Constraint Programming Techniques for Optimal Instruction Scheduling

by

Abid Muslim Malik

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2008

AUTHOR'S DECLARATION FOR SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abid Muslim Malik

Abstract

Modern processors have multiple pipelined functional units and can issue more than one instruction per clock cycle. This puts great pressure on the instruction scheduling phase in a compiler to expose maximum instruction level parallelism. Basic blocks and superblocks are commonly used regions of code in a program for instruction scheduling. Instruction scheduling coupled with register allocation is also a well studied problem to produce better machine code. Scheduling basic blocks and superblocks optimally with or with out register allocation is NP-complete, and is done sub-optimally in production compilers using heuristic approaches. In this thesis, I present a constraint programming approach to the superblock and basic block instruction scheduling problems for both idealized and realistic architectures. Basic block scheduling with register allocation with no spilling allowed is also considered. My models for both basic block and superblock scheduling are optimal and fast enough to be incorporated into production compilers. I experimentally evaluated my optimal schedulers on the SPEC 2000 integer and floating point benchmarks. On this benchmark suite, the optimal schedulers were very robust and scaled to the largest basic blocks and superblocks. Depending on the architectural model, between 99.991% to 99.999% of all basic blocks and superblocks were solved to optimality. The schedulers were able to routinely solve the largest blocks, including blocks with up to 2600 instructions. My results compare favorably to the best previous optimal approaches, which are based on integer programming and enumeration. My approach for basic block scheduling without allowing spilling was good enough to solve 97.496% of all basic blocks in the SPEC 2000 benchmark. The approach was able to solve basic blocks as large as 50 instructions for both idealized and realistic architectures within reasonable time limits. Again, my results compare favorably to recent work on optimal integrated code generation, which is based on integer programming.

Dedicated to my wife Jouweeriea Abid and my son Abdullah Abid.

Acknowledgments

First, I would like to thank my supervisor, Peter van Beek, for his sincere guidance, mentoring and support throughout my work as a graduate student at the University of Waterloo. I would like to thank my examination committee members Jose Nelson Amaral, Peter Buhr, Gordon Cormack and Todd Veldhuizen for their very useful suggestions on my work. Special thanks to Jim McInnes and Marin Litoiu from the IBM Canada for their support during my Ph.D work. Thanks to Tyrel Russell and Michael Chase for their contribution in the work.

A big thank you to my wife, Jouweeriea, for encouraging and supporting me in my research and also helping me to balance between school and other aspects of my life.

I would like to thank my mother Zohra, my brothers Adil, Tariq, Zahid and Shahid, my nieces Saba and Maha, my nephews Ali and Ahmad and my sister Farah for their constant support.

I would like to thank a very special person, my two year old son Abdullah, for providing happiness in the last two years of my Ph.D work with his cute smiles.

This work was supported financially by IBM CAS and IBM Ph.D fellowships.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, I informally introduce my research area: optimal instruction scheduling for multi-issue processors. I give a short introduction to the compilation phase and the instruction scheduling problem for modern processors. Further, I motivate the interest of the optimal instruction scheduling problem and summarize the contributions of my work.

1.1 Compilation phase

A typical compilation path for a compiler is shown in Figure 1.1. A compiler takes as input a source program written in some high-level language and performs lexical analysis (scanning), syntax analysis (parsing) and semantic analysis (type checking) in the front-end. It performs control flow analysis, data flow analysis, various optimizations, instruction scheduling, register allocation and then finally generates machine code in the back-end. These traditional front-end compiler analysis and back-end optimizations steps are quite mature and well understood and are routinely employed in production compilers. The back-end optimization phases play a key role in the quality of the final machine code.

1.2 Instruction scheduling

A key feature in modern processors is multiple pipelined functional units. Examples of functional units include arithmetic-logic units (ALUs), floating point units, memory or load/store units which perform address computations and accesses to memory hierarchy, and branch units which execute branch and call instructions. Having multiple functional units allows the processor to issue more than one instruction in each cycle. This phenomenon is known as *instruction level parallelism*

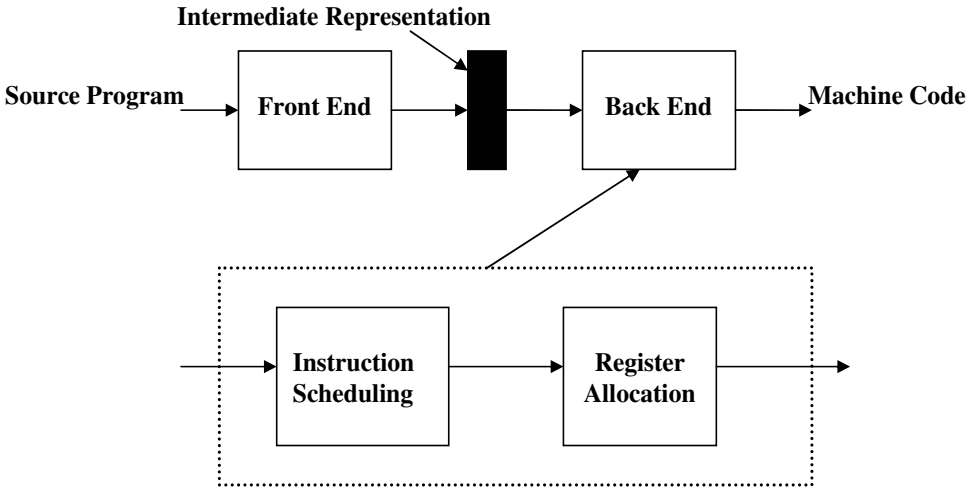


Figure 1.1: Typical compilation path of a compiler for modern processors.

(ILP). Instruction level parallelism increases the throughput of a processor; i.e., the rate at which instructions of an application program can be executed. As we are approaching the technological limits to processor cycle time, ILP has become a key area of research in the compiler field.

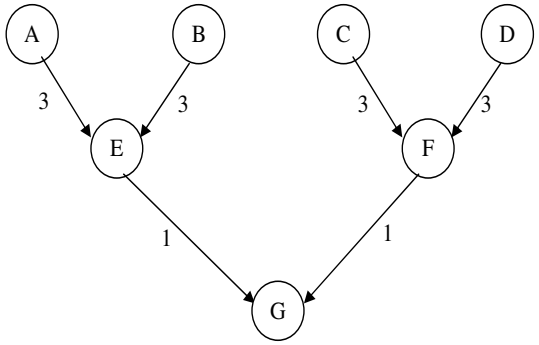
For a pipelined functional unit, each instruction is divided into different sections and each section of the instruction requires different hardware on a functional unit. The number of sections defines the pipeline depth of a functional unit. The number of sections varies depending on processors and instruction types but typically there are four: instruction fetch and decode, address generation and data fetch for memory instructions, instruction execute, and write back. Supposing that n such sections can be identified in an instruction, the pipeline depth for the unit taking care of that instruction is n. Ideally n instructions can be in-flight in each cycle. In-flight instructions are those instructions that have been issued but have not yet been completed. If there are m pipelined functional units in a processor, then there are at most $m \times n$ instructions in-flight in each cycle. A functional unit capable of issuing a new instruction in each cycle is called a *fully pipelined functional unit*. The number of instructions in-flight measures the amount of parallelism that a compiler must provide to keep a processor busy. But in real processors, this is not always true. Sometimes, instructions get hung in one pipeline stage for multiple cycles. There are a number of reasons why this might happen[1]. When it happens, the pipeline is said to be *stalled*. All of the instructions in the stage below the one where the stall happened continue advancing normally, while the stalled instruction just sits in its stage and backs up all the instructions behind it. The number of clock cycles an instruction takes to pass through the pipeline is known as the *instruction latency*. In real processors, the instruction latency is not necessarily equal to the number of pipeline stages. Because, instructions can get hung in one or more pipeline stages

[1]See Chapter 2 for details.

for multiple cycles, each extra cycle that they spend waiting in a pipeline stage adds one more cycle to their latency.

To ensure enough instructions are available to keep all the functional units busy, the compiler rearranges the instructions, which makes instruction scheduling an important phase in any compiler. Instruction scheduling is a code reordering transformation that attempts to hide the latencies inherent in modern processors. The latency of an instruction determines how long the following instructions have to wait to see the result produced by that instruction. This wait or gap needs to be filled with other available instructions that do not depend on the stalled instruction. If the compiler cannot find any such instruction then it inserts NOPs (no operations) to preserve the semantics of the program.

Instruction scheduling is done on certain regions of a program. Commonly used scheduling regions include *basic blocks* and *superblocks*. A basic block is a collection of instructions with a unique entrance and a unique exit point and is used for the local instruction scheduling problem. A superblock is a collection of basic blocks with a unique entrance but multiple exit points and is used for the global instruction scheduling problem. I will discuss both of these problems in detail in the next chapter. Dependencies among instructions in scheduling regions are normally represented by a *directed acyclic graph* (DAG). In a DAG, each node represents an instruction. There is an edge from node A to node B in a DAG, if instruction B is dependent on instruction A. The edge is labeled with an integer number that is the latency associated with instruction A.



(a)

A	$r1 \leftarrow a$
B	$r2 \leftarrow b$
C	$r3 \leftarrow c$
D	$r4 \leftarrow d$
E	$r1 \leftarrow r1 + r2$
F	$r4 \leftarrow r4 \times r3$
G	$r1 \leftarrow r1 + r4$

(b)

Figure 1.2: (a) Dependency DAG associated with the instructions to evaluate $(a + b) + (c \times d)$; (b) assembly code for the DAG.

Example 1.1 *Figure 1.2 shows a simple dependency DAG associated with the instructions to evaluate* $(a + b) + (c \times d)$. *Assume a processor with two fully pipelined functional units capable of*

executing all types of instructions and that loads from memory have a latency of three cycles and all other operations have a latency of one cycle. Two possible schedules are shown in Table 1.1. The optimal schedule requires only four NOPs and is three cycles shorter than the non-optimal schedule. This shows the importance of a good instruction scheduler.

Cycle	Non-Optimal Schedule		Optimal Schedule	
0	$r1 \leftarrow a$	$r2 \leftarrow b$	$r1 \leftarrow a$	$r2 \leftarrow b$
1			$r3 \leftarrow c$	$r4 \leftarrow d$
2				
3	$r1 \leftarrow r1 + r2$	$r3 \leftarrow c$	$r1 \leftarrow r1 + r2$	
4	$r4 \leftarrow d$			$r4 \leftarrow r4 \times r3$
5			$r1 \leftarrow r1 + r4$	
6				
7	$r4 \leftarrow r4 \times r3$			
8	$r4 \leftarrow r1 + r4$			

Table 1.1: Two possible schedules for the DAG in Figure 1.2. Empty slots represent NOPs.

Besides using an instruction scheduler to exploit the parallelism of a processor by rearranging instructions, a compiler attempts to ensure that the data needed and produced by the instructions is available in a memory which is fast and close to the processor. In a memory hierarchy, physical registers are considered the fastest and closest to a processor. The *register allocation* phase is used by a compiler to decide which data values should reside in physical registers. However, there are a limited number of registers in any processor. It is impossible to keep the values of all variables present in a program in the registers all the time. At certain points in a program, the number of variables may exceed the number of available registers. At these points, a compiler has to decide which values should be moved to a cache memory, which is next to registers in any memory hierarchy. This movement of data from registers to memory is called *spilling*. The movement of data between a cache and registers consumes many clock cycles and affects the throughput of a processor. The register allocation phase is usually done after the instruction scheduling phase. Thus, any gain from the instruction scheduling phase due to a shorter schedule may be lost due to excessive spilling in the register allocation phase. Co-ordination between these two phases is an important factor in increasing throughput. This co-ordination is achieved by maintaining the number of data variables alive at each point in a program within certain limits during the instruction scheduling phase. In the compiler literature, one finds the following two definitions for liveness of a variable:

1. A data variable is alive in a register between its first load and last use [51].
2. A data variable is alive in a register between its first load and before the next write update to the register [27].

Cycle	Schedule 1		Schedule 2	
0	$r1 \leftarrow a$	$r2 \leftarrow b$	$r1 \leftarrow a$	$r2 \leftarrow b$
1			$r3 \leftarrow c$	
2				
3	$r1 \leftarrow r1 + r2$	$r3 \leftarrow c$	$r1 \leftarrow r1 + r2$	
4	$r4 \leftarrow d$			$r2 \leftarrow d$
5				
6				
7	$r4 \leftarrow r4 \times r3$		$r2 \leftarrow r2 \times r3$	
8	$r4 \leftarrow r1 + r4$		$r1 \leftarrow r1 + r2$	

Table 1.2: Two possible schedules for the DAG in Figure 1.2 with different maximum number of variables alive. Empty slots represent NOPs.

According to the first definition, variable b in Table 1.1 is alive in $r2$ between cycle 0 and 3 for both non-optimal and optimal schedules. But, it is alive throughout both the schedules according to the second definition. For my work, I consider the second definition. The number of data variables alive at a point in a program is also known as the *register pressure* at that point.

Example 1.2 *Consider the DAG in Figure 1.2 again. Two possible schedules are shown in Table 1.2. In Schedule 1, we have a maximum of four variables alive at any point (e.g., cycle 5). In Schedule 2, we have a maximum of three variables alive at any point (e.g., again cycle 5). Schedule 2 is preferred over Schedule 1, as it only uses three registers.*

1.3 Importance of the work

Instruction scheduling is NP-complete for realistic architectures. This has led to the belief that in production compilers, a heuristic or approximation algorithm approach must be used rather than an exact approach to instruction scheduling [51]. The most commonly used heuristic approach is the *list scheduling algorithm*, which is a greedy algorithm. Although heuristic approaches have the advantage that they are fast, a scheduler which finds provably optimal schedules may be useful when compiling for software libraries, digital signal processing or embedded applications [27]. As well, an optimal scheduler can be used to evaluate the performance of heuristic approaches. Such an evaluation can tell whether there is a room for improvement in a heuristic or not. Finally, an optimal scheduler can be used to automatically create new heuristics using techniques from machine learning.

1.4 Main motivation behind the work

A major challenge when developing an exact approach to an NP-complete problem is to develop a solver that scales and is robust in that it rarely fails to find a solution in a timely manner on a wide selection of real problems. Recently, Wilken, Liu and Heffernan [66] and Shobaki and Wilken [60] showed that through various modeling and algorithmic techniques, integer linear programming and enumeration techniques could be used to produce optimal instruction schedules for large basic blocks and superblocks targeted to a multi-issue processor. However, these approaches are either restricted to single-issue architectures or apply only to idealized architectures. Dynamic programming approaches have also been proposed. However, they are limited to 10 to 40 instructions [42]. Recently, van Beek and Wilken [65] presented a constraint programming approach for single-issue processors that is fast and optimal for larger basic blocks. The results from their work motivated me to apply constraint programming techniques to the harder instruction scheduling problems on more realistic architectures.

1.5 Contributions

In this thesis, I present a constraint programming approach to instruction scheduling for multiple-issue processors that is robust and optimal. In a constraint programming approach, one models a problem by stating constraints on acceptable solutions. A constraint is simply a relation among several unknowns or variables, each taking a value in a given domain. The problem is then usually solved by interleaving a backtracking search with a series of constraint propagation phases. In the constraint propagation phase, the constraints are used to prune the domains of the variables by ensuring that the values in their domains are locally consistent with the constraints. In developing my optimal scheduler, the keys to scaling up to large, real problems were improvements to the constraint model and to the constraint propagation phases.

As already mentioned, attempts have been made to solve instruction scheduling problems optimally. However, test suites used in these works are often small and simple. As well, previous work often assumes a fully pipelined architecture, which is not a realistic architecture. In contrast, in my work I consider both idealized and realistic architectures. Also, the test suites which I use to evaluate my model contain all blocks from the SPEC 2000 benchmark with size as large as 2600 instructions and latency as large as 36 cycles. Scheduling blocks were collected before and after the register allocation phase. The main contributions of this work are:

1. A fast and optimal basic block instruction scheduler for both idealized and realistic multi-issue architectures.

2. A fast and optimal superblock instruction scheduler for both idealized and realistic multi-issue architectures.

3. An optimal basic block instruction scheduler for the combined instruction scheduling and register allocation without spilling problem for both idealized and realistic multi-issue architectures.

With the exception of [31, 32, 65, 66], previous approaches on optimal instruction scheduling have only been evaluated on a few problems with sizes of the problems ranging between 10 and 50 instructions and their experimental results suggest that they would not scale up beyond problems of this size. Further, many previous approaches are for idealized architectures. A major challenge when developing an optimal approach to an NP-complete problem is to develop a solver that scales and is robust in that it rarely fails to find a solution in a timely manner on a wide selection of real problems. In this thesis, I present constraint programming approaches to basic block and superblock scheduling for multiple-issue processors that are robust and optimal for both idealized and realistic architectures. The novelty of my approach is in the extensive computational effort put into a preprocessing stage in order to improve the constraint model and thus reduce the effort needed in backtracking search. I experimentally evaluated my optimal schedulers for basic block and superblock scheduling on the SPEC 2000 integer and floating point benchmarks, using four different idealized architectural models and four realistic architectural models. On this benchmark suite, the optimal schedulers scaled to the largest blocks and were very robust. Depending on the architectural model, at most 3 basic blocks and 15 superblocks out of the hundreds of thousands of blocks used in our experiments could not be solved within a 10-minute time bound. This represents approximately a 50-fold improvement, in terms of number of problems solved, over previous work.

I also present a constraint programming approach for basic block integrated with register allocation. My approach for basic block scheduling without spilling allowed was able to solve 97.496% of all basic blocks in the SPEC 2000 benchmark. The approach was able to solve basic block as large as 50 instructions for both idealized and realistic architectures with in 10 minutes. This compares favorably to the recent work by Bednarski and Kessler [5, 6] on optimal integrated code generation using integer programming. The approach by Bednarski is targeted towards a theoretical idealized multi-issue VLIW processor and is able to solve basic block as large as 50 instructions with in 20 minutes for unit latency and basic blocks as large as 20 instructions with arbitrary latencies.

It should be noted that the scope of this work is limited to instruction scheduling of acyclic code. Cyclic scheduling techniques such as loop unrolling and software pipelining [51] are beyond the scope of this thesis.

1.6 Origin of the thesis

Van Beek and Wilken [65] presented a constraint programming approach for a idealized single-issue processor that is fast and optimal for larger basic blocks. I continued the research on harder

instruction scheduling problems and more realistic architectures. A large part of the material in this thesis originates from the following publications.

- Abid M. Malik, Jim McInnes and Peter van Beek. Optimal Basic Block Instruction Scheduling for Multiple-Issue Processors using Constraint Programming. *International Journal on Artificial Intelligence Tools*, Accepted June 2007. A preliminary version appears in *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, Washington, DC, 2006.
- Abid M. Malik, Tyrel Russell, Michael Chase, and Peter van Beek. Learning List Scheduling Heuristics for Basic Block Scheduling. *Journal of Heuristics*, Accepted January 2007. A preliminary version appears in *Proceedings of the 15th CASCON*, Toronto, 2005.
- Abid M. Malik, Tyrel Russell, Michael Chase, and Peter van Beek. Optimal Superblock Instruction Scheduling for Multiple-Issue Processors using Constraint Programming. Technical Report, CS-2006-37, School of Computer Science, University of Waterloo, 2006.
- Michael Chase, Abid M. Malik, Tyrel Russell and Peter van Beek. An Optimal Scheduler and a Performance Analysis of List Scheduling for Superblocks. In preparation.
- Tyrel Russell, Abid M. Malik, Michael Chase, and Peter van Beek. Learning List Scheduling Heuristics for Superblock Scheduling. 19 pages. Submitted to *IEEE Transactions on Knowledge and Data Engineering*. Through one round of reviewing.

1.7 Organization of the thesis

The rest of the thesis is organized as follows: Chapter 2 gives the technical background necessary for understanding the problems addressed in this thesis. Chapter 3 describes the constraint programming approach for basic block instruction scheduling. Chapter 4 describes the constraint programming approach for superblock scheduling. Chapter 5 describes the constraint programming approach for basic block instruction scheduling without spilling. Chapter 6 concludes the thesis and discusses potential future work.

Chapter 2

Background

In this chapter, I describe the technical background needed to understand the instruction scheduling problems addressed in this thesis. I define the main terms used in this thesis, present the main issues in the instruction scheduling problems, and introduce constraint programming and how constraint programming can be applied to model an instruction scheduling problem.

2.1 Fundamental constraints

An instruction scheduler reorders instructions to maximize instruction level parallelism. However, in order to preserve the semantics of a given program the reordering of instructions is done under certain constraints. A fundamental problem that arises in compilers is to find a better instruction schedule subject to data dependency, control and resource constraints.

2.1.1 Data dependency constraints

Two instructions I_1 and I_2 are data dependent if data produced by one instruction is used by the other instruction. There are three types of data dependency:

- Instruction I_2 is *flow dependent* on instruction I_1 if I_1 computes a value that I_2 uses, as in the following two instructions:

  $$(I_1) \quad r1 \leftarrow r2 + r3$$
  $$(I_2) \quad r4 \leftarrow r5 + r1$$

  If I_2 is scheduled before I_1 then the data in register $r1$ will be different from what is required for I_2. Hence, I_2 would compute the wrong result.

- Instruction I_2 is *anti-dependent* on instruction I_1 if I_1 reads some location (register or memory cell) that I_2 writes to, as in the following two instructions:

$$(I_1) \quad r1 \leftarrow r2 + r3$$
$$(I_2) \quad r2 \leftarrow r5 + r4$$

  Instruction I_2 overwrites the data in $r2$. Instruction I_1 would compute the wrong result if I_2 is scheduled ahead of I_1.

- Instruction I_2 is *output dependent* on instruction I_1 if I_1 and I_2 write to the same location (register or memory cell), as in the following two instructions:

$$(I_1) \quad r1 \leftarrow r2 + r3$$
$$(I_2) \quad r1 \leftarrow r5 + r4$$

  If I_2 is scheduled before I_1 then all the instructions after I_2 using $r1$ get the result of the wrong computation.

2.1.2 Control constraints

A control dependency occurs between a branch and subsequent instructions. An instruction is dependent on a branch if the outcome of that branch controls whether the instruction is executed or not. Example 2.1 explains control dependency among instructions.

Example 2.1 *Consider the code in Figure 2.1. Here the two move instructions, I_2 and I_3, are control dependent on the branch instruction I_1. However, the add instruction I_4 is not, since it will be executed regardless of the branch.*

(I_1) if $r1 == 0$ goto L1
(I_2) $r2 \leftarrow 1$
(I_3) L1: $r2 \leftarrow 2$
(I_4) $r1 \leftarrow r1 + r2$

Figure 2.1: Control dependency constraint.

2.1.3 Resource constraints

A resource dependency occurs when two instructions need the same functional unit. If there is a resource dependence between two instructions then they cannot execute at the same time, but in the absence of other dependencies, they can execute in any order. Example 2.2 explains resource dependency among instructions.

Example 2.2 *Consider the code in Figure 2.2. Assume both instructions I_1 and I_2 are fixed point instructions and require a fixed point functional unit for their execution. Suppose there is only one fixed point functional unit available in the processor. Only one of the instructions can be executed at a time. Since there does not exist any dependency between these two instructions, they can be issued in any order.*

$$\begin{array}{ll} (I_1) & r1 \leftarrow r1 + r2 \\ (I_2) & r3 \leftarrow r4 + r5 \end{array}$$

Figure 2.2: Resource dependency constraint.

2.2 Instruction scheduling regions

This section discusses the regions used in the instruction scheduling problems. In a compiler, a program is represented by a *call graph*. A call graph is an abstract data structure. In a call graph, each procedure of the program is represented by a node, and edges between nodes indicate that one procedure calls another procedure. Each procedure node in a call graph is represented by a *control flow graph* (CFG) which is also an abstract data structure. Directed edges in a CFG represent jumps within a procedure or program. A CFG is essential to several compiler optimizations based on global dataflow analysis. Both call graphs and CFGs are not used directly in instruction scheduling, but they help in building various regions for instruction scheduling.

Each node in a CFG is a *basic block*. A basic block is a sequence of instructions with a unique entrance and a unique exit point. A basic block is for local instruction scheduling by a compiler. Other instruction scheduling regions are formed by combining basic blocks in different ways. Figure 2.3(b) shows a CFG for the procedure in Figure 2.3(a). In Figure 2.3(b), instructions 1 through 4 form basic block B_1. Instructions 8 through 12 form basic block B_6. Instruction 13 forms B_2, instruction 5 forms B_3, instruction 6 forms B_4 and instruction 7 forms B_5. Section 2.4 discusses the issues related with basic block instruction scheduling in detail.

On modern processors, the main job of an instruction scheduler is to increase instruction level parallelism (ILP). Basic block scheduling exposes a limited amount of ILP. By combining basic blocks, ILP can be increased. In Figure 2.3(b) instructions in basic block B_2 are independent

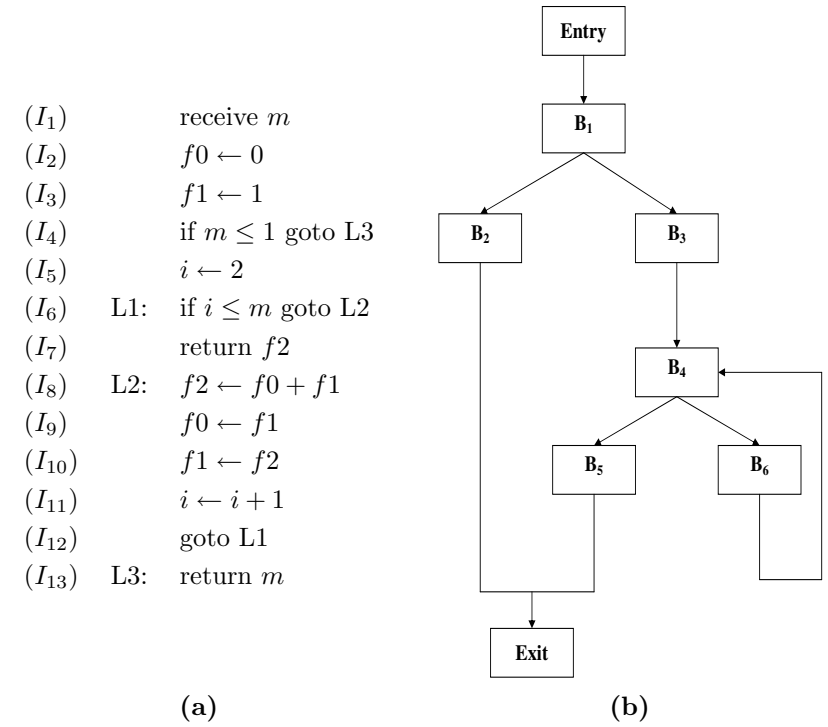


Figure 2.3: (a) Assembly code for a procedure computing Fibonacci numbers; (b) *control flow graph* for the code.

of the instructions in basic blocks B_3, B_4, B_5, and B_6. The ILP can be increased by inserting instructions from B_2 into the free slots available in the schedule for B_3, B_4, B_5, and B_6. This is only possible if one schedules instructions in all basic blocks at the same time. In global instruction scheduling, more than one basic block is considered simultaneously.

Fisher [22] introduced the concept of *trace* for global instruction scheduling. A trace is the most frequently executed loop-free path and is determined by profiling. Unlike a basic block, a trace may contain more than one entry point and exit point. The side entry points are also known as *join points*. Figure 2.4 shows a trace consisting of basic blocks A, B, D, E, G and H. We have join points in basic blocks D (from C) and H (from F). In trace scheduling, a trace is scheduled independently ignoring side exit and side entrance points. This may move some instructions across side exit and side entrance points. Bookkeeping is done to ensure the correct execution of the program. Figure 2.5 explains the bookkeeping process for downward movement of an instruction across a side exit point. Trace-1 and Trace-2 are connected by the control flow from instruction 2 (exit point in Trace-1) to instruction 3 (side entrance point in Trace-2). Trace scheduling moves instruction 1 across instruction 2 in the downward direction. In order to ensure execution of instruction 1, even if the program jumps from instruction 2 to instruction

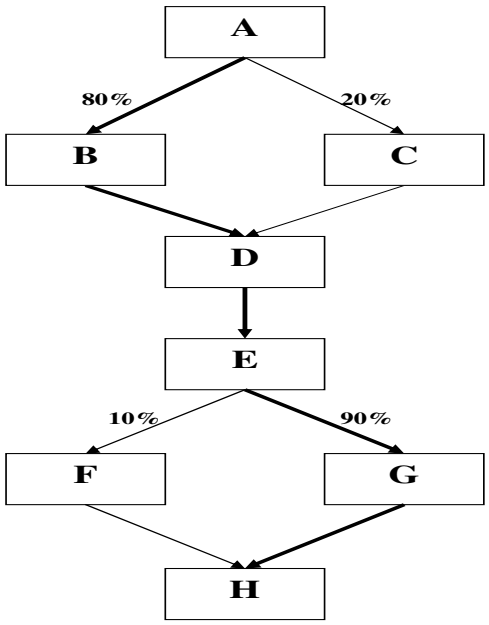


Figure 2.4: Path $A \rightarrow B \rightarrow D \rightarrow E \rightarrow G \rightarrow H$ has the highest execution probability. The path forms a trace.

3, a copy of instruction 1 is placed between instruction 2 and instruction 3. Upward movement of an instruction across a side exit point is known as *speculation* and the moved instruction is called a *speculative instruction*. An instruction is allowed to be a speculative instruction if (i) the destination of the speculative instruction is not used before it is redefined when the exit point is taken and (ii) the speculative instruction will never cause an exception that may terminate program execution when the exit point is taken. Special hardware support is needed to handle speculative instructions, but no bookkeeping is done.

Complex bookkeeping is done when an instruction is moved across a side entrance. In Figure 2.6(a), instruction 5 is moved upward across the side entrance point. To ensure the execution of instruction 5, if the control is entering the trace through the side entrance, a copy of instruction 5 is placed at the entrance point. In Figure 2.6(b), instruction 1 is moved downward across the entrance point, and placed after instruction 4. For correct execution of the program, if the control enters the trace through the side entrance, instruction 1 should not be executed. The entrance point is moved down after instruction 1 and copies of instruction 3 and instruction 4 are placed at the side entrance. Bookkeeping for side entrance points makes other compiler optimization phases more difficult. This can be avoided by removing side entrance points in traces.

Hwu et al. [35] gave a solution by introducing *superblocks* which have unique entrance and multiple exit points. Superblocks are built from traces, and tail duplication is performed to remove the side entrances into a trace. All blocks from the side entrance to the end of the trace

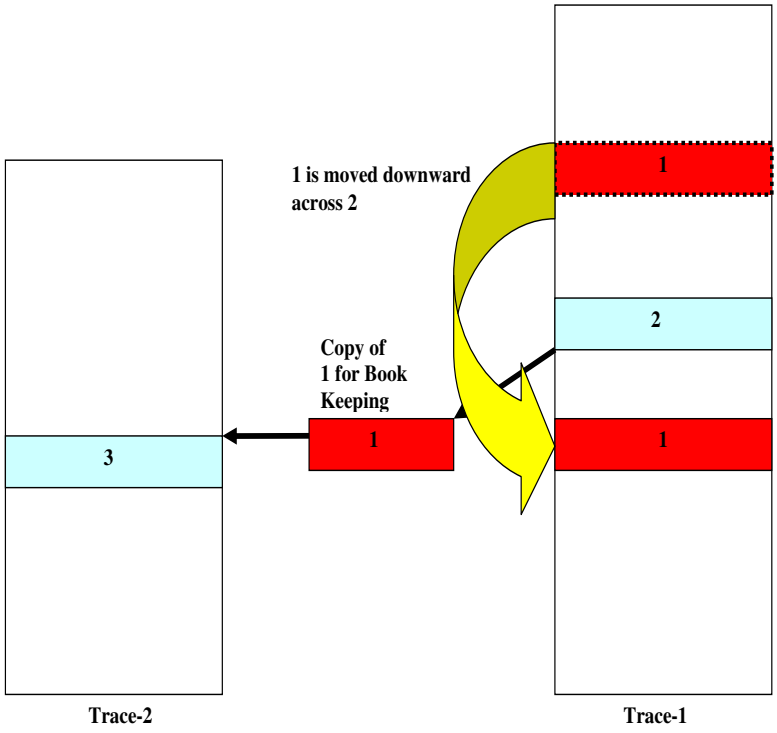


Figure 2.5: Bookkeeping for the downward movement across a side exit point: Instruction 2 is a side exit point for Trace-1 and instruction 3 is a side entrance point in Trace-2. Instruction 1 is moved downward across instruction 2. A copy of instruction 1 is placed between instruction 2 and instruction 3.

are duplicated, and all side entrances are redirected into the copy. Therefore, a superblock can have a single entry point but might have more than one exit point. Figure 2.7 shows the formation of a superblock. Basic blocks B_1, B_2 and B_4 form superblock S_1. Basic blocks B_3 and B_4' form superblock S_2. Section 2.5 describes issues related with superblock scheduling in more detail.

Mahlke et al. [46] introduced *hyperblocks*. A hyperblock is very similar to a superblock. The difference is that the instructions within a hyperblock are predicated instructions. While, a superblock contains only instructions from one path of control, a hyperblock combines basic block from multiple paths of control. Thus, optimizations using hyperblock are not biased towards any exit instruction. Hyperblocks require special support from the hardware, such as special registers, to execute predicated instructions. If-conversion replaces a set of basic blocks containing conditional control flow between the blocks with a single hyperblock of predicated instructions. Figure 2.8(b) illustrates a resultant flow graph after if-conversion is applied to the control flow graph in Figure 2.8(a).

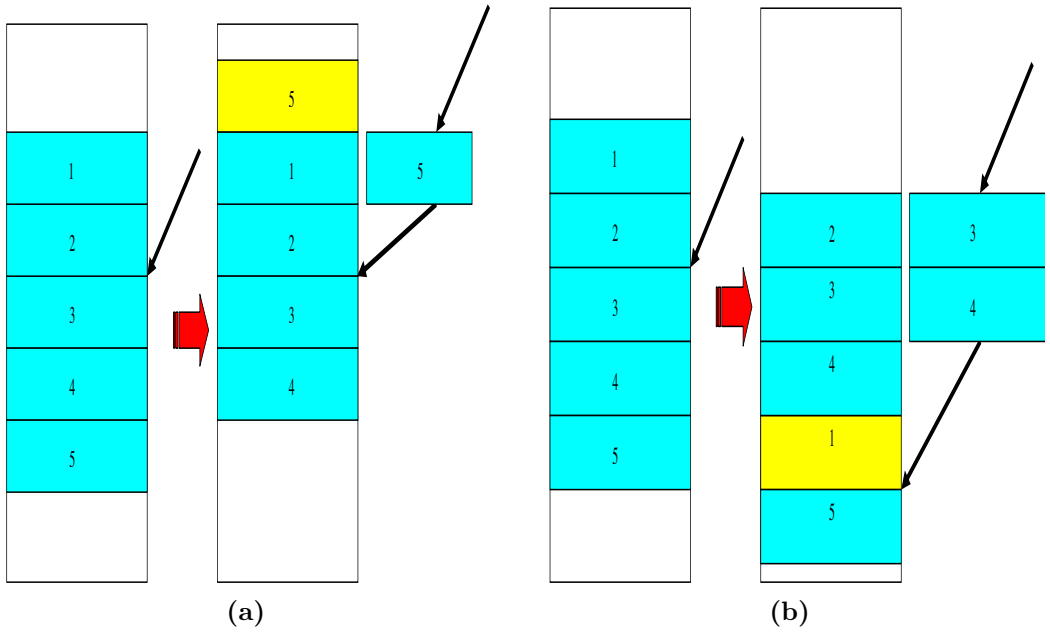


Figure 2.6: Bookkeeping: (a) upward movement across side entrance instruction; (b) downward movement across side entrance instruction.

For my work, I selected superblocks because they are simple, which makes them an attractive choice for global instruction scheduling in compilers [60].

2.3 Target machine architecture

There are two major architecture types, superscalar and VLIW (very large instruction word). Both allow ILP. The major difference between superscalar and VLIW is that superscalar does not parallelize instructions until runtime, while VLIW combines the instructions during compilation and issues them in a block or group. A superscalar processor is capable of executing instructions out of the sequence given by a compiler. This is known as *out-of-order* execution. A VLIW processor strictly follows the sequence of instructions given by a compiler. This is known as *in-order* execution. Both types of architecture benefit from reordering the instructions to improve the overall schedule cost.

In this thesis, I consider multiple-issue processors. On such processors, there are multiple functional units, and multiple instructions can be issued (begin execution) in each clock cycle. The maximum number of instructions that can be issued in one cycle is known as the processors' *issue width*. The issue width for a particular architecture must be less than or equal to the number of functional units. Associated with each instruction is a delay or *latency* between when

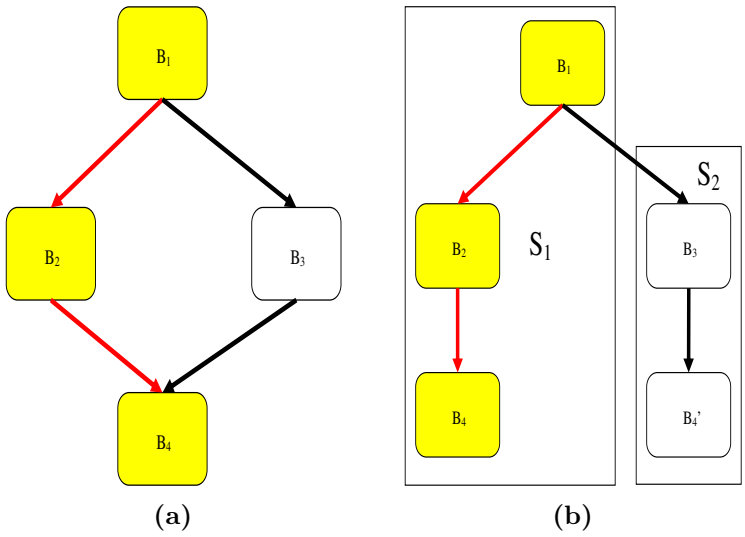


Figure 2.7: Superblock formation: B_i is a basic block in a CFG (a) Path $B_1 \rightarrow B_2 \rightarrow B_4$ has the highest probability of execution; (b) in order to remove the entrance from B_3 to path $B_1 \rightarrow B_2 \rightarrow B_4$, a copy of B_4 is created, called tail duplication, and the flow from B_3 is directed towards B_4'.

the instruction is issued and when the result is available for other instructions that use the result. I consider both fully pipelined and non-fully pipelined units. In a fully pipelined unit, one can begin a instruction on the unit on each cycle. In a non-fully pipelined functional unit one cannot begin a new instruction on the unit on each cycle. I assume that both instructions and functional units are typed and instructions of a given type only execute on one type of functional unit. Examples of types of instructions are load/store, integer, floating point, and branch instructions.

The IBM PowerPC 604 [36] is a superscalar multi-issue processor, which has the aforementioned architectural properties. The IBM PowerPC is the target machine architecture for this work. It has six functional units: a branch unit, two integer units for simple instructions, an integer unit for more complex instructions, a floating point unit, and a load/store unit for data transfer to and from memory. The PowerPC 604 has an issue width of 4, so not every functional unit will begin executing a new instruction every cycle. Most instructions are fully pipelined, and thus the PowerPC 604 can often dispatch a new instruction for execution each cycle on the same functional unit. However, some instructions are not fully pipelined and monopolize a functional unit for the entire duration of their execution. For example, a floating point division takes 36 cycles and during this period no other floating point instruction can be issued on the floating point functional unit [36].

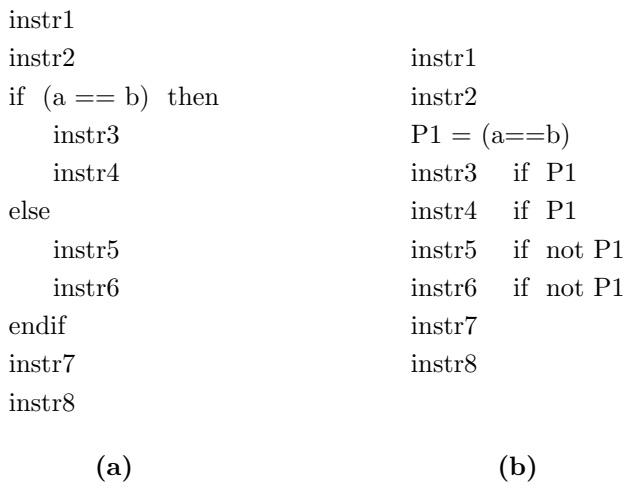


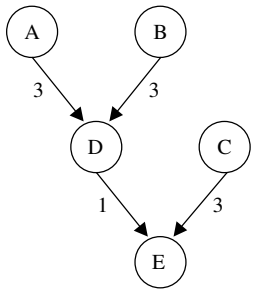
Figure 2.8: Hyperblock formation: (a) A region consisting of four blocks. Instruction *instr*1 and *instr*2 forms $B1$, *instr*3 and *instr*4 form $B2$, *instr*5 and *instr*6 form $B3$, *instr*7 and *instr*8 form $B4$. Basic block $B2$ and $B3$ are independent of each other. (b) The region is converted into a hyperblock using control registers $P1$ and $P2$. Basic block $B2$ and $B3$ are combined using $P1$ and $P2$. Control dependencies have been converted into data dependencies using $P1$ and $P2$.

2.4 Basic block scheduling

This section discusses instruction rearrangement within a basic block. I use the standard labeled directed acyclic graph (DAG) representation of a basic block. Each node corresponds to an instruction and there is an edge from i to j labeled with a non-negative integer $l(i, j)$ if j must not be issued until i has executed for $l(i, j)$ cycles. In particular, if $l(i, j) = 0$, j can be issued in the same cycle as i; if $l(i, j) = 1$, j can be issued in the next cycle after i has been issued; and if $l(i, j) > 1$, there must be some intervening cycles between when i is issued and when j is subsequently issued. These cycles can possibly be filled by other instructions.

The *critical-path distance* from a node i to a node j in a DAG, denoted $cp(i, j)$, is the maximum sum of the latencies along any path from i to j. A node i is a *predecessor* of a node j if there is a directed path from i to j; if the path consists of a single edge, i is also called an *immediate predecessor* of j. A node j is a *successor* of a node i if there is a directed path from i to j; if the path consists of a single edge, j is also called an *immediate successor* of i. A *sink* node is a node with no successors. For convenience, I assume that a fictitious sink node, hereafter called *the* sink node, is added to each DAG and that an edge is added from each node i in the DAG to the sink node, where the label on the edge is the latency of instruction i.

Given a labeled dependency DAG for a basic block, a *schedule* for a multiple-issue processor specifies an issue or start time for each instruction or node such that the latency constraints are



(a)

A	$r1 \leftarrow a$	
B	$r2 \leftarrow b$	
	NOP	
	NOP	
D	$r1 \leftarrow r1 + r2$	
C	$r3 \leftarrow c$	
	NOP	
	NOP	
E	$r1 \leftarrow r1 + r3$	

(b)

A	$r1 \leftarrow a$	
B	$r2 \leftarrow b$	
C	$r3 \leftarrow c$	
	NOP	
D	$r1 \leftarrow r1 + r2$	
E	$r1 \leftarrow r1 + r3$	

(c)

Figure 2.9: (a) Dependency DAG associated with the instructions to evaluate $(a + b) + c$ on a processor where loads from memory have a latency of 3 cycles and integer operations have a latency of 1 cycle; (b) non-optimal schedule for a single-issue processor; (c) optimal schedule.

satisfied and the resource constraints are satisfied. The latter are satisfied if, at every time cycle, the number of functional units that can execute a set of instruction types is greater than or equal to the number of instructions of those types issued at that cycle. The *length* of a schedule is the number of the cycle in which the sink node is issued.

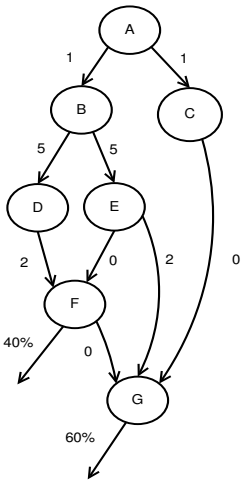
Definition 2.1 (Basic block instruction scheduling) *Given a labeled dependency DAG for a basic block, the* basic block instruction scheduling problem *is to find a schedule with minimum length.*

The basic block instruction scheduling problem for fully pipelined functional units is a special case of resource-constrained project scheduling (see, e.g., [20]) where all of the activities have unit execution times (i.e., unit latency) and we seek a schedule which minimizes the makespan.

Example 2.3 *Figure 2.9 shows a simple dependency DAG and two possible schedules for the DAG, assuming a single-issue processor that can execute all types of instructions. The schedule (b) requires four* NOP *instructions (null operations) because the values loaded are used by the following instructions. The better schedule (c), the optimal or minimum length schedule, requires only one* NOP *and completes in three fewer cycles.*

2.5 Superblock scheduling

This section discusses instruction rearrangement within a superblock. As for basic blocks, I use the standard labeled directed acyclic graph (DAG) representation for a superblock. There are



(a)

Time Slot	Instruction
1	A
2	B
3	C
4	NOP
5	NOP
6	NOP
7	D
8	E
9	F
10	G

(b)

Figure 2.10: A simple superblock and corresponding minimum cost schedule for a single issue processor. A NOP (No OPeration) instruction is used as a place holder when their are no instructions available to be scheduled.

some special nodes in the DAG of a superblock, known as *exit nodes*. Exit nodes represent branch instructions. Some weightage is associated with each exit node. The weightage represents the chance that the flow of control will leave the superblock through this exit point and is calculated using profiling. The weightage for an exit node e_i, denoted by w_i, is also known as the *exit probability*. Figure 2.10 shows a DAG for a superblock.

When scheduling a basic block in local instruction scheduling, the objective is to minimize the schedule length of the basic block. In the case of global scheduling with superblocks, the objective is to minimize the *weighted completion time* (WCT); i.e., the number of cycles from the entry point to each exit point, weighted by the exit probability. The weighted completion time is referred to as the cost function for the *superblock scheduling problem*.

Definition 2.2 (Weighted completion time) *The* weighted completion time *or cost of a superblock schedule is* $\sum_{i=1}^{n} w_i e_i$*, where* n *is the number of exit nodes,* w_i *is the weight of exit* e_i*, and* e_i *is the clock cycle in which exit* i *will be issued in the schedule.*

A *schedule* for a superblock is an assignment of a clock cycle to each instruction such that the precedence, latency and resource constraints are satisfied.

Definition 2.3 (Superblock instruction scheduling) *The superblock instruction scheduling problem is to construct a schedule with minimum weighted completion time.*

Example 2.4 *Consider the superblock shown in Figure 2.10. The two exits from the graph are from instructions F and G. Each exit is marked with a corresponding exit probability. The minimum cost schedule is shown for a single issue processor. Instruction F is scheduled at time cycle nine and instruction G is scheduled at time cycle ten. Thus, the cost of the schedule is* $\sum_{b \in B} w_b e_b = 0.40 \times 9 + 0.60 \times 10 = 9.60$ *clock cycles.*

2.6 List scheduling

Finding an optimal solution to both the local and global instruction scheduling problems is NP-complete [33]. Hennessy and Gross [33] were the first to give a non-optimal polynomial algorithm for the instruction scheduling problem with worst-case runtime of $O(n^4)$, where n is the number of instructions. Gibbons and Muchnick [51] improved the worst-case runtime to $O(n^2)$. This refined algorithm, known as the *list scheduling* algorithm, has become the most popular instruction scheduling algorithm, and is used almost exclusively in production compilers.

The list scheduling algorithm is so-called because of its use of a *ready list*. The algorithm iterates through machine cycles sequentially, and at each cycle it populates the ready list with the set of all *candidate* instructions which could begin execution in the current cycle. It then selects the best instructions from the ready list to begin execution in the current cycle, subject to resource constraints [51]. The algorithm also makes use of an *execution list*: whenever an instruction is issued, it is placed on the execution list, a list of all instructions currently being executed. When an instruction i finishes executing, any successor j of i becomes a candidate instruction as long as all other predecessors of j have also finished executing. The execution list is used to easily identify instructions that have finished executing, so it can quickly be determined if j may be added to the ready list. Algorithm 2.1 presents a formal representation of the list scheduling algorithm [27].

The method **selectBestInstruction** is the main part of the list scheduling algorithm. It returns the best instruction available in the ready list for the time cycle under consideration. A number of heuristics have been developed to select the best instruction. If there is no instruction among the ready instruction that can be issued in the current cycle, the method returns NOP. This process is continued until all instructions are scheduled.

2.6.1 Common scheduling heuristics

When the list scheduler chooses an instruction to be scheduled in the current cycle, it uses a *heuristic* to choose the best instruction from the ready list of instructions [51]. Each instruction has a set of *features*. A feature is a significant property of the instruction. Features may be *static* or *dynamic*. The values of static features do not change during the scheduling phase while the values of dynamic features may change during the scheduling. A list scheduling heuristic is a

Algorithm 2.1: List Scheduling algorithm.

```
input  : A DAG G = (V, E), issue width W, number of functional units f(t) of type t.
output: A valid schedule satisfying the precedence constraints of G and the architectural
        constraints of W and f(t) of type t.
cycle = 0;
ready-list = all source nodes in G;
execution-list = empty;
while ( ready-list or execution-list are not empty ) do
    op_i = selectBestInstruction( ready-list );
    while ( op_i is not null ) do
        remove op_i from ready-list and add to execution-list;
        for all instructions op_j such that (op_i, op_j) ∈ E do
            Add op_j to ready-list if op_j is ready to be executed;
        op_i = selectBestInstruction( ready-list );
    cycle = cycle + 1;
    for ( op_i = all nodes in execution-list ) do
        if ( op_i finishes in cycle cycle ) then
            remove op_i from execution-list;
            for all instructions op_j such that (op_i, op_j) ∈ E do
                Add op_j to ready-list if op_j is ready to be executed;
```

function that takes as input a pair of instructions, and based on the features of those instructions, gives a preference of one instruction over the other. Not all possible features for instructions are used in a heuristic. If two instructions agree on every feature in the heuristic, one is chosen arbitrarily.

Smotherman et al. [62] provide a survey of common scheduling heuristics used for local instruction scheduling. They also describe a large number of features that can be used for both local and global instruction scheduling. I describe here the features used in the heuristics I compare against my constraint programming model during experimentation. For more detail on these features see [62].

Critical path distance to sink: The critical path distance between two nodes in a DAG is defined as the maximum length path between the two nodes, where path length is the sum of latencies encountered along the path. Critical path distance to sink refers to the distance between any node and the sink in the DAG. This feature is what is generally meant by "critical path," and is labeled as such throughout the remainder of this thesis. For example, the critical path distance from node B to node E in the DAG in Figure 2.11 is 4.

Dependence height: The dependence height of a node in a DAG is the number of nodes on the longest path from the node to the sink [23].

Earliest starting time: A static estimate of the earliest cycle in which an instruction can begin execution. The root node has an EST of 1, the first cycle for scheduling. Any other node j has an EST of $\max\{l(i,j) + EST(i)\}$ for any parent i of j.

Instruction type: In my critical path heuristic, I favor floating point instructions above all other instructions and treat remaining instructions equally with respect to their instruction type, as done in [9].

Maximum latency: This feature is simply the maximum latency of an edge from a DAG node to any other DAG node. Maximum latency can also be thought of as the longest possible time that any other node in a DAG will have to wait for a result from the current node once the current node begins execution.

Number of successors: The total number of nodes reachable by following a single edge from the current node.

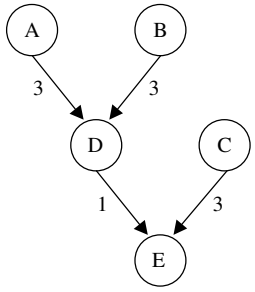


Figure 2.11: Critical path distance to sink.

Example 2.5 *Consider the basic block and its DAG shown in Figure 2.9. Table 2.1 and Table 2.2 show the step by step simulation of list scheduling for a single-issue processor and for a double-issue processor, respectively. The priority of a node is calculated by the critical path distance between the node and the sink of the basic block.*

Cycle	1	2	3	4	5	6
Candidates	A, B	B	D, C	D	D	E
Schedule	A	B	C		D	E

Table 2.1: Schedule for a *single-issue processor* for Example 2.5. Empty slot represents a NOP.

In Table 2.1 and Table 2.2, the first row shows the available cycles. The second row gives the available ready instructions against each cycle. The third row gives the actual instruction picked by the list scheduler.

Cycle	1	2	3	4
Candidates	A, B	C, D	D	E
Schedule	A, B	C	D	E

Table 2.2: Schedule for a *double-issue processor* for Example 2.5.

List scheduling is simple and efficient. Its main short coming is the list scheduling algorithm itself. As already said, the algorithm greedily constructs a single schedule from the solution space of all possible schedules based on a given priority heuristic. As a result, the schedule may not be the best possible schedule.

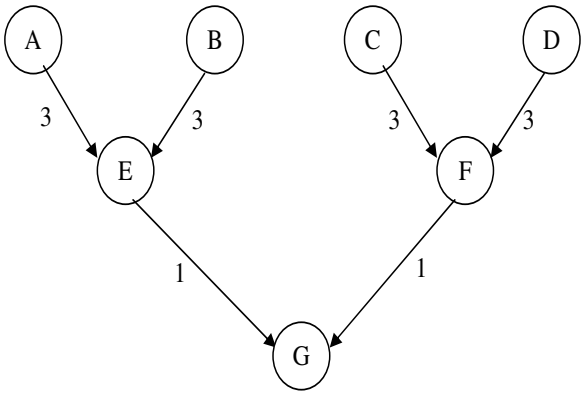
2.7 Instruction scheduling under register pressure constraint

Like instruction scheduling, register allocation is an important optimization phase in any compiler. A register is fast computer memory used to speed the execution of computer programs by providing quick access to commonly used values. Registers are limited in number. Almost every optimization phase assumes unlimited registers for its implementation. These unlimited registers are known as *symbolic registers*. The register allocation phase decides which symbolic register should be mapped to a real physical register.

Definition 2.4 (Register pressure) *The* register pressure *of an instruction schedule is the maximum number of variables alive simultaneously at any time slot in the schedule.*

Register allocation is often done using a graph coloring algorithm and an *interference graph*. An interference graph is a special graph for a given schedule of instructions in which each node represents a symbolic register in the given schedule and each edge indicates a pair of symbolic registers that cannot be assigned to the same register. Usually, this phase of optimization is done after the instruction scheduling phase. The main disadvantage of assigning registers first is the creation of false dependences in the code, limiting the possibilities to reorder instructions. Performing register allocation after instruction scheduling allows the greatest freedom to the instruction scheduler, but some bookkeeping has to be done in order to add spill code. Adding spill code after scheduling is a critical task, which must be done carefully to avoid degradation in performance. There is a chance that spill code insertion is unavoidable since instruction scheduling before register allocation tends to lengthen the live ranges of values, thereby increasing the register pressure. Example 2.6 explains the struggle between the two phases.

Example 2.6 *Figure 2.12 shows a simple dependency DAG associated with the instructions to evaluate* $(a + b) + (c \times d)$*. Assume a processor with two fully pipelined functional units capable of executing all types of instructions and that loads from memory have a latency of three cycles*



(a)

A	$r1 \leftarrow a$
B	$r2 \leftarrow b$
C	$r3 \leftarrow c$
D	$r4 \leftarrow d$
E	$r1 \leftarrow r1 + r2$
F	$r4 \leftarrow r4 \times r3$
G	$r1 \leftarrow r1 + r4$

(b)

Figure 2.12: (a) Dependency DAG associated with the instructions to evaluate $(a + b) + (c \times d)$; (b) assembly code for the DAG.

Cycles	Schedule S_1		Schedule S_2	
0	$r1 \leftarrow a$	$r2 \leftarrow b$	$r1 \leftarrow a$	$r2 \leftarrow b$
1			$r3 \leftarrow c$	$r4 \leftarrow d$
2				
3	$r1 \leftarrow r1 + r2$	$r3 \leftarrow c$	$r1 \leftarrow r1 + r2$	
4	$r2 \leftarrow d$			$r4 \leftarrow r4 \times r3$
5			$r1 \leftarrow r1 + r4$	
6				
7	$r2 \leftarrow r2 \times r3$			
8	$r2 \leftarrow r1 + r2$			

Table 2.3: Two possible schedules for the DAG in Figure 2.12. Empty slots represents NOPs.

and all other operations have latency of one cycle. Two possible schedules are shown in Table 2.3. Schedule S_1 is obtained by performing register allocation before instruction scheduling. Schedule S_2 is obtained by performing instruction scheduling before register allocation. Schedule S_1 has a register pressure of three, while S_2 has a register pressure of 4.

Which phase to run first is largely dependent on the target architecture and the code being compiled. One problem is that DAG graphs are different from interference graphs, and edges and nodes have different meanings. Therefore, it is not easy to combine the DAG and the interference graph, and new data structures have to be used to achieve good results. There are two approaches: integrated and cooperative. In an integrated approach, the phase ordering problem is solved by performing instruction scheduling and register allocation simultaneously. In a cooperative

approach, instruction scheduling and register allocation are done as separate phases but one phase is sensitive to the needs of the other. One method for cooperation is through register pressure in which instruction scheduling is done by keeping the register pressure within a certain range, which makes the register allocation phase more effective.

Chen [14] divides the register pressure, with respect to the instruction scheduling problem, into four groups. Consider an instruction sequence I with a schedule S:

1. *Register pressure of a given fixed schedule (RP)*: The register pressure RP is the maximum register pressure over all time slots of a given instruction schedule.

2. *Length-restricted minimum register pressure (LP)*: The length-restricted minimum register pressure LP is the minimum RP value for all schedules of I that are bounded to be of a schedule length less than or equal to the schedule length of S. The difference between RP and LP is known as excessive register pressure EP. EP is the pressure that we pay when we do not realize that there is an equivalent schedule with less register pressure. EP is the part that we should eliminate first.

3. *Minimum register pressure (MP)*: The minimum register pressure MP is the minimum RP value for all schedules of I. The difference between LP and MP is parallel register pressure PP. PP represents the register pressure that we pay to achieve a certain amount of parallelism.

4. *Constant register pressure (CRP)*: The constant register pressure CRP is the maximum number of register operands required by an instruction in I.

Definition 2.5 (Basic block instruction scheduling without spilling) *The basic block instruction scheduling without spilling problem is to construct a schedule with minimum length that has register pressure RP equal to or less than the number of available physical registers.*

Figure 2.13 explains the basic block scheduling without spilling. Point A represents an optimal schedule S_{op} for a given basic block without any register pressure constraint. Let P_a be the length-restricted minimum register pressure at point A, i.e., $P_a = LP(S_{op})$. Point B represents a schedule S with register pressure P_b such that $P_b = MP(S)$ and $P_a \geq P_b$. Beyond point B, the length-restricted minimum register pressure remains constant. Let P_c be the available number of physical registers. If $P_a \geq P_c \geq P_b$, then optimal schedule for the given basic block without any spilling will be a point between A and B, where register pressure is less than or equal to P_c. An optimal schedule without spilling ensures no spilling during the register allocation phase. Instruction scheduling in the case where $P_c \leq P_b$ is beyond the scope of this thesis, as this would require spilling during the register pressure phase.

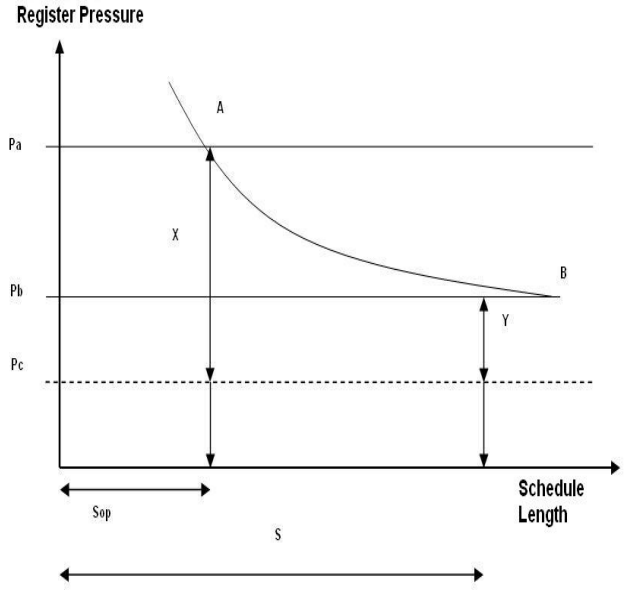


Figure 2.13: Breakdown of register pressure (from Chen [14]).

2.8 Constraint programming

This section is a short introduction to the constraint programming approach. For details on this area, consult [48, 58]. Constraint programming is the study of computational systems based on constraints. The idea of constraint programming is to solve a problem by stating constraints on acceptable solutions. The set of constraints on acceptable solutions is also referred to as a constraint satisfaction problem (CSP). In the past few years, constraint programming has attracted much attention from many areas including compiler optimization because of its potential for solving hard real-life problems.

Definition 2.6 (Constraint satisfaction problem) *A* constraint satisfaction problem *consists of a set of variables* $X = \{x_1, \cdots, x_n\}$*; for each variable* x_i*, a finite set* $domx_i$ *of possible values (its domain); and a set of constraints restricting the values that the variables can simultaneously take on.*

A solution to a CSP is an assignment of a value from its domain to every variable in such a way that all constraints are satisfied at once. Example 2.7 shows a simple application of constraint programming.

Example 2.7 *Graph coloring can be formulated as a CSP. Consider the graph in Figure 2.14 and suppose I wish to color each node of the graph red, yellow or blue such that no two adjacent nodes*

receive the same color. In the CSP, there would be a variable X_i for each node i, $i = 1, \ldots, 6$, and the domain of each variable would be $red, yellow, blue$. The following constraints capture that adjacent nodes should not be filled with the same color.

$$X_1 \neq X_2; X_1 \neq X_4; X_2 \neq X_4$$
$$X_2 \neq X_3; X_3 \neq X_4; X_3 \neq X_5$$
$$X_4 \neq X_5; X_5 \neq X_6; X_4 \neq X_6$$

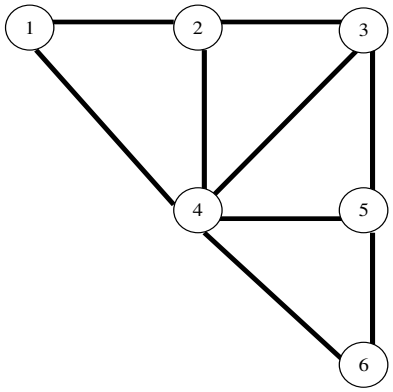


Figure 2.14: Graph for Example 2.7.

Solutions to a CSP can be found by searching through the possible assignments of values to variables. *Backtracking* is the usual method for solving CSP problems. It starts with an empty assignment set, i.e., no variables has been assigned a value and incrementally extends the solution by initializing the variables one by one in such a way that the constraints are not violated. This process is continued until a complete solution is found. If at any stage a partial solution violates any of the given constraints, backtracking is performed to the most recently initialized variable that still has alternative values available in the domain. Whenever a partial solution violates a constraint, backtracking eliminates a subspace from the Cartesian product of all variable domains. A backtracking algorithm can be improved by removing the values from the domains of variables which are not consistent with the constraints of a model. This process is known as constraint propagation. One form of constraint propagation is *b*ounds consistency.

Definition 2.7 (Bounds consistency) *A constraint C over the variables $x_1, ..., x_n$ with domains $dom(x_1), ..., dom(x_n)$ is bounds consistent with respect to x_i with domain $dom(x_i) = \{l, ..., r\}$ ($i \in \{1, ..., n\}$) iff:*

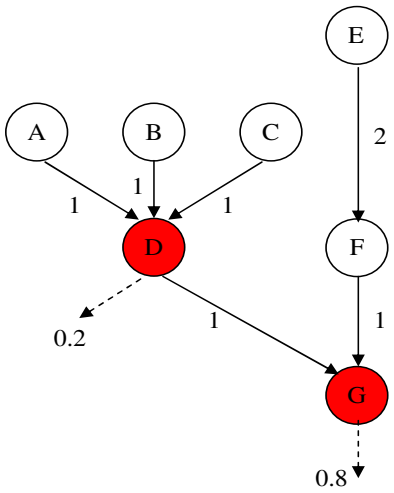
$$\exists d \in dom(x_1) \times ... \times dom(x_n) \text{ such that } d(x_i) = l \text{ and } d \in C$$

and

$$\exists d \in dom(x_1) \times ... \times dom(x_n) \text{ such that } d(x_i) = r \text{ and } d \in C.$$

Example 2.8 *Let* $x \in \{3, ..., 6\}$, $y \in \{2, 3\}$, $z \in \{5, ..., 9\}$, $x + y = z$ *is bounds consistent while* $x \in \{2, 3\}$, $y \in \{3, ..., 6\}$, $z \in \{1, ..., 19\}$, $3 \times x = y + z$ *is not bounds consistent.*

Example 2.9 explains how constraint programming can be used to model an instruction scheduling problem.



(a)

cycle	S	
1	A	E
2	B	C
3	D	F
4		G

(b)

Figure 2.15: (a) Superblock (taken from [49]) for Example 2.9. Nodes D and G are branch instructions; (b) a possible schedule for Example 2.9.

Example 2.9 *Consider the constraint model of the small instruction scheduling problem in Figure 2.15(a) with variables* $\mathrm{A}, \ldots, \mathrm{G}$, *each with domain* $\{1, 2, 3, 4\}$, *and the constraints,*

C_1: $\mathrm{D} \geq \mathrm{A} + 1$, C_3: $\mathrm{D} \geq \mathrm{C} + 1$, C_5: $\mathrm{G} \geq \mathrm{F} + 1$,
C_2: $\mathrm{D} \geq \mathrm{B} + 1$, C_4: $\mathrm{F} \geq \mathrm{E} + 2$, C_6: $\mathrm{G} \geq \mathrm{D} + 1$,
C_7: $gcc(\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, width = 2)$,

where constraint C_7, *a global cardinality constraint (gcc), enforces that at most two instructions can be issued in any cycle. The constraints are not bounds consistent. For example, the minimum value 1 in the domain of* D *does not have a support in constraint* C_1, C_2 *and* C_3 *as there is no corresponding values for* A, B *and* C *that satisfies the constraints. Enforcing bounds consistency using constraints* C_1 *through* C_6 *reduces the domains of the variables as follows:* $dom(\mathrm{A}) = \{1, 2\}$, $dom(\mathrm{B}) = \{1, 2\}$, $dom(\mathrm{C}) = \{1, 2\}$, $dom(\mathrm{D}) = \{2, 3\}$, $dom(\mathrm{E}) = \{1, 2\}$, $dom(\mathrm{F}) = \{3\}$,

and $dom(\mathrm{G}) = \{4\}$*. I am considering a dual-issue fully pipelined processor. Enforcing bounds consistency using* C_7 *reduces the domain of* D *to* $dom(\mathrm{D}) = \{3\}$*. Now, arbitrarily picking*[1] A *and* E *for the first cycle and enforcing bounds consistency using* C_7 *again will reduce the domains of variables as follows:* $dom(\mathrm{A}) = \{1\}$*,* $dom(\mathrm{B}) = \{2\}$*,* $dom(\mathrm{C}) = \{2\}$*,* $dom(\mathrm{D}) = \{3\}$*,* $dom(\mathrm{E}) = \{1\}$*,* $dom(\mathrm{F}) = \{3\}$ *and* $dom(\mathrm{G}) = \{4\}$*, which is schedule* S *given in Figure 2.15(b).*

2.9 Summary

In this chapter, I introduced the basic block and superblock instruction scheduling problems. I also introduced the basic block scheduling without spilling problem. I also covered the material necessary to understand the constraint programming techniques. In the next chapter, I discuss the related work on basic block instruction scheduling and I present my fast and optimal scheduler for basic blocks.

[1] In constraint programming, special heuristics are adopted to select variables.

Chapter 3

Basic Block Scheduling for Multi-Issue Processors

In this chapter, I present my constraint programming approach for basic block instruction scheduling for both idealized and realistic architectures. I discuss the related work in the field and present the experimental results using the constraint programming model with different architectures and compare it with the previous work.

3.1 Related work

Basic block instruction scheduling for realistic multiple-issue processors is NP-complete [33], and most compilers use a heuristic approach. The list scheduling algorithm is the most commonly used algorithm and is considered near optimal. Theoreticians have worked out some upper bounds on its optimality. R.L. Graham [29] proves that for n identical processors, an upper bound for the optimality of the list scheduling algorithm is $2 - 1/n$. Bernstein and Gertner [7] present a polynomial algorithm that exactly solves the instruction scheduling problem for the special case where the maximum latency, m, of any instruction is 2. Their result also acts as an approximation algorithm when $m > 2$, with the schedule produced having length within $2 - 2/m$ times that of an optimal schedule.

3.1.1 Approaches for optimal solutions

Previous work on optimal approaches to basic block instruction scheduling can be categorized by those approaches that are targeted only towards idealized architectural models and those

approaches which have been developed for more realistic architectural models. Broadly speaking, previous work has shown that (i) for an idealized single-issue processor, optimal approaches can scale up to the largest basic blocks which arise in practice, and (ii) for more realistic architectures, optimal approaches can be used but do not yet scale up. In my work, I present a constraint programming approach which applies to realistic architectures and scales up to the largest blocks. I now present previous work in more detail.

Wilken et al. [66] presented an integer programming approach for basic block scheduling. They tested their model on SPEC95 floating point benchmark. The basic blocks were obtained by compiling the benchmark using the Gnu Compiler Collection(GCC) with the highest level of optimization. The largest basic block contained up to 1200 instructions. The target architecture was an idealized single issue, fully pipelined processor, with a maximum instruction latency of 3. Their work shows that integer programming techniques scale well for a simple single issue architecture but is quite slow for large basic blocks. Their model was able to improve 0.39% of total basic blocks in terms of schedule length when compared with the list scheduler.

Van Beek and Wilken [65] presented a constraint programming model for the same idealized single issue processor. Constraint programming led to a simpler, more efficient solution. All basic blocks were optimally scheduled, and 0.39% of basic blocks were improved over list scheduling, as in [66]. My work builds on their approach.

Heffernan and Wilken [31] presented graph transformations to reduce the work required for the scheduler. They tested their transformation on SPEC 2000 and MediaBench benchmarks, compiled by the GCC with the highest level of optimization. The largest basic block contained up to 1200 instructions. In their work, they targeted idealized single-issue, 2-issue, and 4-issue processors with a maximum latency of 4. For each basic block, they compute a lower bound and use critical path list scheduling. They found that up to 13.2% of the non-trivial basic blocks were improved after graph transformations. But, only a small percentage of the evaluated basic blocks were non-trivial.

Ertl and Krall [21] developed an approach to instruction scheduling using constraint programming. Their approach is targeted towards the Motorola 88100 processor, which is a multi-issue RISC architecture with maximum instruction latency of six cycles. The work did not use the standard SPEC benchmark and compared the model against the GCC scheduler using five, relatively small applications. About 80% of the basic blocks scheduled by the GCC scheduler were found to be optimal. However, as our experiments confirm, the constraint model presented by the work does not scale beyond 50 instructions.

Kästner and Winkel [41] used an integer programming approach for the Intel Itanium architecture. They adopted a two-phase integer programming formulation to find an optimal solution. They implemented their approach in Intel Itanium compiler and compared it against their scheduler. They used nine benchmark applications from the SPEC95 suite. Their basic blocks were small in size with an average length of a basic block around 10 instructions. They found about

96% of the basic blocks scheduled by the list scheduler to be optimal.

Liu and Chow [44] presented an optimal scheduler using enumeration for the VISC architecture. The VISC has 4 functional units and a maximum instruction latency of 1. Liu and Chow compared their scheduler to a list scheduler with a benchmark of five network application developed in-house by Cognigine. A total of 487 basic blocks were scheduled, with an overall average of 9 instructions per basic block. The enumeration scheduler was guided by a critical path heuristic, and ties were broken by choosing the instruction involved in the most constraints and then the instruction with the highest number of successors. The list scheduler used the same heuristic as the enumeration scheduler. Their enumeration scheduler outperformed the list scheduler, producing 13.2-14.6% fewer cycles over an entire benchmark application.

Previous work on optimal approaches to basic block instruction scheduling can also be categorized by the approach taken, including branch-and-bound enumeration [15, 30, 31, 44], dynamic programming [42], integer linear programming [2, 11, 41, 43, 66], and constraint programming [21, 65]. However, with the exception of [31, 65, 66], to which I do detail comparisons later in this chapter, these previous approaches have only been evaluated on a few problems with sizes of the problems ranging between 10 and 50 instructions. Further, their experimental results suggest that none of them would scale up beyond problems of this size. A major challenge when developing an optimal approach to an NP-complete problem is to develop a solver that scales and is robust in that it rarely fails to find a solution in a timely manner on a wide selection of real problems. In this chapter, I present a constraint programming approach to basic block scheduling for multiple-issue processors that is robust and optimal. The novelty of my approach is in the extensive computational effort put into a preprocessing stage in order to improve the constraint model and thus reduce the effort needed in backtracking search.

I experimentally evaluated my optimal scheduler on the SPEC 2000 integer and floating point benchmarks, using four idealized architectural models and four realistic architectural models. The results for the idealized architectural models are presented to allow a comparison with previous work. On the SPEC 2000 benchmark suite, the optimal scheduler scaled to the largest basic block and was very robust. Depending on the architectural model, at most a few (between zero and 22) basic blocks out of the hundreds of thousands of basic blocks used in our experiments could not be solved within a 10-minute time bound. This represents a 50-fold improvement over work. As well, the scheduler was able to routinely solve the largest basic blocks that I found in practice, including basic blocks with up to 2600 instructions.

3.2 Constraint programming model for basic block scheduling

In this section, I present my constraint model of the basic block instruction scheduling problem. In the constraint programming methodology a problem is modeled in terms of variables, values, and constraints. The choice of variables defines the search space and the choice of constraints defines how the search space can be reduced so that it can be effectively searched using backtracking search.

I model each instruction by a variable with names $1, \ldots, n$ (we use i to refer interchangeably to variable i, instruction i, and node i in the DAG). The domain of each variable $dom(i)$ is a subset of $\{1, \ldots, m\}$ which are the available time cycles. Assigning a value $d \in dom(i)$ to a variable i has the intended meaning that instruction i will be issued at time cycle d. The domain $dom(i) = \{a, \ldots, b\}$ of a variable i is represented by the endpoints of the interval $[a, b]$. I use the notation $lower(i)$ and $upper(i)$ to refer to these endpoints.

I now specify the six types of constraints in the model: latency, resource, distance, predecessor and successor, safe pruning, and dominance constraints. Some of the notation I use is summarized in Table 3.1. As will be clear, for a minimal correct model of the instruction scheduling problem all that is needed are the latency and resource constraints. However, it is now well-established that adding implied (or redundant) constraints and dominance constraints to a constraint model can greatly improve the efficiency of the search for a solution (see, e.g., [63]). Implied constraints are constraints which do not change the set of solutions to the constraint model. Dominance constraints do not necessarily preserve the set of solutions but do preserve at least one of the solutions. Both types of constraints can increase the amount of constraint propagation and so cause the domains of the variables to be further restricted. In my context, adding the distance, predecessor and successor, safe pruning, and dominance constraints was found to be essential in improving the efficiency of the backtracking search for a schedule—without them, only small problems could be consistently solved. For example, for a single-issue architecture (the simplest version of the problem), the minimal model without any redundant constraints and dominance constraints does not scale beyond 40 instructions. With the redundant constraints and dominance constraints, the improved model scales up to instances with 2600 instructions (the largest that I have found in practice) on multiple-issue architectures. Many instances of each of these four types of constraints are added in an extensive preprocessing stage.

3.2.1 Latency constraints

Given a labeled dependency DAG $G = (N, E)$, for each pair of variables i and j such that $(i, j) \in E$, a latency constraint of the form $j \geq i + l(i, j)$ is considered for addition to the constraint model. A latency constraint is added if it is not redundant. A latency constraint

Table 3.1: Notation used in specifying the constraints.

$lower(i)$	lower bound of domain of variable i
$upper(i)$	upper bound of domain of variable i
$type(i)$	type of node/instruction i
k_t	number of functional units of type t
$l(i,j)$	latency on edge between nodes i and j
$cp(i,j)$	critical-path distance between nodes i and j
$d(i,j)$	lower bound on distance between nodes i and j
$onpath(i,j,t)$	set of all nodes of type t that are on some path from node i to node j. Note that $i \in onpath(i,j,t)$ if $type(i) = t$ and $j \in onpath(i,j,t)$ if $type(j) = t$. These are all of the instructions of type t that must be issued with or after node i is issued and must all be issued with or before node j is issued.
$pred(i)$	set of all immediate predecessors of node i
$succ(i)$	set of all immediate successors of node i
$pred(i,t)$	set of all immediate predecessors of node i that are of type t
$succ(i,t)$	set of all immediate successors of node i that are of type t
$I([a,b],t)$	set of all variables of type t whose domains intersect the interval $[a,b]$. These are all of the instructions of type t that may need these time cycles to execute on functional units of type t.

between i and j is redundant if there exists a $k < j$ such that, $l(i,j) \leq l(i,k) + cp(k,j)$. In other words, the constraint is redundant if there is a path from i to j that goes through k that is equal to or longer than the direct path $l(i,j)$. (If the constraint is redundant, adding it will have no effect as the remaining latency constraints will derive a stronger result.) Since I am enforcing bounds consistency, the actual form of the constraints added to the constraint model is,

$$lower(j) \geq lower(i) + l(i,j)$$

and its symmetric version,

$$upper(i) \leq upper(j) - l(i,j).$$

The latency constraints are easy to propagate when establishing lower and upper bounds for the variables, and easy to propagate incrementally during the backtracking search.

3.2.2 Resource constraints

For each type t of instruction/functional unit a resource constraint is needed to ensure that the number of instructions of type t issued at each time cycle does not exceed the number of functional units of type t. Such resource constraints are a special case of a well-studied constraint called the global cardinality constraint [57]. A global cardinality constraint over a set of variables and values states that the number of variables instantiating to a value must be between a given upper and lower bound, where the bounds can be different for each value. Here, for each type t a global cardinality constraint over all variables of type t is added to the constraint model, where all of the lower bounds are set equal to zero and all of the upper bounds are set equal to the number of functional units of type t. Note that when all of the upper bounds are set equal to one—in my case, when there is a single functional unit for some type t—the global cardinality constraint is equivalent to the well-known all-different constraint, which enforces that its arguments are pair-wise different.

Fast algorithms for enforcing bounds consistency on a global cardinality constraint have been proposed. In my implementation, I used the efficient algorithm presented in [45, 56]. The algorithm runs in time $O(t + n)$, where t is the time to sort the bounds of the domains of the variables and n is the number of variables. I note that for scheduling basic blocks, it has been shown that bounds consistency is dramatically better than other, more expensive, forms of consistency [45, 56].

3.2.3 Distance constraints

For each pair of nodes i and j, a distance constraint of the form $j \geq i + d(i, j)$ is considered for addition to the constraint model. A distance constraint is added if it is an improvement over the critical-path distance; i.e., $d(i, j) > cp(i, j)$. (If the distance is not greater than the critical-path distance, adding the constraint will have no effect as the latency constraints will derive a stronger result.) The distance constraints are lower bounds on the number of cycles that must elapse between when i is scheduled and j is scheduled. Although syntactically identical to latency constraints and hence propagated in the same manner, they are conceptually distinct and are key factors in effectively reducing the size of the search space. The distance constraints differ in that they take into account the architecture's resource constraints and can be much stronger than critical-path distances.

In what follows, I am interested in subgraphs called regions [66], which are induced from a given dependency DAG. Basic blocks typically contain many such regions embedded within them, with larger blocks containing many thousands.

Definition 3.1 (Region [66]) *A pair of nodes i, j in a DAG define a* region *if there is more than one path between i and j and there does not exist a node k distinct from i and j such that*

Table 3.2: Additional notation used in specifying the distance constraints.

$r_1(i,j,t)$	The minimum number of cycles that must elapse before the first instruction in $onpath(i,j,t)$ can be issued; i.e., $\min\{cp(i,k) \mid k \in onpath(i,j,t)\}$, the minimum critical-path distance from node i to any node in $onpath(i,j,t)$.
$r_2(i,j,t)$	The minimum number of cycles to issue all of the instructions in $onpath(i,j,t)$; i.e., $\lceil \lvert onpath(i,j,t) \rvert / k_t \rceil$, the size of the set of instructions divided by the number of functional units that can execute instructions of type t, rounded up to the next highest integer value.
$r_3(i,j,t)$	The minimum number of cycles that must elapse between when the last instruction in $onpath(i,j,t)$ is issued and node j can be issued; i.e., $\min\{cp(k,j) \mid k \in onpath(i,j,t)\}$, the minimum critical-path distance from any node in $onpath(i,j,t)$ to node j.

every path between i and j goes through k.

Given a region defined by nodes i and j, I wish to add a distance constraint $j \geq i + d(i,j)$, for some integer value $d(i,j)$. Following [66], if the region is small enough, I solve the region exactly (in isolation) and determine the optimal value for $d(i,j)$. To solve a region in isolation, I use the same constraint solver as for an entire basic block, but the constraint model is restricted to just the latency and resource constraints, plus any distance constraints that have been found so far. The regions in the DAG are examined in an "inside-out" manner so that distance constraints for inner regions can be used when solving larger outer regions.

For larger regions, I estimate the value, ensuring that my estimate is always less than or equal to the optimal value. I found that a threshold of 25 nodes worked well in practice; for regions larger than this the distance was estimated. Consider the notation shown in Table 3.2. For larger regions, initially I estimate $d(i,j)$ using,

$$d(i,j) = \max_t \{r_1(i,j,t) + r_2(i,j,t) + r_3(i,j,t) - 1\},$$

where I am finding the maximum over all instruction types t. Note that the nodes that are on a path from node i to node j can be determined quickly given the critical-path distances between all pairs of nodes, since a node k is on a path from i to j iff $cp(i,k) \geq 0$ and $cp(k,j) \geq 0$. The estimate of the distance can sometimes be improved by "removing" a small number of nodes (between one and three nodes) from $onpath(i,j,t)$. This was done whenever removing these nodes led to an increase in the value of $d(i,j)$; i.e., the decrease in $r_2(i,j,t)$ was more than offset by the increase in $r_1(i,j,t) + r_3(i,j,t)$. The estimate is a generalization and improvement over the distance constraints presented in [65], to handle multiple-issue, multiple types of instructions, and zero latency edges.

Example 3.1 *Consider the dependency DAG shown in Figure 3.1 where the clear nodes are of one instruction type and the shaded (yellow) nodes are of a different instruction type. Assume there is a single functional unit for each type of instruction. For the region defined by A and F, the initial estimate of the distance is* $d(\mathrm{A},\mathrm{F}) = 4$. *Similarly, for the region defined by A and G, the initial estimate of the distance is* $d(\mathrm{A},\mathrm{G}) = 5$. *The estimate of the distance* $d(\mathrm{A},\mathrm{G})$ *can be improved to* $d(\mathrm{A},\mathrm{G}) = 6$ *by "removing" node G from onpath*$(\mathrm{A},\mathrm{G},\mathrm{shaded})$. *The distance constraints* $F \geq A + 4$ *and* $G \geq A + 6$ *would be added to the constraint model, as both* $d(\mathrm{A},\mathrm{F})$ *and* $d(\mathrm{A},\mathrm{G})$ *are improvements over the critical-path distances between those nodes.*

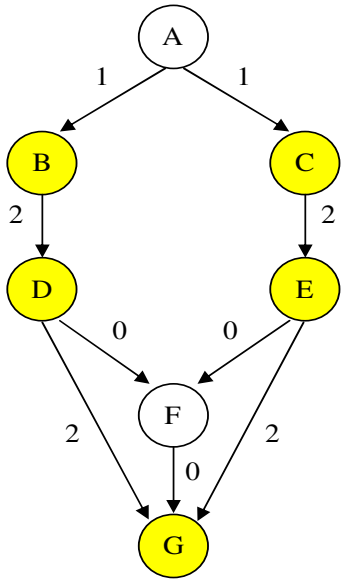


Figure 3.1: Example of adding distance constraints between nodes that define regions. The constraints $\mathrm{F} \geq \mathrm{A} + 4$ and $\mathrm{G} \geq \mathrm{A} + 6$ would be added to the constraint model.

3.2.4 Predecessor and successor constraints

For each node i which has more than one immediate predecessor, a single predecessor constraint of the following form is added,

$$\begin{aligned} \min(dom(i)) \quad \geq \quad & \min\{\min(dom(k)) \mid k \in P\} \\ & + \ \lceil |P|/k_t \rceil - 1 \\ & + \ \min\{l(k,i) \mid k \in P\} \end{aligned}$$

for every type t and every subset P of $pred(i,t)$ where $|P| > k_t$,

where the operator $\lceil x \rceil$ returns the smallest integral value not less than x. It can be seen that a predecessor constraint can be propagated in $O(|pred(i)|^2)$ time by first sorting the predecessors of i by increasing lower bounds and then stepping through the lower bounds, each time finding the minimum latency among the remaining predecessors. A symmetric version, called successor constraints, for the immediate successors of a node is given by,

$$\begin{aligned} \max(dom(i)) \quad \leq \quad & \max\{\max(dom(k)) \mid k \in P\} \\ & - \lceil |P|/k_t \rceil + 1 \\ & - \min\{l(i,k) \mid k \in P\}, \end{aligned}$$

for every type t and every subset P of $succ(i,t)$ where $|P| > k_t$.

The predecessor and successor constraints are propagated in a preprocessing stage and also during search. They can be viewed as an adaptation of edge-finding rules (see [4]) and are an easy generalization of the similarly named constraints presented in [65] to handle multiple-issue and multiple types of instructions.

Example 3.2 *Consider the partial DAG shown in Figure 3.2, where the domains of the variables are as shown. Assume there is a single functional unit for each type of instruction. Propagating the predecessor constraint associated with node E improves the lower bound of the variable. The earliest that the set $P = \{C, D\}$ of immediate predecessors of node E can be scheduled is cycle 8, and, therefore, cycle 9 is the earliest that the last of its predecessors could be scheduled. Therefore, the earliest that E can be scheduled is cycle 11.*

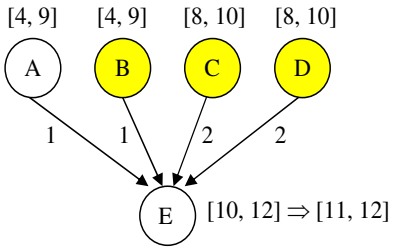


Figure 3.2: Example of improving the lower bound of a variable using a predecessor constraint.

3.2.5 Safe pruning constraint

Given a constraint model, I say that it is *safe* to add a constraint to a constraint model whenever it is the case that, if there was a solution to the constraint model before adding the constraint, there is still a solution after adding the constraint. Adding safe pruning constraints is based on the following theorem.

Theorem 3.1 *Suppose that all of the latency and resource constraints have been propagated. If there exists an interval* $[a, b]$ *such that,*

(i) for all $i \in I([a, b], t)$, $\min(dom(i)) = a$,

(ii) for all $i \in I([a, b], t)$, *for all* $k \in pred(i)$, $\max(dom(k)) + l(k, i) \leq \min(dom(i))$,

(iii) $\mid I([a, b], t) \mid \quad \leq \quad (b - a + 1) \times k_t$,

then it is safe to prune the upper bounds of the variables $i \in I([a, b], t)$ *as follows,*

$$\max(dom(i)) = \min(\max(dom(i)), b).$$

Proof. Suppose there was a solution to the constraint model before pruning. Call this the original solution. There are two cases.

1. Suppose that in the original solution each variable in $I([a, b], t)$ is assigned a value from its domain that is less than or equal to b. Clearly this is still a solution after pruning.

2. Suppose that in the original solution there exist variables in $I([a, b], t)$ that have been assigned values from their domains that are greater than b. We will show that each of these variables can be given a consistent value from $[a, b]$.

    a. *Latency constraints*: I will show that any value in $[a, b]$ satisfies the latency constraints. Let i be any variable that has been reassigned a value. Let k be an immediate predecessor of i and consider the latency constraint $k + l(k, i) \leq i$. Lowering the value of i cannot violate the constraint since $\max(dom(k)) + l(k, i) \leq \min(dom(i))$ (by condition (ii)) and I assumed that the latency constraints have been propagated. Thus, any value in the domain of i will satisfy this constraint. Let k be an immediate successor of i and consider the latency constraint $i + l(i, k) \leq k$. Lowering the value of i cannot violate this constraint.

    b. *Resource constraints*: I will show that it is possible to reassign values to these variables from $[a, b]$ and satisfy the relevant resource constraint. Condition (i) implies that before pruning there is no variable i of type t such that $\min(dom(i)) < a$ and $a \leq \max(dom(i))$; i.e., before pruning there is no variable whose domain intersects both $[c, a - 1]$ and $[a, d]$ where $c < a \leq d \leq b$. I also know that after pruning there is no variable whose domain intersects both $[c, b]$ and $[b + 1, d]$ where $a \leq c \leq b < d$. This means that I can look at the resource constraint over the variables in $I([a, b])$ in isolation; whatever values are assigned to the variables in this set cannot impact the values that variables outside of this set can be assigned. Condition (iii) ensures there are enough values so that all of the variables in $I([a, b], t)$ can be assigned a value such that the resource constraint is satisfied.

□

Corollary 3.1 *Suppose that all of the latency and resource constraints have been propagated. If there exists an interval* $[a, b]$ *such that,*

(i) for all $i \in I([a, b], t)$, $\max(dom(i)) = b$,

(ii) for all $i \in I([a, b], t)$, *for all* $k \in succ(i)$, $\max(dom(i)) + l(i, k) \leq \min(dom(k))$,

(iii) $\mid I([a, b], t) \mid \quad \leq \quad (b - a + 1) \times k_t$,

then it is safe to prune the lower bounds of the variables $i \in I([a, b], t)$ *as follows,*

$$\min(dom(i)) = \max(\min(dom(i)), a).$$

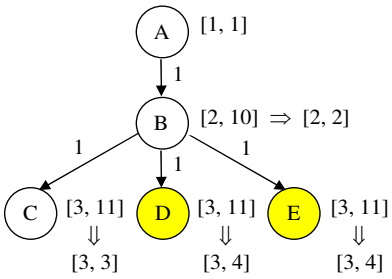


Figure 3.3: Improving bounds of variables using safe pruning constraints.

Example 3.3 *Consider the partial DAG shown in Figure 3.3, where the domains of the variables are as shown. Assume there is a single functional unit for each type of instruction. The safe pruning constraint can be applied iteratively as follows. First, the interval [2,2], where* $I([2, 2], \text{clear}) = \{B\}$, *satisfies the theorem. Hence, node B can have its domain pruned to [2,2]. Second, the interval [3,3], where* $I([3, 4], \text{clear}) = \{C\}$, *now satisfies the theorem. Hence, node C can have its domain pruned to [3,3]. Third, the interval [3,4], where* $I([3, 4], \text{shaded}) = \{D, E\}$, *also now satisfies the theorem. Hence, nodes D and E can have their domains pruned to [3,4].*

3.2.6 Dominance constraints

Heffernan and Wilken [31] present a set of graph transformations for dependency DAGs for basic blocks and show that optimally scheduling the transformed DAGs using branch-and-bound enumeration is faster and more robust. The DAG transformations reduce the search space while preserving optimality. I found that adaptation of these transformations also worked well in my

constraint programming approach. In my context, the transformations add simple constraints to the model of the form $i \geq j$, which I call dominance constraints.

In what follows, I am interested in pairs of disjoint, isomorphic subgraphs A and B induced from a given dependency DAG. Subgraphs A and B are isomorphic if there is a mapping from the node set of A to the node set of B such that A and B are identical (identical instruction types, edges, and latencies on the edges).

Using terminology similar to that for the safe pruning constraint, I say that it is *safe* to add a constraint to a constraint model whenever it is the case that, if there was a solution to the constraint model before adding the constraint, there is still a solution after adding the constraint. Adding dominance constraints, when it is safe to do so, is based on the following theorem.

Theorem 3.2 (Heffernan and Wilken [31]) *Let A and B be isomorphic subgraphs with node sets $V(A) = \{a_1, \ldots, a_r\}$ and $V(B) = \{b_1, \ldots, b_r\}$. If,*

(i) a_i is neither a predecessor or a successor of b_i, $1 \leq i \leq r$,

(ii) for all $k \in pred(a_i)$ such that $k \notin V(A)$, $l(k, a_i) \leq cp(k, b_i)$, $1 \leq i \leq r$,

(iii) for all $k \in succ(b_i)$ such that $k \notin V(B)$, $l(b_i, k) \leq cp(a_i, k)$, $1 \leq i \leq r$,

(iv) for any edge (b_i, a_j), $l(b_i, a_j) \leq cp(a_i, b_j)$,

then adding the constraints $a_i \leq b_i$, $1 \leq i \leq r$ is safe.

Example 3.4 *Consider the DAG shown in Figure 3.4a. Dominance constraints can be added iteratively as follows. First, the subgraphs with nodes $V(A) = \{B, D\}$ and $V(B) = \{C, E\}$ are isomorphic and satisfy the conditions of the theorem. Hence, the constraints $B \leq C$ and $D \leq E$ can be added to the model. Adding these constraints updates the critical path distances. In particular, $cp(\mathrm{D}, \mathrm{E})$ was $-\infty$ and is now 0. Second, the subgraphs with nodes $V(A) = \{F\}$ and $V(B) = \{E\}$ are isomorphic and now satisfy the conditions of the theorem. Hence, the constraint $F \leq E$ can be added to the model.*

Heffernan and Wilken [31] find isomorphic subgraphs that satisfy the theorem using backtracking search with a time cutoff. The search starts with isomorphic subgraphs that consist of single nodes (i.e., they have the same instruction type) that satisfy condition (i) of the theorem and either condition (ii) or condition (iii). These nodes are called seed nodes. The backtracking search expands these subgraphs to adjacent nodes, maintaining isomorphism, until either (a) all of the conditions of the theorem are satisfied (in which case, dominance constraints can be added), or (b) the subgraphs cannot be expanded any further or the time cutoff is reached (in which case, this pair of seed nodes leads to failure and we try another pair of seed nodes).

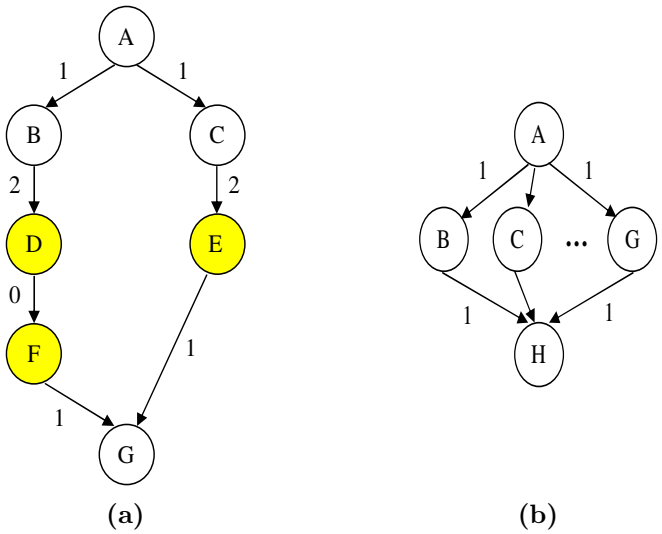


Figure 3.4: Examples of adding dominance constraints: (a) (adapted from [31]) the constraints B $\leq$ C, D $\leq$ E, and F $\leq$ E would be added to the constraint model; (b) the constraints B $\leq$ C, C $\leq$ D, ..., F $\leq$ G would be added to the constraint model.

In my work, I find isomorphic subgraphs by focusing on regions (see Definition 3.1). Given a region defined by nodes i and j, I conceptually remove the source node i and the sink node j of the region and perform a depth-first search to find the separate components or subgraphs of the region. I then check whether pairs of components are isomorphic and satisfy the conditions of the theorem (or can be made to do so by dropping a few nodes). I focus on separate components of regions as during the backtracking search for a solution, often both orderings of these components must be tried to verify that there is no solution. Thus, the dominance constraints, by establishing an ordering on the variables between these components, can greatly reduce the search space.

Testing sub-graph isomorphism is NP-complete in general. Here, a fast heuristic test is used to determine whether two components are isomorphic. The nodes in each component are independently sorted based on features of the nodes, and the order of the nodes constitutes a potential isomorphism mapping, which is then verified. Observe that whenever the heuristic (sort) test returns true, the pair of subgraphs is isomorphic, and that sometimes the heuristic returns false even though there exists a true mapping. However, experimental evidence suggests that the heuristic works well. Consider the following sets S_1 and S_2, where $S_1 \subseteq S_2$. Construct the first set S_1 as follows. For all pairs of components, add only those pairs to S_1 that pass the heuristic test. This gives some of the pairs of components that are isomorphic (although it may miss some); i.e., S_1 is a subset of the set of all isomorphic pairs of components. Construct the second set S_2 as follows. For all pairs of components, add only those pairs to S_2 that have the same numbers of instructions of each instruction type. This gives the pairs of components that are *potentially*

isomorphic (although some may not be); i.e., S_2 is a superset of the set of all isomorphic pairs of components. I found that the difference $S_2 - S_1$ was most often empty and always small, thus providing evidence that the heuristic test catches almost all isomorphic pairs of components.

A special case of the theorem was found to occur often in practice. Consider the DAG shown in Figure 3.4b where the region defined by A and H contains many nodes all of the same type and all at the same latencies. All of these nodes are symmetric and the dominance constraints that would be added are equivalent to so-called symmetry-breaking constraints. I recognize this special case as follows. For each instruction type t, we sort the variables by their lower bounds, and then step through all instructions with the same lower bound and check if the pairs of nodes satisfy the theorem. If so, dominance constraints are added.

Overall, I found that my techniques often discovered many pairs of components within a basic block that satisfied the theorem, sometimes with several hundred nodes each. I also found that the dominance constraints that were added greatly improved the efficiency of the search for a schedule, thus providing additional evidence for the effectiveness of the graph transformations proposed by Heffernan and Wilken [31].

3.3 Additional constraints for realistic architectures

The model presented in Section 3.2 assumes the issue width is equal to the number of functional units in a processor and that each unit is fully pipelined. My group extended the model to a more realistic architecture. Chase [12] introduced *issue width, non-fully pipelined and serialize instruction constraints* for a realistic architecture. I will briefly discuss these constraints here. Definitions and examples are taken from [12].

3.3.1 Issue width constraint

On many architectures, such as the IA-64 [41] and PowerPC [36], the issue width does not equal the number of available functional units. Adding an issue width constraint is a straight-forward modification to the initial model. The initial model already uses global cardinality constraints to ensure that the number of instructions of any type t issued each cycle does not exceed $f(t)$, the number of functional units of that type. To ensure that solutions are consistent with respect to the issue width, we added a global cardinality constraint involving all variables.

3.3.2 Non-fully pipelined processor constraint

Almost all modern architectures are not fully pipelined, including the Intel Pentium and Itanium and the PowerPC architectures. A fully pipelined architecture requires every instruction to have an

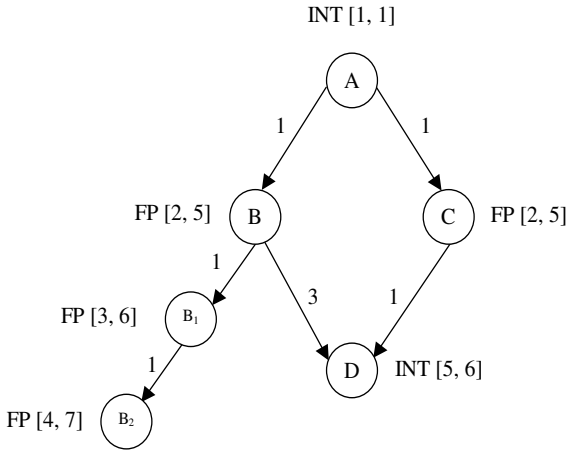


Figure 3.5: Example DAG with additional nodes B_1 and B_2 corresponding to pipeline variables.

execution time of 1. On most architectures, some instructions require significantly more processing time than the majority. For example, compare integer addition with floating point square root on the PowerPC 604: addition takes one cycle while square root takes 32. In order for every instruction to have an execution time of one cycle, the cycle time would have to be long enough for the longest instruction in the architecture to complete. In each cycle where one of those long instructions was not scheduled, computing power would be wasted. To be more efficient, most architectures are not fully pipelined, and so there will be cycles in which instructions cannot be issued on a particular functional unit, since the unit will still be executing a previously-issued instruction. To model this feature, Chase [12] introduced *pipeline variables*, special variables that are added to the CSP.

Definition 3.2 (Pipeline variables) Suppose an instruction i with corresponding CSP variable X_i has bounds $[a_i, b_i]$ and execution time $e(i)$, with $e(i) > 1$. Insert variables $p_{i,j}$ into the CSP, for $1 \leq j \leq e(i) - 1$. Each variable $p_{i,j}$ is of functional unit type $u(i)$. $p_{i,j}$ has bounds $[a_i + j, b_i + j]$. Also add all pipeline variables of type t to the functional unit constraint for type t.

Example 3.5 The DAG in Figure 3.5 has nodes B_1 and B_2 added to illustrate the use of pipeline variables. Suppose the target architectural model has one floating point functional unit and that instruction B is scheduled in cycle 2. As nodes B_1 and B_2 correspond to pipeline variables, they must be issued on the floating point unit in cycles 3 and 4, and instruction C cannot be issued until instruction 5.

3.3.3 Serializing instruction constraint

A *serial schedule* is a schedule in which only one instruction is issued per cycle and no two instructions execute at the same time. Instruction scheduling may produce non-serial schedules when scheduling for architectures with an issue width greater than one, but the behavior of the scheduled code must be exactly equivalent to the behavior of a serial schedule. Access to architectural resources may force a schedule to be partially serialized, or for only one instruction to be issued in a given cycle if the instruction has certain properties. For example, architectures may have only one of a particular resource, such as the condition register on the PowerPC 604 [37], and need to ensure that only one instruction is accessing that resource at a time. Some architectures may enforce instruction ordering on the processor, by stalling some instructions until it is safe to issue them. Other architectures, including the PowerPC [36, 37] and Intel Pentium [40] and Itanium [38], require order to be enforced by the compiler, either by providing instructions that serialize the processor or by creating a serial-friendly schedule.

The PowerPC Compiler Writer's Guide [36] describes four types of instructions that require some sort of serialization and occur on the PowerPC architecture. This type of serializing instruction, labeled *execution serialization* in PowerPC literature [36, 37], describes a set of instructions that require exclusive access to the processor in the cycle in which they are issued. These instructions are held in a queue on the processor until they are the oldest uncompleted instruction on the processor (in other words, until all previously executing instructions have completed), and then they are issued. In the cycle in which they are issued, no other instruction can be issued, meaning that for one cycle, the instruction has sole access to the processor and its resources. Instructions having these exact properties will be referred to as *serializing instructions* throughout this thesis.

Serializing instructions can be modeled in a CSP in a manner similar to that of instructions with a execution time larger than one. Chase [12] introduced a *serial variable* to implement serializing instruction constraint.

Definition 3.3 (Serial variables) Suppose an instruction i with corresponding CSP variable X_i has bounds $[a_i, b_i]$ and represents a serial instruction. Insert variables $s_{i,j}$ into the CSP, for $1 \leq j \leq F-1$, where F is the total number of functional units. There is one serial variable for every functional unit except for the one on which instruction i is issued; the functional unit type of each serial variable is assigned accordingly. $s_{i,j}$ has bounds $[a_i, b_i]$. Also add all serial variables of type t to the functional unit constraint for type t.

Example 3.6 Consider the DAG in Figure 3.6, where node B corresponds to a serial instruction. Suppose the target architectural model has an issue width of 2. If neither B nor C corresponded to serial instructions, they could both be issued in cycle 2, allowing D to be issued in cycle 3. However, since B corresponds to a serial instruction, no other instruction can be issued in the

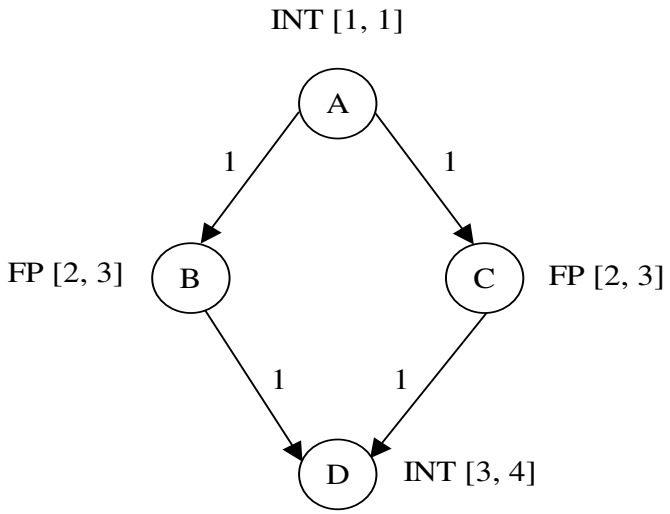


Figure 3.6: Example DAG with serial instruction B.

same cycle as B. Thus, if B is issued in cycle 2, C must be issued in cycle 3 and D must be issued in cycle 4.

3.4 Solving an instance

Solving an instance of an instruction scheduling problem is divided into several phases.

In phase one, I construct the constraint model and use the constraints to establish the lower bounds of the variables and a lower bound on the length m of an optimal schedule. Given m, the upper bounds of the variables are similarly established and the constraint model is passed to the backtracking algorithm. The backtracking search interleaves constraint propagation with branching on variables. During constraint propagation, bounds consistency is enforced on the constraints until no further changes result. A dynamic variable ordering is used that selects as the next variable to instantiate the variable with the least number of values remaining in its domain, breaking ties by choosing the variable that participates in the most constraints. Given a selected variable x, the backtracking search first branches on x assigned to $lower(x)$, then on x assigned to $lower(x)+1$, and so on, until either a solution is found or the domain of x is exhausted. If no solution is found, a length m schedule does not exist and the value of m is incremented, the upper bounds of the variables are re-established using the new value of m, and the new constraint model is passed to the backtracking algorithm. This is repeated, each time incrementing m until a solution is found, an upper bound on the length of a schedule is reached, or a time limit is exceeded. An upper bound on the length of a schedule is established by running a list-scheduling

algorithm using a critical-path heuristic. If a solution is found or the upper bound on the length of a schedule is reached, a provably optimal solution has been found. If, instead, the time limit is exceeded, I proceed to phase two of the solution process.

In phase two, the level of constraint propagation during backtracking search is increased to a variation of singleton consistency [18]. In singleton consistency, a variable is temporarily instantiated to a single value and the constraint model is tested for consistency. If the consistency test fails, the value can be removed from the domain of the variable. In my work, I iteratively instantiated and tested the consistency of the lower and upper bounds of the domains of the variables. The consistency test consisted of enforcing bounds consistency on the constraints. I found that singleton consistency sometimes dramatically reduced the domains of the variables during search. As well, when testing the consistency of the bounds, I record the number of changes that are made during the bounds consistency propagation. This information is used in phase two to select the next variable to branch on. The goal is to branch on a variable that causes the most reductions in the domains of the other variables. As for phase one, if a solution is found or the upper bound on the length of a schedule is reached, a provably optimal solution has been found.

In phase three, the level of constraint propagation during backtracking search is increased once again to perform singleton consistency to a depth of two. Each variable is temporarily instantiated to a single value and I test whether the constraint model is singleton consistent. This level of propagation is expensive and is viable only for smaller but difficult basic blocks.

In my experiments, I found that the following scheme worked best for stepping through the phases. First, if the basic block contains 300 or fewer instructions, phase one is allocated 5 seconds, phase two is allocated 15 seconds, and the remaining time is allocated to phase three. Second, if the basic blocks contains more than 300 instructions, phase one is allocated 5 seconds and the remaining time is allocated to phase two.

3.5 Experimental evaluation

Now I present an experimental evaluation of the performance of my optimal scheduler for both idealized and realistic architectures.

The constraint programming model was implemented and evaluated on all of the basic blocks from the SPEC 2000 integer and floating point benchmarks (www.spec.org). The benchmarks were compiled using IBM's Tobey compiler [9] targeted towards the IBM PowerPC processor [36], and the basic blocks were captured as they were passed to Tobey's instruction scheduler. The basic blocks contain four types of instructions: branch, load/store, integer, and floating point. The range of the latencies is: all 1 for branch instructions, 1–12 for load/store instructions (the largest value is for a store-multiple instruction, which stores to memory the values in a sequence of registers), 1–37 for integer instructions (the largest value is for division), and 1–38 for floating

point instructions (the largest value is for square root). The Tobey compiler performs instruction scheduling before global register allocation and once again afterward, and our test suite contains both versions of the basic blocks. The compilations were done using Tobey's highest level of optimization, which includes aggressive optimization techniques such as software pipelining and loop unrolling.

3.5.1 Experiments for idealized architectural models

The following table shows the four idealized architectural models I used in my evaluation. Consistent with previous work, I assumed that all functional units were fully pipelined and that the issue width of the processor was equal to the number of functional units.

1-issue	processor executes all types of instructions.
2-issue	processor with one floating point functional unit and one functional unit that can execute integer, load/store, and branch instructions.
4-issue	processor with one functional unit for each type of instruction.
6-issue	processor with the following functional units: two integer, one floating point, two load/store, and one branch.

The optimal constraint programming scheduler was compared experimentally with list scheduling, the most popular heuristic method for scheduling basic blocks in compilers [27]. List scheduling is a greedy algorithm which uses a heuristic for which instruction to schedule next. Following Muchnick [51], my heuristic used critical-path distance as the primary feature and earliest start time as a tie-breaker. Although a popular heuristic, the primary reason for adopting this heuristic is that critical-path heuristics were also used in previous work [31, 65, 66], thus allowing a fairly direct comparison of previous experimental results with our experimental results.

Wilken, Liu, and Heffernan [66] and van Beek and Wilken [65] present experimental results for a 1-issue processor. Note that, although both of these solvers could solve all of the basic blocks in the SPEC95 floating point benchmarks in seconds, when the solver in [65] was applied to the current test suite of basic blocks, hundreds of problems could not be solved. I speculate that the current test suite contains more difficult problems for the following four reasons. First, the current test suite contains longer and more varied latencies (in [66], the latencies were uniformly 1 for integer instructions, 2 for floating point instructions, and 3 for memory instructions). Second, the current test suite contains shorter latencies (our DAGs contain many latency 0 edges, which are used to capture anti-dependencies and output dependencies between two instructions). Third, the current test suite contains many larger basic blocks (previous work used the GCC compiler and the largest DAG was approximately 1000 instructions). Fourth, the current test suite contains blocks from both before and after register allocation (previous work only used blocks from after register allocation).

Heffernan and Wilken [31] were the first to present experimental results on solving large basic

blocks targeted towards a multiple-issue processor. Their test suite contains the basic blocks from the SPEC 2000 floating point benchmarks (with the Fortran90 benchmarks omitted) and are from after register allocation. They report the number of basic blocks where their optimal scheduler failed to complete within a time limit of 100 seconds. In their worst case, a 2-issue processor model, their optimal solver failed on over 200 basic blocks. They used a 3-GHz Pentium 4 processor with 512 MB of main memory for experimentation. If I restrict my experimental results to the same benchmarks and the same time limit, my optimal solver failed on only 4 basic blocks, a 50-fold improvement. For my experimentation, I used a 2.40 GHz Pentium 4 processor with 1GB of main memory.

Table 3.3 gives number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of basic blocks with improved schedules (%), for various idealized architectural models. Table 3.4 gives average and maximum percentage improvements in schedule length of optimal schedule over schedule found by list scheduler using the heuristic, for various idealized architectural models. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule. To systematically study the scaling behavior of the optimal scheduler, I report the results broken down by increasing size ranges of the basic blocks as well. For reference, the number of basic blocks in each size range is also given (see Tables 3.5 & 3.6). It can be seen that the optimal scheduler scales well, finding improved solutions for large basic blocks. Not surprisingly, as the basic block size increases, the heuristic method has more opportunities to make a mistake and the fraction of basic blocks improved by the optimal scheduler increases. For the largest basic blocks, up to 32.7% of the schedules are improved by the optimal scheduler (see the 4-issue architecture in Table 3.5). Table 3.6 shows the average and maximum gain in schedule length against the heuristic schedule.

Depending on the architectural model, the optimal scheduler took between 2:31:44 (hh:mm:ss) and 6:44:00 to schedule all of the basic blocks in the entire SPEC benchmark (see Table 3.7). While such long compile times would not be tolerable in everyday use, these times are well within acceptable limits when compiling for software libraries, embedded applications, or final release builds. I note that adding the implied distance constraints and the safe pruning and dominance constraints were critical to achieving this performance. Without these constraints, many individual basic blocks could *not* be solved within the amount of time that I can now solve the entire ensemble of basic blocks. Table 3.8 gives percentage of all basic blocks in the SPEC 2000 benchmark suite which were solved to optimality, for various idealized architectural models and time limits for solving each basic block.

Table 3.3: *Critical path heuristic.* Number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of basic blocks with improved schedules (%), for various idealized architectural models.

		1-issue		2-issue		4-issue		6-issue	
	#blocks	imp.	%	imp.	%	imp.	%	imp.	%
ammp	6,587	121	1.8	160	2.4	122	1.9	108	1.6
applu	1,387	68	4.9	101	7.3	92	6.6	60	4.3
apsi	4,860	102	2.1	218	4.5	221	4.5	129	2.7
art	841	3	0.4	5	0.6	13	1.5	0	0.0
bzip2	2,032	15	0.7	15	0.7	30	1.5	10	0.5
crafty	10,104	136	1.3	136	1.3	176	1.7	57	0.6
eon	9,481	131	1.4	161	1.7	212	2.2	133	1.4
equake	989	8	0.8	12	1.2	16	1.6	9	0.9
facerec	2,657	41	1.5	124	4.7	147	5.5	73	2.7
fma3d	21,314	593	2.8	669	3.1	829	3.9	408	1.9
galgel	11,489	266	2.3	396	3.4	367	3.2	198	1.7
gap	40,354	343	0.8	343	0.8	297	0.7	105	0.3
gcc	88,251	448	0.5	446	0.5	566	0.6	214	0.2
gzip	3,333	40	1.2	40	1.2	69	2.1	5	0.2
lucas	1,929	87	4.5	103	5.3	105	5.4	68	3.5
mcf	771	21	2.7	21	2.7	20	2.6	1	0.1
mesa	31,381	363	1.2	447	1.4	560	1.8	248	0.8
mgrid	428	12	2.8	31	7.2	27	6.3	18	4.2
parser	7,496	50	0.7	50	0.7	50	0.7	16	0.2
perlbmk	33,992	276	0.8	277	0.8	273	0.8	76	0.2
sixtrack	23,518	803	3.4	1,270	5.4	1,265	5.4	640	2.7
swim	733	10	1.4	26	3.5	21	2.9	7	1.0
twolf	15,163	163	1.1	167	1.1	150	1.0	39	0.3
vortex	24,753	159	0.6	157	0.6	269	1.1	116	0.5
vpr	7,023	68	1.0	70	1.0	84	1.2	23	0.3
wupwise	1,245	58	4.7	78	6.3	68	5.5	12	1.0
Total	352,111	4,385	1.2	5,523	1.6	6,049	1.7	2,773	0.8

Table 3.4: *Critical path heuristic.* Average and maximum percentage improvements in schedule length of optimal schedule over schedule found by list scheduler using the heuristic, for various idealized architectural models. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

	1-issue		2-issue		4-issue		6-issue	
	ave.	max.	ave.	max.	ave.	max.	ave.	max.
ammp	3.6	20.0	3.5	20.0	4.0	20.0	3.3	11.1
applu	2.7	20.0	3.3	20.0	3.5	14.8	2.9	13.3
apsi	4.7	20.0	5.3	20.0	5.7	37.5	5.0	25.0
art	9.4	11.1	10.1	11.1	8.2	11.1		0.0
bzip2	6.9	20.0	6.9	20.0	6.7	30.0	5.1	10.0
crafty	5.4	20.0	5.4	20.0	6.0	27.3	5.2	16.7
eon	3.8	20.0	3.9	20.0	5.8	25.0	4.5	11.1
equake	6.3	16.7	5.0	16.7	5.8	20.0	3.3	7.7
facerec	7.0	20.0	6.5	20.0	5.6	15.1	4.9	11.9
fma3d	4.0	20.0	4.2	24.1	5.4	25.0	4.3	15.0
galgel	5.5	20.0	6.3	33.3	5.1	20.0	5.7	25.0
gap	8.0	20.0	8.0	20.0	9.7	20.0	8.3	25.0
gcc	7.5	21.4	7.5	21.4	7.7	38.9	9.8	33.3
gzip	10.0	20.0	10.0	20.0	10.4	20.0	10.8	16.7
lucas	4.1	10.5	5.8	12.8	5.3	12.8	2.9	6.7
mcf	6.8	20.0	6.8	20.0	8.7	25.0	7.7	7.7
mesa	4.4	20.0	5.6	27.3	7.2	28.6	5.6	32.4
mgrid	3.7	11.1	4.2	11.1	4.3	11.1	4.6	12.5
parser	9.3	20.0	9.3	20.0	10.5	25.0	8.5	14.3
perlbmk	7.4	20.0	7.5	20.0	7.5	20.0	7.1	25.0
sixtrack	4.3	20.0	5.0	25.0	4.9	33.3	3.9	20.7
swim	2.9	9.1	3.4	9.1	3.4	12.5	2.0	4.4
twolf	6.8	20.0	6.9	20.0	7.2	32.0	6.4	17.6
vortex	6.6	20.0	6.7	20.0	6.7	16.7	7.0	16.7
vpr	5.8	20.0	6.0	20.0	7.3	20.0	6.1	11.1
wupwise	3.8	20.0	5.5	33.3	4.5	11.5	6.0	10.0
Overall	5.5	21.4	5.7	33.3	6.2	38.9	5.2	33.3

Table 3.5: *Critical path heuristic.* Number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of basic blocks with improved schedules (%), for ranges of basic block sizes and various idealized architectural models.

		1-issue		2-issue		4-issue		6-issue	
range	#blocks	imp.	%	imp.	%	imp.	%	imp.	%
3–5	179,056	338	0.2	350	0.2	182	0.1	0	0.0
6–10	94,066	804	0.9	907	1.0	736	0.8	69	0.1
11–15	32,069	754	2.4	834	2.6	882	2.8	189	0.6
16–20	14,433	364	2.5	392	2.7	741	5.1	345	2.4
21–30	13,911	619	4.4	781	5.6	962	6.9	584	4.2
31–50	9,760	628	6.4	853	8.7	1,013	10.4	615	6.3
51–100	5,669	536	9.5	790	13.9	915	16.1	538	9.5
101–250	2,789	270	9.7	505	18.1	501	18.0	337	12.1
251–2750	358	72	20.1	111	31.0	117	32.7	96	26.8
Total	352,111	4,385	1.2	5,523	1.6	6,049	1.7	2,773	0.8

Table 3.6: *Critical path heuristic.* Average and maximum percentage improvements in schedule length of optimal schedule over schedule found by list scheduler using the heuristic, for ranges of block sizes and various idealized architectural models. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

	1-issue		2-issue		4-issue		6-issue	
range	ave.	max.	ave.	max.	ave.	max.	ave.	max.
3–5	16.5	20.0	16.8	33.3	15.6	25.0		0.0
6–10	8.8	20.0	9.3	30.0	11.0	33.3	15.0	33.3
11–15	6.0	21.4	6.6	27.3	8.4	27.3	10.3	20.0
16–20	4.5	15.8	4.6	15.8	7.2	37.5	8.3	25.0
21–30	3.6	13.6	4.3	19.0	5.7	28.6	6.1	17.6
31–50	2.6	15.6	3.7	25.0	4.3	32.0	4.4	21.4
51–100	1.9	10.3	2.8	20.0	2.9	38.9	3.1	28.6
101–250	1.3	9.7	2.3	27.3	2.3	27.5	2.2	32.4
251–2750	0.2	0.9	1.9	24.1	1.4	16.3	0.7	4.9
Overall	5.5	21.4	5.7	33.3	6.2	33.3	5.2	33.3

Table 3.7: Total time (hh:mm:ss) to schedule all basic blocks in the SPEC 2000 benchmark suite, for various idealized architectural models and time limits for solving each basic block.

	1 sec.	10 sec.	1 min.	10 min.
1-issue	50:06	1:20:50	1:43:31	2:31:44
2-issue	50:39	1:32:17	2:19:19	6:44:08
4-issue	46:31	1:26:31	2:09:16	5:31:21
6-issue	47:01	1:27:13	2:00:12	4:37:01

Table 3.8: Percentage of all basic blocks in the SPEC 2000 benchmark suite which were solved to optimality, for various idealized architectural models and time limits for solving each basic block.

	1 sec.	10 sec.	1 min.	10 min.
1-issue	99.792	99.969	99.997	99.999
2-issue	99.771	99.960	99.989	99.993
4-issue	99.791	99.965	99.990	99.995
6-issue	99.792	99.968	99.993	99.996

3.5.2 Experiments for realistic architectural models

The following table shows the four realistic architectural models I used in my evaluation. In these architectures, the functional units are not fully pipelined and the issue width of the processor is not equal to the number of functional units.

architecture	issue width	simple int. units	complex int. units	memory units	branch units	floating point units
1r-issue	1	1				
PowerPC 603e (ppc603e)	2	1		1	1	1
PowerPC 604 (ppc604)	4	2	1	1	1	1
6r-issue	6	2		2	3	2

Table 3.9 gives number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of basic blocks with improved schedules (%), for various realistic architectural models. Table 3.10 gives average and maximum percentage improvements in schedule length of optimal schedule over schedule found by list scheduler using the heuristic, for various realistic architectural models. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule. Tables 3.11 give the number of basic blocks where the optimal scheduler found an improved schedule over the best schedule for the realistic architecture. One can see that the optimal scheduler is performing much better than the heuristic for realistic architecture than non-realistic architecture. Tables 3.12 summarize the percentage improvements in schedule length of the optimal schedule over the schedule found by a list scheduling algorithm using the critical-path heuristic.

Depending on the architectural model, the optimal scheduler took between 35:20:21 (hh:mm:ss) and 233:24:03 to schedule all of the basic blocks in the entire SPEC benchmark (see Table 3.14). Table 3.13 gives percentage of all basic blocks in the SPEC 2000 benchmark suite which were solved to optimality, for various idealized architectural models and time limits for solving each basic block[1].

Figure 3.7 shows the percentage of basic blocks, for each architecture, where the list scheduler was within a given percentage of optimal when instruction scheduling was performed before register allocation. For example, the list scheduler is within 10% of optimal for 99.7% of all basic blocks on the 1r-issue architecture. As another example, for the PowerPC 604, the list scheduler finds an optimal schedule (i.e. is within 0% of optimal) for 96.5% of all basic blocks. Figure 3.8 shows the percentage of basic blocks, for each architecture, where the list scheduler was within

[1]For my experimentation, I used 2.40 GHz Pentium 4 processor with 1GB of main memory.

a given percentage of optimal when instruction scheduling was shown after register allocation. These two figures indicate clearly that list scheduling is nearly optimal when scheduling basic blocks, even on a more realistic architectural model, as list scheduling is optimal at least 94% of the time.

3.6 Summary

I presented a constraint programming approach to basic block instruction scheduling for multiple-issue processors. The problem is considered intractable, yet my approach is optimal and robust on large, real problems. The key to scaling up to large, real problems was in the development of an improved constraint model and the application of more powerful constraint propagation techniques. I experimentally evaluated my optimal scheduler on the SPEC 2000 integer and floating point benchmarks. On this benchmark suite, the optimal scheduler was very robust and scaled to the largest basic blocks. Depending on the architectural model, between 99.991% to 99.999% of all basic blocks were solved to optimality. The scheduler was able to solve the largest basic blocks, including blocks with up to 2600 instructions. This compares favorably to the best previous approach due to Heffernan and Wilken [31]. I also compared the performance of a list scheduler for basic block scheduling using the critical path heuristic. When scheduling for the idealized architectural model, the list scheduler solved 98.6%-99.9% of the basic blocks in the benchmark suite optimally. For the realistic architectural model, the list scheduler produced optimal schedules for 94.2%-97.8% of the basic blocks. In the next chapter, I will present my constraint programming model for superblock instruction scheduling problem.

Table 3.9: *Critical path heuristic.* Number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of basic blocks with improved schedules (%), for various realistic architectural models.

		1r-issue		ppc603e		ppc604		6r-issue	
	#blocks	imp.	%	imp.	%	imp.	%	imp.	%
ammp	6,587	220	3.3	334	5.1	317	4.8	189	2.9
applu	1,387	104	7.5	209	15.1	205	14.8	138	9.9
apsi	4,860	250	5.1	790	16.3	809	16.6	649	13.4
art	841	5	0.6	57	6.8	67	8.0	54	6.4
bzip2	2,032	18	0.9	100	4.9	67	3.3	46	2.3
crafty	10,104	142	1.4	420	4.2	428	4.2	294	2.9
eon	9,481	280	3.0	641	6.8	782	8.2	583	6.1
equake	989	14	1.4	27	2.7	24	2.4	21	2.1
facerec	2,657	58	2.2	357	13.4	395	14.9	335	12.6
fma3d	21,314	964	4.5	2,508	11.8	2,145	10.1	1,667	7.8
galgel	11,489	329	2.9	1,218	10.6	1,363	11.9	1,043	9.1
gap	40,354	817	2.0	1,410	3.5	967	2.4	827	2.0
gcc	88,251	474	0.5	1,737	2.0	1,597	1.8	1,391	1.6
gzip	3,333	42	1.3	169	5.1	133	4.0	113	3.4
lucas	1,929	89	4.6	297	15.4	321	16.6	171	8.9
mcf	771	19	2.5	53	6.9	42	5.4	31	4.0
mesa	31,381	547	1.7	1,801	5.7	1,723	5.5	1,416	4.5
mgrid	428	12	2.8	43	10.0	52	12.1	30	7.0
parser	7,496	50	0.7	214	2.9	179	2.4	174	2.3
perlbmk	33,992	273	0.8	1,108	3.3	947	2.8	781	2.3
sixtrack	23,518	1,196	5.1	2,877	12.2	2,578	11.0	1,788	7.6
swim	733	28	3.8	83	11.3	73	10.0	43	5.9
twolf	15,163	217	1.4	610	4.0	500	3.3	328	2.2
vortex	24,753	173	0.7	913	3.7	963	3.9	878	3.5
vpr	7,023	122	1.7	317	4.5	272	3.9	233	3.3
wupwise	1,245	69	5.5	152	12.2	150	12.0	91	7.3
Total	352,111	6,512	1.8	18,445	5.2	17,099	4.9	13,314	3.8

Table 3.10: *Critical path heuristic.* Average and maximum percentage improvements in schedule length of optimal schedule over schedule found by list scheduler using the heuristic, for various realistic architectural models. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

	1r-issue		ppc603e		ppc604		6r-issue	
	ave.	max.	ave.	max.	ave.	max.	ave.	max.
ammp	3.32	16.67	6.15	25.00	5.64	25.00	8.60	25.00
applu	2.88	16.67	4.95	23.53	4.64	23.53	6.51	27.91
apsi	2.97	16.67	5.56	37.50	5.85	30.00	7.53	74.42
art	5.38	10.00	7.99	25.00	8.19	25.00	10.46	25.00
bzip2	5.73	16.67	8.21	33.33	8.07	25.00	7.73	25.00
crafty	4.98	16.67	7.82	33.33	7.15	36.67	6.50	20.00
eon	3.81	17.14	6.00	27.78	5.63	25.00	5.97	27.87
equake	3.79	14.29	4.97	16.67	3.92	11.11	6.02	10.53
facerec	6.33	16.67	6.34	25.00	6.71	25.81	7.16	33.33
fma3d	3.56	16.67	5.32	27.27	5.21	35.00	5.79	44.83
galgel	5.03	16.67	5.97	25.00	6.65	40.00	7.60	40.51
gap	6.25	16.67	10.19	33.33	11.91	25.00	13.10	25.00
gcc	6.69	16.67	8.39	33.33	9.39	30.00	9.63	33.33
gzip	9.15	16.67	10.03	25.00	11.71	25.00	11.90	25.00
lucas	4.16	9.09	6.11	13.33	6.45	21.95	6.27	12.90
mcf	6.62	16.67	7.82	20.00	10.47	20.00	10.02	20.00
mesa	4.85	28.57	7.64	51.72	8.33	51.72	9.51	53.57
mgrid	2.91	10.00	3.87	10.00	5.35	19.23	4.63	10.00
parser	8.29	16.67	9.29	25.00	9.95	27.27	9.77	25.00
perlbmk	6.70	16.67	9.50	33.33	9.60	30.00	10.83	25.00
sixtrack	4.03	16.67	5.73	25.00	5.91	25.00	7.04	50.00
swim	3.36	9.09	4.44	13.89	5.12	12.33	3.96	11.54
twolf	5.88	22.67	9.11	27.69	9.17	26.56	9.16	26.56
vortex	6.14	16.67	7.92	28.00	8.04	25.00	8.68	36.36
vpr	5.02	16.67	7.12	25.00	6.80	25.00	6.77	33.33
wupwise	3.77	16.67	5.69	25.00	5.56	20.00	6.19	20.00
Overall	4.82	28.57	7.10	51.72	7.35	51.72	8.27	74.42

Table 3.11: *Critical path heuristic.* Number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of basic blocks with improved schedules (%), for ranges of basic block sizes and various realistic architectural models.

		1r-issue		ppc603e		ppc604		6r-issue	
range	#blocks	imp.	%	imp.	%	imp.	%	imp.	%
3–5	182,113	374	0.2	1,559	0.9	1,122	0.6	1,016	0.6
6–10	91,807	1,121	1.2	2,445	2.7	1,674	1.8	1,222	1.3
11–15	31,610	1,350	4.3	2,911	9.2	2,444	7.7	1,490	4.7
16–20	14,323	590	4.1	2,150	15.0	2,130	14.9	1,689	11.8
21–30	13,767	903	6.6	3,042	22.1	3,179	23.1	2,610	19.0
31–50	9,703	988	10.2	3,016	31.1	3,098	31.9	2,537	26.1
51–100	5,645	771	13.7	2,168	38.4	2,181	38.6	1,767	31.3
101–250	2,786	343	12.3	1,057	37.9	1,171	42.0	937	33.6
251–2750	357	72	20.2	123	34.5	129	36.1	62	17.4
Total	352,111	6,512	1.8	18,471	5.2	17,128	4.9	13,330	3.8

Table 3.12: *Critical path heuristic.* Average and maximum percentage improvements in schedule length of optimal schedule over schedule found by list scheduler using the heuristic, for ranges of block sizes and various realistic architectural models. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

	1r-issue		ppc603e		ppc604		6r-issue	
range	ave.	max.	ave.	max.	ave.	max.	ave.	max.
3–5	13.2	16.7	19.5	33.3	21.2	25.0	22.9	33.3
6–10	7.5	18.2	10.5	25.0	11.7	33.3	13.4	33.3
11–15	5.5	28.6	7.4	51.7	8.3	51.7	9.4	74.4
16–20	4.1	15.8	6.2	37.5	7.1	28.6	7.6	36.4
21–30	3.8	22.7	5.6	27.8	6.5	35.0	7.4	51.1
31–50	2.8	15.2	4.6	33.3	5.1	30.0	5.6	36.6
51–100	2.0	10.2	3.2	25.0	3.6	40.0	4.3	50.0
101–250	1.6	10.0	2.1	25.9	2.1	28.8	2.5	31.6
251–2750	0.4	4.3	1.7	9.9	1.4	12.9	2.0	12.0
Overall	4.8	28.6	7.1	51.7	7.3	51.7	8.3	74.4

Table 3.13: Percentage of all basic blocks in the SPEC 2000 benchmark suite which were solved to optimality, for various realistic architectural models and time limits for solving each basic block.

	1 sec.	10 sec.	1 min.	10 min.
1r-issue	99.479	99.727	99.905	99.969
ppc603e	97.822	98.847	99.566	99.805
ppc604	97.831	98.856	99.579	99.838
6r-issue	97.321	98.643	99.442	99.738

Table 3.14: Total time (hh:mm:ss) to schedule all basic blocks in the SPEC 2000 benchmark suite, for various realistic architectural models and time limits for solving each basic block.

	1 sec.	10 sec.	1 min.	10 min.
1r-issue	48:27	4:05:52	11:16:46	35:20:21
ppc603e	2:31:50	16:25:58	47:33:35	179:08:17
ppc604	2:32:16	16:24:20	46:05:36	162:27:33
6r-issue	3:03:45	19:50:42	57:34:06	233:24:03

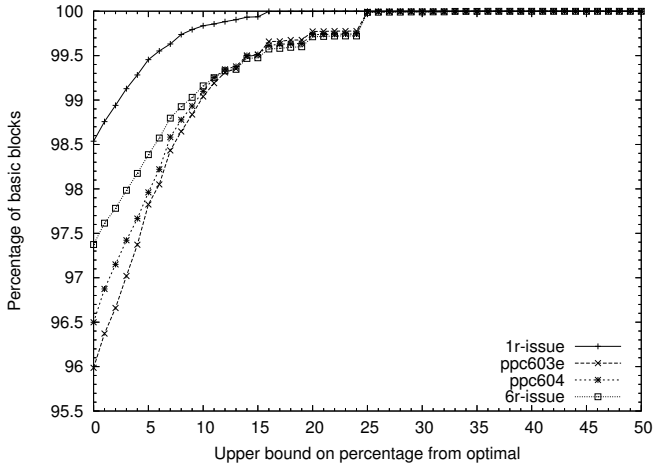


Figure 3.7: *Basic block scheduling before register allocation.* Performance guarantees for the list scheduler using the critical path heuristic in terms of worst-case factors from optimal, for various architectures.

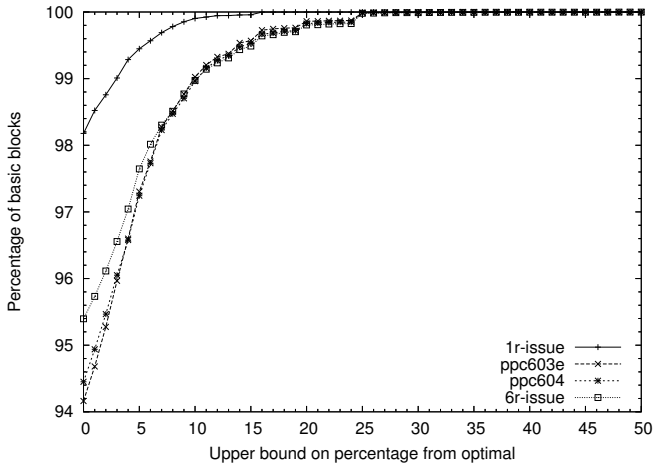


Figure 3.8: *Basic block scheduling after register allocation.* Performance guarantees for the list scheduler using the critical path heuristic in terms of worst-case factors from optimal, for various architectures.

Chapter 4

Superblock Scheduling for Multi-Issue Processors

In this chapter, I present my model for the superblock scheduling problem. I discuss the changes made to the basic block instruction scheduler to solve the superblock scheduling problem, and evaluate it for the idealized and realistic architectures presented in Chapter 3.

4.1 Related work

Superblock instruction scheduling for realistic multiple-issue processors is NP-complete [33], and most compilers use a heuristic approach. Researchers have also proposed to solve it optimally. I will discuss both the approaches separately.

4.1.1 Scheduling heuristics

List scheduling is the commonly used algorithm for superblock scheduling. As already stated in Chapter 2 it maintains a priority queue of ready instructions which are instructions with no predecessors. The priority of an instruction is calculated using a heuristic. For a given clock cycle, the list scheduler picks the top instructions in the priority queue. The number of picked instructions depend upon the number of functional units and the available free slots in the given cycle. If it could not find any instruction, it inserts a NOP (No OPeration). It continues this process until all instructions are scheduled. Many heuristics have been crafted to find a good schedule including critical path [35], successive retirement [13], dependence height and speculative yield [22], G^* [13], speculative hedge [19] and balance scheduling [49]. In my work I did a detailed

analysis of critical path, G^*, dependence height and speculative yield and speculative hedge heuristics with respect to their success for finding optimal solutions. I dropped the balance scheduling and successive retirement heuristics because of their high computational cost.

Superblocks, when introduced [35], were scheduled using the *critical path* heuristic. The critical path heuristic is good when the aim is to minimize the distance between the *root* and the *sink* node. In superblock scheduling the objective is to minimize the *weighted completion time*. *Exit nodes*, which define the weighted completion time, may not be on the critical path of a DAG representing a superblock. Hence, this heuristic may not be a good choice for finding a good schedule for superblock scheduling.

The *dependence height and speculative yield* (DHASY) [22] heuristic is a modified version of the critical path heuristic for superblock scheduling. Instead of a plain critical path, a weighted critical path to all exit points is used to prioritize the instruction nodes in a superblock. The priority of an instruction x is calculated as,

$$priority(x) = \sum_{e \in B} (w_e(cp(1,n) + 1 - ((cp(1,e) - cp(x,e)))$$

where B is the set of exit nodes that are descendants of x, $cp(1,n)$ is the critical path distance between the root and the sink node, $cp(1,e)$ is the critical path distance between the root node and exit node e and $cp(x,e)$ is the critical path distance between instruction x and exit node e.

In the *G* heuristic* [13], a superblock is scheduled using the critical path heuristic. The rank for each exit point is then calculated by dividing the cycle in which the exit point is scheduled by the sum of the exit probabilities for the exit point under consideration and its preceding exit points. The exit points are sorted in ascending order. The final schedule for the superblock is obtained by taking an exit point from the sorted list one by one and scheduling it as early as possible with its predecessors.

The *speculative hedge* [19] heuristic calculates the priority of an instruction by the sum of the weights of the branches that it helps schedule early. Speculative hedge investigates each operation to determine whether it helps still unscheduled exit points or not. An operation can help an exit point in two ways: (i) the operation is on the critical path to the exit point and delaying the operation will delay the exit point, and (ii) the operation uses a critical resource that is critical to the exit point, and preferring some other operation will delay the exit point. An operation's priority is the sum of the exit probabilities helped by the operation.

4.1.2 Approaches for optimal solutions

Even the best heuristics can produce sub-optimal solutions. In contrast to basic block instruction scheduling, there has been little work on optimal global instruction scheduling where the region

of code to schedule consists of more than one basic block—such as in traces, superblocks, or hyperblocks.

Winkel [67] presents an integer linear programming model for global instruction scheduling for Itanium processors. However, the approach has a number of deficiencies. First, the evaluation of the approach is limited, consisting of only nine scheduling regions with sizes up to 200 instructions. Second, and more importantly, the approach minimizes the *length* of the schedule. This measure is appropriate for basic blocks, which consist of straight line code. But it is not appropriate for global regions which contain multiple exits and whose paths of execution may rarely fall through to the last instruction (see Section 2.5 and the discussion of weighted completion time).

Shobaki and Wilken [60, 61] were the first to develop a robust optimal scheduler for superblocks that scaled up to large superblocks. Their experimental work is limited to superblocks with size up to 1236 instructions. For their work, instruction latencies are 2 cycles for floating point (FP) adds, 3 cycles for loads and FP multiplies, 9 cycles for FP divides and 1 cycle for all other instructions. My test suite, obtained from the IBM TOBEY compiler, contains larger superblocks with size up to 2600 instructions and more varied latencies. My test suite also contains zero latency edges, which are used to capture anti-dependencies and output dependencies[1] between two instructions. This makes the optimal superblock scheduling problem more challenging. As well, Shobaki and Wilken's work is targeted to idealized architectures which assume that the functional units are fully pipelined and that the issue width of the processor is equal to the number of functional units.

In my work, I remove these assumptions and present the first optimal superblock scheduling approach for realistic architectures. Further, even though the target architectures are realistic, my approach scales up to more difficult and larger superblocks than in previous work. I experimentally evaluated my optimal scheduler on the SPEC 2000 integer and floating point benchmarks, using four idealized architectural models and four realistic architectural models. The results for the idealized architectural models are presented to allow a comparison with previous work. On the SPEC 2000 benchmark suite, the optimal scheduler scaled to the largest superblocks and was very robust. Depending on the architectural model, at most 15 superblocks out of 187,334 superblocks used in my experiments could not be solved within a 10-minute time bound per superblock. In my experiments I also performed a detailed analysis of several state-of-the-art heuristics for superblock scheduling in comparison to the optimal scheduler.

[1] An anti-dependency occurs when an instruction requires a value that is later updated; an output dependency occurs when the ordering of instructions will affect the final output value of a variable.

4.2 Constraint programming model for superblock scheduling

In this section, I present my constraint model of the superblock instruction scheduling problem. The six main types of constraints in the model are latency, resource, distance, predecessor and successor, safe pruning, and dominance constraints. Except the dominance and distance constraints, all other constraints are taken from the model for basic block scheduling. The dominance and distance constraints are modified for superblock scheduling. I discuss here only the dominance constraints and distance constraints for subgraphs in a DAG for a superblock.

4.2.1 Dominance constraints

Heffernan and Wilken [31] present a set of graph transformations for dependency DAGs for basic blocks and show that optimally scheduling the transformed DAGs using branch-and-bound enumeration is faster and more robust. The DAG transformations reduce the search space while preserving optimality and hence are safe. I implemented it successfully for my basic block scheduling model[2]. In this section, I will talk about this constraint with reference to superblock scheduling.

Adding a dominance constraint in a dependency DAG for a superblock is safe, if it does not change the optimal cost function value; i.e, weighted completion time (WCT) of the DAG. The number of speculative instructions across an exit node define the speculative characteristic of the exit node. The speculative characteristic of exit nodes and their schedule length affect the WCT value. Let G be a DAG of a superblock S. Let l_i^* be the minimum schedule length of exit node e_i from the root node of G for all schedules of S. If there are n exit nodes in S, then a lower bound, C^*, on the value of WCT can be calculated as:

$$C^* = \sum_{i=1}^{n} w_i l_i^*.$$

Let l_i be the schedule length of exit node e_i from the root node of G in any schedule of S. The following relationship holds between C^* and the value of WCT, C, for any schedule:

$$C = \sum_{i=1}^{n} w_i l_i \geq C^* = \sum_{i=1}^{n} w_i l_i^*.$$

Let C' be the difference between C and C^* i.e.,

$$\begin{aligned} C' &= C - C^*, \\ C' &= \sum_{i=1}^{n} w_i (l_i - l_i^*), \\ C' &= \sum_{i=1}^{n} w_i \delta_i, \end{aligned}$$

[2]See Chapter 3 for detail.

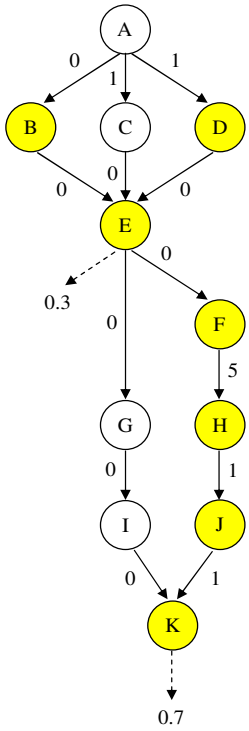


Figure 4.1: DAG for a superblock with node E and node K as exit nodes.

where $\forall i, \delta_i = l_i - l_i^*$. δ_i gives the distance of e_i from its minimum schedule length in any schedule for the superblock. In order to ensure an optimal solution for G after a transformation, I have to ensure that δ_i does not change after the transformation. The value of δ_i depends upon the number of instructions that can be moved across e_i; i.e., the speculative characteristic of e_i and the minimum schedule length of e_i. If I preserve the speculative characteristic and the minimum schedule length of e_i, I preserve the value of δ_i and hence the optimal cost function value for G after transformation.

Definition 4.1 (Immediate predecessor exit node) *If all paths from an exit node e_i to a node j do not contain any other exit node, then e_i is an immediate predecessor exit node of j.*

Definition 4.2 (Immediate successor exit node) *If all paths from a node j to an exit node e_i do not contain any other exit node, then e_i is an immediate successor exit node of j.*

Example 4.1 *Consider Figure 4.1. Nodes E and K are exit nodes. Node E is the immediate predecessor exit node for nodes F, G, H, I and J. Node K is the immediate successor exit node for nodes F, G, H, I and J.*

I restate the theorem by Heffernan and Wilken [31] for dependency DAGs for superblocks [3].

Theorem 4.1 *Let A and B be isomorphic subgraphs in a DAG G_s of a superblock S, with node sets $V(A) = \{a_1, \ldots, a_r\}$ and $V(B) = \{b_1, \ldots, b_r\}$. If, (i) a_i is neither a predecessor or a successor of b_i, $1 \leq i \leq r$, (ii) for all $k \in pred(a_i)$ such that $k \notin V(A)$, $l(k, a_i) \leq cp(k, b_i)$, $1 \leq i \leq r$, (iii) for all $k \in succ(b_i)$ such that $k \notin V(B)$, $l(b_i, k) \leq cp(a_i, k)$, $1 \leq i \leq r$, (iv) for any edge (b_i, a_j), $l(b_i, a_j) \leq cp(a_i, b_j)$, and (v) neither a_i nor b_i, $1 \leq i \leq r$, are exit nodes. Then adding the constraints $a_i \leq b_i$, $1 \leq i \leq r$ in G_s is safe.*

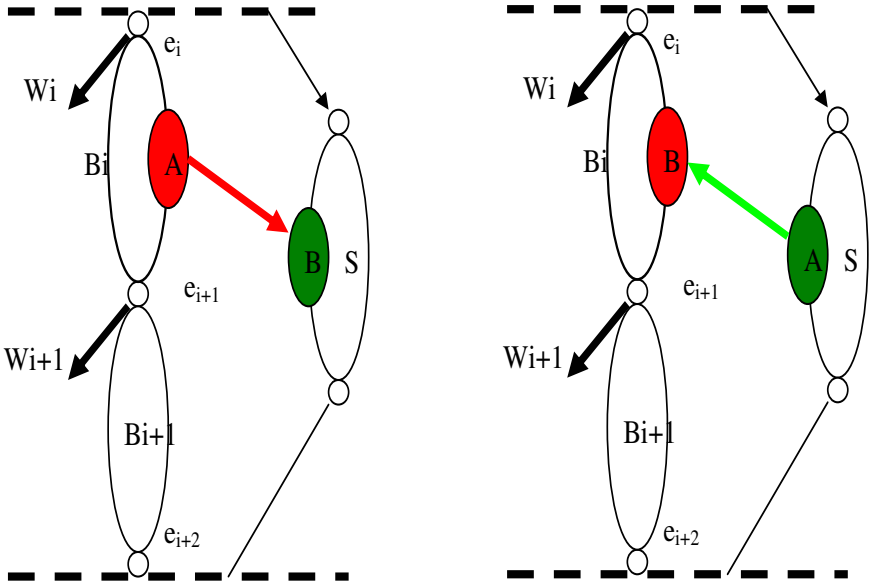


Figure 4.2: Adding dominance constraints in a superblock. A and B are isomorphic graphs; (a) case-1: $V(B)$ consists of speculative nodes; (b) case-2: $V(A)$ consists of speculative nodes.

Proof. Figure 4.2 shows the two possibilities of adding a dominance constraint in a superblock. Subgraph A and B are the same as in Theorem 4.1; i.e., A is superior to B. To show that the transformations are safe, I must show that the transformations preserve the speculative characteristic and the minimum schedule length of the exit nodes.

Part-1: Preserving the speculative characteristic of exit nodes.

Case-1: *V(B) consists of speculative nodes that can be moved across basic blocks.* By inserting zero-latency edges from V(A) to V(B) (adding dominance constraints $V(A) \leq V(B)$), I am restricting the movement of $b_i \in V(B)$ to be below e_i; i.e., e_i, which is an immediate predecessor exit node for $a_i \in V(A)$, now is also the immediate predecessor exit node for $b_i \in V(B)$. According

[3]The theorem has been proved independently by Heffernan and Wilken [32]

to condition (ii) of Theorem 4.1, there is a path from each predecessor of $a_i \in V(A)$ to $b_i \in V(B)$. As there is a path from e_i to each predecessor of $a_i \in V(A)$, then there is also a path from e_i to $b_i \in V(B)$, which also makes e_i the immediate predecessor exit for $b_i \in V(B)$. Thus, the transformations do not change the speculative characteristic of the exit nodes in the superblock.

Case-2: *V(A) consists of speculative nodes that can be moved across basic blocks.* By inserting zero-latency edges from V(A) to V(B), I am restricting the movement of $a_i \in V(A)$ to be above e_{i+1}; i.e., e_{i+1}, which is the immediate successor exit node for $b_i \in V(B)$, now is also the immediate successor exit node for $a_i \in V(A)$. According to condition (iii) of Theorem 4.1, there is a path from $a_i \in V(A)$ to successors of $b_i \in V(B)$. As there is a path from each successor of $b_i \in V(B)$ to e_{i+1}, then there is also a path from each $a_i \in V(A)$ to e_{i+1}, which also makes e_{i+1} the immediate successor exit for $a_i \in V(A)$. Thus, the transformations do not change the speculative characteristic of the exit nodes in the superblock.

Part-2: Preserving the minimum schedule length of exit nodes.

The minimum schedule length of e_{i+1} can be determined by scheduling a subgraph G' containing e_{i+1} and all its predecessors using an optimal scheduler. According to Theorem 4.1, adding dominance constraint within G' preserves the minimum schedule length of e_{i+1}.

Case-1: *V(B) consists of speculative nodes that can be moved across basic blocks.* This case can be further divided into following three sub-cases:

- When for every $b_i \in V(B)$, there is a path from b_i to e_{i+1}. This makes each $b_i \in V(B)$ a predecessor of e_{i+1}, i.e., $b_i \in V(G')$. Then the transformations are within subgraph G'. The transformations preserve the minimum schedule length of e_{i+1}.
- When for every $b_i \in V(B)$, there is no path from b_i to e_{i+1}. It means $b_i \ni V(G')$. Then the transformations are outside of subgraph G'. The transformations do not change G'. The transformations preserve the minimum schedule length of e_{i+1}.
- When for some $b_i \in V(B)$, there is a path from b_i to e_{i+1} and for some $b_i \in V(B)$ there is no path from b_i to e_{i+1}. Let B_1 be a subgraph of B consisting of $b_i \in V(B)$ which has a path to e_{i+1}. Let A_1 be a subgraph of A which is isomorphic to B_1. Then adding dominance constraints from $a_i \in V(A_1)$ to $b_i \in V(B_1)$ are within G' and dominance constraints from $a_i \in V(A - A_1)$ to $b_i \in V(B - B_1)$ are outside of G'. The transformations preserve the minimum schedule length of e_{i+1}.

Case-2: *V(A) consists of speculative nodes that can be moved across basic blocks.* In this case all immediate successors of $b_i \in V(B)$ are predecessors of e_{i+1}. According to condition (iii) of Theorem 4.1, there is a path from $a_i \in V(A)$ to successors of $b_i \in V(B)$. As there is a path from each successor of $b_i \in V(B)$ to e_{i+1}, then there is also a path from each $a_i \in V(A)$ to e_{i+1}, which also makes e_{i+1} the immediate successor exit for $a_i \in V(A)$. The transformations are within subgraph G'. The transformations preserve the minimum schedule length of e_{i+1}.

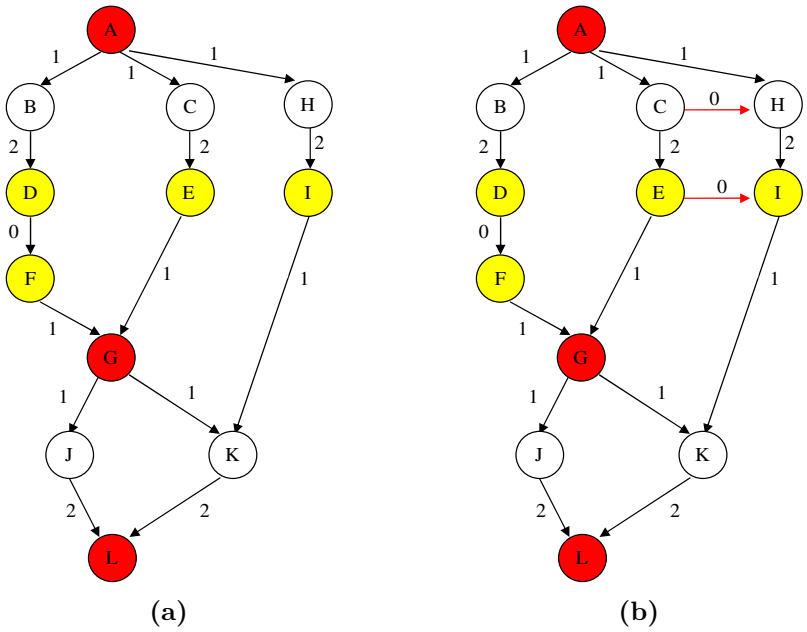


Figure 4.3: Example of adding dominance constraints in a superblock: (a) actual DAG; (b) the constraints C $\leq$ H and E $\leq$ I (zero latency edges) would be added to the constraint model. Nodes A, G and L are exit nodes.

Thus, the transformations are safe. □

Example 4.2 *Consider the DAG shown in Figure 4.3(a). Nodes H and I are speculative nodes as they can be moved across exit node G. Hence, the number of speculative instructions across exit node G is* 2*. The subgraphs with nodes* $V(A) = \{C, E\}$ *and* $V(B) = \{H, I\}$ *are isomorphic and satisfy the conditions of Theorem 4.1. Hence, the constraints* $C \leq H$ *and* $E \leq I$ *can be added to the model. Figure 4.3(b) shows the DAG with the added constraints. The added constraints do not change the speculative characteristic of exit node G, as node H and node I still can be moved across node G.*

Testing isomorphism is NP-complete in general. In Chapter 3, I explained my fast heuristic to determine whether two components are isomorphic. For superblock constraint, I adopted the same strategy for finding isomorphic graphs to add dominance constraints.

4.2.2 Upper bound distance constraints

Wilken et al. [66] introduced the concept of *region* in their work for optimal basic block scheduling using integer programming. Using the concept of region, van Beek and Wilken [65] introduced the

distance constraint in their work for optimal basic block scheduling using constraint programming. The distance constraint improves the lower bound for the distance that must exist between a pair of instructions, defining the region, in any schedule. I give an upper bound for the distance constraint for a region between a pair of articulation nodes. Consider articulation nodes x_i and x_j in a superblock, with no exit node in between them, a distance constraint of the form $x_i + d_{ij} \geq x_j$ is added to the constraint model. If there is no resource contention at x_i, then d_{ij} is the minimum schedule length, l^*_{ij}, between x_i and x_j in any legal schedule for the superblock. If there is resource contention at x_i, then $d_{ij} = l^*_{ij} + 1$. Adding upper bound distance constraints for such regions is based on Theorem 4.2.

Definition 4.3 (Articulation node) *Let G be a graph. Node $V_i \in V(G)$ is an* articulation node *for G, if the subgraph of G induced by $V(G)/\{V_i\}$ is unconnected.*

Definition 4.4 (Resource contention) *Let G be a DAG for superblock. Let B_{i-1} and B_i be two basic blocks in G connected by exit node e_i. If instructions from B_{i-1} and B_i compete for slots available at the clock cycle in which e_i can be issued, then there is said to be* resource contention *at the exit node e_i.*

Example 4.3 *Consider Figure 4.4. Assume a fully pipelined processor with issue-width equal to four. Basic block B_1 consists of nodes A, B, C, D and E. Basic block B_2 includes of nodes E, F, G and H. Node E is an articulation node. There is resource contention at E in Figure 4.4(b), as nodes B, C, D from B_1 and nodes F, G from B_2 compete for the slots in the cycle in which E can be issued. There is no resource contention at E in Figure 4.4(a).*

Theorem 4.2 *The schedule length of a region between two consecutive articulation nodes, with no exit nodes in-between them, cannot be more than $l^* + 1$, where l^* is the optimal schedule length of the region when scheduled independently.*

Proof. Consider a region r_{ij} between exit nodes x_i and x_j in a superblock S. Where x_i is a predecessor of x_j, and there are no exit nodes between x_i and x_j as per the statement of Theorem 4.2. When r_{ij} is scheduled independently, all resources are considered at the disposal of region r_{ij} at clock cycle 1. An optimal scheduler will give optimal schedule length l^* for r_{ij}. In any schedule for S, r_{ij} cannot have schedule length less than l^*. When there is no resource contention at x_i, the situation is the same as if the region was being scheduled independently. If there is resource contention, then some resources might still be occupied by the instructions which are predecessors of x_i. Let t_i be the clock cycle of x_i. If I insert a free slot at $t_i + 1$, then all resources will be available for the region r_{ij}, and all the nodes in r_{ij} can be scheduled within $t_i + l^* + 1$ schedule length from x_i. Thus, the distance between x_i and x_j cannot be more than $l^* + 1$ in any schedule for S. □

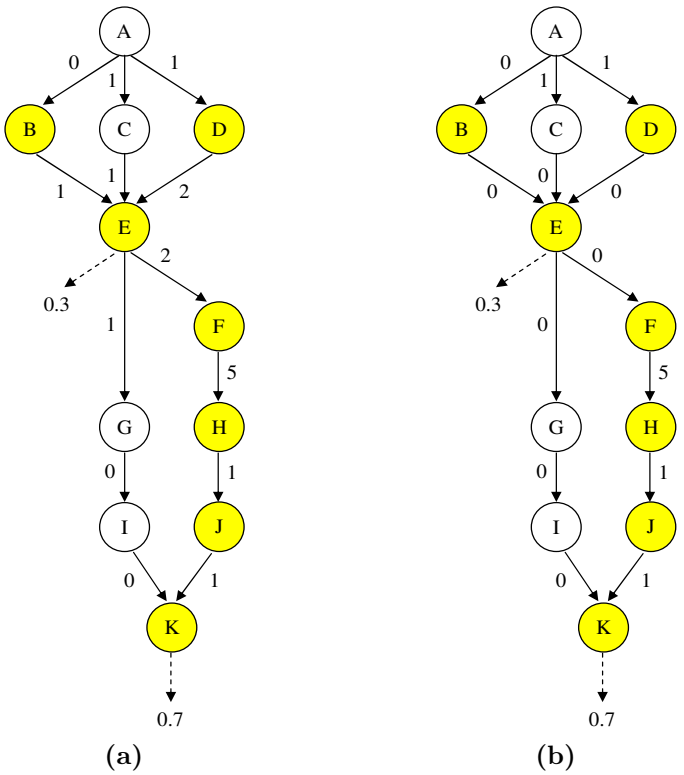


Figure 4.4: Articulation node and resource contention: (a) no resource contention at the articulation node E; (b) resource contention at the articulation node E.

Example 4.4 *Figure 4.5(a) is a region in a superblock bounded by articulation nodes A and E. Nodes X and Y are predecessor nodes of A. With latency of more than zero between A and its predecessor nodes, there is no resource contention at node A. The distance between A and E cannot be less than the minimum schedule length of the region scheduled independently by an optimal scheduler. Figure 4.5(b) shows the region in isolation. Considering a dual-issue processor, Table 4.1 gives a minimum length schedule for the region. In Figure 4.5(c), the latency between A and its predecessor nodes is zero. This gives rise to resource contention at A. The worst case is when one of the predecessor node takes the slot parallel to node A. But, I can still schedule all the nodes in the region (excluding A) within the minimum length after the node A issue slot.*

4.2.3 Improved lower and upper bounds for cost variables

The cost function, *cost*, is given by,

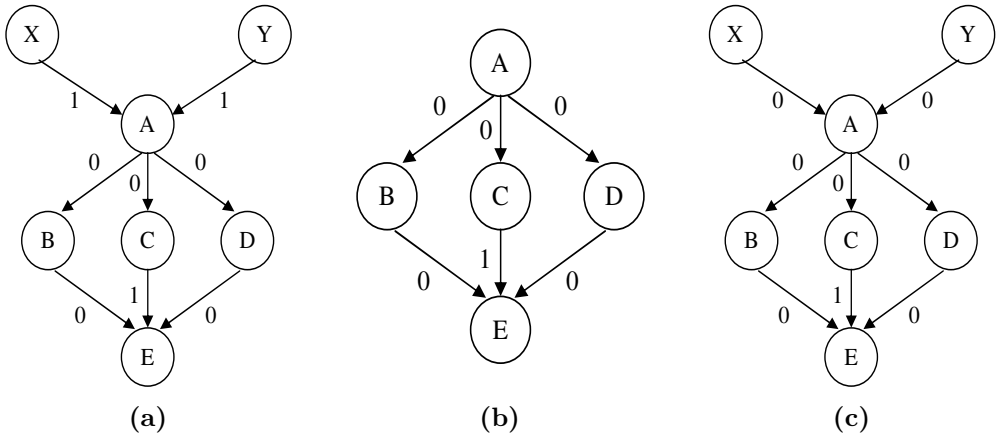


Figure 4.5: Region for Example 4.4: (a) no resource contention at the articulation node A; (b) region between node A and E in isolation; (c) resource contention at the articulation node A.

cycle	*Schedule*	
1	A	B
2	C	D
3	E	

Table 4.1: An optimal schedule for the region in Figure 4.5(b).

$$cost = \sum_{i=1}^{n} w_i x_i,$$

where w_i is the exit probability of exit node e_i with schedule length x_i. The cost function value from any efficient heuristic approach can be used as an upper bound. Given an upper bound, c, on *cost* and bounds on the variables in the cost function, i.e. exit nodes; it is straightforward to improve upper bounds for each cost function variable by considering each exit node at their minimum domain value except the exit node under consideration. An upper bound improvement for exit node e_j can be calculated as:

$$
\begin{aligned}
c &\geq \sum_{i=1}^{n} w_i x_i \\
c &\geq w_j x_j + \sum_{\substack{i=1 \\ i \neq j}}^{n} w_i x_i \\
\frac{c - \sum_{\substack{i=1 \\ i \neq j}}^{n} w_i x_i}{w_j} &\geq x_j \\
\frac{c - \sum_{\substack{i=1 \\ i \neq j}}^{n} w_i \cdot \min(x_i)}{w_j} &\geq x_j \\
\left\lfloor \frac{c - \sum_{\substack{i=1 \\ i \neq j}}^{n} w_i \cdot \min(x_i)}{w_j} \right\rfloor &\geq x_j
\end{aligned}
$$

I also use singleton consistency to prune the upper bounds of the cost variables. In singleton consistency, a variable is temporarily instantiated to a single value and the constraint model is tested for consistency. If the consistency test fails, the value can be removed from the domain of the variable. In my work, I iteratively instantiated and tested the consistency on the upper bounds of the domains of the variables. Let x_j be a cost variable and $dom(x_j) = [a, b]$. I temporarily instantiate $x_j \leftarrow b$ and test whether the CSP is consistent by propagating the constraints and also by testing, once the constraints have been propagated and the lower bounds have potentially been updated, whether Equation 4.1 is satisfied. If the CSP is not consistent or the equation is not satisfied, the domain of x_j is set to $[a, b-1]$ and the process repeats.

Given a lower bound c on *cost* and bounds on the variables in the cost function, the lower bound for each cost function variable can be improved by considering each exit node at their maximum domain except the node under consideration. A lower bound improvement for exit node e_j can be determined by Equation 4.1.

$$
\left\lceil \frac{c - \sum_{\substack{i=1 \\ i \neq j}}^{e+1} c_i \cdot \max(x_i)}{w_j} \right\rceil \leq x_j \tag{4.1}
$$

I also use singleton consistency to prune the lower bounds of the cost variables. Let x_j be a cost variable and $dom(x_j) = [a, b]$. I temporarily instantiate $x_j \leftarrow a$ and test whether the CSP is consistent by propagating the constraints and also by testing, once the constraints have been propagated and the lower bounds have potentially been updated, whether Equation 2 is satisfied. If the CSP is not consistent or the equation is not satisfied, the domain of x_j is set to $[a+1, b]$ and the process repeats.

4.3 Solving an instance

I construct the constraint model and use the constraints to establish the lower bounds of the variables and a lower bound on the length m of an optimal schedule. Given m, the upper bounds of the variables are similarly established. A lower bound on the value of the cost function of an optimal schedule is given by:

$$\begin{aligned} cost &= \sum_{i=1}^{n} w_i x_i \\ &\geq \sum_{i=1}^{n} w_i \cdot \min(x_i) \end{aligned}$$

An upper bound on the value of the cost function is found by a heuristic approach. If $lower(cost) = upper(cost)$, then the schedule given by the heuristic approach is optimal. If $lower(cost) \neq upper(cost)$, an optimal schedule can be determined depending upon the characteristic of the superblock. For a superblock with all exit points as articulation points, Algorithm 4.1 is adopted. In Algorithm 4.1, *ConstraintModel* constructs a constraint model of DAG G. *SubConstraintModel* gives a sub-constraint model of G with exit node e_i as final exit point. *OptimalSchedule* gives the optimal schedule length for exit node e_i using sub-constraint model CM$'$. The optimal scheduler for basic blocks, given in Chapter 3, has been used as *OptimalSchedule*. *UpdateDomain* makes the domain of exit node e_i the singleton domain equal to L_i.

Algorithm 4.1: Algorithm for finding an optimal schedule for a superblock with exit points as articulation points.

```
input  : Dependency DAG G for a superblock S
output: Optimal schedule for S
CM = ConstraintModel (G);
for i = 0 to n do
    CM' = SubConstraintModel (CM, i);
    L_i = OptimalSchedule(CM', schedule);
    UpdateDomain(i, L_i, CM);
return schedule;
```

Example 4.5 *Consider Figure 4.6. Using Algorithm 4.1, an optimal schedule for the superblock can be obtained by first finding the minimum schedule length of exit node e_1, then fixing this node at its minimum schedule length slot and finding the minimum schedule length for next exit node e_2 and so on. This methodology ensures an optimal solution by the following theorems.*

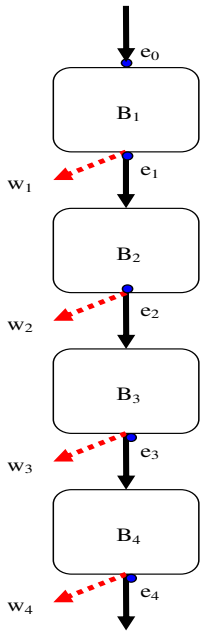


Figure 4.6: Superblock for Example 4.5. All exit nodes are articulation points.

Lemma 4.1 *When scheduling a superblock using Algorithm 4.1, the first basic block of the superblock will have schedule length equal to l^*, where l^* is the optimal schedule length of the basic block when scheduled independently.*

Proof. When a superblock is scheduled using Algorithm 4.1, the resource condition for the first basic block is same as when it is scheduled independently using an optimal scheduler. □

Theorem 4.3 *Scheduling a superblock with all exit nodes as articulation nodes using Algorithm 4.1, each exit node in the superblock is at the minimum schedule length from the root node of the superblock.*

Proof. Let S be a schedule for a superblock obtained using Algorithm 4.1. Let there be n exit nodes in the superblock with e_0 as the root node. Let L_i, $0 \leq i \leq n$, be the schedule length of exit node e_i from e_0 in S. Let S' be any other feasible schedule of the superblock. Let L'_i, $0 \leq i \leq n$, be the schedule length of exit node e_i from e_0 in S'. My claim is:

$$\forall i, L_i \leq L'_i.$$

I will use a proof by contradiction approach. Suppose there exist values of i for which this claim is not true. Let j be the smallest such value; i.e. for $\forall i < j$, the claim is true, and e_j is the first exit node which contradicts the claim. For this condition to be true, I have to examine two cases.

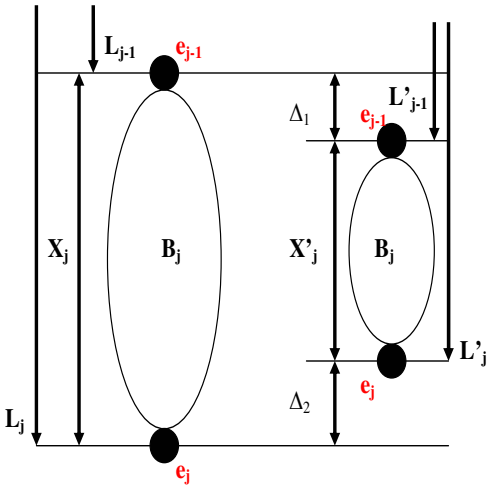


Figure 4.7: Case-2: $L_j > L'_j$ exists when $L_{j-1} < L'_{j-1}$.

Case-1: $L_j > L'_j$ exists when $L_{j-1} = L'_{j-1}$.

This is not possible. As I am using a true optimal scheduler. Had there been a schedule which gives $L'_j < L_j$, when $L_{j-1} = L'_{j-1}$, the optimal scheduler would have found it.

Case-2: $L_j > L'_j$ exists when $L_{j-1} < L'_{j-1}$.

Figure 4.7 explains *case-2* graphically. Let X_j be the schedule length of the basic block B_j, that exists between exit nodes e_j and e_{j-1}, in the schedule S, that is found using Algorithm 4.1. Let X'_j be the schedule length of the same basic block in the schedule S', having $L_j > L'_j$, found by any other heuristic. The relationship between X_j and X'_j can be expressed as:

$$X_j = X'_j + \triangle_1 + \triangle_2, \tag{4.2}$$

where $\triangle_1 = L'_{j-1} - L_{j-1}$ and $\triangle_2 = L_j - L'_j$. I know that $L_j > L'_j$ and $L_{j-1} < L'_{j-1}$. Therefore, $\triangle_1, \triangle_2 \geq 1$. Let $\triangle_{total} = \triangle_1 + \triangle_2$. Then I can safely say that $\triangle_{total} \geq 2$. According to Theorem 4.2, $X_{op_j} + 1 \geq X_j \geq X_{op_j}$, where X_{op_j} is the optimal schedule length of basic block B_j when scheduled independently. When there is resource contention at e_{j-1}, then $X_j = X_{op_j} + 1$ and Equation 4.2 can be written as:

$$X_{op_j} = X'_j + \triangle_{total} - 1 \tag{4.3}$$

In Equation 4.3, $\triangle_{total} - 1 \geq 1$. Thus, $X_{op_j} > X'_j$. This cannot be true. Now, if there is no resource contention at e_j, then $X_j = X_{op_j}$. Equation 4.2 can be written as,

$$X_{op_j} = X'_j + \triangle_{total} \tag{4.4}$$

In Equation 4.4, $\triangle_{total} \geq 2$. Thus, $X_{op_j} > X'_j$. This is not possible. Hence, *case-2* is not possible.

Thus, the contradiction, $L_i > L'_i$, is not true. □

Theorem 4.4 *Scheduling a superblock with all exit nodes as articulation nodes using Algorithm 4.1 will give an optimal schedule for the superblock.*

Proof. Using Theorem 4.3. □

If $lower(cost) \neq upper(cost)$, and each exit point in a superblock is not an articulation node, I used backtracking along with constraint propagation, as in my constraint programming approach for basic block instruction scheduling, to find an optimal solution.

4.4 Experimental evaluation

In this section, I present experimental results gained from scheduling 154,651 superblocks. As in Chapter 3, the data was obtained from compiling the entire SPEC 2000 benchmark suite in IBM's TOBEY compiler backend. Superblocks were collected before instruction scheduling was performed, both before and after register allocation were performed. Each superblock was scheduled on several different architectures using both the idealized and realistic architectural models. The same set of architectures that were used for basic block scheduling, were used again for superblock instruction scheduling.

4.4.1 Experiments for idealized architectural models

Shobaki and Wilken [60] were the first to present experimental results on solving large superblocks targeted towards a multiple-issue processor. Their test suite contains the superblocks from the SPEC 2000 integer and floating point benchmarks. They reported that on average 98.7% of the superblocks were scheduled optimally within one second. Also, on average they were not able to solve about 1.3% of superblocks. They also stated that they were able to improve 80% of the hard problems (the problems that were passed to the enumerator). Comparing with Shobaki and Wilken's work, I speculate that my test suite contains more difficult problems for the following four reasons:

- My test suite contains longer and more varied latencies.
- My test suite contains shorter latencies (our DAGs contain many latency 0 edges, which are used to capture anti-dependencies and output dependencies between two instructions).

- My test suite contains many larger basic blocks (work [60] used the GCC compiler and the largest DAG was 1236 instructions).
- My test suite contains more speculation (more instructions that can move up to higher basic blocks) as there is little speculation after register allocation.

Table 4.2 gives the total time (hh:mm:ss) to schedule all superblocks in the SPEC 2000 benchmark suite, for various idealized architectural models and time limits for solving each superblock. Depending on the architectural model, the optimal scheduler took between 3:18:13 and 4:59:10 to schedule all of the superblocks in the entire SPEC benchmark. Table 4.3 gives the percentage of all superblocks in the SPEC 2000 benchmark suite which were solved to optimality, for various idealized architectural models and time limits for solving each superblock.

Table 4.4 gives the number of superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of superblocks with improved schedules (%), for various idealized architectural models over DHASY heuristic. One can see that the optimal scheduler is able to improve at least 3 superblocks and at most 1059 superblocks in one application. Overall, the improvement is from 2640 to 10,024 superblocks.

Table 4.5 gives an average and maximum percentage improvements in schedule cost of optimal schedule over schedule found by list scheduler using the heuristic, for various idealized architectural models. The average is over *only* the superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule. The minimum and maximum percentage improvement per superblock is 4.3 and 49 respectively. The average minimum and maximum percentage improvement per application is 0.3 and 7.2 respectively. Overall, the improvement is from 4.0 to 49 percent.

Table 4.6 gives the number of superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of superblocks with improved schedules (%), for ranges of superblock sizes and various idealized architectural models.

Table 4.7 gives the average and maximum percentage improvements in schedule cost of optimal schedule over schedule found by list scheduler using the heuristic, for ranges of block sizes and various idealized architectural models. The average is over *only* the superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

In summary, global instruction scheduling is a more complex problem than local instruction scheduling, and even on an idealized architectural model, list scheduling does not perform as well for global instruction scheduling as it did for local instruction scheduling.

Table 4.2: Total time (hh:mm:ss) to schedule all superblocks in the SPEC 2000 benchmark suite, for various idealized architectural models and time limits for solving each superblock.

	1 sec.	10 sec.	1 min.	10 min.
1-issue	56:30	2:11:39	2:41:13	3:18:13
2-issue	56:54	2:22:30	3:21:46	4:59:10
4-issue	43:16	1:44:18	2:39:38	4:29:38
6-issue	38:29	1:20:52	1:48:14	2:50:52

Table 4.3: Percentage of all superblocks in the SPEC 2000 benchmark suite which were solved to optimality, for various idealized architectural models and time limits for solving each superblock.

	1 sec.	10 sec.	1 min.	10 min.
1-issue	99.225	99.897	99.990	99.999
2-issue	99.196	99.846	99.983	99.995
4-issue	99.454	99.878	99.981	99.995
6-issue	99.578	99.923	99.988	99.997

Table 4.4: *Dependence height and speculative yield heuristic.* Number of superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of superblocks with improved schedules (%), for various idealized architectural models.

		1-issue		2-issue		4-issue		6-issue	
	#blocks	imp.	%	imp.	%	imp.	%	imp.	%
ammp	2,972	147	4.9	150	5.0	103	3.5	94	3.2
applu	306	40	13.1	46	15.0	32	10.5	23	7.5
apsi	1,727	109	6.3	123	7.1	112	6.5	112	6.5
art	439	30	6.8	30	6.8	5	1.1	3	0.7
bzip2	1,087	95	8.7	95	8.7	49	4.5	17	1.6
crafty	4,773	444	9.3	444	9.3	313	6.6	103	2.2
eon	2,514	110	4.4	100	4.0	109	4.3	72	2.9
equake	227	18	7.9	21	9.3	12	5.3	4	1.8
facerec	1,125	70	6.2	83	7.4	91	8.1	43	3.8
fma3d	12,380	757	6.1	832	6.7	670	5.4	313	2.5
galgel	3,839	279	7.3	300	7.8	191	5.0	96	2.5
gap	19,651	840	4.3	840	4.3	676	3.4	254	1.3
gcc	43,509	2,972	6.8	2,963	6.8	1,529	3.5	317	0.7
gzip	1,339	128	9.6	128	9.6	79	5.9	38	2.8
lucas	1,057	34	3.2	42	4.0	48	4.5	21	2.0
mcf	337	18	5.3	18	5.3	16	4.7	7	2.1
mesa	11,555	697	6.0	660	5.7	624	5.4	236	2.0
mgrid	132	13	9.8	14	10.6	16	12.1	7	5.3
parser	3,198	186	5.8	186	5.8	115	3.6	24	0.8
perlbmk	16,915	1,059	6.3	1,059	6.3	680	4.0	195	1.2
sixtrack	7,372	444	6.0	459	6.2	613	8.3	423	5.7
swim	81	6	7.4	7	8.6	8	9.9	3	3.7
twolf	6,832	361	5.3	356	5.2	214	3.1	55	0.8
vortex	8,061	823	10.2	824	10.2	366	4.5	97	1.2
vpr	2,830	198	7.0	197	7.0	141	5.0	65	2.3
wupwise	393	40	10.2	47	12.0	33	8.4	18	4.6
Total	154,651	9,918	6.4	10,024	6.5	6,845	4.4	2,640	1.7

Table 4.5: *Dependence height and speculative yield heuristic.* Average and maximum percentage improvements in schedule cost of optimal schedule over schedule found by list scheduler using the heuristic, for various idealized architectural models. The average is over *only* the superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

	1-issue		2-issue		4-issue		6-issue	
	ave.	max.	ave.	max.	ave.	max.	ave.	max.
ammp	4.3	47.4	4.2	47.4	3.2	23.3	3.0	14.3
applu	1.2	12.0	1.8	12.0	4.7	12.5	7.1	16.7
apsi	2.5	16.2	2.6	16.2	2.6	16.7	3.2	12.5
art	4.4	48.1	4.4	48.1	3.7	12.4	2.4	5.3
bzip2	4.4	48.8	4.4	48.8	4.9	14.6	4.7	11.8
crafty	5.7	49.0	5.7	49.0	4.7	36.8	3.9	13.3
eon	3.8	48.5	3.9	48.5	3.4	26.5	4.4	26.5
equake	1.5	12.9	2.2	12.9	4.5	11.1	1.6	4.3
facerec	4.1	47.6	4.6	47.6	4.0	15.4	4.4	14.2
fma3d	4.8	48.5	4.6	48.5	3.4	17.4	4.1	17.4
galgel	6.4	49.0	6.2	49.0	5.0	16.7	3.8	14.3
gap	4.7	49.0	4.7	49.0	3.8	16.5	5.5	20.0
gcc	5.8	49.0	5.8	49.0	3.8	80.7	5.5	49.0
gzip	3.5	47.4	3.5	47.4	3.6	21.1	6.9	16.7
lucas	3.6	47.4	3.4	47.4	3.3	10.7	5.5	11.2
mcf	0.3	2.0	0.3	2.0	1.9	8.0	4.6	10.1
mesa	5.7	49.0	6.0	49.0	5.0	26.4	4.7	12.3
mgrid	0.4	1.5	0.8	6.5	2.5	13.8	1.1	4.3
parser	7.2	48.5	7.2	48.5	5.7	20.0	5.0	16.7
perlbmk	4.2	49.0	4.2	49.0	4.1	28.0	4.4	16.8
sixtrack	2.9	48.1	3.1	48.1	3.2	16.7	3.1	19.0
swim	3.0	5.9	2.8	6.7	4.9	8.9	9.4	11.1
twolf	4.9	48.6	5.0	48.6	3.3	16.5	4.8	24.5
vortex	3.9	48.8	3.9	48.8	4.2	14.3	3.3	14.3
vpr	6.2	47.8	6.2	47.8	4.6	29.7	5.1	16.5
wupwise	7.1	25.0	7.1	25.0	7.2	16.7	3.6	11.1
Overall	5.0	49.0	5.0	49.0	4.0	39.0	4.3	49.0

Table 4.6: *Dependence height and speculative yield heuristic.* Number of superblocks where the optimal scheduler found an improved schedule (imp.), and the percentage of superblocks with improved schedules (%), for ranges of superblock sizes and various idealized architectural models.

		1-issue		2-issue		4-issue		6-issue	
range	#blocks	imp.	%	imp.	%	imp.	%	imp.	%
3–5	13,629	36	0.3	36	0.3	0	0.0	0	0.0
6–10	39,948	866	2.2	849	2.1	168	0.4	21	0.1
11–15	32,105	1,457	4.5	1,449	4.5	831	2.6	166	0.5
16–20	21,461	1,699	7.9	1,686	7.9	888	4.1	278	1.3
21–30	22,752	2,330	10.2	2,305	10.1	1,680	7.4	575	2.5
31–50	14,876	1,887	12.7	1,939	13.0	1,697	11.4	644	4.3
51–100	7,321	1,155	15.8	1,174	16.0	1,085	14.8	616	8.4
101–250	2,231	398	17.8	484	21.7	399	17.9	260	11.7
251–2750	328	90	27.4	102	31.1	97	29.6	80	24.4
Total	154,651	9,918	6.4	10,024	6.5	6,845	4.4	2,640	1.7

Table 4.7: *Dependence height and speculative yield heuristic.* Average and maximum percentage improvements in schedule cost of optimal schedule over schedule found by list scheduler using the heuristic, for ranges of block sizes and various idealized architectural models. The average is over *only* the superblocks where the optimal scheduler found an improved schedule.

	1-issue		2-issue		4-issue		6-issue	
range	ave.	max.	ave.	max.	ave.	max.	ave.	max.
3–5	20.2	26.9	20.2	26.9		0.0		0.0
6–10	14.8	49.0	14.9	49.0	9.0	49.0	10.1	49.0
11–15	7.7	48.6	7.8	48.6	6.1	18.9	7.8	20.0
16–20	5.7	47.6	5.7	47.6	5.1	25.6	7.0	24.5
21–30	3.6	46.5	3.6	46.5	4.3	28.0	5.7	26.5
31–50	2.6	45.6	2.7	45.6	3.3	36.8	4.1	20.3
51–100	1.2	40.7	1.3	40.7	2.3	29.7	2.5	17.4
101–250	0.9	13.1	1.1	13.1	1.7	80.7	1.6	9.4
251–2750	0.9	13.3	1.1	13.3	1.7	17.8	0.8	3.9
Overall	5.0	49.0	5.0	49.0	4.0	39.0	4.3	49.0

4.4.2 Experiments for realistic architectural models

There are no new changes needed to the improved architectural model to accommodate global instruction scheduling.

Table 4.8 gives the total time (hh:mm:ss) to schedule all superblocks in the SPEC 2000 benchmark suite, for various realistic architectural models and time limits for solving each superblock. Depending on the architectural model, the optimal scheduler took between 22:35:33 and 752:28:12 to schedule all of the superblocks in the entire SPEC benchmark. Table 4.9 gives the percentage of all superblocks in the SPEC 2000 benchmark suite which were solved to optimality, for various realistic architectural models and time limits for solving each superblock.

Table 4.10 gives the number of superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of superblocks with improved schedules (%), for various realistic architectural models for the DHASY heuristic. One can see that the optimal scheduler is able to improve between 2.7% and 44.4% of all superblocks, depending on the application. Across all benchmarks, the total improvement is from 8,165 to 32,128 superblocks.

Table 4.11 gives an average and maximum percentage improvements in schedule cost of optimal schedule over schedule found by list scheduler using the heuristic, for various realistic architectural models. The average is over *only* the superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule. The maximum percentage improvement in an application ranges from 91.0% to 417.0%. The average percentage improvement in an application ranges from 2.5% to 20.2%. Overall, the average improvement is from 5.0% to 6.5%.

Table 4.12 gives the number of superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of superblocks with improved schedules (%), for ranges of superblock sizes and various idealized architectural models. Table 4.13 gives the average and maximum percentage improvements in schedule cost of optimal schedule over schedule found by list scheduler using the heuristic, for ranges of block sizes and various idealized architectural models. The average is over *only* the superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

Table 4.8: Total time (hh:mm:ss) to schedule all superblocks in the SPEC 2000 benchmark suite, for various realistic architectural models and time limits for solving each superblock.

	1 sec.	10 sec.	1 min.	10 min.
1r-issue	1:37:30	8:27:31	14:08:21	22:35:33
ppc603e	4:31:32	36:41:56	128:10:03	752:28:12
ppc604	2:43:33	21:18:18	74:11:46	399:51:12
6r-issue	3:11:40	26:18:09	91:40:36	534:32:10

Table 4.9: Percentage of all superblocks in the SPEC 2000 benchmark suite which were solved to optimality, for various realistic architectural models and time limits for solving each superblock.

	1 sec.	10 sec.	1 min.	10 min.
1r-issue	97.022	99.128	99.880	99.978
ppc603e	90.410	92.593	96.750	97.455
ppc604	94.451	95.647	98.147	98.709
6r-issue	93.305	94.486	97.650	98.227

Table 4.10: *Dependence height and speculative yield heuristic.* Number of superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of superblocks with improved schedules (%), for various realistic architectural models.

		1r-issue		ppc603e		ppc604		6r-issue	
	#blocks	imp.	%	imp.	%	imp.	%	imp.	%
ammp	2,972	653	22.0	381	12.8	212	7.1	171	5.8
applu	306	110	35.9	98	32.0	38	12.4	62	20.3
apsi	1,727	644	37.3	425	24.6	292	16.9	244	14.1
art	439	85	19.4	39	8.9	17	3.9	12	2.7
bzip2	1,087	267	24.6	159	14.6	52	4.8	40	3.7
crafty	4,773	1,085	22.7	662	13.9	254	5.3	241	5.0
eon	2,514	604	24.0	453	18.0	299	11.9	266	10.6
equake	227	58	25.6	45	19.8	20	8.8	13	5.7
facerec	1,125	330	29.3	217	19.3	142	12.6	152	13.5
fma3d	12,380	2,479	20.0	2,162	17.5	1,039	8.4	1,108	8.9
galgel	3,839	984	25.6	765	19.9	370	9.6	292	7.6
gap	19,651	3,482	17.7	1,940	9.9	653	3.3	602	3.1
gcc	43,509	7,538	17.3	4,242	9.7	1,546	3.6	1,409	3.2
gzip	1,339	323	24.1	227	17.0	107	8.0	77	5.8
lucas	1,057	221	20.9	93	8.8	42	4.0	36	3.4
mcf	337	70	20.8	43	12.8	24	7.1	12	3.6
mesa	11,555	2,155	18.6	1,540	13.3	859	7.4	653	5.7
mgrid	132	56	42.4	31	23.5	16	12.1	16	12.1
parser	3,198	709	22.2	449	14.0	179	5.6	160	5.0
perlbmk	16,915	3,478	20.6	1,844	10.9	1,151	6.8	796	4.7
sixtrack	7,372	2,169	29.4	1,701	23.1	943	12.8	720	9.8
swim	81	36	44.4	22	27.2	22	27.2	8	9.9
twolf	6,832	1,311	19.2	655	9.6	355	5.2	323	4.7
vortex	8,061	2,446	30.3	1,964	24.4	672	8.3	546	6.8
vpr	2,830	731	25.8	503	17.8	221	7.8	192	6.8
wupwise	393	104	26.5	67	17.0	32	8.1	14	3.6
Total	154,651	32,128	20.8	20,727	13.4	9,557	6.2	8,165	5.3

Table 4.11: *Dependence height and speculative yield heuristic.* Average and maximum percentage improvements in schedule length of optimal schedule over schedule found by list scheduler using the heuristic, for various realistic architectural models. The average is over *only* the superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

	1r-issue		ppc603e		ppc604		6r-issue	
	ave.	max.	ave.	max.	ave.	max.	ave.	max.
ammp	8.4	124.1	5.7	81.3	4.9	47.4	4.9	43.0
applu	8.8	55.7	3.5	41.7	4.5	32.3	2.7	11.3
apsi	8.3	71.2	5.6	53.7	7.0	62.8	6.1	61.9
art	8.3	63.8	7.5	27.3	4.9	25.0	5.2	21.4
bzip2	5.5	36.7	5.4	40.0	6.0	37.5	6.1	34.6
crafty	6.6	74.9	4.2	40.0	3.7	40.5	3.9	41.2
eon	8.9	62.1	5.1	40.0	5.3	34.9	6.1	45.7
equake	3.7	37.2	5.0	12.3	2.5	6.2	3.0	11.0
facerec	8.0	49.2	4.9	39.6	6.6	46.2	5.6	50.0
fma3d	9.4	137.3	5.2	92.1	6.3	91.0	6.2	91.0
galgel	9.9	119.3	5.1	155.0	6.4	154.5	5.1	40.4
gap	6.4	115.0	5.6	139.4	4.9	45.0	4.6	45.0
gcc	4.7	417.4	4.4	70.0	4.6	56.6	4.3	50.0
gzip	5.7	87.7	4.4	90.0	6.8	88.2	6.4	88.2
lucas	5.1	52.3	3.7	39.8	5.3	40.6	3.7	29.6
mcf	5.0	47.2	5.3	52.7	8.3	53.3	9.2	55.3
mesa	7.6	73.1	5.2	52.8	5.5	52.8	5.4	51.4
mgrid	4.7	21.7	4.1	17.4	5.5	18.9	3.6	12.5
parser	4.6	34.8	4.5	26.2	4.7	36.4	4.4	36.4
perlbmk	4.4	54.0	4.4	46.5	4.9	43.5	4.9	43.5
sixtrack	9.3	341.7	5.3	141.9	5.1	141.9	5.2	77.6
swim	20.2	68.5	6.3	36.1	5.6	19.6	9.5	13.6
twolf	7.0	66.7	5.8	42.9	5.1	40.9	4.6	37.9
vortex	6.3	282.4	5.3	126.4	6.2	62.5	6.2	68.6
vpr	7.2	74.1	5.0	59.5	4.9	44.7	5.7	42.1
wupwise	6.4	23.1	4.0	16.0	4.4	11.0	3.2	13.8
Overall	6.5	417.4	5.0	155.0	5.3	154.5	5.2	91.0

Table 4.12: *Dependence height and speculative yield heuristic.* Number of superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule (imp.), and the percentage of superblocks with improved schedules (%), for ranges of superblock sizes and various realistic architectural models.

		1r-issue		ppc603e		ppc604		6r-issue	
range	#blocks	imp.	%	imp.	%	imp.	%	imp.	%
3–5	14,811	153	1.0	14	0.1	9	0.1	9	0.1
6–10	40,984	2,460	6.0	615	1.5	242	0.6	179	0.4
11–15	31,622	5,155	16.3	2,375	7.5	669	2.1	559	1.8
16–20	20,717	4,973	24.0	3,095	14.9	1,031	5.0	694	3.3
21–30	22,147	7,410	33.5	5,477	24.7	2,156	9.7	1,889	8.5
31–50	14,610	6,716	46.0	5,971	40.9	3,130	21.4	2,945	20.2
51–100	7,221	3,918	54.3	3,452	47.8	2,343	32.4	1,968	27.3
101–250	2,211	1,185	53.6	1,013	45.8	621	28.1	573	25.9
251–2750	328	174	53.0	140	42.7	103	31.4	89	27.1
Total	154,651	32,144	20.8	22,152	14.3	10,304	6.7	8,905	5.8

Table 4.13: *Dependence height and speculative yield heuristic.* Average and maximum percentage improvements in schedule length of optimal schedule over schedule found by list scheduler using the heuristic, for ranges of block sizes and various realistic architectural models. The average is over *only* the superblocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved schedule.

	1r-issue		ppc603e		ppc604		6r-issue	
range	ave.	max.	ave.	max.	ave.	max.	ave.	max.
3–5	5.0	58.6	8.8	57.1	0.7	3.6	0.7	3.6
6–10	6.6	214.2	6.6	58.1	5.7	43.2	4.3	22.2
11–15	7.2	417.4	6.7	59.4	5.8	49.1	5.1	37.6
16–20	6.3	101.0	5.5	84.0	5.8	51.5	5.9	51.5
21–30	6.1	282.4	5.1	141.9	5.8	141.9	5.5	51.4
31–50	6.9	137.3	4.6	89.6	5.7	88.2	5.7	61.9
51–100	6.0	341.7	4.0	155.0	4.6	154.5	4.5	77.6
101–250	6.2	127.6	3.3	92.1	3.4	91.0	3.5	91.0
251–2750	6.8	87.7	4.0	93.6	2.4	88.2	3.7	88.2
Overall	6.5	417.4	5.0	155.0	5.3	154.5	5.2	91.0

I also evaluated my optimal scheduler with respect to the number of cycles saved against the two heuristic schedulers. Table 4.15 gives the number of cycles saved by the optimal scheduler over the list scheduler using the dependence height and speculative yield heuristic ($\times 10^9$), and the percentage reduction (%), for various realistic architectural models after register allocation. Table 4.16 gives the number of cycles saved by the optimal scheduler over the list scheduler using the critical path heuristic ($\times 10^9$), and the percentage reduction (%), for various realistic architectural models after register allocation. However, this may not translate into time saved because of cache misses and unlimited registers. I compiled the SPEC 2000 benchmark with the training data set associated with the benchmark using the Tobey compiler. The compiler uses the training data to construct a profile for each branch instruction. The profile is used to calculate the information regarding the number of times each basic block is executed in the benchmark. Example 4.6 explains how I calculate the number of cycles saved in Table 4.16 and Table 4.15.

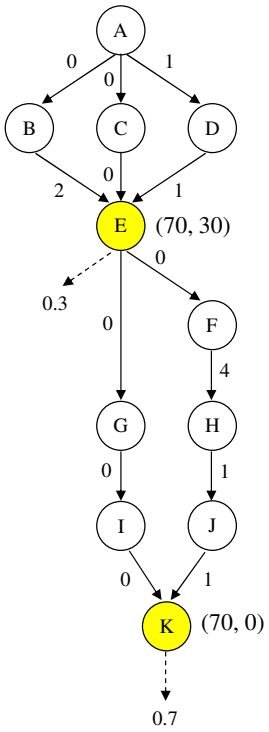


Figure 4.8: Superblock for Example 4.6.

Example 4.6 *Consider the superblock in Figure 4.8. The superblock consists of two basic blocks, B_1 and B_2. Basic block B_1 consists of nodes A, B, C, D and E. Basic block B_2 consists of nodes F, G, H, I, J and K. Nodes E and K are branch instructions. Assume a fully pipelined processor with two functional units which are capable of issuing any type of instruction. The processor can issue two instructions in each cycle. Let B_1 be executed 100 times. The numbers beside node*

Cycle	S_1		S_2	
0	A	C	A	B
1	B	D	C	D
2			E	F
3	E	G	G	
4	F	I	I	
5				
6			H	
7			J	
8	H		K	
9	J			
10	K			

Table 4.14: Two possible schedules for the DAG in Figure 4.8. Empty slots represent NOPs. S_1 is a non-optimal schedule. S_2 is an optimal schedule.

E, (70, 30), give the information that the branch is taken—i.e., the flow of control exits at node E—30 times and is not taken—i.e., the flow of control falls through—70 times. Similarly, the numbers beside node K tell how many times this branch is taken and not taken. Basic block B_2 is executed only when the branch instruction at node E is not taken. Hence, B_2 is executed 70 times. Table 4.14 shows an optimal and a non-optimal schedule for the given superblock. The schedule length of B_1 in S_2, an optimal schedule, is 1 cycle shorter than in S_1, a non-optimal schedule. As B_1 is executed 100 times, the saving in B_1 is 100 cycles with S_2 . Similarly, the schedule length of B_2 in S_2 is 1 cycle shorter than in S_1. As B_2 is executed 70 times, the saving in this basic block is 70 cycles with S_2. In total 170 cycles are being saved with S_2.

Figure 4.9 and Figure 4.10 show performance guarantees for the list scheduler using the dependence height and speculative yield (DHASY) heuristic in terms of worst-case factors from optimal, for various architectures. For example, consider the 1r-issue architecture. The list scheduler finds an optimal schedule (i.e. is within 0% of optimal) for approximately 84% of all superblocks before register allocation and approximately 88% of all superblocks after register allocation. Further, the list scheduler is within 10% of optimal for approximately 95% of all superblocks before register allocation and approximately 97% of all superblocks after register allocation, for this architecture.

Figure 4.11 and Figure 4.12 show performance guarantees for the list scheduler using the critical path heuristic in terms of worst-case factors from optimal, for various architectures. Consider once again the 1r-issue architecture. The list scheduler finds an optimal schedule (i.e. is within 0% of optimal) for approximately 54% of all superblocks before register allocation and approximately 65% of all superblocks after register allocation. Further, the list scheduler is within 10% of optimal for approximately 70% of all superblocks before register allocation and approximately 80% of all

superblocks after register allocation, for this architecture.

Table 4.15: *Superblock scheduling after register allocation.* For the SPEC 2000 benchmark suite, number of cycles saved by the optimal scheduler over the list scheduler using the dependence height and speculative yield heuristic ($\times 10^9$), and the percentage reduction (%), for various realistic architectural models.

	1r-issue		ppc603e		ppc604		6r-issue	
	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%
ammp	56.3	0.2	669.1	3.2	227.1	1.1	375.7	2.2
applu	5.2	0.4	0.9	0.1	0.6	0.1	3.5	0.6
apsi	52.4	1.1	56.0	1.3	50.2	1.2	45.2	1.4
art	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bzip2	51.8	0.3	282.1	1.8	281.6	1.9	137.5	0.9
crafty	50.9	0.7	63.2	1.1	31.3	0.6	32.5	0.6
eon	303.1	2.7	96.9	1.0	53.0	0.6	188.2	2.2
equake	22.4	0.5	12.2	0.3	11.8	0.3	0.2	0.0
facerec	19.4	0.3	27.1	0.5	3.6	0.1	1.6	0.0
fma3d	36.4	0.4	52.7	0.7	88.9	1.2	29.9	0.5
galgel	1.4	0.1	0.7	0.1	0.5	0.1	0.1	0.0
gap	18.9	0.0	124.2	0.0	48.7	0.0	31.6	0.0
gcc	28.7	0.6	33.7	0.8	20.9	0.5	17.6	0.4
gzip	5.9	0.0	37.3	0.3	22.6	0.2	30.5	0.2
lucas	0.0	0.0	0.2	0.1	0.0	0.0	0.0	0.0
mcf	54.1	1.5	43.9	1.4	38.9	1.2	34.3	1.1
mesa	34.0	0.2	65.2	0.5	32.2	0.3	13.9	0.1
mgrid	1.7	0.5	0.3	0.1	0.0	0.0	0.0	0.0
parser	483.1	1.9	577.7	2.8	507.9	2.6	326.3	1.8
perlbmk	67.8	0.2	391.1	1.5	117.2	0.5	79.1	0.3
sixtrack	122.6	3.5	6.3	0.2	4.0	0.1	1.3	0.0
swim	0.0	0.2	0.1	1.7	0.1	1.9	0.0	0.0
twolf	288.5	1.5	56.9	0.3	88.6	0.6	76.6	0.5
vortex	40.4	0.4	275.4	3.4	286.5	3.8	227.9	3.4
vpr	57.4	0.5	30.1	0.3	41.1	0.5	6.1	0.1
wupwise	23.8	0.3	30.3	0.4	20.8	0.3	14.2	0.2

Table 4.16: *Superblock scheduling after register allocation.* For the SPEC 2000 benchmark suite, number of cycles saved by the optimal scheduler over the list scheduler using the critical path heuristic ($\times 10^9$), and the percentage reduction (%), for various realistic architectural models.

	1r-issue		ppc603e		ppc604		6r-issue	
	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%
ammp	467.4	1.8	953.2	4.5	243.6	1.2	401.5	2.4
applu	23.7	1.9	8.3	0.7	0.5	0.0	3.8	0.6
apsi	341.5	7.1	95.7	2.2	88.1	2.1	74.5	2.2
art	2.7	0.1	1.1	0.0	1.1	0.0	1.1	0.0
bzip2	300.1	1.5	405.0	2.5	356.7	2.3	167.2	1.1
crafty	162.1	2.3	120.5	2.1	55.5	1.0	50.0	1.0
eon	610.6	5.5	329.1	3.4	271.2	2.9	286.7	3.4
equake	20.6	0.5	1.6	0.0	1.2	0.0	0.3	0.0
facerec	28.7	0.5	33.4	0.7	3.6	0.1	2.1	0.0
fma3d	51.8	0.5	94.7	1.2	108.8	1.4	34.9	0.6
galgel	4.8	0.4	2.2	0.2	1.3	0.1	0.7	0.1
gap	99.9	0.0	131.2	0.0	49.8	0.0	29.1	0.0
gcc	65.9	1.3	52.3	1.2	28.9	0.7	23.4	0.6
gzip	151.8	1.0	51.2	0.4	22.7	0.2	18.5	0.1
lucas	4.5	1.2	0.5	0.2	0.0	0.0	0.0	0.0
mcf	89.0	2.5	93.7	2.9	94.9	3.0	58.1	1.9
mesa	85.3	0.6	40.2	0.3	31.0	0.3	12.6	0.1
mgrid	3.1	0.9	0.5	0.2	0.0	0.0	0.0	0.0
parser	956.8	3.8	855.9	4.1	526.0	2.7	388.9	2.1
perlbmk	181.7	0.6	448.0	1.8	142.0	0.6	109.2	0.5
sixtrack	655.4	18.6	19.8	0.6	7.1	0.2	5.2	0.2
swim	8.5	99.8	6.6	102.3	3.2	58.1	3.1	62.7
twolf	689.0	3.5	390.3	2.3	199.8	1.3	25.7	0.2
vortex	211.1	2.0	398.7	4.9	330.2	4.4	230.0	3.4
vpr	224.2	2.1	114.8	1.3	74.1	0.8	46.1	0.5
wupwise	463.8	5.5	211.0	2.9	69.3	1.0	15.0	0.2

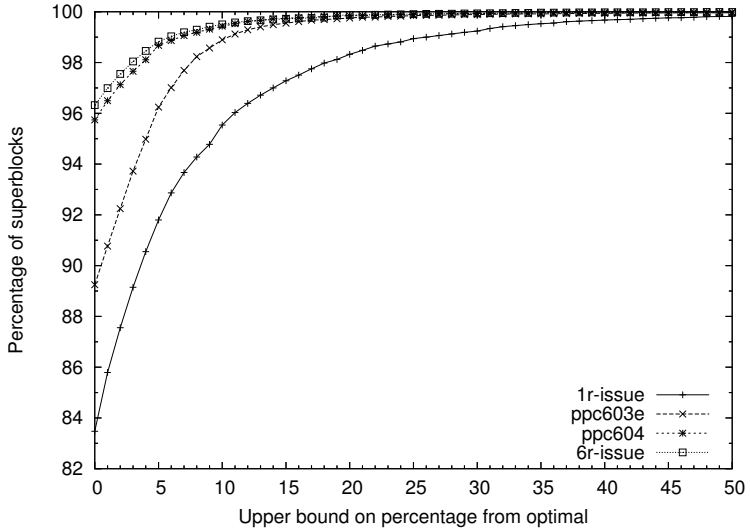


Figure 4.9: *Superblock scheduling before register allocation.* Performance guarantees for the list scheduler using the dependence height and speculative yield heuristic in terms of worst-case factors from optimal, for various realistic architectures.

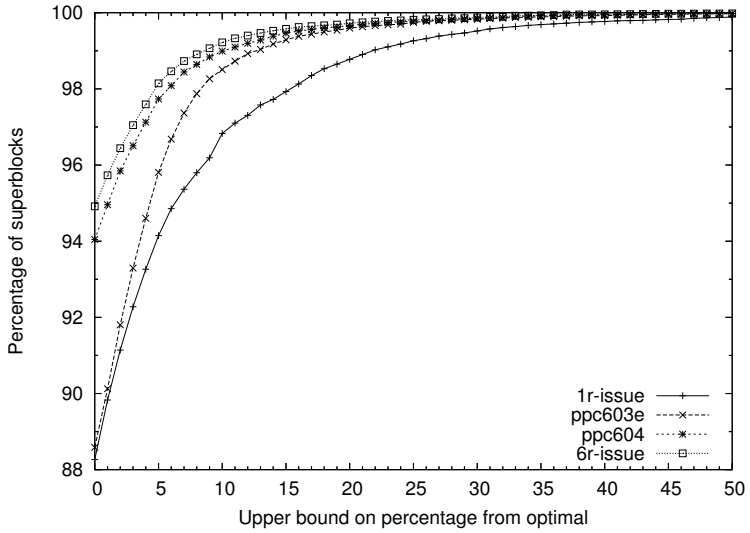


Figure 4.10: *Superblock scheduling after register allocation.* Performance guarantees for the list scheduler using the dependence height and speculative yield heuristic in terms of worst-case factors from optimal, for various realistic architectures.

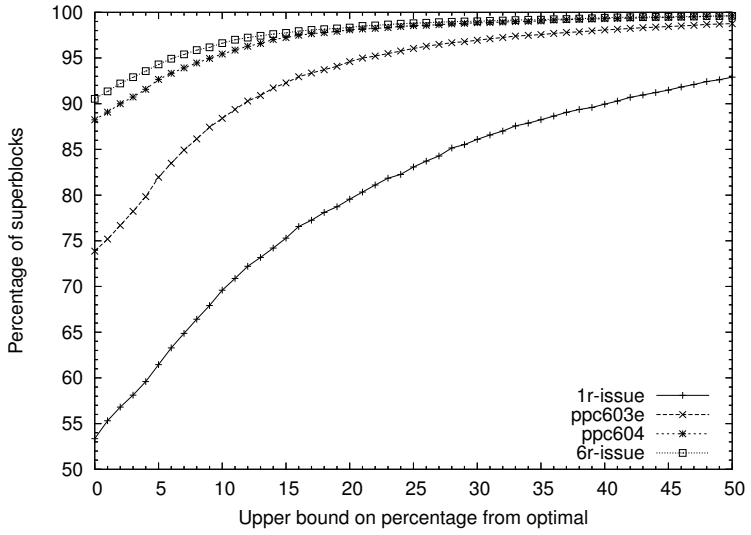


Figure 4.11: *Superblock scheduling before register allocation.* Performance guarantees for the list scheduler using the critical path heuristic in terms of worst-case factors from optimal, for various realistic architectures.

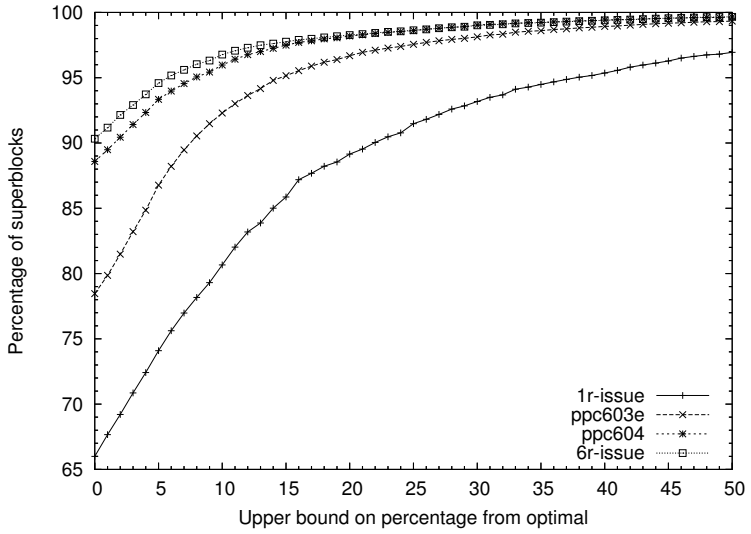


Figure 4.12: *Superblock scheduling after register allocation.* Performance guarantees for the list scheduler using the critical path heuristic in terms of worst-case factors from optimal, for various realistic architectures.

4.5 Summary

I presented a constraint programming approach to superblock instruction scheduling for multiple-issue processors for both idealized and realistic architectures. The problem is considered intractable, yet my approach is optimal and robust on large, real problems. The key to scaling up to large, real problems was in the development of an improved constraint model and the application of more powerful constraint propagation techniques. I experimentally evaluated my optimal scheduler on the SPEC2000 integer and floating point benchmarks. On this benchmark suite, the optimal scheduler was very robust and scaled to the largest basic blocks. Depending on the architectural model, between 99.991% to 99.999% of all superblocks were solved to optimality. The scheduler was able to routinely solve the largest superblocks, including blocks with up to 2600 instructions. This compares favorably to the best previous approach by Shobaki [61]. Shobaki's work considered idealized architectures with latency from 1 cycle to 9 cycles with maximum basic block size of 1200 instructions. For the global instruction scheduling problem, the critical path heuristic and the DHASY heuristic were used for comparison against the optimal scheduler. When scheduling for the idealized architectural model, the list scheduler solved 91.2%-97.5% of superblocks optimally. However, the list scheduler was only optimal for 54%-96% of superblocks when scheduling for a realistic architectural model. The schedules produced by the optimal schedule showed an improvement of 0%-3.8% on average over DHASY heuristic and an improvement of 0%-102% on average over the critical path heuristic. As expected, the heuristic DHASY yielded better schedules than the critical path heuristic. In the next chapter, I will present my constraint programming model for basic block scheduling without spilling.

Chapter 5

Basic Block Scheduling without Spilling for Multi-Issue Processors

In this chapter, I present an approach to finding an optimal schedule for a basic block with register pressure less than or equal to the available physical registers. The approach builds on the constraint programming model for basic block scheduling given in Chapter 3 and is for both idealized and realistic architectures.

5.1 Introduction

The goals of an instruction scheduler and a register allocator are orthogonal to each other. An instruction scheduler attempts to increase the instruction level parallelism by re-ordering the given instruction sequence. However, packing the given instructions into the shortest possible schedule may increase the register pressure. On the other hand, a register allocator attempts to separate the live ranges of data variables—the time interval within which the variable is used—in order to decrease the register pressure which may have the side effect of reducing the instruction level parallelism. Which phase should be done first? This is an important question in high performance computing. The question is commonly known as the *phase ordering problem*. No matter which optimization is done first, the earlier phase has to make decisions without knowing what the later phase will do. The combined instruction scheduling and register allocation approach is an answer to this problem. In a combined instruction scheduling and register allocation approach, instruction scheduling and register allocation are done side by side. The goal is to have a schedule of instructions with the minimum length that will introduce the minimum spill code[1].

[1]The movement of data from registers to memory is called *spilling*.

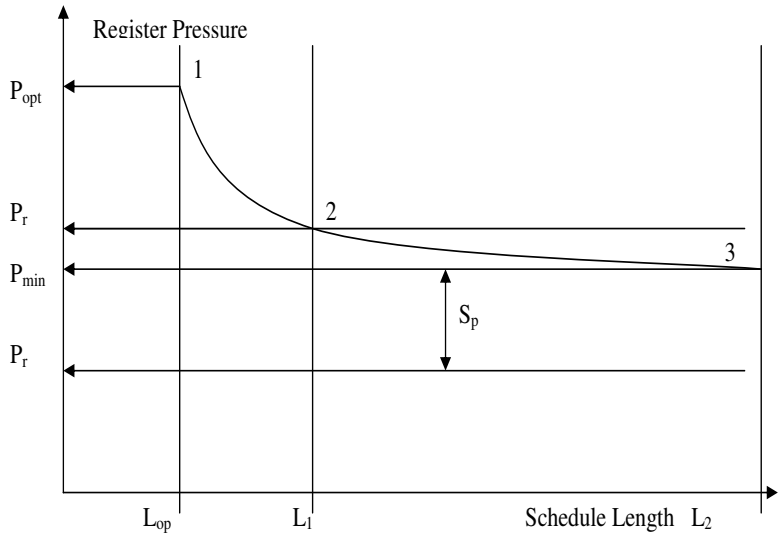


Figure 5.1: Basic block scheduling without spilling.

Definition 5.1 (Minimum register requirement) *Given a labeled dependency DAG for a basic block, the* minimum register requirement *is the minimum register pressure over all schedules for the block.*

Consider Figure 5.1. For a given basic block, let L_{op} be the minimum schedule length and P_{opt} be the minimum register requirement for a schedule with length L_{op}. Point P_{min} on the register pressure axis gives the minimum register requirement over all schedules for the given basic block. Let P_r be the available number of physical registers. There are two cases.

- **Case 1:** $P_r < P_{min}$**.** The number of available physical registers is less than the minimum register requirement. The difference S_p gives the register pressure that has to be eliminated in order to get a legal schedule for a given basic block.
- **Case 2:** $P_r \geq P_{min}$**.** The number of available physical registers is greater than or equal to the minimum register requirement. In this case, one has to find a schedule which has the minimum schedule length with register pressure less than or equal to P_r. Point 2 represents this situation.

For my work, I consider the second case only. I define the basic block scheduling without spilling problem as follows.

Definition 5.2 (Basic block scheduling without spilling) *Given a labeled dependency DAG for a basic block and the number of available physical registers, the* basic block scheduling without

spilling problem *is to find a schedule with minimum length over all schedules with register pressure less than or equal to the available physical registers, if such a schedule exists.*

My interest in the problem is motivated by the challenges being faced in compilers for modern processors. Although modern processors have many registers which can be used to break false dependencies, sometimes instruction level parallelism must be sacrificed to avoid spilling. Reducing spilling reduces the number of loads and stores executed, which in turn is important for architectures that either have a small cache or a large cache miss penalty; for minimizing memory bandwidth usage; for instruction level parallelism, as the elimination of some spill instructions frees instruction slots to issue other useful instructions; for power dissipation, as load and store instructions contribute to a significant portion of the power consumed; and for multi-threading for some architectures (e.g., IA-64) as minimal register usage lowers the cost of context switching.

In this chapter, I solve the basic block scheduling without spilling problem by combining the concept of lineage developed by Govindarajan et al. [28] and the constraint programming model for basic block scheduling developed in Chapter 3. The experimental results show that the approach is good enough to solve basic blocks as large as 50 instructions within a time bound of 10 minutes for both idealized and realistic architectures. This compares favorably to the recent work by Bednarski and Kessler [5, 6] on optimal integrated code generation using integer programming. The work by Bednarski and Kessler is targeted only towards an idealized multi-issue VLIW processor and can solve basic blocks as large as 50 instructions within 20 minutes for unit latency and basic blocks as large as 20 instructions with arbitrary latencies. In my work, I consider both idealized and realistic multi-issue processors with arbitrary latencies ranging from 0 cycles to 36 cycles and am able to solve basic blocks as large as 50 instructions within 10 minutes.

5.2 Related work

The ordering of the instruction scheduling and register allocation phases has been studied extensively for both in-order (VLIW) and out-of-order (superscalar) architectures. The combined instruction scheduling and register allocation problem is NP-complete [50]. Little work has been done on solving it optimally because of the complexity of the problem. Previous work has mostly focused on heuristic or sub-optimal approaches. Integrated techniques that attempt to minimize register pressure while exposing instruction level parallelism have also been proposed. First, I will discuss the heuristic approaches and then the work related to finding optimal solutions.

Goodman and Hsu [26] propose solving the phase ordering problem by maintaining register pressure within a certain limit. Their algorithm consists of two modes. One mode does the instruction scheduling and increases the instruction level parallelism as long as the register pressure is below the limit. When the register pressure crosses the limit, the scheduler switches to the second mode which does the instruction scheduling in favor of the register allocation phase; i.e,

it prefers instructions that reduce the register pressure. This mode is continued until the register pressure is less than or equal to the limit. Usually, the limit is equal to the number of available physical registers. This approach has been widely adopted in commercial compilers, including the IBM Tobey compiler [9]. However, Touati [64] has shown in experiments that this method can result in much spilling.

Bradlee, Eggers, and Henry [10] introduce a three pass system called RASE (*register allocation with schedule estimate*). The first pass of the RASE system computes schedule lengths for each basic block for a range of register pressure limits. This information is then used by a global register allocator (the second phase) in deciding what register pressure limit to impose on each basic block. The final phase schedules each basic block and performs the register assignment. However, the implementation cost of the approach is high [64].

Norris and Pollock [52] introduce a global allocator based on coloring an interference graph. The constructed interference graph is more conservative as they assume that a variable which is alive at entry and exit points of basic block is alive throughout the basic block. This is not the case if this variable is redefined inside. This assumption produces false interference and hence the graph has more edges which slows down the coloring algorithm. They propose to add serial arcs into the DAG to reduce the interferences; for instance, arcs induced by resource constraints. When no legal coloring is found, the node with the greatest number of neighbors is selected to add false dependences. If there does not exist enough possibilities to eliminate interferences so that the node is colorable, no arcs are added and a minimal-cost node is selected for spilling. The main short coming of this method is its conservative assumptions. Extra interference edges result in over-estimating register requirement.

Pinter [55] combines information from a DAG and a register interference graph to create a *parallel interference graph*. First, the transitive closure of the given DAG is taken, and all edges become undirected. Then edges are added between operations that have resource constraints imposed by the architecture. Edges in the complement of this graph represent operations that could be run in parallel. This new interference graph takes into account all possible parallelism in the program. By using this graph, register allocation is carried out which is then followed by an instruction scheduling phase. An optimal coloring of this graph ensures that no false dependence can be introduced. While coloring the graph, if it is found that spill code has to be added, the scheduling edges in the graph are removed one at a time to avoid spilling, thus reducing instruction parallelism. Unfortunately, coloring algorithms are costly, especially in this method since the number of edges in the parallel interference graph may be very high. No experiments have been provided to support the technique.

Berson, Gupta, and Soffa [8] present a URSA (*unified resource allocator*) approach for VLIW processors. The approach deals with the instruction scheduling phase and the register allocation phase simultaneously. The method uses three phases: measure phase, reduce phase, and allocation phase. In the first phase the allocator measures register requirement at each point of a given

program. In the second phase, effort is made to reduce these requirements. In the third phase the allocation is done. URSA was extended to global scheduling with code motion in CFGs.

Motwani et al. [50] give an algorithm known as *alpha-beta combined heuristics*. They define a value called the *register rank* for each operation. This value is similar to the list scheduling priority function, except it is used to reduce the register pressure. The register rank and the traditional list scheduling priority function are added together (weighted by the coefficients) to create a new list scheduling priority value. The list scheduling algorithm is then performed in the usual way using these new priority values. The register allocation is then performed on the given block. Experiments on randomly generated DAGs show that this technique is better than a strictly late or prior register allocation.

Govindarajan et al. [28] propose a heuristic approach based on the concept of lineages and propose a solution to the *minimum register instruction scheduling problem*. I use this heuristic approach to compare against my model.

Definition 5.3 (Minimum register instruction scheduling [28]) *Given a labeled dependency DAG for a basic block, the* minimum register instruction scheduling problem *is to find a schedule with minimum register requirement.*

Definition 5.4 (Lineage [28]) *An instruction lineage* $L_v = [v_1, \ldots, v_n]$ *is a set of nodes such that there exist edges* $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ *in the DAG.*

I discuss this approach in more detail in the next section. A lineage is a collection of nodes in a DAG which is allocated to one register. The approach attempts to minimize the register pressure by first sequencing the lineages and then performing instruction scheduling.

Definition 5.5 (Anti-chain [64]) *Let* $G = (V, E)$ *be a labeled dependency DAG, where* V *is the set of nodes and* E *is the set of edges. A subset of the nodes* $A \subseteq V$ *is an* anti-chain *iff all nodes in* A *are parallel. An anti-chain is called a* maximal *anti-chain iff its size in terms of number of nodes is maximal.*

Touati [64] proposes an approach for performing register allocation and instruction scheduling in a single phase by generating a schedule that maximizes the number of values alive at the same time. His reasoning is that when more values are simultaneously alive the scheduler will find more opportunities to find instruction level parallelism. Thus, instead of forming long lineages like Govindarajan [28], he creates anti-chains. When an anti-chain results in register pressure that is more than the available physical registers, Touati uses heuristics to serialize the anti-chain and reduce the requirement.

Efforts have been made to solve the register allocation problem optimally. Recently, Fu and Wilken [24] and Appel and George [1] use integer programming (IP) to solve the register allocation

problem optimally. Fu and Wilken target a RISC architecture with 24 real registers and are able to solve blocks as large as 1000 instructions using the SPEC92 integer benchmark. Their work introduces new techniques which identify and reduce redundant constraints from the IP model while preserving an optimal solution. Appel and George [1] target a CISC architecture and their approach is able to solve regions as large as 1500 instructions. But, the work does not use any standard SPEC benchmarks. Their approach reduces the IP model complexity by decomposing the register allocation problem into two sub-problems: spill code placement and register assignment, and solves each subproblem using IP. Although the IP based allocator is faster, the decomposition results in an allocation which may not be optimal. Work by Bednarski and Kessler [5, 6], using dynamic programming and integer programming, is the only noticeable work in this area which is targeted towards DSP and VLIW processors. The work is optimal for basic blocks as large as 50 instructions but with the restriction of unit latencies. Generalizing to arbitrary latencies makes the problem harder and their approach finds optimal solution for smaller DAGs (up to 25 nodes). Chang, Chen, and King [11] present a model using integer programming for the basic block scheduling without spilling problem. However, the model is not efficient and robust. In their work they tested it on two examples and did not use any real benchmark data set.

My approach for basic block scheduling without spilling is good enough to solve 97.496% of all basic blocks in the SPEC2000 benchmark. The approach is able to solve basic blocks as large as 50 instructions for both idealized and realistic architectures within 10 minutes with available physical registers from 8 to 32. This compares favorably to the work by Bednarski and Kessler [5, 6] on optimal integrated code generation using integer programming. In my experiments I also perform a detailed analysis of Govindarajan et al.'s [28] work in comparison to the optimal scheduler for basic block scheduling without spilling.

5.3 Minimum register instruction sequencing

I selected Govindarajan et al.'s [28] work on the minimum register instruction sequencing problem to build upon and to compare against my approach. The reason for its selection is that if no spilling condition is added in Definition 5.3 then it becomes the basic block scheduling without spilling problem. Also, their experimental results show that it is near-optimal for most of the basic blocks in their test suite. Govindarajan et al.'s [28] method is based on the following two steps:

1. *Instruction lineage formation:* The concept of an instruction lineage is based on a chain of instructions which allows the sharing of a register among instructions. A lineage is a set of nodes in the DAG that can reuse the same destination register.

2. *Lineage interference graph:* The concept of a lineage interference graph captures the definite overlap relationship (see the next section for the definition) between the live ranges of

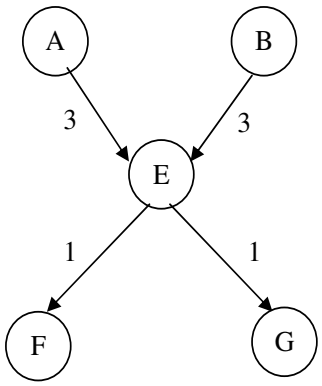


Figure 5.2: Invalid lineage formation.

lineages. It is used to facilitate sharing of registers across lineages that do not intersect with each other.

5.3.1 Properties of a lineage

A lineage has the following properties. Interested readers can consult [28] for more details and proofs on these properties.

1. A lineage has a unique starting instruction.
2. No two lineages can cross each other. Consider Figure 5.2. Lineage formation L_1 = [A, E, G] and L_2 = [B, E, F] are not possible because they cross each other at node E.
3. An instruction can be an end point for more than one lineage.
4. The live ranges of two lineages $L_u = [u_1, \ldots, u_m]$ and $L_v = [v_1, \ldots, v_n]$ overlap if u_1 reaches v_n and v_1 reaches u_m.
5. Two lineages $L_u = [u_1, \ldots, u_m]$ and $L_v = [v_1, \ldots, v_n]$ can be fused into a single lineage if u_1 reaches v_n and v_1 does not reach u_m. Fused lineages share the same register.
6. The fusion of lineages does not introduce any cycle.
7. Two lineages $L_u = [u_1, \ldots, u_m]$ and $L_v = [v_1, \ldots, v_n]$ cannot be fused if u_1 reaches v_n and v_1 reaches u_m.

Algorithm 5.1: Lineage Formation for a given DAG (from [28]).

```
input  : DAG G = (V, E)
output: DAG G' = (V, E') with each v_i ∈ V in a lineage and additional sequencing edges
        added to E
mark all nodes in the DAG as not in any lineage;
compute the height of every node in the DAG;
while ( there is a node not in any lineage ) do
    recompute height ← false;
    v_i ← highest node not in lineage;
    start a new lineage containing v_i;
    mark v_i as in a lineage;
    while ( v_i has a descendant ) do
        v_j ← lowest descendant of v_i;
        if ( v_i has multiple descendants ) then
            recompute height ← true;
            for each descendant v_k ≠ v_j of v_i do
                add sequencing edge from v_k to v_j;
        if ( v_j is already marked as in a lineage or v_j has different type from v_i ) /* the
        second test condition is added by me to take care of different types of lineages */
        then
            end lineage with v_j as the last node;
            break;
        mark v_j as in a lineage;
        v_i ← v_j;
    if ( recompute height = true ) then
        recompute the height of every node in the DAG;
```

5.3.2 Heuristic for lineage formation

Algorithm 5.1 gives pseudo-code for a heuristic method for lineage formation. Govindarajan et al. [28] (hereafter, just Govindarajan) target a single-issue processor and assume only one type of lineage. I consider both fixed point and floating point types of lineages for my work. The type of target register in an instruction defines the type of lineage. In Govindarajan's approach, if a node is already in a lineage then it is the end point for the lineage under consideration and the node is marked as a node with a lineage. For considering both types of lineages, the heuristic has to be modified. One possible modification is to consider the next node of same type with the lowest height. Let us call this modification M_1. The second possible modification is, if the type of node

is different from the type of lineage under consideration then it is the end point for the lineage under consideration. In this case, the end point is not marked as a node with a lineage. Let us call this modification M_2. I use M_2 for my work. Example 5.1 compares M_1 and M_2.

Definition 5.6 (Height of a node) *The* height *of a node in a DAG is the number of edges in the longest path from that node to the sink node of the DAG.*

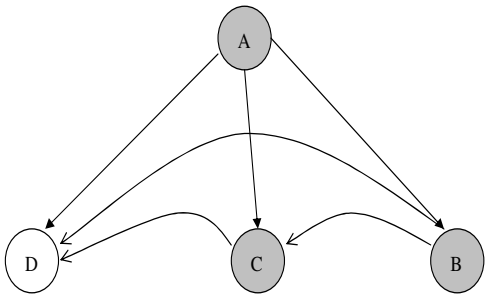


Figure 5.3: Comparison of M_1 and M_2.

Example 5.1 *Consider the DAG shown in Figure 5.3. Assume nodes A, B and C are of fixed point type and node D is of floating point type. Nodes D, C, B and A have height 1, 2, 3 and 4, respectively. Assuming unit latency with each edge, the only possible schedule for the given DAG is $S = A \rightarrow B \rightarrow C \rightarrow D$. According to M_1, A and C will share the same registers. Node D needs data from A, B and C for its data computation. If A and C are sharing the same registers, then data from A will be over written by the data from C. With this situation, schedule S will not preserve the correctness of the program represented by the DAG. With M_2, data from each node will initiate a new lineage. This will preserve the semantics of the program with schedule S.*

Lineage formation in Algorithm 5.1 depends upon potential killer nodes, unique killer nodes, the height of nodes and the height of siblings in the given DAG. I next define the relevant concepts[2].

Definition 5.7 (Potential killer node) *Let S be the set of successor nodes for a given node N. A node $x \in S$ is a* potential killer *of the value generated by N iff for every $y \in S - \{x\}$ there is no path from x to y.*

Definition 5.8 (Unique killer node) *Let S be the set of successor nodes for a given node N. A node $x \in S$ is the* unique killer *of the value generated by N iff for every $y \in S - \{x\}$ there is a path from y to x.*

[2]The terms potential killer and unique killer node also appear in the work by Touati [64]. Here they are being defined independently.

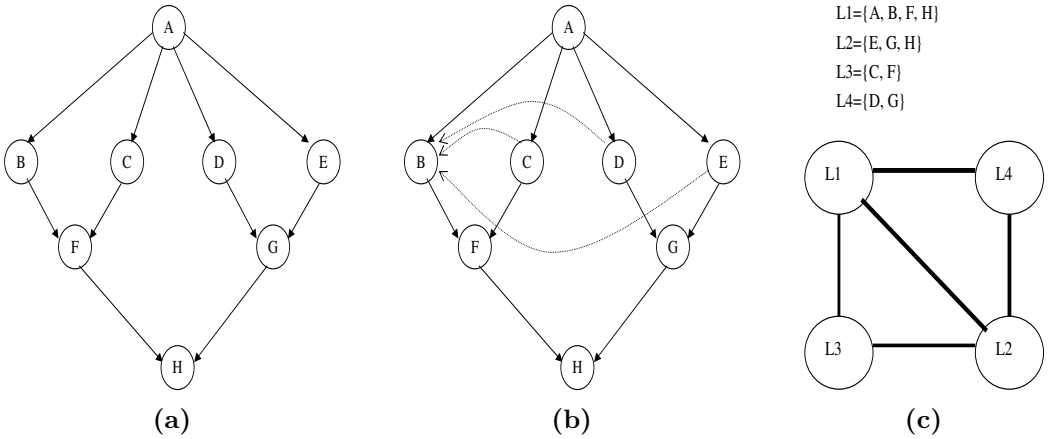


Figure 5.4: Lineage formation and lineage interference graph: (a) original DAG (from [28]); (b) transformed DAG; (c) lineage interference graph.

Example 5.2 *Consider the DAG in Figure 5.4(a). Nodes B, C, D and E are the potential killer nodes for node A. Node F is the unique killer node for both B and C. Similarly, node G is the unique killer node for both D and E. The height of node A is 3. Nodes B, C, D and E have height 2. Both nodes D and E have height 1 and the sink node H has height 0.*

5.3.3 Lineage interference graph

The lineage interference graph is similar to the interference graph used in register allocation. However, in the lineage interference graph, each node represents a lineage. There will be an edge between two lineages if they overlap each other. Two lineages that do not have an edge between them or do not overlap can share the same registers; i.e, they can be fused. The lineage interference graph can be colored using a heuristic graph coloring algorithm. The number of colors required to color a lineage interference graph is called the *heuristic register bound* (HRB). Example 5.3 illustrates Algorithm 5.1 and the formation of the lineage interference graph.

Example 5.3 *Consider the DAG in Figure 5.4(a). For simplicity assume all nodes are of the same type. Starting from node H, calculate the height of each node in the DAG. Algorithm 5.1 selects node A as the starting node for Lineage L_1. For the next node in L_1, there is more than one potential killer node with the same height. The algorithm selects the next node arbitrarily. Let node B be selected. As a sole killer of the value generated by A, the siblings of B have to see the value generated by A before it is killed by B. To ensure this, a scheduling constraint is added by inserting edges (broken edges in Figure 5.4(b)) from each sibling node C, D and E to B. Next F and H are added in L_1 ($L_1 = \{A, B, F, H\}$). The height of the nodes is recomputed. With the added edges, the heights of C, D and E become 4. The algorithm selects the highest node not in*

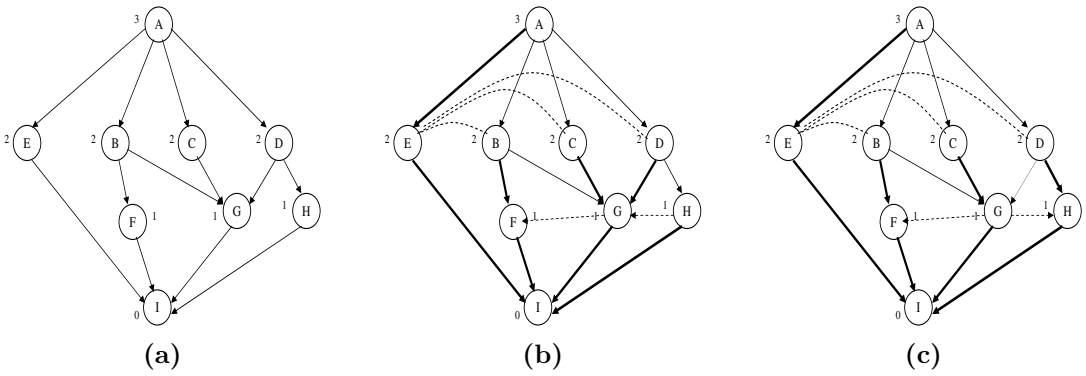


Figure 5.5: The heuristic method by Govindarajan et al. [28] is not optimal: (a) original DAG; (b) transformed DAG with five lineages; (c) transformed DAG with four lineages.

any lineage as the starting node for a new Lineage L_2. Let it select E. The algorithm then picks G and finally H for L_2 ($L_2 = \{E, G, H\}$). Similarly lineages $L_3 = \{C, F\}$ and $L_4 = \{D, G\}$ are created. Keeping in view the properties of lineages in Section 5.3.1, L_1, L_2 and L_3 interfere with each other and hence cannot share the same register. Similarly, L_1 and L_4, L_4 and L_2 interfere with each other. Figure 5.4(c) gives the lineage interference graph for the given DAG. Lineages L_3 and L_4 can be fused as there is no edge between them. To do so, an edge is introduced from the last node of L_4 to the first node of L_3. The HRB value for the given lineage interference graph is three.

Algorithm 5.1 does not guarantee the minimum register requirement for a given basic block. This can be shown using Example 5.4.

Example 5.4 *Consider Figure 5.5. The numbers beside each node are the heights. One possible lineage formation from Algorithm 5.1 consists of five lineages; i.e, $L_1 = \{A, E, I\}$, $L_2 = \{B, F, I\}$, $L_3 = \{C, G, I\}$, $L_4 = \{D, G\}$, and $L_5 = \{H, I\}$ as shown in Figure 5.5(b). However, lineage formation with four lineages is possible; i.e., $L_1 = \{A, E, I\}$, $L_2 = \{B, F, I\}$, $L_3 = \{C, G, I\}$, and $L_4 = \{D, H, I\}$ as shown in Figure 5.5(c).*

5.3.4 Instruction scheduling

Govindarajan adopt the traditional list scheduling approach to schedule the modified DAG (the DAG with added edges to take care of lineages). The only change to the list scheduling algorithm is the addition of a ready list of registers. Like the ready list of instructions, it is a list of available registers that can be assigned to lineages. The ready list of registers is initialized equal to the HRB value calculated using the lineage interference graph. A register is allocated to a lineage

when the starting instruction of the lineage is scheduled. The register is given back to the list when the last instruction of the lineage is scheduled. If during scheduling no legal schedule is found then the value of HRH is incremented and the scheduling is done again. The process is repeated until the scheduler finds a legal schedule for the modified DAG.

5.4 My solution for the problem

In this section, I discuss my approach for the problem. The approach uses the concept of lineage and the constraint programming (CP) approach for basic block scheduling given in Chapter 3. Through out this section, I assume the following:

Assumption 5.1 *Each basic block has a unique entrance and a unique exit point. If a given basic block does not have a unique entrance and exit point then I am inserting them in the DAG of the given basic block. The edges from or to the inserted nodes in the DAG are treated as dependency edges and are considered in the lineage formation.*

Govindarajan in his work also inserts unique entrance and exit points in the DAG of a given basic block if it does not have one. However, Govindarajan does not treat the edges from or to the inserted nodes as dependency edges and does not consider them in the lineage formation heuristic. There might be instructions inside a basic block that are data dependent on the variables defined outside of the block. Similarly, variables defined inside a basic block might be alive outside of the block. Considering the edges to the source and sink nodes as dependency edges takes care of this situation. Example 5.5 explains in more detail.

Example 5.5 . *Figure 5.6(a) gives a DAG for a basic block with no unique entrance and exit points. Unique entrance node R and unique exit node S are added to the DAG. According to Govindarajan's assumption, edges from R to A and B and from C and D to S are not dependency edges and will not be considered in the lineage formation heuristic. Figure 5.6(b) gives a lineage formation using Govindarajan's assumption. Lineage $L_1 = \{A, C\}$ and $L_2 = \{B, D\}$ can be fused as they do not interfere with each other. Hence, with Govindarajan's assumption the minimum register requirement for the give basic block is one. According to Assumption 5.1, edges from R to A and B and from C to D and S are dependency edges and are be considered in the lineage formation heuristic. Figure 5.6(c) gives one possible lineage formation with this assumption. There are two lineages; $L_1 = \{R, A, C, S\}$ and $L_2 = \{B, D, S\}$ and both cannot be fused as they interfere with each other. The minimum register requirement using Assumption 5.1 is two.*

Now, I present the characteristics of lineages that are used in my approach.

Lemma 5.1 *Any legal lineage formation for a DAG of a given basic block, with unique entrance and exit points, is a valid register assignment.*

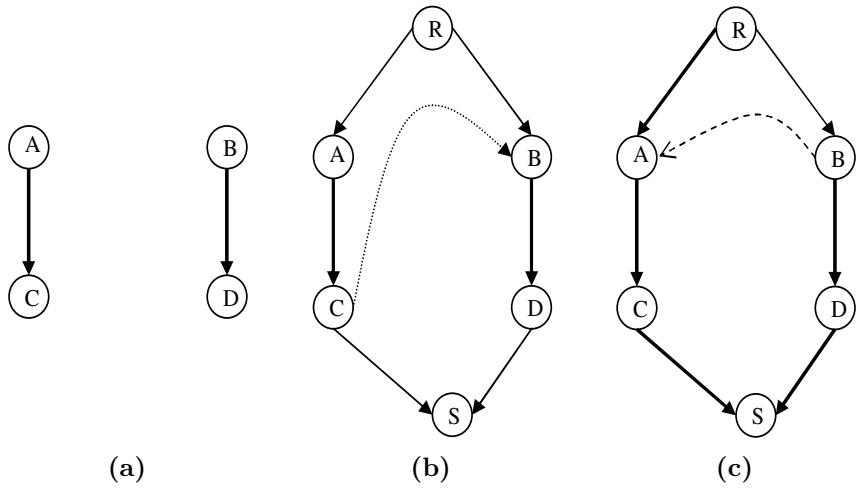


Figure 5.6: The difference in the assumption made by Govindarajan and the assumption made in this work. (a) original DAG; (b) lineage formation using Govindarajan's assumption; (c) lineage formation using Assumption 5.1.

Proof. The proof is by contradiction. Suppose there exists a lineage formation which does not give a valid register assignment. In the register allocation problem, a register assignment is not valid if two data variables that are alive simultaneously are assigned to the same register. In a lineage formation if two data variables are alive side by side in any instruction schedule then they belong to different lineages. Two different lineages that overlap cannot be assigned to the same register. Hence, this is not possible. □

Lemma 5.2 *The minimum register requirement for a given basic block, with unique entrance and exit points, is equal to the minimum number of lineages for the DAG.*

Proof. Let S be a set of all possible lineage formations for the DAG of a given basic block. Let N_i be the number of lineages in a lineage formation $T_i \in S$. Let N_{min} be the minimum number of lineages in S given by a lineage formation $T_{min} \in S$. Let R_{min} be the minimum register requirement for the given basic block.

The proof is by contradiction. There are two possibilities:

- Case 1 ($N_{min} < R_{min}$): This is not possible. According to Lemma 5.1, T_{min} is a valid register assignment. This means R_{min} is not the minimum register requirement of the DAG for a given basic block.
- Case 2 ($N_{min} > R_{min}$): Each lineage represents the flow of a data value in the DAG. Set S represents all possible flow patterns for all data values in the DAG. This means N_{min} is the minimum register requirement. Again, this is not true.

Hence, $N_{min} = R_{min}$. □

Lemma 5.3 *If each node in a DAG of a given basic block, with unique entrance and exit points, has a unique killer node then the DAG has a unique lineage formation.*

Proof. The proof is by contradiction. Suppose there exists a lineage formation other than the unique lineage formation which has a node in a lineage which is not the unique killer node of its parent. This is not possible, as this will create a cycle between the unique successor and the selected node. □

Heffernan and Wilken [31] present a set of graph transformations for dependency DAGs for basic blocks and show that optimally scheduling the transformed DAGs using branch-and-bound enumeration is faster and more robust. The DAG transformations reduce the search space while preserving optimality and hence are safe. I am reproducing the transformations given by Heffernan and Wilken here. For more detail see Section 2.3 or the work [31].

Theorem 5.1 (Heffernan and Wilken [31]) *Let A and B be isomorphic subgraphs with node sets $V(A) = \{a_1, \ldots, a_r\}$ and $V(B) = \{b_1, \ldots, b_r\}$. If,*

(i) a_i is neither a predecessor or a successor of b_i, $1 \leq i \leq r$,

(ii) for all $k \in pred(a_i)$ such that $k \notin V(A)$, $l(k, a_i) \leq cp(k, b_i)$, $1 \leq i \leq r$,

(iii) for all $k \in succ(b_i)$ such that $k \notin V(B)$, $l(b_i, k) \leq cp(a_i, k)$, $1 \leq i \leq r$,

(iv) for any edge (b_i, a_j), $l(b_i, a_j) \leq cp(a_i, b_j)$,

then adding the constraints $a_i \leq b_i$, $1 \leq i \leq r$ is safe.

Theorem 5.2 shows that the transformations also preserve the minimum register requirements. The transformations can be used to add extra edges among the siblings and hence reduce the number of potential killer nodes in a given DAG. The edges added by the transformations are sequential edges and are not considered in the lineage formations.

Theorem 5.2 *The transformations introduced by Heffernan and Wilken [31] when applied to a DAG G of a basic block do not change the minimum register pressure of G.*

Proof. Consider Figure 5.7. Node D and node E satisfy the conditions of Heffernan's transformation and node D is superior to node E. Thus, one can have an edge with zero latency from node D to node E. What I wish to prove is that this does not change the minimum register requirement of G. The proof is by contradiction. Suppose to the contrary that when a zero latency edge is introduced from node E to node D, the minimum register pressure with this transformation is less than the minimum register pressure due to the transformation when a zero latency edge is introduced from node D to node E. There are two possible cases for this to happen:

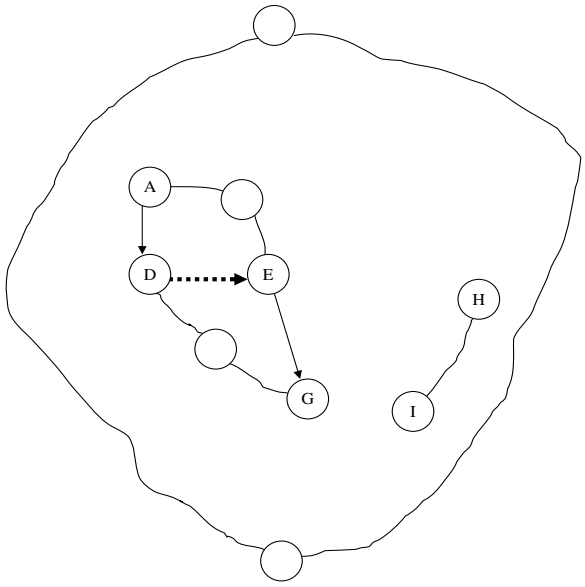


Figure 5.7: Heffernan and Wilken [31] transformation. Node D is superior to node E.

- **Case 1:** Let $IPred_D$ and $ISucc_E$ be the *immediate predecessors* and *immediate successors* of node D and node E, respectively. In Figure 5.7, A $\in IPred_D$ and G $\in ISucc_E$ form a region $R_{A,G}$. One possibility is that the register requirement for $R_{A,G}$, when there is an edge from D to E, is more than the register requirement when there is an edge from E to D. This is not possible. Node D and node E are independent of each other; i.e, no path exists between the two nodes. Thus, they belong to two different lineages. The question is whether these two lineages can be fused together when an edge from D to E or E to D is inserted. The lineage containing node D will contain a node $x \in IPred_D$ and a node $y \in ISucc_D$. Similarly, the lineage containing node E will contain a node $w \in IPred_E$ and a node $z \in ISucc_E$. According to the transformation introduced by Heffernan and Wilken [31], if D is superior to E then there is a path from any $x \in IPred_D$ to E and there is a path from D to any $z \in ISucc_E$. Thus, there is always a path from the starting node of the lineage containing D to the end node of the lineage containing E. The same is true for the lineage containing node E; i.e., there is always a path from the starting node of the lineage containing E to the end node of the lineage containing D. According to Property 7 in Section 5.3.1, the two lineages cannot be fused. Hence, the register requirement for $R_{A,G}$ remains constant. This is true for all regions $R_{i,j}$ such that $i \in IPred_D$, $j \in ISucc_E$.

- **Case 2:** The other possibility is when an edge from E to D is inserted, there is a lineage, say from node H to node I in G that can be fused with a lineage containing D or E. However, this fusion is not possible when an edge is introduced from D to E. As already mentioned

in Case 1, if D dominates E then there is a path from any $x \in IPred_D$ to E and there is a path from D to any $z \in ISucc_E$. Thus, the addition of an edge between D and E does not change the reachability of both nodes to any node in the given DAG. Let V_D be the set of vertices that can be reached from D and V_E be the set of vertices that can be reached from E before adding an edge between D and E. Then, V_D and V_E do not change after inserting an edge between D and E. Thus, the fusion is independent of the situation whether node E is scheduled before or after node D.

Hence, the transformation preserves the minimum register pressure. □

Govindarajan does not consider the optimal lineage fusion problem.

Definition 5.9 (Optimal lineage fusion) *Let S be a set of lineages in a DAG for a given basic block and let n be the number of lineages; i.e., $n = |S|$. Let r be the number of physical registers and suppose that $n > r$. An* optimal lineage fusion *is a fusion of at least $n - r$ pairs of lineages, if there is a possibility to do so, such that the minimal schedule length is preserved.*

Example 5.6 explains the concept of optimal lineage fusion.

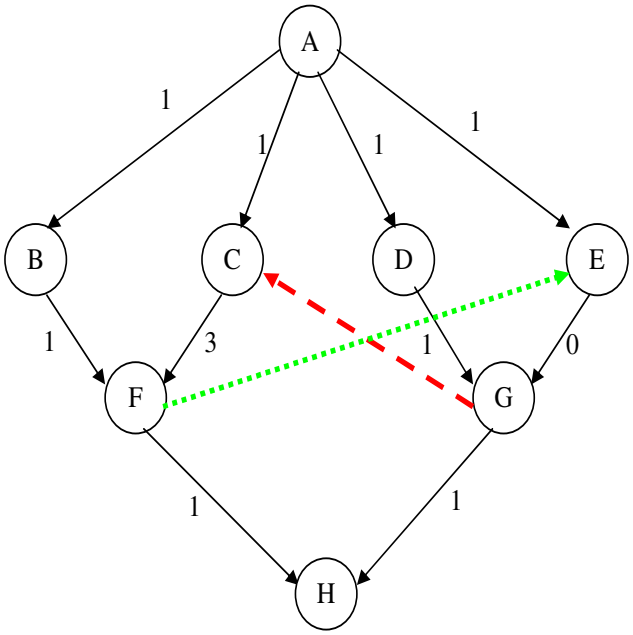


Figure 5.8: Optimal lineage fusion.

Example 5.6 *Consider Figure 5.8. Let Algorithm 5.1 give the four lineages: $L_1 = \{A, B, F, H\}$, $L_2 = \{C, F\}$, $L_3 = \{D, G, H\}$ and $L_4 = \{E, G\}$. Consider a fully pipelined single-issue processor*

with three physical registers. The minimal schedule length for the DAG is eight cycles. In order to reduce the register requirement, lineage fusion, if possible, must be done. Lineage L_2 and L_4 can be fused. However, fusing L_4 with L_2 by adding an edge with zero latency from node G to node C will increase the minimal schedule length to nine cycles as the best possible schedule that can be achieved under this fusion is (A, D, E, G, C, B, NOP, F, H). However, fusing L_2 with L_4 by adding an edge with zero latency from node F to node E will not increase the schedule length. The fusion of L_2 with L_4 is optimal as it preserves the minimal schedule length of eight cycles—one such schedule is given by (A, C, B, D, F, E, G, H)—and has a register requirement of three registers.

5.5 Searching for a solution

I reformulate the basic block scheduling without spilling problem into finding a lineage formation for the DAG of the given basic block that will give the minimum schedule length and number of lineages less than or equal to the number of available physical registers. Thus, finding a solution for the problem consists of two parts: (i) a DAG transformation for a given basic block that ensures the register requirement is less than or equal to the available physical registers, and (ii) a schedule of the transformed DAG from that gives the minimum schedule length. The constraints for part (i) are the following.

- Each node should have a unique killer node.
- All successors of a node N should be scheduled before the unique killer node of N; i.e., there should be a dependency edge with zero latency between the successor nodes and the unique killer node.
- All immediate predecessors of a node should belong to different lineages.
- The number of lineages should be less than or equal to the available physical registers.

Chang et al. [11] use the same constraints to limit the register requirement. In order to find a lineage formation with the number of lineages less than or equal to the available registers, one has to check every possible lineage formation by considering every possible DAG transformation; i.e., one has to permute every possible DAG transformation using each node using all of its potential killers. The search space is exponential and is a function of the number of potential killers in a given DAG. Reducing the number of potential killers reduces the search space. Using the transformations of Theorem 5.2, one can add extra edges among siblings and can reduce the number of potential killer for a given DAG. I applied these transformations to a given DAG before starting the search process for an optimal solution for the problem.

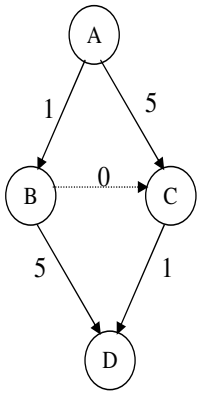


Figure 5.9: Reducing the number of potential killers.

Example 5.7 *Consider Figure 5.9. Node A has two successors, B and C. According to Theorem 5.2, adding an edge makes C the unique killer of A but will not increase the register pressure.*

To find an optimal solution, the optimal scheduler in Chapter 3 starts with a lower bound value on the cost function. At each cost function value, the scheduler attempts to find a solution. If there is a solution then it is returned; otherwise the value of the cost function is incremented. For basic block scheduling, there is one criteria to minimize via the cost function, i.e., the schedule length. For the basic block without spilling problem, there are two criteria to minimize via the cost function: the number of lineages and the schedule length. The upper bound on the number of lineages is the number of available physical registers and the lower bound is the minimum register requirement for a given basic block. Finding the minimum register requirement is a hard problem. I use the maximum number of immediate predecessors of a node in the DAG as a lower bound. If the lower bound on the register pressure of a given DAG is more than the available physical registers then there does not exist a solution. An upper bound on the schedule length is the length given by Govindarajan's modified list scheduling heuristic and the lower bound is the optimal schedule length under no register pressure constraint; i.e., the schedule length given by the optimal scheduler given in Chapter 3. One way to find a solution to this problem is to start with the lower bound on schedule length and see whether there exists a DAG transformation with number of lineages less than or equal to the available physical registers. If no lineage formation is found then increment the schedule length and repeat the process till a solution is found. This would be an expensive process. The other way is to check all transformations and pick the best solution, i.e., a transformation DAG which has the minimum schedule length with number of lineages less than or equal to the available physical registers. I followed this latter approach.

Algorithm 5.2: Generate(G, $Pkiller$, N, R)

```
input  : DAG G = (V, E), array Pkiller, node N, number of physical registers R
output: True if a solution is found; otherwise false
if (N is the sink node of G ) then
    /* DAG transformation is completed with each node having a unique killer node */;
    Algorithm 5.1 is called to give the number of lineages;
    if (number of lineages > R) then
        generate all possible fusion combination for a given set of lineages ;
        select the combinations which are possible and give number of lineages ≤ R;
        if (no combination exists) then
            return false;
        for each combination do
            apply the CP approach from Chapter 3;
            if (schedule length = lower bound) then
                return true;
            else
                update the record regarding the minimum schedule length;
    else
        apply the CP approach from Chapter 3;
        if (schedule length = lower bound) then
            return true;
        else
            update the record regarding the minimum schedule length;
else
    for each v_i ∈ Pkiller[N] do
        make v_i the unique killer of N by adding edges from v_j ∈ Pkiller[N] − v_i to v_i;
        update Pkiller because of the new edges added;
        if (Generate( G, Pkiller, N + 1, R ) then
            return true;
        else
            remove the edges from v_j ∈ Pkiller[N] − v_i to v_i;
            update Pkiller ;
return false;
```

If the schedule length from the heuristic is equal to the lower bound and number of lineages is less than or equal to the physical registers then the solution given by the heuristic is optimal. If not then I use Algorithm 5.2 to generate all possible lineage formations for the given DAG. A transformation of DAG is completed by recursively calling the function Generate on each node until the sink node is called. This give each node in the graph a unique killer. Algorithm 5.1 is applied to find the number of lineages in the transformation. According to Lemma 5.3, if each node has a unique killer node, then the number of lineages given by Algorithm 5.1 is the minimum for the given DAG. If number of lineages is more than the available physical registers then I use a brute force approach to find an optimal lineage fusion. I generate every possible lineage fusion combination. The constraint programming approach for basic block scheduling given in Chapter 3 is used to determine the minimum schedule length for the given DAG transformation with every possible lineage fusion that reduces the register requirement to less than or equal to the available physical registers. A data structure is maintained to keep the current minimum number of lineages during the search phase and the optimal schedule length associated with it. It is initialized to the number of lineages and schedule length determined by the heuristic. This record is updated if during the search phase there is a DAG transformation with number of lineages less than or equal to the recorded minimum number of lineages and the optimal schedule length for the transformation less than or equal to the recorded schedule length. If no DAG transformation with number of lineages less than or equal to the available physical registers is found after the search ends then according to Lemma 5.2 the minimum register requirement of given DAG is more than the available registers. The time limit to find a solution is 10 minutes. If a solution could not be found within this limit then the best solution found so far is returned. Example 5.8 explains the implementation of Algorithm 5.2.

Example 5.8 *Consider again Figure 5.8. Assume a single-issue processor with three physical registers. The optimal schedule length assuming no register constraints for the given DAG is eight cycles. This is the lower bound on the schedule length. The maximum number of immediate predecessors for a node in the DAG is two. This is a lower bound on register pressure. Let the heuristic give three lineages after fusion:* $L_1 = \{A, B, F, H\}$, $L_3 = \{D, G, H\}$ *and* L_4 *fused to* $L_2 = \{E, G, C, F\}$. *With this fusion a schedule* $(A, B, D, E, C, NOP, NOP, F, G, H)$ *with schedule length of 10 cycles is found using Govindarajan's modified list scheduling heuristic. This is an upper bound on the schedule length. The function Generate is first called with the root node A. The array Pkiller maintains the list of potential killers for each node. At this level, node A has four potential killers; i.e.,* $Pkiller[A] = \{B, C, D, E\}$. *Similarly,* $Pkiller[B] = Pkiller[C] = \{F\}$, $Pkiller[D] = Pkiller[E] = \{G\}$, $Pkiller[F] = Pkiller[G] = \{H\}$ *and* $Pkiller[H] = \{\}$. *Node B is selected as the unique killer node for node A and sequential edges are added from C, D and E to B. The information in Pkiller is updated to be* $Pkiller[A] = \{B\}$, $Pkiller[B] = Pkiller[C] = \{F\}$, $Pkiller[D] = Pkiller[E] = \{G\}$, $Pkiller[F] = Pkiller[G] = \{H\}$, $Pkiller[H] = \{\}$. *The function Generate is called with node B. As node B has a unique killer node, nothing happens. The same is true when Generate is called with C, D, E, F and G. With node H, which is the sink*

node of the DAG, the transformation is completed. Algorithm 5.1 is called and it gives four lineages: $L_1 = \{A, B, F, H\}$, $L_2 = \{C, F\}$, $L_3 = \{D, G, H\}$ *and* $L_4 = \{E, G\}$. *As there are three registers, lineage fusion is required. With the available DAG transformation only two fusion combinations are possible; i.e.,* L_2 *fused to* L_4 *and* L_4 *fused to* L_2. *Both combinations are tried and the transformed DAG is given to the CP scheduler for basic blocks to determine the optimal schedule length. The fusion of* L_2 *to* L_4 *gives a DAG transformation for which there exists a schedule* (A, C, B, D, F, E, G, H), *which has a schedule length of eight cycles. As this is equal to the lower bound, this is an optimal solution. In case the schedule length is not equal to the lower bound, the record keeping the minimum schedule length and number of lineages is updated. Backtracking is done and the sequential edges which were added would be removed. A new potential killer node would be selected and the process repeated until an optimal solution was found, if one exists.*

5.6 Experimental evaluation

In this section, I present experimental results gained from scheduling 343,295 basic blocks from the SPEC 2000 benchmark. The data contains basic blocks both before and after register allocation. Each basic block was scheduled on several different architectures using both the idealized and realistic architectural models. The same set of architectures that were used for basic block and superblock scheduling, were used again for basic block instruction scheduling without spilling problem. Bednarski and Kessler [5, 6] were the first to present integrated optimal code generation using integer and dynamic programming. Their model was targeted toward a VLIW processor. They used a theoretical VLIW target platform with issue width of three instructions per clock cycle. The architecture has two ALUs, two multiply-and-accumulate and two load/store units and eight general purpose registers. They were able to solve basic block as large as 50 instructions with unit latency and as large as 25 instructions with arbitrary latencies. This work cannot be used as a reference work to compare my model as it considers instruction selection in the model along with instruction scheduling and register allocation. This makes the model very complex. However, this work can be used as a reference to compare the robustness of my model. For my model I consider the PowerPC architecture, which is a realistic architecture, with issue width from zero to six instructions per clock cycle. The latency for various operations in my model varies from zero to 36 cycles. I tested my approach on a difficult test suite.

I used Govindarajan's work to compare against my work. They presented the result from two sets of experiments. In the first set of experiments, they compared the register requirement calculated by their approach with the minimum register requirement calculated by an integer programming (IP) approach. Because of the time complexity of an IP algorithm, instead of using whole SPEC benchmarks, they used a set of 675 DAGs extracted from the benchmark (all from the SPEC92 integer benchmark). The DAGs considered varied widely in size with a median of

10 nodes, a geometric mean of 12 nodes, and an arithmetic mean of 19 nodes per DAG. For 650 out of the 675 DAGs, Algorithm 5.1 found an instruction sequence that used the minimum number of registers. In the second set of experiments, they implemented their approach in the SGI MIPSpro compiler. They presented the performance results for a machine with 32 integer and 32 floating-point registers and for a machine with 32 integer and 16 floating-point registers. The fusion of lineages reduced the number of spill operations in the code by 63.1% and 55.9% respectively for 32 integer and 16 floating-point registers and without lineage fusion by 52.6% and 42.9% respectively.

5.6.1 Experiments for realistic architectural models

I did extensive experimentation for both idealized and realistic architectures. Here, I present only the results for the realistic architectures. Similar results were obtained for the idealized architectures.

Tables 5.1 to 5.3 give the performance of the approach for realistic architectures. The first column contains the applications in the SPEC 2000 benchmark and number of basic blocks with size from 2 to 50 instructions in each application. Column (a) gives the number of basic blocks where my approach was able to find register requirement less than the register requirement found by the heuristic and Column (b) gives the number of basic blocks where my approach is able to improve the schedule length over the heuristic with register requirement less than or equal to the register requirement. As the issue width increases, the gain in schedule length decreases. Depending upon the architecture and register requirement, the approach was able to improve at most 502 basic blocks in terms of register requirement which is less than 1% of the total blocks. The maximum gain was 2 registers per basic block over the heuristic. The approach was able to improve schedule length of at least 7% of basic blocks with register requirement less than or equal to the register requirement determined by the heuristic where the maximum gain was 3 cycles per basic block. The results show that Govindarajan's work is almost optimal for the basic block scheduling without spilling problem on our data set. I speculate that the reason my approach is not more successful is partly due to the extra non-data dependency edges introduced by the TOBEY compiler to control the register pressure during the instruction scheduling phase. These extra edges reduce the search space for the heuristic and restrict the improvements that can be made.

Tables 5.4 to 5.6 give the gain broken down by different ranges of basic block. The maximum gain is for basic blocks in the range of 16 to 30 instructions. Tables 5.7 to 5.9 give the percentage of all basic blocks in the SPEC 2000 benchmark suite which were solved to optimality, for various realistic architectural models and time limits for solving each basic block using 8 to 32 physical registers. Table 5.10 gives total time (hh:mm:ss) to schedule basic blocks in the SPEC 2000 benchmark suite, for various realistic architectural models within a 10 minute time limit.

Table 5.1: *Basic block scheduling without spilling for realistic architecture with 8 integer and 8 floating-point registers.* Number of basic blocks in the SPEC 2000 benchmark suite with 2 to 50 instructions where (a) the approach found a schedule with register requirement less than the heuristic, and (b) the approach found a schedule with better schedule length than the heuristic with the minimum register requirement within a 10-minute time limit, for various architectures.

		1r-Issue		ppc603e		ppc604		6r-issue	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	6,466	3	363	3	363	3	300	3	299
applu	1,173		31		28		31		15
apsi	4,416	6	271	6	271	6	228	6	143
art	830	2	57	2	52	2	50	2	48
bzip2	1,986	3	153	3	153	3	150	3	137
crafty	9,886	2	418	2	518	2	600	2	656
eon	8,937	6	636	6	768	6	747	6	551
equake	947		117		148		45		43
facerec	2,461	14	203	14	210	14	222	14	76
fma3d	19,259	13	376	13	376	13	501	13	307
galgel	10,939	4	721	4	721	4	770	4	460
gap	40,063	47	792	47	792	47	780	47	689
gcc	87,925	42	3,333	42	3,300	42	3,300	42	3,000
gzip	3,309		261		261		240		239
lucas	1,805	4	100	4	200	4	162	4	60
mcf	763		50		50		50		50
mesa	30,719	44	948	44	900	44	890	44	890
mgrid	383	1	14	1	14	1	14	1	11
parser	7,478	11	604	11	614	11	600	11	486
perlbmk	33,848	29	532	29	500	29	501	29	500
sixtrack	21,393	11	644	11	609	11	748	11	579
swim	677		11		11		8		5
twolf	14,957	21	253	21	653	21	700	21	700
vortex	24,621	3	379	3	470	3	400	6	300
vpr	6,904	7	304	7	400	7	400	7	3
wupwise	1,166		126		75		65		126
Totals	343,295	273	11,647	273	12,472	273	12,802	273	10,373

Table 5.2: *Basic block scheduling without spilling for realistic architecture with 16 integer and 16 floating-point registers.* Number of basic blocks in the SPEC 2000 benchmark suite with 2 to 50 instructions where (a) the approach found a schedule with register requirement less than the heuristic, and (b) the approach found a schedule with better schedule length than the heuristic with the minimum register requirement within a 10-minute time limit, for various architectures.

		1r-Issue		ppc603e		ppc604		6r-issue	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	6,466	11	500	11	441	11	366	11	307
applu	1,173	5	58	5	50	5	69	5	50
apsi	4,416	13	390	13	313	13	304	13	254
art	830	11	85	11	85	11	102	11	83
bzip2	1,986	5	212	5	187	5	150	5	191
crafty	9,886	7	1,080	7	929	7	900	7	910
eon	8,937	14	1,036	14	968	14	957	14	1,051
equake	947		127		148		56		61
facerec	2,461	16	253	16	210	16	188	16	119
fma3d	19,259	27	1,958	27	1,150	27	1,431	27	1,150
galgel	10,939	48	907	48	728	48	700	48	660
gap	40,063	47	2,430	47	1,179	47	3,232	47	2,732
gcc	87,925	48	5,852	48	5,078	48	7,067	48	5,100
gzip	3,309	7	308	7	282	7	389	7	304
lucas	1,805	4	125	4	212	4	169	4	69
mcf	763		60		50		71		50
mesa	30,719	48	2,582	48	2,239	48	2,000	48	1,998
mgrid	383	2	25	2	18	2	17	2	17
parser	7,478	13	751	13	687	13	650	13	557
perlbmk	33,848	21	3,050	21	2,765	21	1,201	21	2,000
sixtrack	21,393	27	1,604	27	1,220	27	1,101	27	1,100
swim	677	17	32	17	18	17	13	17	11
twolf	14,957	27	991	27	819	27	712	27	700
vortex	24,620	6	1,890	6	1,791	6	968	6	900
vpr	6,904	7	543	7	540	7	385	7	377
wupwise	1,166	6	150	6	100	6	100	6	126
Totals	343,295	407	26,999	407	23,304	407	23,298	407	20,877

Table 5.3: *Basic block scheduling without spilling for realistic architecture with 32 integer and 32 floating-point registers.* Number of basic blocks in the SPEC 2000 benchmark suite with 2 to 50 instructions where (a) the approach found a schedule with register requirement less than the heuristic, and (b) the approach found a schedule with better schedule length than the heuristic with the minimum register requirement within a 10-minute time limit, for various architectures.

		1r-Issue		ppc603e		ppc604		6r-issue	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	6,466	11	500	11	480	11	366	11	307
applu	1,173	5	58	5	58	5	69	5	50
apsi	4,416	13	390	13	323	13	304	13	254
art	830	11	85	11	85	11	102	11	83
bzip2	1,986	5	212	5	197	5	150	5	191
crafty	9,886	7	1,100	7	959	7	900	7	910
eon	8,937	14	1,056	14	978	14	957	14	951
equake	947		127		148		56		61
facerec	2,461	16	253	16	210	16	188	16	119
fma3d	19,259	32	1,998	32	1,950	32	1,431	32	1,150
galgel	10,939	58	907	58	928	58	700	58	660
gap	40,063	57	2,530	57	2,479	57	2,373	57	2,032
gcc	87,925	58	6,852	58	6,678	58	6,067	58	5,007
gzip	3,309	7	308	7	282	7	389	7	304
lucas	1,805	4	125	4	212	4	169	4	69
mcf	763		60		50		71		50
mesa	30,719	58	2,682	58	2,639	58	2,000	58	1,498
mgrid	383	2	25	2	18	2	17	2	17
parser	7,478	13	751	13	687	13	650	13	557
perlbmk	33,848	31	3,150	21	2,865	21	1,201	21	2,000
sixtrack	21,393	67	1,704	67	1,720	67	1,101	67	1,100
swim	677	17	32	17	18	17	13	17	11
twolf	14,957	27	991	27	819	27	712	27	700
vortex	24,620	6	1,990	6	1,991	6	968	6	900
vpr	6,904	7	543	7	540	7	385	7	377
wupwise	1,166	6	150	6	112	6	100	6	126
Totals	343,295	502	28,579	502	27,426	502	25,157	502	22,270

Table 5.4: *Using 8 integer and 8 floating-point registers.* Number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved register requirement (imp1.), and number of basic blocks with improved schedule length (imp2.), for ranges of basic block sizes and various realistic architectural models.

		1r-issue		ppc603e		ppc604		6r-issue	
range	#blocks	imp1.	imp2.	imp1.	imp2.	imp1.	imp2.	imp1.	imp2.
3–5	182,113	0	200	0	175	0	179	0	180
6–10	91,807	3	338	3	308	3	299	3	298
11–15	31,610	16	1,127	16	1,201	16	1,198	16	1,007
16–20	14,323	119	8,799	119	8,900	119	8,997	119	7,779
21–30	13,767	125	1,019	125	1,768	125	1,976	125	1,009
31–50	9,703	10	164	10	120	10	153	10	100
Total	343,295	273	11,647	273	12,472	273	12,802	273	10,373

Table 5.5: *Using 16 integer and 16 floating-point registers.* Number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved register requirement (imp1.), and number of basic blocks with improved schedule length (imp2.), for ranges of basic block sizes and various realistic architectural models.

		1r-issue		ppc603e		ppc604		6r-issue	
range	#blocks	imp1.	imp2.	imp1.	imp2.	imp1.	imp2.	imp1.	imp2.
3–5	182,113	0	200	0	175	0	179	0	180
6–10	91,807	3	338	3	308	3	299	3	298
11–15	31,610	16	1,211	16	1,201	16	1,098	16	1,007
16–20	14,323	170	12,700	170	11,001	170	11,000	170	9,789
21–30	13,767	180	11,991	180	9,999	180	10,000	180	9,099
31–50	9,703	38	643	38	680	38	622	38	504
Total	343,295	407	26,999	407	23,304	407	23,298	407	20,877

Table 5.6: *Using 32 integer and 32 floating-point registers.* Number of basic blocks in the SPEC 2000 benchmark suite where the optimal scheduler found an improved register requirement (imp1.), and number of basic blocks with improved schedule length (imp2.), for ranges of basic block sizes and various realistic architectural models.

		1r-issue		ppc603e		ppc604		6r-issue	
range	#blocks	imp1.	imp2.	imp1.	imp2.	imp1.	imp2.	imp1.	imp2.
3–5	182,113	0	200	0	175	0	179	0	180
6–10	91,807	3	338	3	308	3	299	3	298
11–15	31,610	16	1,127	16	1,201	16	1,198	16	1,007
16–20	14,323	211	12,567	211	11,990	211	11,780	211	10,990
21–30	13,767	220	13,561	220	12,990	220	10,990	220	9,111
31–50	9,703	52	786	52	762	52	711	52	684
Total	343,295	502	28,579	502	27,426	502	25,157	502	22,270

Table 5.7: Percentage of all basic blocks in the SPEC 2000 benchmark suite which were solved to optimality, for various realistic architectural models and time limits for solving each basic block using 8 registers.

	1 sec.	10 sec.	1 min.	10 min.
1r-issue	90.479	90.727	90.905	97.496
ppc603e	90.822	90.847	90.866	97.496
ppc604	90.831	90.856	90.979	97.496
6r-issue	90.321	90.643	90.742	97.496

Table 5.8: Percentage of all basic blocks in the SPEC 2000 benchmark suite which were solved to optimality, for various realistic architectural models and time limits for solving each basic block using 16 registers.

	1 sec.	10 sec.	1 min.	10 min.
1r-issue	90.179	90.701	90.905	97.496
ppc603e	90.722	90.747	90.766	97.496
ppc604	90.731	90.756	90.779	97.496
6r-issue	90.121	90.543	90.642	97.496

Table 5.9: Percentage of all basic blocks in the SPEC 2000 benchmark suite which were solved to optimality, for various realistic architectural models and time limits for solving each basic block using 32 registers.

	1 sec.	10 sec.	1 min.	10 min.
1r-issue	90.123	90.133	90.321	97.496
ppc603e	90.522	90.647	90.666	97.496
ppc604	90.731	90.756	90.779	97.496
6r-issue	90.321	90.343	90.342	97.496

Table 5.10: Total time (hh:mm:ss) to schedule basic blocks in the SPEC 2000 benchmark suite, for various realistic architectural models within a 10 minute time limit.

	with 8 registers	with 16 registers	with 32 registers
1r-issue	59:45:52	70:26:46	87:28:21
ppc603e	91:55:58	98:53:35	130:58:17
ppc604	100:04:20	120:05:36	140:37:33
6r-issue	110:58:42	130:34:06	180:44:03

5.7 Summary

I presented a constraint programming approach for solving the basic block scheduling without spilling problem for multiple-issue processors. I performed an extensive experimental evaluation using the SPEC2000 benchmark for both the realistic and idealized architectures. My model was able to solve basic blocks as large as 50 instructions within a 10 minute time limit per basic block. I was able to solve 97.496% of all basic blocks to optimality. This compares favorably to the recent work by Bednarski and Kessler [5, 6] on optimal integrated code generation using integer programming. Bednarski's work, targeted towards idealized architecture, was able to solve basic blocks as large as 50 instructions within 20 minutes with the latency ranging from unit cycle to 9 cycles. I also compared my optimal approach against the heuristic approach of Govindarajan. Depending upon the architecture and register requirement, my approach was able to show an improvement of 0.0%-7.8% over the heuristic.

Chapter 6

Conclusion and Future Work

In this chapter, I summarize my contributions. I describe the potential applications of the proposed optimal schedulers. At the end, I outline some possible extensions of my work.

6.1 Conclusion

This thesis gives an optimal instruction scheduler for multi-issue processors for both simplistic and realistic architectural models using constraint programming techniques. Both local and global instruction scheduling problems are considered. Global instruction scheduling is done using superblocks. Combined instruction scheduling and register allocation using basic block with out spilling is also considered. Previous work on optimal instruction scheduling is mainly targeted towards idealized architectures. Also, the previous work in this area are not fast and robust enough to be incorporated in commercial compilers. The main contribution of this work is a presentation of an optimal scheduler which is fast and robust for both idealized and realistic architectures.

For the local instruction scheduling problem, basic blocks were collected before and after register allocation phase using the IBM TOBEY compiler using the SPEC 2000 benchmark. I experimentally evaluated my optimal scheduler on the SPEC 2000 integer and floating point benchmarks. On this benchmark suite, the optimal scheduler was very robust and scaled to the largest basic blocks. Depending on the architectural model, between 99.991% to 99.999% of all basic blocks were solved to optimality. The scheduler was able to solve the largest basic blocks, including blocks with up to 2600 instructions. This compares favorably to the best previous approach due to Heffernan and Wilken [31]. I also compare the performance of a list scheduler for the basic block scheduling using the critical path heuristic. When scheduling for the idealized architectural model, the list scheduler solved 98.6%-99.9% of the basic blocks in the benchmark

suite optimally. For the realistic architectural model, the list scheduler produced optimal schedules for 94.2%-97.8% of the basic blocks.

For the global instruction scheduling problem, superblocks were collected before and after register allocation phase using the IBM TOBEY compiler. I experimentally evaluated my optimal scheduler on the SPEC 2000 integer and floating point benchmarks. On this benchmark suite, the optimal scheduler was very robust and scaled to the largest basic blocks. Depending on the architectural model, between 99.991% to 99.999% of all superblocks were solved to optimality. The schedulers were able to routinely solve the largest superblocks, including blocks with up to 2600 instructions. This compares favorably to the best previous approach by Shobaki [61]. For the global instruction scheduling problem, the critical path heuristic and the DHASY heuristic were used for comparison against the optimal scheduler. When scheduling for the idealized architectural model, the list scheduler solved 91.2%-97.5% of superblocks optimally. However, the list scheduler was only optimal for 54%-96% of superblocks when scheduling for a realistic architectural model. The schedules produced by the optimal schedule showed an improvement of 0%-3.8% on average over DHASY heuristic and an improvement of 0%-102% on average over the critical path heuristic. As expected, the heuristic DHASY yielded better schedules than the critical path heuristic.

For the basic block scheduling with out spilling problem, I tested my model on basic blocks from the SPEC 2000 benchmarks collected before and after register allocation phase using the IBM TOBEY compiler. The model was able to solve basic block as large as 50 instructions within a 10 minute time limit. I was able to solve 97.496% of all basic blocks to optimality. This compares favorably to the recent work by Bednarski and Kessler [5, 6] on optimal integrated code generation using integer programming. I also compared my optimal approach against the heuristic approach of Govindarajan. Depending upon the architecture and register requirement, my approach was able to show an improvement of 0.001%-7.8% over the heuristic.

The most significant conclusion of this thesis is that list scheduling is sufficiently close to optimality in practice for local instruction scheduling but not for global instruction scheduling. There is almost no need for optimal schedulers of any kind when scheduling basic blocks, as the cost of invoking an optimal scheduler will generally outweigh the cost of list scheduling, and there will only be benefits for a small number of blocks which may not even be significant to the execution time of a particular application. This is not the case for global instruction scheduling, and other superblock scheduling algorithms must be investigated in order to produce lower-cost schedules for superblocks being scheduled on realistic architectures. The other significant conclusion of this work is that constraint programming methodology can be a fruitful approach for solving NP-hard compiler optimization problems.

6.2 Applications

Although heuristic approaches have the advantage that they are fast, a scheduler which finds provably optimal schedules may still be useful in practice. The optimal approaches given in this thesis can be used in the following areas.

1. An optimal scheduler may be useful when longer compiling times are tolerable. With the appropriate setting of the time limit, the optimal model could be used at advanced levels of optimizations to schedule performance-critical regions of a program, or when compiling for software libraries, digital signal processing or embedded applications.
2. An optimal scheduler can be used to evaluate the performance of heuristic approaches. Such an evaluation can tell whether there is a room for improvement in a heuristic or not.
3. An optimal scheduler can be used to automatically create new list scheduling heuristics using techniques from supervised machine learning. In this approach the optimal scheduler provides the correct answer from which a heuristic approach can be learned.
4. An optimal scheduler can be used by computer architects to study the limits of instruction-level parallelism. An optimal instruction scheduler measures the maximum amount of ILP that a compiler can exploit with the scope of a given scheduling region.

6.3 Future Work

There are several interesting extensions of this work, including:

1. *Extending the model to optimally schedule non-linear regions:* In this work, I consider only acyclic regions. Software pipelining is a technique to extract instruction level parallelism within cyclic regions. A common example of a cyclic region is a nested loop. In software pipelining, iterations of a loop in a source program are continuously initiated at constant intervals without having to wait for preceding iterations to complete. That is, multiple iterations, in different stages of their computations, are in progress simultaneously. Software pipelining is NP-complete. Work can be done to develop a model to solve the software pipelining problem optimally.
2. *Extending the model to combined instruction and register allocation problem with spilling allowed:* In this work I presented a very basic model. The model is able to solve basic block as large as 50 instructions. Work can be done to improve this model. The model can be extended to solve the combined instruction and register allocation problem with spilling allowed using constraint programming techniques. The model can be extended for superblocks as well.

3. *Optimal instruction selection:* Instruction selection is a compiler optimization that transforms an intermediate representation of a program into the final compiled code, either in binary or assembly format. Instruction scheduling combined with instruction selection is an important problem for embedded processors where the emphasis is to produce compact code.

4. *Using traces and hyperblock for global instruction scheduling:* My model can be extended to solve the global instruction scheduling problem using traces and hyperblocks.

5. *Optimal power consumption problem:* Reducing energy consumption has become an important issue in designing hardware and software systems in recent years. Although low power hardware components are critical for reducing energy consumption, the switching activity, which is the main source of dynamic power dissipation in electronic systems, is largely determined by the software running on these systems. Instruction scheduling algorithms can take into account energy considerations [54]. The current models could be extended to find a schedule for a given scheduling region that requires minimum power consumption.

Bibliography

[1] A. W. Appel and L. George. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 243–253, ACM press, 2001.

[2] S. Arya. An Optimal Instruction Scheduling Model for a Class of Vector Processors. *IEEE Transactions on Computers*, C-34:(11), pp. 981–995, 1985.

[3] S. Banerjia, W. A. Havanki, and T. M. Conte. Treegion Scheduling for Highly Parallel Processors. In *Proceedings of the European Conference on Parallel Computing*, pp. 1074–1078, Passau, Germany, 1997.

[4] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer, 2001.

[5] A. Bednarski. Optimal Integrated Code Generation for Digital Signal Processors. PhD thesis, Linköping University - Institute of Technology, Linköping, Sweden, 2006.

[6] A. Bednarski and C. Kessler. Optimal Integrated VLIW Code Generation with Integer Linear Programming. In *Proceedings of the European Conference on Parallel Computing*, pp. 461–472, Dresden, Germany, 2006.

[7] D. Bernstein and I. Gertner. Scheduling Expressions on a Pipelined Processor with a Maximal Delay of One Cycle. *ACM Transactions on Programming Languages and Systems*, 11:(1), pp. 57–66, 1989.

[8] D. A. Berson, R. Gupta, and M. Soffa. URSA: A Unified Resource Allocator for Registers and Functional Units in VLIW Architectures. In *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism.*, pp. 243–254, Holland, 1993.

[9] R. J. Blainey. Instruction Scheduling in the TOBEY Compiler. *IBM J. Res. Develop.*, 38:(5), pp. 577–593, 1994.

[10] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, Santa Clara, California, 1991.

[11] C.-M. Chang, C.-M. Chen, and C.-T. King. Using Integer Programming for Instruction Scheduling and Register Allocation in Multi-issue Processors. *Computers and Mathematics with Applications*, 34:(9), pp. 1–14, 1997.

[12] J. M. Chase. On the Near-Optimality of List Scheduling Heuristics for Local and Global Instruction Scheduling. M.Sc. thesis, University of Waterloo, 2006.

[13] C. Chekuri, R. Johnson, R. Motwani, B. K. Natarajan, B. R. Rau, and M. Schlansker. Profile-driven Instruction Level Parallel Scheduling with Applications to Superblocks. In *Proceeding of the 29th Annual International Symposium on Microarchitecture*, pp. 58–67, Paris, France, 1996.

[14] G. Chen Effective Instruction Scheduling using Limited Register Pressure. PhD Thesis, Harvard University, 2001.

[15] H. Chou and C. Chung. An Optimal Instruction Scheduler for Superscalar Processors. *IEEE Transactions on Parallel and Distributed Systems*, 6:(3), pp. 303–313, 1995.

[16] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[17] J. M. Crawford, M. L. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking Predicates for Search Problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 148–159, San Francisco, California, 1996.

[18] R. Debruyne and C. Bessière. Domain Filtering Consistencies. *J. Artificial Intelligence Research*, 14, pp. 205–230, 2001.

[19] B. L. Deitrich and W. W. Hwu. Speculative Hedge: Regulating Compile-time Speculation Against Profile Variations. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 70–79, Paris, France, 1996.

[20] U. Dorndorf. *Project Scheduling with Time Windows*. Physica-Verlag, 2002.

[21] M. A. Ertl and A. Krall. Optimal Instruction Scheduling using Constraint Logic Programming. In *Proceedings of 3rd International Symposium on Programming Language Implementation and Logic Programming*, pp. 75–86, Passau, Germany, 1991.

[22] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30, pp. 478-490, 1981.

[23] J. A. Fisher. Global Code Generation for Instruction Level Parallelism: Trace Scheduling-2. Technical Report HPL-93-43, Hewlett-Packard Laboratories, 1993.

[24] C. Fu and K. Wilken A Faster Optimal Register Allocator. In *Proceedings of the International Symposium on Microarchitecture*, pp. 245–256, Istanbul, Turkey, 2002.

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[26] J. R. Goodman and W. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the* 2^{nd} *International Conference on Supercomputing*, pp. 442–452, St. Malo, France, 1988.

[27] R. Govindarajan. Instruction Scheduling. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook*, pp. 631–687, CRC Press, 2003.

[28] R. Govindarajan, H. Yang, J. N. Amaral, C. Zhang, and G. R. Gao. Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architectures. *IEEE Transactions on Computers*, 52:(1), pp. 4–20, 2003.

[29] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics*, 17:(2), pp. 263–269, 1969.

[30] S. Haga and R. Barua. EPIC Instruction Scheduling Based on Optimal Approaches. In *1st Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, Austin, Texas, 2001.

[31] M. Heffernan and K. Wilken. Data-dependency Graph Transformations for Instruction Scheduling. *Journal of Scheduling*, 8, pp. 427–451, 2005.

[32] M. Heffernan, K. Wilken, and G. Shobaki. Data-dependency Graph Transformations for Superblock Scheduling. In Proceedings of the *39th Annual International Symposium on Microarchitecture*, pp. 77–88, Orlando, Florida, 2006.

[33] J. Hennessy and T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5:(3), pp. 422–448, 1983.

[34] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[35] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *J. Supercomputing*, 7:(1), pp. 229-248, May 1993.

[36] S. Hoxey, F. Karim, B. Hay, and H. Warren. *The PowerPC Compiler Writer's Guide*. Warthman Associates, 1996.

[37] IBM. *PowerPC 604e RISC Microprocessor User's Manual*. 1998.

[38] Intel. *Intel Itanium Architecture Software Developer's Manual, Volume 2: System Architecture*. 2002.

[39] Intel. *Intel Itanium Architecture Software Developer's Manual, Volume 3: Instruction Set Reference*. 2002.

[40] Intel. *IA-32 Intel Architecture Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*. 2006.

[41] D. Kästner and S. Winkel. ILP-based Instruction Scheduling for IA-64. In *Proceedings of the SIGPLAN 2001 Workshop on Languages Compilers, and Tools for Embedded Systems*, pp. 145–154, Snowbird, Utah, 2001.

[42] C. W. Kessler. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, 24:(1), pp. 33–53, 1998.

[43] R. Leupers and P. Marwedel. Time-constrained Code Compaction for DSPs. *IEEE Trans. VLSI Systems*, 5:(1), pp. 112–122, 1997.

[44] J. Liu and F. Chow. A Near-optimal Instruction Scheduler for a Tightly Constrained, Variable Instruction Set Embedded Processor. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pp. 9–18, Grenoble, France, 2002.

[45] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A Fast and Simple Algorithm for Bounds Consistency of the Alldifferent Constraint. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pp. 245–250, Acapulco, Mexico, 2003.

[46] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In Proceedings of the *25th Annual International Symposium on Microarchitecture*, pp. 45–54, Portland, Oregon, 1992.

[47] A. M. Malik, J. McInnes, and P. van Beek. Optimal Basic Block Instruction Scheduling for Multiple-Issue Processors using Constraint Programming. Technical Report CS-2005-19, School of Computer Science, University of Waterloo, 2005.

[48] K. Marriott and P. J. Stuckey. *Programming with Constraints*. The MIT Press, 1998.

[49] W. M. Meleis and A. E. Eichenberger. Balance Scheduling: Weighting Branch Tradeoffs in Superblocks. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 272–283, 1999.

[50] R. Motwani, V. Krishna, V. Sarkar, and S. Reyen. Combining Register Allocation and Instruction Scheduling. Technical Note STAN/CSTN-95-22, Stanford University, Department of Computer Science, 1995.

[51] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[52] C. Norris and L. Pollock. Experiences with Cooperating Register Allocation and Instruction Scheduling. *International Journal of Parallel Programming*, 26:(3), pp. 241–283, 1998.

[53] K. Neumann, C. Schwindt, and J. Zimmermann. *Project Scheduling with Time Windows and Scarce Resources*. Springer, second edition, 2003.

[54] A. Parikh, S. Kim, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Instruction Scheduling for Low Power. *The Journal of VLSI Signal Processing*, 37:(1), pp. 129–149, 2004.

[55] S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *J. of Programming Languages*, 4:(1), pp. 21-38, 1996.

[56] C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S. B. Sadjad. An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pp. 600–614, Kinsale, Ireland, 2003.

[57] J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 209–215, Portland, Oregon, 1996.

[58] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*, Elsevier, 2006.

[59] T. Russell, A. M. Malik, M. Chase, and P. van Beek. Learning Basic Block Scheduling Heuristics from Optimal Data. In *Proceedings of the 15th CASCON*, pp. 242–253, Toronto, 2005.

[60] G. Shobaki and K. Wilken. Optimal Superblock Scheduling Using Enumeration. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 283–293, Portland, Oregon, 2004.

[61] G. Shobaki. *Optimal Global Instruction Scheduling Using Enumeration*. PhD thesis, University of California, Davis, 2006.

[62] M. Smotherman, S. Krishnamurthy, P. S. Aravind, and D. Hunnicutt. Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling. In *Proceedings of the 24th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 93–102, Albuquerque, New Mexico, 1991.

[63] B. M. Smith. Modelling. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, Chapter 11. Elsevier, 2006.

[64] S.-A.-A. Touati. Register Saturation in Instruction Level Parallelism. *International Journal of Parallel Programming*, 33:(4), pp. 393–449, 2005.

[65] P. van Beek and K. Wilken. Fast Optimal Instruction Scheduling for Single-issue Processors with Arbitrary Latencies. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pp. 625–639, Paphos, Cyprus, 2001.

[66] K. Wilken, J. Liu, and M. Heffernan. Optimal Instruction Scheduling using Integer Programming. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp. 121–133, Vancouver, 2000.

[67] S. Winkel. Optimal Global Scheduling for Itanium Processor Family. In *Proceedings of the* 2^{nd} *EPIC Compiler and Architecture Workshop*, pp. 59–70, Istanbul, Turkey, November 2002.