a task is approximately 5 nodes [18], the program sizes were selected to show both the average size, as well as harder to solve cases covering over 90% of task graphs normally found in programs. [17]. The programs were distributed to give as much granularity to the more common program sizes.

The graph in Fig. 5.5 shows that the algorithms are unable to find feasible solutions close to the 60 node mark, twice the size that [20] states as being the point where ILP based methods become useless. For the ITS and CSA algorithms,

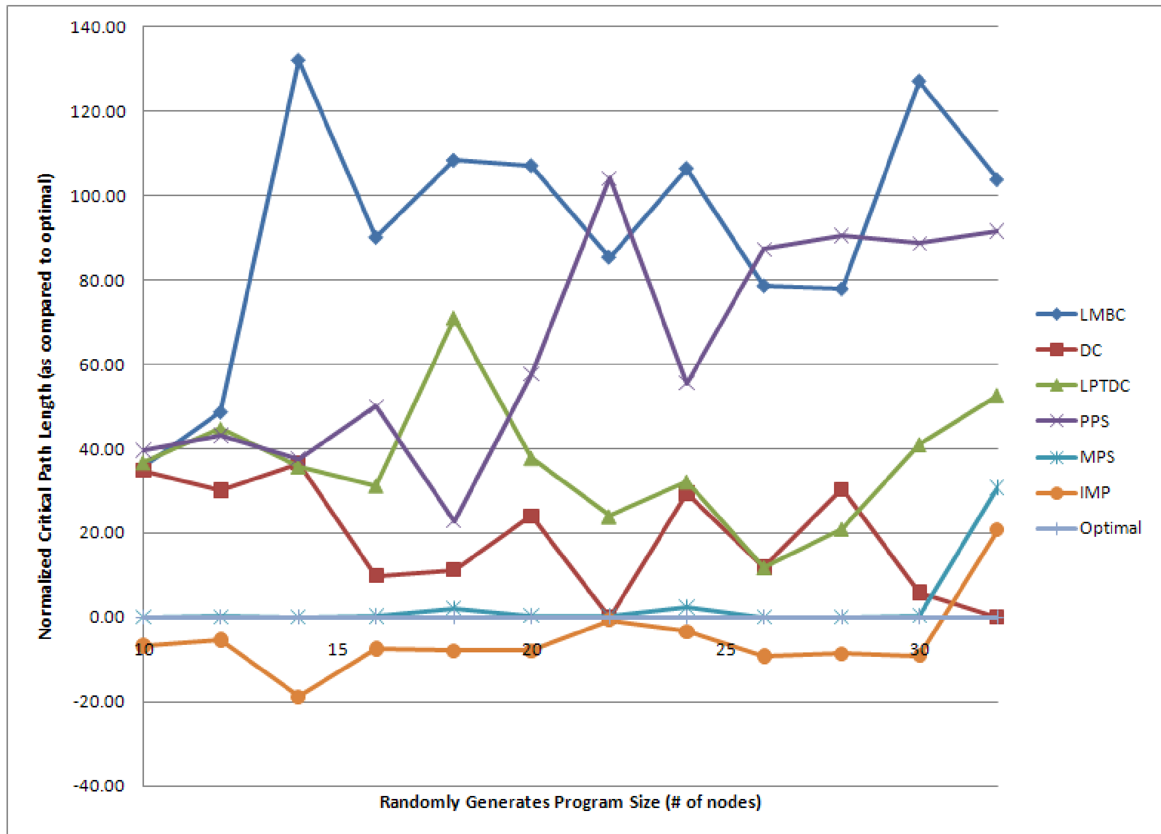

Figure 5.4: Comparison of the results in Table 5.3.

this is comparable to over 400 nodes when taking into account the instruction level scheduling that takes place. This number was found by measuring the size of each basic block in the program containing 60 basic blocks.

With small program sizes, Fig. 5.5 shows a smaller speedup when compared to sizes closer to 20-40 nodes. This is due to the fact that a single large node can be larger than the sum of all other nodes. Even if all other nodes were scheduled in the second processor, the giant node in the first processor would still take longer to execute. This was mentioned in the Section 4.4 when creating new constraints on the makespan.

The results of all four algorithms quickly rise to speedup numbers close to 2 and 4 (the maximum speedup for non ISE results) as the size of large nodes becomes less

of an issue when calculating the schedule. As the model sizes approach the 60 node mark, the speedup begins to drop. This is due to the solver no longer being able to find a feasible solution in the time given.

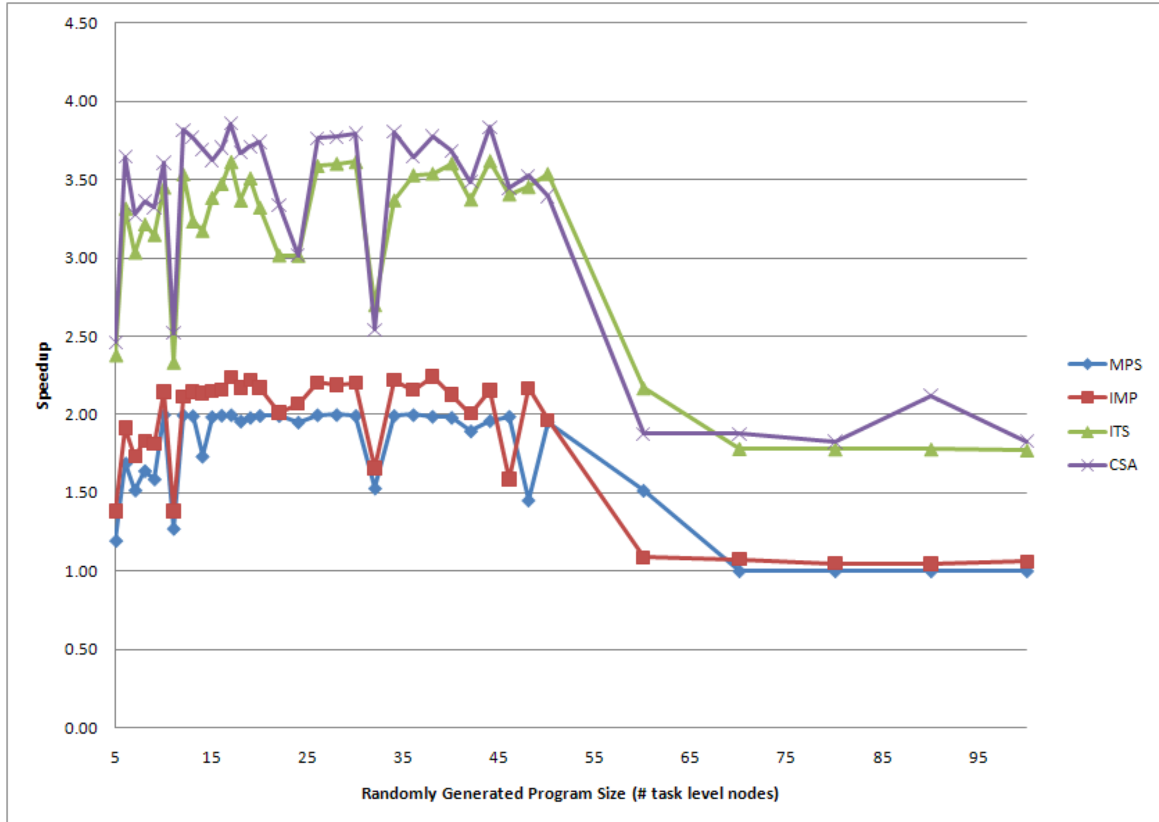

Figure 5.5: Comparisons of speedup for MPS, IMP, ITS and CSA.

Averaging the speedup of each of the four algorithms for values between 5 and 100 nodes shows that all four algorithms perform well, with the instruction level algorithms performing best. The addition of ISEs provides a smaller gain in comparison to the gain due to the scheduling, as can be seen in Fig. 5.3. The ISEs were chosen with a hardware constraint that would limit the system to a small handful of ISEs (preventing the system from simply turning the entire program into a single ISE). Additionally, a random program is harder to find highly beneficial ISEs as the chance for repetition in the code is lower. This was purposefully done to show the results

for a set of programs that would be hard for the algorithms to solve. In the case of programs with higher repetition in code, the results for the IMP and CSA algorithms would be improved, as the ISEs found would be more beneficial. The average results were generated for cases up to 60 nodes, as 60 nodes is the last point where the solver gives a feasible solution within the time constraint.

Table 5.4: Average speedup calculated for MPS, IMP and ITS up to and including 100 nodes. This calculation is with respect to a uniprocessor approach. Note that for ITS and CSA the model assumes two issue slots per processor, whereas all other results are assumed to have one issue slot per processor.

MPS	IMP	ITS	CSA
1.74	1.89	3.10	3.28

5.3 Variation in Model Processing Time

While the values of 60 minutes per subsection of the program were given as the maximum cutoff times for the purposes of showing how high a speedup each algorithm could attain, it should also be noted that due to the constraints placed on the models, many of the programs were able to have solutions found within a few minutes. This is especially true of the smaller sized programs. As the average basic block is approximately 5-5.5 nodes in size [18, 25], the ability to solve the scheduling of small programs optimally within a small time constraint is of interest. A comparison of each of the algorithms at the 5, 15 and 60 minute marks is shown in this section.

As can be seen from Fig. 5.6, the solver finds solutions approaching the optimal within five minutes for models of sizes ranging between 20 and 30 nodes, with some

good solutions up to programs with 40 nodes. Letting the solver run for fifteen minutes extends the set of good solutions to model sizes near 44-46 nodes. At the sixty minute mark, the solver fails to find good solutions for programs with 60 nodes, which is twice that of the values produced by ILP based methods in the work of [20]. It should be noted that while many of the solutions are very close to a 2 times speedup, they are not proven to be optimal, as the solver had not completed.

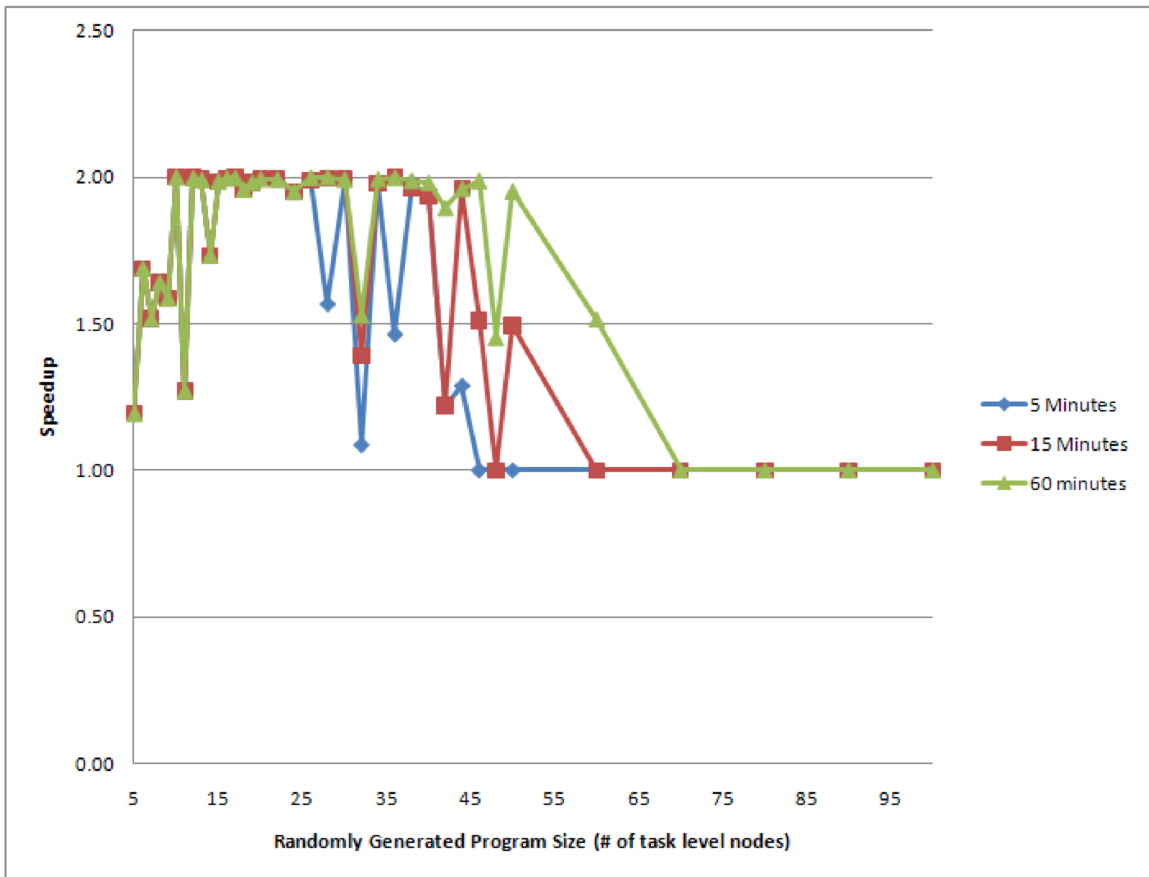

Figure 5.6: Comparison of Speedup Results for MPS with Varying Cutoff Times

Similar to Fig. 5.6, the results in Fig. 5.7 show that the five minute run begins to lose ground around the 30-34 node mark (slightly better than MPS). IMP continues to provide answers up to the 50 node mark, again somewhat better than MPS. For

compiler execution times of 15 minutes, the IMP algorithm provides decent solutions until the 50 node mark as well, performing better than MPS, which begins to fail around the 44-46 node mark. However, at the 60 node mark, the IMP algorithm fails to find a decent solution, whereas the MPs algorithm does find a solution at 60 nodes.

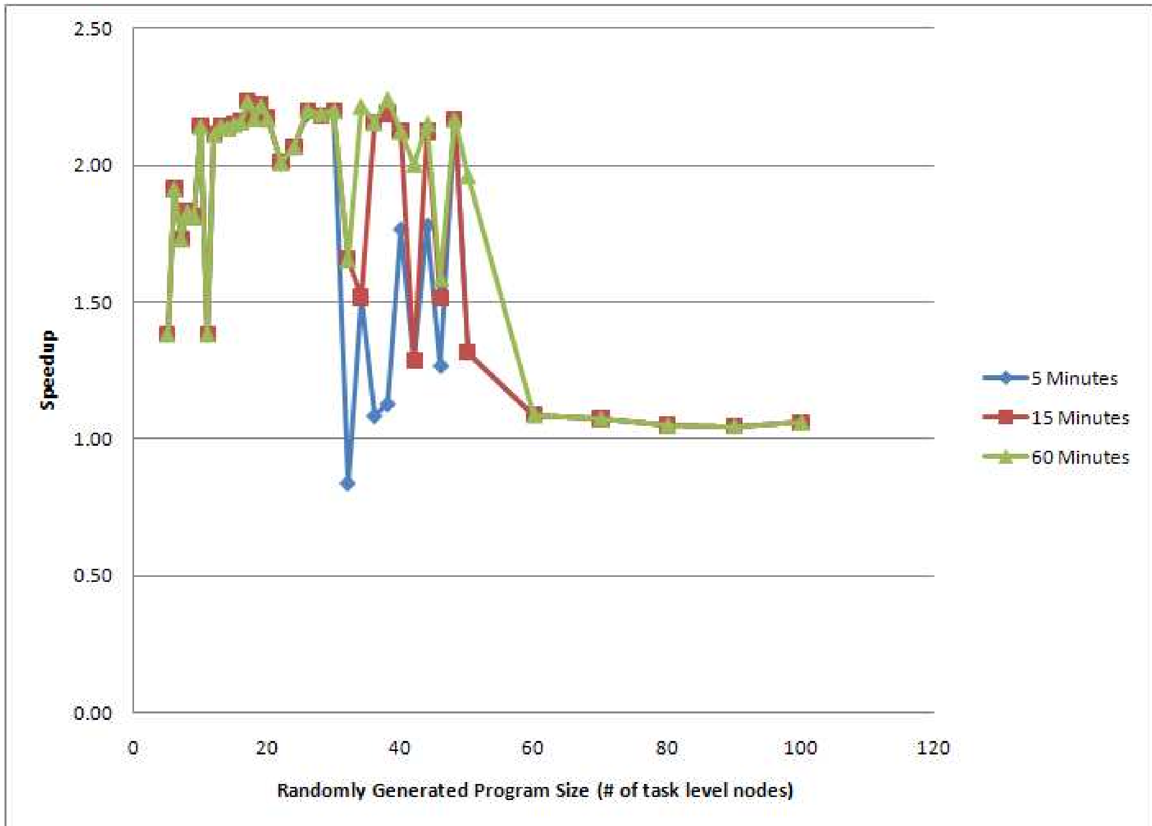

Figure 5.7: Comparison of Speedup Results for IMP with Varying Cutoff Times

The timed comparison of the ITS algorithm can be seen in Fig. 5.8. With the exception of a single failure to find a solution at a problem size of 7 and 8, the 5 minute cutoff run of ITS performs well until about the 38 node mark, with solutions degrading rapidly after that point. The 15 minute run of ITS follows a similar path (with no failure at 7 or 8 nodes), and begins to rapidly degrade in performance at about the 40 node mark. The 60 minute run performs very well until the 60 node

mark.

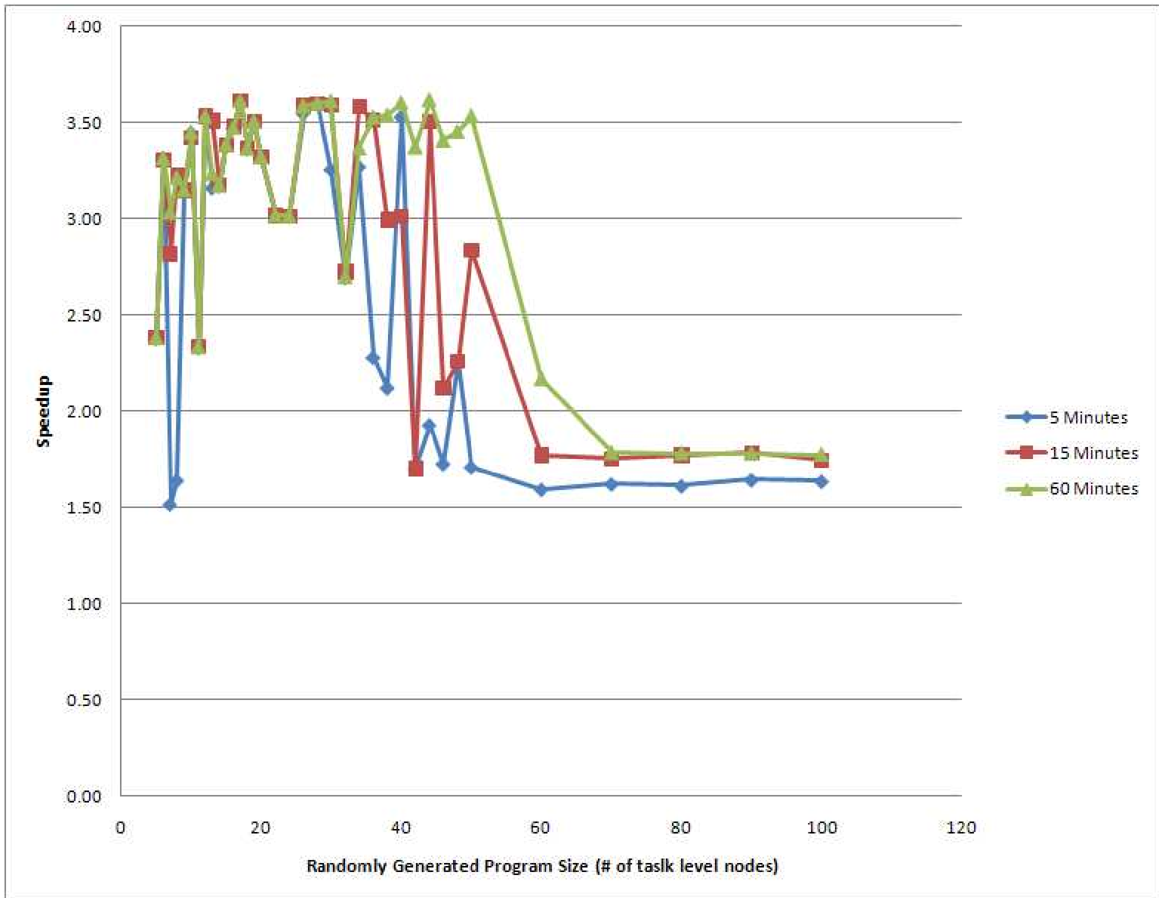

Figure 5.8: Comparison of Speedup Results for ITS with Varying Cutoff Times

Following a similar curve as 5.8, the results of the 5 minute run of the CSA algorithm in Fig. 5.9 show good performance until the 40 node mark, with performance degrading rapidly thereafter. The 15 minute run has three poor results at the 32, 42, and 46 node marks, but otherwise performs as well as the 60 minute run, with both failing to find adequate solutions at the 60 node mark. The 60 minute run can be seen to show a solution (albeit a marginal one) for a program of 90 task level nodes. This is equal to a solution of over 550 nodes at both task and instruction level.

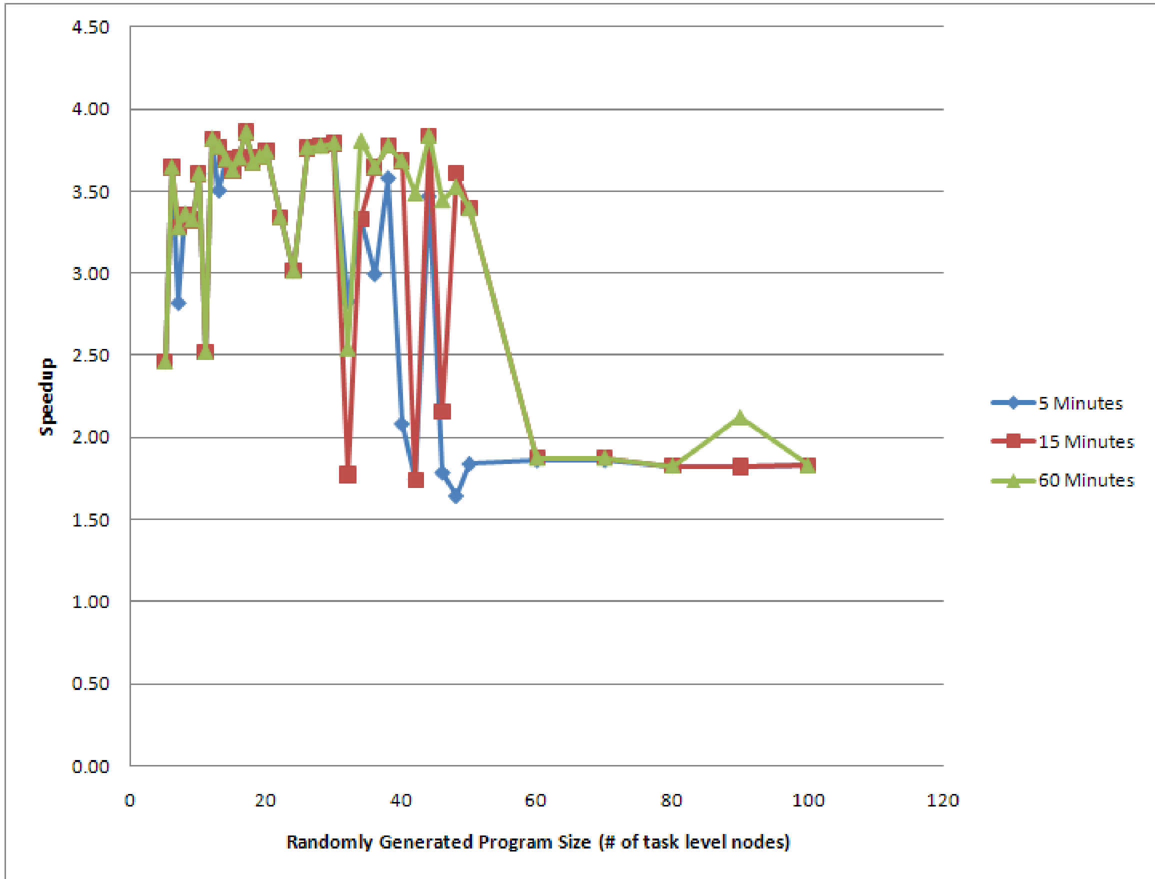

Figure 5.9: Comparison of Speedup Results for CSA with Varying Cutoff Times

5.4 Scheduling of Fast Fourier Transform Benchmark

In this section we discuss the results of the Fast Fourier Transform (FFT) benchmark and compare it to the work performed in [24]. The results presented in [24] were done across multiple functional units for a 64-point FFT. The 64-point FFT was chosen as there was a direct comparison with recent literature. A larger number of points could be chosen, but would require more chunking. Additionally, several hand made ISEs were added to the system, a list of which can be seen in Table 5.5. The authors of

[24] justify their choices of hand made ISEs by stating “that a precision of 16 binary digits is sufficient for almost all tasks in wireless base-band processing, the small set of special instructions ... promises high speed up and high data path utilization.”

Table 5.5: List of hand optimized ISEs used in [24]

op-code	Intrinsic Function
brev	bit reversed index calc.
cadd2	16-bit vector addition
sub2x	16-bit vector subtraction
cmul	complex multiplication
casr	complex arith. shift right
ccon	conjugate complex

As these ISEs were implemented in advance with knowledge that they perform well in the domain of digital signals processing, they were expected to outperform the ISEs generated by SHIRA for scheduling using the four methods presented in this work. The ISEs generated by SHIRA are formulaic, and would not likely find highly specific ISEs (such as the butterfly). A list of the results of the FFT scheduled across 1, 2, and 4 functional units can be seen in Table 5.6.

Table 5.6: Speedup of FFT scheduling with ISEs in [24]

FFT64	1 FU	2 FU	4 FU
Standard	1	1.93	3.65
With ISEs	2.53	4.82	8.84

As can be seen, the results of the four algorithms are quite similar to the random

program results generated by the MPS and ITS algorithms for 2 and 4 functional units back in Fig. 5.5. While this work did not cover implementation of a single functional unit with ISEs, comparison of a single functional unit with ISEs was given as the sequential time for the IMP program (that is, with ISEs added, but without multiprocessor scheduling performed).

Table 5.7 shows the results with each complex portion of the FFT chunked into 8, 16, and 32 pieces. As can be seen in the results, the benefit of scheduling at the instruction level only becomes apparent when the number of task level chunks increases, making each chunk easier to solve at the instruction level.

In the case of pure scheduling, the algorithms perform similarly to the results of [24]. As the ISEs for this thesis were generated automatically, they do not provide as large a benefit as the hand made ones in [24].

Table 5.7: Speedup of FFT scheduling with and without ISEs in this thesis

FFT64		1 FU	2 FU	4 FU
Chunk 8	Standard	1	1.97	1.97
	With ISEs	1.27	2.35	2.38
Chunk 16	Standard	1	1.95	1.92
	With ISEs	1.30	2.59	3.47
Chunk 32	Standard	1	1.70	1.91
	With ISEs	1.29	2.24	4.05

In comparison to the baseline, the best performer of the four models was the CSA algorithm, run on the FFT chunked into 32 pieces. CSA benefited from the instruction level speedup and the ISE speedup

5.5 Summary

In this chapter we covered the generation of the random test programs, and the results of running the four algorithms (MPS, IMP, ITS, and CSA) against both the random programs and the 64-point FFT. We show that all four versions of the algorithm perform in a similar fashion, rapidly rising towards the optimal, and beginning to fail at the 50 node mark. The algorithms begin to fail at the 50 node mark as the number of combinations at programs of that size creates a search space too large for the ILP to find a valid solution to within the time limit. While it may be possible to add constraints to prune the search space dependent on the type of application being scheduled, research into application specific constraints was not within the scope of this research.

When running the algorithms against the FFT, the algorithms performed as expected with the exception of the ITS algorithm. This is likely due to the instruction level scheduling failing to find solutions for each basic block. These basic blocks would therefore remain unoptimized. The task level model would then resemble the original task level model from MPS, which has no instruction level optimizations to shrink the basic block sizes. ITS may perform better with a bit more time, but that would be going beyond the time limitation placed on the other algorithms.

In all cases, the algorithms were run on a 2-issue, 2-processor system. This was chosen to compare with existing literature. A larger number of processors or functional units can be used. The only change that would occur would be a longer run time, as a larger number of processors would produce a larger search space, as more combinations are possible (both in terms of number of processors available, and the combination of communication possible between processors).

Chapter 6

Conclusions and Future Work

In this chapter we present the conclusions of running the algorithms on both the test programs and the FFT. We then present a set of possible extensions to this research that could make the algorithms more robust, produce better results, and be able to handle a wider variety of programs.

6.1 Conclusion

We presented a scalable method for scheduling static, instruction set extended code onto single and multi-issue processors, for use as an optimization pass in the SHIRA toolchain. Four such ILP-based algorithms were developed in the process of creating an algorithm to combine scheduling of both task and instruction level code with ISEs. Each iteration of the algorithms developed increased the speedup of the programs tested when compared to its predecessor. We assumed that the system was composed of 2 or 4 identical functional units divided across two processors that could be extended with ISEs, and were able to communicate with one another with a symmetrical set of delays (i.e. the delays for sending from processor 1 to processor 2 would be the same if sent the other way).

We also made the assumption that the work from [25] would be the only source of ISEs, and that no ISEs would be created by hand in an effort to boost speedup (though the code itself was formatted to be easier to exploit parallelism). While the scheduling algorithms themselves would not need to be modified, the generation of the dataflow graph that inputs to the scheduling algorithms would require modification. This could either be accomplished by modifying the ISE generation portion to use either a list of hand made ISEs, or to use a mix of hand made ISEs and automatic ISE generation. As the work of generating ISEs was performed by another graduate student in the CARG research group, it is beyond the scope of this work. Finally we assumed that the SHIRA toolchain would not be modified in order to produce three address code optimized for work in this thesis.

The first algorithm, MPS, was able to statically schedule tasks into a multiprocessor system. MPS was then improved in two separate ways. First, the IMP algorithm was created to schedule tasks in a program that had ISEs added to the instruction mix. Second, the ITS algorithm was created, allowing for the instructions in each task to first be scheduled into the functional units of a processor, before the tasks themselves were scheduled into the multiprocessor system. Finally, these improvements were combined into the CSA, allowing for instruction and task scheduling of programs with ISEs.

We discovered that the ILP based method for scheduling was best suited to smaller problems, and that the act of splitting the problem of instruction scheduling into small pieces (those inside of a given task) allowed for larger problems to be solved. All four of the algorithms managed to double the size of programs able to be run in comparison to the work in [20], and in the case of CSA, were able to obtain results for a program with 90 nodes; three time that of the ILP based method used in [20].

In comparison to the heuristic methods presented in [7], the ILP methods implemented in this thesis were significantly closer to the optimal or theoretical optimal solution. Even the simplest of the four algorithms developed, MPS, achieved nearly

optimal results with each test program for sizes up to 60 nodes.

Comparisons involving IMP and CSA went beyond the 2 and 4 times speedup, which would be the optimal without ISEs for a two processor system scheduling only tasks and a two-issue, two-processor system scheduling tasks and instructions, respectively. This is due to their ability to exploit the use of instruction set extensions to create better speedup.

When comparing to a hand crafted solution for a 64-point FFT presented in [24], the automated methods of scheduling with ISEs proved to be less effective than the ISEs optimized for the FFT program. However, the results corresponded to those generated from the set of randomly generated programs. This benchmark shows that all four algorithms continue to operate as expected when used in a real application. While not implemented in hardware, a similar speedup would be expected for programs where cache and communication issues such as bus contention would not be a major source of delay. Once these are taken into account, the schedule lengths would likely increase, as other runtime issues would force the processor to switch tasks.

6.2 Future Work

This section details the possible routes for expansion based on the results obtained form the scheduling algorithms presented. As the MPS, IMP, ITS and CSA algorithms are all designed to statically schedule code for a symmetrical, homogeneous multiprocessor system, several new research and development opportunities can continue from where this work leaves off.

6.2.1 Combination with Dynamic Scheduling

As the four algorithms presented in this thesis are designed to schedule small blocks of static code, they would fit well into the SHIRA toolchain as a pass called by a higher

level dynamic scheduling algorithm. Larger portions of a program may contain calls to user inputs or have other unknowable delays in the code, neither of which can be statically scheduled. A heuristic for dynamically scheduling the portions of the code that are not static would be the best option for these large program calls.

By having the smaller portions of static code within each program call be scheduled by the aforementioned static scheduling algorithms, optimized program calls can be delivered to the dynamic scheduler. This would result in the dynamic scheduler having an easier time, and would result in less waste during runtime scheduling, as the static code was scheduled at compile time.

6.2.2 Heterogeneous Processors

The ability to schedule instructions and tasks onto processors which are non identical will be crucial in extending the work done in this thesis. Allowing for differences in the processors will decrease the overall size of certain projects, as the instruction set extensions will not need to be applied to all functional units in all processors. This will save space for other additional ISEs, co-processors, and/or additional processors.

Additionally, different processors are able to perform similar calculations in different amounts of time. For programs that contain multiple types of work, some of which are performed well in some processors, and poorly in others, will be able to benefit from a multiprocessor system making use of the processors that will benefit it best. An example of this would be a game engine that requires both physics calculations for the physics engine and video processing calculations for generation of shaders and textures. The physics calculations may be done on a physics card, and the video calculations on a video card.

This work will add another layer of complexity to the generation of the schedule, as the ILP will require a list of processors available for use, the amount of space available, and a profile of which types of instructions are best executed on which

processor(s). Additionally, it will need to permute through all possible combinations of instruction set extended functional units and processors in order to find the most beneficial.

6.2.3 Asymmetrical Network Topologies

As systems add more and more processors, the need to allow for scheduling across an asymmetrical network becomes essential. It is easy to calculate asymmetrical delays in a two processor system, but such an arrangement is not likely to be found in real world applications. A real world system, such as a cellphone, is likely to have more than two processors. By calculating the amount of delay between each processor in the system and arranging it in a structure than can be easily used in the Lingo models, static scheduling for non-symmetrical networks can be achieved.

The ability to schedule code across an asymmetrical network has applications in embedded systems for modern portable devices. Embedded systems are often a collection of multiple heterogeneous processors spaced unevenly on a board, but can have predetermined delays between processors. Additionally, asymmetrical computing will require the ability to schedule code over networks with asymmetrical bandwidth. A cloud of computers communicating over an ADSL line will have differing upload and download speeds between nodes. The scheduling algorithm will need to account for these differences, sending more variables to be calculated downstream, while the less common results of the calculations can be sent upstream.

6.2.4 Balancing the Tradeoff between ISEs speedup and Schedule length

In scheduling across multiple processors, and multiple issue lanes in a processor with code that has instruction set extensions, a tradeoff calculation between parallelism and development of ISEs will need to be taken into account to maximize the efficiency.

This tradeoff between parallelism and hardware is also applicable to larger custom hardware blocks. Research into balancing scheduling, ISEs and custom hardware would be required.

The motivating example for this tradeoff is as follows: a set of independent instructions which sequential schedule to a length of X. Finding an ISE in this set of instructions for which the ISE uses a subset of nodes and results in a beneficial speedup compared to the nodes' sequential runtime. In order to provide benefits, the ISE must be faster than or equal to the speedup offered by parallelism. In the case of a two processor system that means an ISE must incorporate at least two independent nodes, as the system can schedule these nodes to run concurrently (one on each processor).

As the ISE has a hardware cost associated with it, implementing an ISE with a speedup of two (based on two independent nodes) in a two-processor system is not beneficial, as the ISE has a higher hardware cost than simply scheduling the nodes across the processors. This number increases proportionally with the number of processors in the system. A value for the benefit of the ISE must be factored in when selecting which ISE candidates should be implemented in order to maximize the hardware usage.

In addition to the above case, another area for potential expansion would be research into whether or not an ISE can be selected that is actually worse than the speedup offered by instruction level parallelism in cases where dependencies exist. The motivating example in Figs. 6.1 and 6.2 illustrate a set of fictional ISEs that would, if implemented, provide a worse speedup than simple parallelism.

In Fig. 6.1 we make the assumption of no communication costs, and a runtime of one cycle per instruction. Any instruction that is not dependent on another instruction can be executed in parallel, requiring only one cycle. As can be seen in Fig. 6.1, the schedule can run in 4 cycles, requiring A to be run first, followed by B and C which can be run in parallel. For instructions D, E, and F all three can be

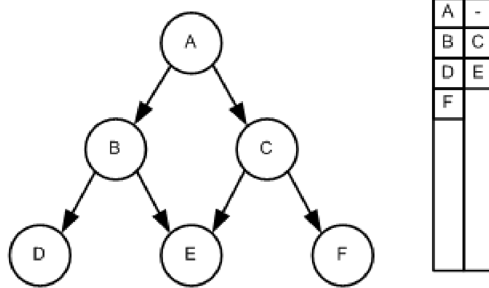

Figure 6.1: Motivating Example Schedule (no ISEs)

run in parallel, but the system has only two execution slots, limiting the number of instructions that can be run in parallel to two, with F being forced to run one cycle later.

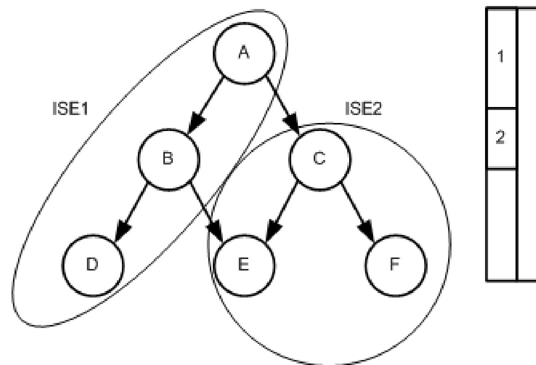

Figure 6.2: Motivating Example Schedule with ISEs

As the benefit of ISEs is measured in comparison to the sequential runtime of the instructions which make up the ISE, the ISEs in Fig 6.2 have a speedup of 1 and 1.5 respectively. As the communication for ISEs is done at the beginning and end of the running of each ISE, $ISE2$ must be run after $ISE1$ is finished. This leads to a runtime of 5 cycles, which is longer than the time required without ISEs.

6.2.5 Coprocessor Scheduling

The CARG research group currently has members developing the ability to add custom coprocessors to SoCs. The current state of this development allows for the identification and creation of a two input one output fuzzy logic coprocessor to be implemented [31].

Eventually, this coprocessor development should be expanded to include several beneficial coprocessors. By combining the coprocessor extension with the instruction set extensions, a toolchain should be able to determine the optimal tradeoff between the number of dedicated coprocessors and the number of ISEs.

As coprocessors may require multiple clock cycles to return a value, this delay will need to be considered when scheduling portions of the code. If the delay is not factored into the scheduling calculations, there may be several clock cycles of downtime while the rest of the system waits for the results from a coprocessor. Dependencies will need to be calculated so that variables not dependent on the results of the coprocessor can be executed in parallel.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers; Principles, Techniques, and Tools*. Addison Wesley, Reading, Mass., 1988.

[2] Altera Inc. *Nios II Processor Reference Handbook*. 2009. http://www.altera.com/literature/hb/nios2.

[3] CDS Technology Inc. Fourier Transform Example, 2004. http://www.ece.uvic.ca/~ece499/2004a/group05/html/background.html.

[4] J. W. Chinneck. *Practical Optimization: a Gentle Introduction*. 2003. http://www.sce.carleton.ca/faculty/chinneck/po.html.

[5] COINS-project. COINS: A Compiler Infrastructure Project, 2010. http://www.coins-project.org/international/.

[6] P. Crescenzi and V. Kann. A compendium of NP optimization problems, 2005. http://www.ensta.fr/~diam/ro/online/viggo_wwwcompendium/.

[7] T. Davidović and T. G. Crainic. Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multiprocessor systems. *Comput. Oper. Res.*, 33(8):2155–2177, 2006.

[8] L. D. English and G. S. Halford. *Mathematics education: models and processes*. Lawrence Erlbaum Associates Inc. Publishers, Mahwah, NJ, USA, 1995.

[9] J. Fenalson and R. Stallman. GNU gprof: the GNU Profiler. `http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html`.

[10] C. Galuzzi and K. Bertels. The instruction-set extension problem: A survey. In *Proc. 4th Int. workshop on Reconfigurable Computing, ARC '08:*, pages 209–220, 2008.

[11] C. Galuzzi, D. Theodoropoulos, R. J. Meeuws, and K. Bertels. Algorithms for the automatic extension of an instruction-set. In *Design, Automation and Test in Europe (DATE 2009)*, April 2009.

[12] L. M. Gambardella, ric Taillard, and G. Agazzi. Macs-vrptw: A multiple colony system for vehicle routing problems with time windows. In *New Ideas in Optimization*, pages 63–76. McGraw-Hill, 1999.

[13] M. Gries. Methods for evaluating and covering the design space during early design development. *Integr. VLSI J.*, 38(2):131–183, 2004.

[14] R. Hada, K. Tanigawa, A. Kojima, and T. Hironaka. An adaptive compiler method for scheduling and place-and-route for VLIW-based dynamic reconfigurable processor. In *Proceedings of the 12th WSEAS international conference on Computers*, pages 61–69, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[15] M. Johnson and J. Zelenski. Intermediate representation, 2008. http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/16-Intermediate-Rep.pdf.

[16] N. Lall. FPGA judgment day: Rise of second generation structured ASICs. http://www.opensystems-publishing.com/e-letter/dsp/2008/03/easic.pdf.

[17] S. McFarling. WRL research report 91/5, procedure merging with instruction caches. Technical report, Western Research Laboratory, 100 Hamilton

Avenue Palo Alto, California, 94301, USA, 1991. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-91-5.pdf.

[18] J. E. Miller and A. Agarwal. Flexicache: Software-based instruction caching for embedded processors, 2006. groups.csail.mit.edu/cag/raw/documents/ASPLOS06_Icache.ppt.

[19] M. Montcalm, D. Shapiro, V. Groza, and M. Bolic. ITS: An ILP-based combined instruction/task static scheduling algorithm. Technical report, Computer Architecture Research Group, University of Ottawa, Ottawa, Ont., Canada, Jan. 2010. Report No. TR-2010-2.

[20] N. Satish, K. Ravindran, and K. Keutzer. A decomposition-based constraint optimization approach for statically scheduling task graphs with communication delays to multiprocessors. In *Proc. DATE '07*, pages 1–6, April 2007.

[21] N. R. Satish, K. Ravindran, and K. Keutzer. Scheduling task dependence graphs with variable task execution times onto heterogeneous multiprocessors. In *Proc. 8th Int. Conf on Embedded software. EMSOFT '08*, pages 149–158, New York, NY, USA, 2008. ACM.

[22] E. Schnarr and J. R. Larus. Instruction scheduling and executable editing. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 288–297, Washington, DC, USA, 1996. IEEE Computer Society.

[23] L. Schrage. *Optimization Modeling with LINGO*. Lindo Systems Inc., Chicago, IL., 6th edition, 2006.

[24] T. Schuster, B. Bougard, D. N. Bruna, E. Matus, L. V. D. Perre, and G. Fettweis. An exploration methodology for VLIW architecture targeting multi-

mode wireless baseband processing. 2006. `http://mns.ifn.et.tu-dresden.de/publications/2006/Matus_E_SPS_06.pdf`.

[25] D. Shapiro. Design and implementation of instruction set extension identification for a multiprocessor system-on-chip hardware/software co-design toolchain. Master's thesis, School of Information Technology and Engineering, Faculty of Engineering, University of Ottawa, Ottawa, Canada, January 2009.

[26] D. Shapiro, M. Montcalm, M. Bolic, and V. Groza. Static task scheduling for configurable multiprocessors. Technical report, Computer Architecture Research Group, University of Ottawa, Ottawa, Ont., Canada, Jan. 2010. Report No. TR-2010-1.

[27] G. Sih and E. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 4(2):175–187, Feb 1993.

[28] S. S. Skiena. *The algorithm design manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[29] K. Srinivasan and K. S. Chatha. Integer linear programming and heuristic techniques for system-level low power scheduling on multiprocessor architectures under throughput constraints. *Integration, the VLSI Journal*, 40(3):326 – 354, 2007.

[30] B. D. Sutter and Vakgroep Elektronica. General-purpose architecture instruction scheduling techniques. Technical report, ELIS, 1998.

[31] V. Thareja, M. Bolic, and V. Groza. Design of a fuzzy logic coprocessor using Handel-C. In *Proceedings of the 2nd International Workshop on Soft Computing Applications (SOFA)*, August 2007.

[32] The Eclipse Foundation. Eclipse IDE, 2009. `http://www.eclipse.org/`.

[33] R. Wain et al. An overview of FPGAs and FPGA programming; Initial experiences at Daresbury. Technical report, Computational Science and Engineering Department, CCLRC Daresbury Laboratory, Daresbury, Warrington, Cheshire, WA4 4AD, UK, 2006.

[34] L. Wang, Y. Huang, X. Chen, and C. Zhang. Task scheduling of parallel processing in CPU-GPU collaborative environment. In *ICCSIT '08: Proceedings of the 2008 International Conference on Computer Science and Information Technology*, pages 228–232, Washington, DC, USA, 2008. IEEE Computer Society.

[35] C. Wells. DOING MATH: Zooming and chunking, 2009. http://www.abstractmath.org/MM/ MMZoomChunk.htm.

[36] W. Xu and R. Tessier. Tetris: A new register pressure control technique for VLIW processors. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, San Diego, CA, USA, 2007.

[37] D. Zaretsky, G. Mittal, X. Tang, and P. Banerjee. Evaluation of scheduling and allocation algorithms while mapping assembly code onto FPGAs. In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 397–400, New York, NY, USA, 2004. ACM.

Appendix A

Example Calculations

This Appendix contains portions of the calculations involved in finding a schedule for an example problem. Figure A.1 shows the example dataflow graph, and the initial calculations of the entire graph. Figure A.2 shows the calculations involved with dependent nodes, while Figure A.3 shows the calculations needed for independent nodes.

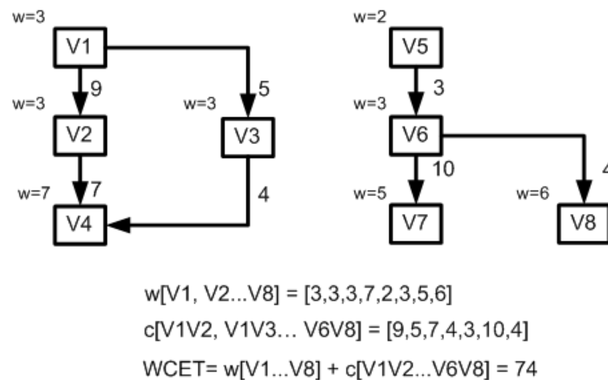

Figure A.1: Calculation of W, C, and WCET for example problem

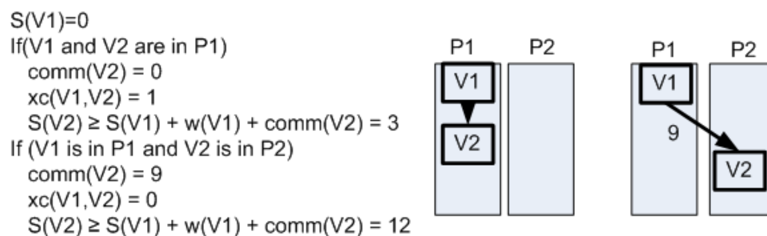

Figure A.2: Calculation of COMM, XC and S for dependent nodes

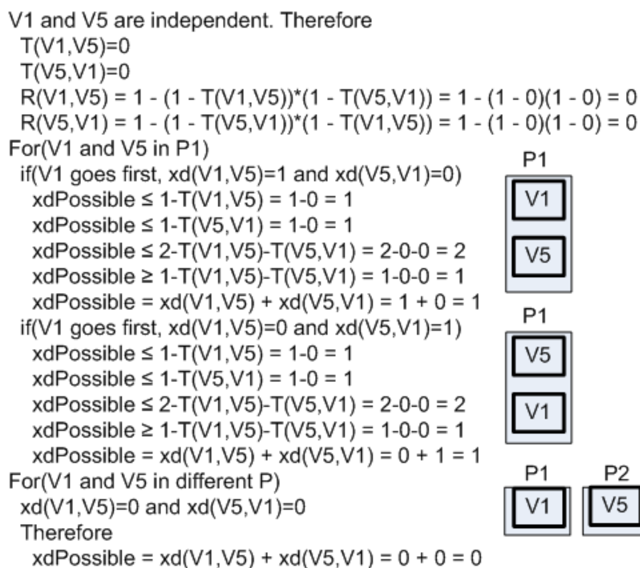

Figure A.3: Calculation of T, R, XD and XDPossible for independent nodes

Appendix B

Algorithm Coverage

This Appendix contains a layout of all the steps covered by each of the algorithms. The smallest numbers of steps is performed by the MPS algorithm, with the ITS and IMP algorithms each adding an additional set of steps to their calculations. The CSA algorithm performs all calculations.

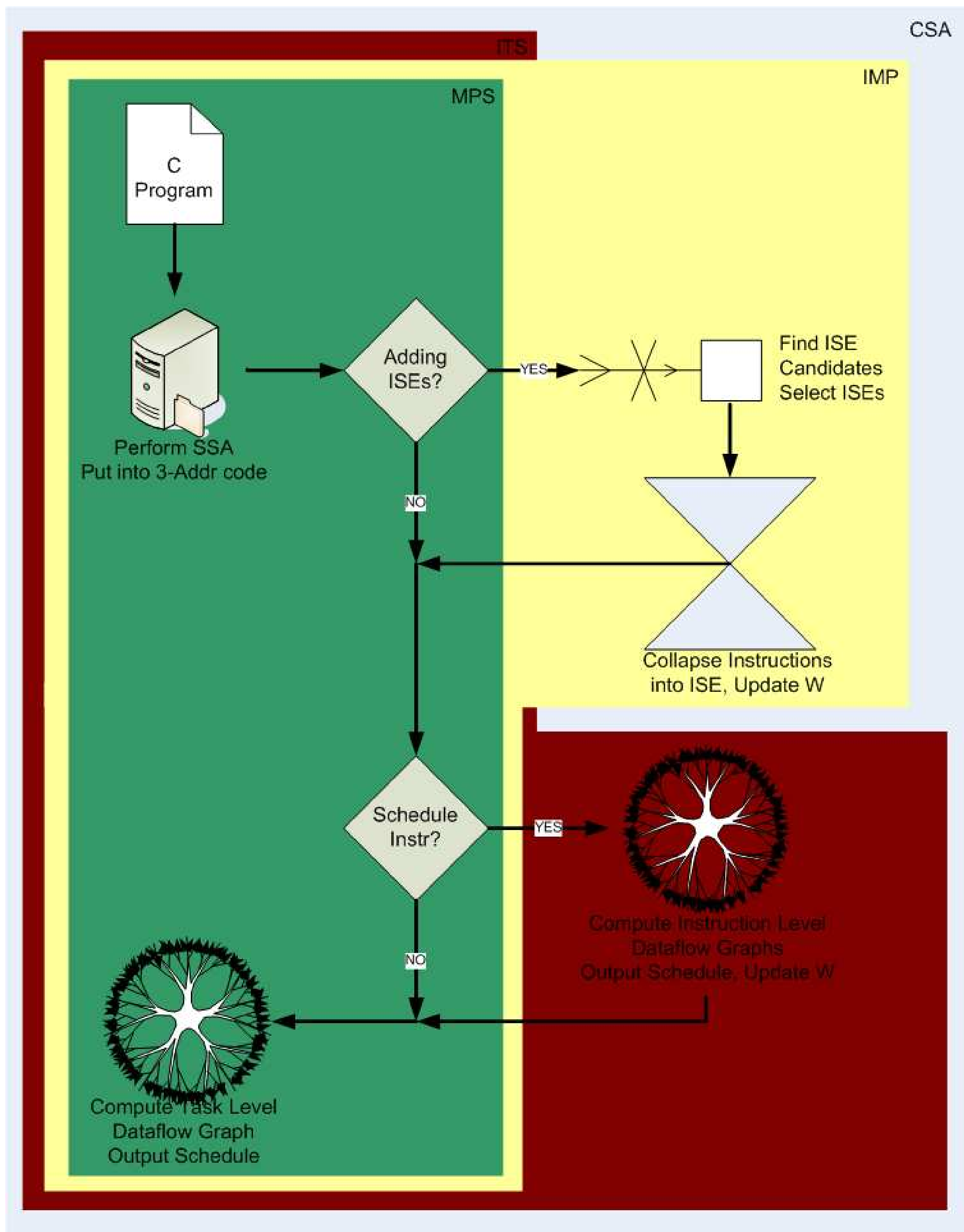

Figure B.1: Steps covered by each algorithm