SEARCH-BASED OPTIMIZATION FOR COMPILER MACHINE-CODE GENERATION

by

ARAN CLAUSON

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

September 2013

DISSERTATION APPROVAL PAGE

Student: Aran Clauson

Title: Search-based Optimization for Compiler Machine-code Generation

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Dr. Christopher Wilson	Chair
Dr. Matthew L. Ginsberg	Advisor
Dr. Michal Young	Core Member
Dr. Bart Massey	Core Member
Dr. David A. Levin	Institutional Representative

and

Kimberly Andrews Espy	Vice President for Research & Innovation/ Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded September 2013

DISSERTATION ABSTRACT

Aran Clauson

Doctor of Philosophy

Department of Computer and Information Science

September 2013

Title: Search-based Optimization for Compiler Machine-code Generation

Compilation encompasses many steps. Parsing turns the input program into a more manageable syntax tree. Verification ensures that the program makes some semblance of sense. Finally, code generation transforms the internal abstract program representation into an executable program. Compilers strive to produce the best possible programs. Optimizations are applied at nearly every level of compilation.

Instruction Scheduling is one of the last compilation tasks. It is part of code generation. Instruction Scheduling replaces the internal graph representation of the program with an instruction sequence. The scheduler should produce some sequence that the hardware can execute quickly. Considering that Instruction Scheduling is an NP-Complete optimization problem, it is interesting that schedules are usually generated by a greedy, heuristic algorithm called List Scheduling.

Given search-based algorithms' successes in other NP-Complete optimization domains, we ask whether search-based algorithms can be applied to Instruction Scheduling to generate superior schedules without unacceptably increasing compilation time.

To answer this question, we formulate a problem description that captures practical scheduling constraints. We show that this problem is NP-Complete given modest requirements on the actual hardware. We adapt three different search algorithms to Instruction Scheduling in order to show that search is an effective Instruction Scheduling technique. The schedules generated by our algorithms are generally shorter than those generated by List Scheduling. Search-based scheduling does take more time, but the increases are acceptable for some compilation domains.

CURRICULUM VITAE

NAME OF AUTHOR: Aran Clauson

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:
University of Oregon, Eugene
Western Washington University, Bellingham

DEGREES AWARDED:
Doctor of Philosophy, Computer Science, 2013, University of Oregon
Master of Science in Computer Science, 2007, Western Washington University
Bachelor of Science in Computer Science, 1997, Western Washington University

AREAS OF SPECIAL INTEREST:
Search-based Optimization and Machine-code Generation

PROFESSIONAL EXPERIENCE:

Software Engineer, On Time Systems, 2008–Present

Lecturer, Western Washington University, 2007–2008

Graduate Teaching Fellow, Western Washington University, 2005–2007

Software Engineer, Tieto Enator, 2004–2005

Software Engineer, Amazon.com, 2004

Software Engineer, Majiq, 2001–2004

PUBLICATIONS:

Apple, Jim, Chang, Paul, Clauson, Aran, Dixon, Heidi E., Fakhoury, Hiba, Ginsberg, Matthew L., Keenan, Erin, Leighton, Alex, Scavezze, Kevin and Smith, Bryan. "Green Driver: AI in a Microcosm." Paper presented at the meeting of the AAAI, 2011.

ACKNOWLEDGEMENTS

My educational adventure is something like a line of falling dominoes. Each accomplishment is leading directly to a new challenge. Along the way there are numerous people who have kept the tiles falling in the right direction. I would like to take this opportunity to thank a few of those people.

To begin, I would like to thank Matthew Ginsberg and David Etherington for supporting me throughout this work. They have listened, commented, directed, and they have given me endless examples of how to conduct independent research. Further, Matt and David created the Computational Intelligence Research Laboratory and On Time Systems (OTS). This creative, challenging, and dynamic environment is full of talented and supportive people with similar interests. The people at OTS have provided an endless supply of support, discussion, and criticism. One person at OTS deserves special attention: Heidi Dixon. Heidi has been a good friend, trenchant critic, and stellar example of a computer scientist.

On a more personal note, I would like to thank my wife, Stacy Clauson. Without her support and confidence this work would never have been started, let alone finished. Of course I must acknowledge my parents, Richard Clauson and Gretchen Luosey without whom I would not be possible. Finally, I would like to thank Kathy Gunnerson who I believe knocked over the first of my dominoes.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. BACKGROUND 7

2.1. Code Generation 7
2.2. Scheduling Algorithms 11
2.3. Search-Based Optimization 16
2.4. Summary 20

III. THE SCHEDULING PROBLEM 21

3.1. Basic Instruction Scheduling 23
3.2. Special Purpose Registers 25
3.3. Limited Registers 29
3.4. Problem Complexity 33
3.5. Summary 39

IV. SEARCH-BASED OPTIMIZATION 41

4.1. List Scheduling 42
4.2. Limited Discrepancy Search 49

Chapter Page

4.3. Squeaky Wheel Optimization 53

4.4. Iterative Flattening 57

4.5. Summary 61

V. EXPERIMENTAL RESULTS 62

5.1. Measuring Performance 63

5.2. Success Rates 65

5.3. Schedule Length 69

5.4. Scheduling Time 74

5.5. Register Pressure 79

5.6. Multi-Objective Comparison 81

5.7. Simulated Execution 85

5.8. Just-In-Time Compilation 87

5.9. Summary 89

VI. RELATED AND FUTURE WORK 92

6.1. Future Work 96

VII. CONCLUSION 98

APPENDIX: BENCHMARK STATISTICS 100

Chapter Page

REFERENCES CITED 104

LIST OF FIGURES

Figure Page

1.1. Unoptimized Dependency Graph for the Quadratic Formula. 4

1.2. Optimized Dependency Graph for the Quadratic Formula. 5

2.1. Quadratic Formula 9

2.2. QUADRATIC's Control Flow Graph 10

2.3. Dependency Graph for Quadratic's Basic Block BB2. 11

2.4. QUADRATIC's Control Flow Graph 13

3.1. The C code fragment in (a) could be translated to the assembly fragment in (b). Note that the body of the if statement is elided in both fragments. 27

3.2. Incorrect schedule for Figure 3.1 and the actual corresponding source fragment. 27

3.3. Reduction outline showing Instruction Scheduling NP-complete. 34

4.1. Naive List Scheduling Algorithm 43

4.2. Special Purpose Register Aware List Scheduling Algorithm 47

4.3. Scheduling Problem with Special Purposes Register conflict. 47

4.4. List Scheduling Algorithm with Backtracking 48

4.5. Limited Discrepancy Search search tree. 50

4.6. Limited Discrepancy Search 51

4.7. Squeaky Wheel Optimization's view of solution vs priority space. . . . 54

4.8. Squeaky-Wheel 56

4.9. Iterative Flattening 59

4.10. Example Minimum Conflict Sets 60

5.1. List Scheduling success rate by timeout limits 66

Figure Page

5.2. LDS scheduling success rate by timeout limit 68

5.3. SW scheduling success rate by timeout limit 69

5.4. IFlat scheduling success rate by timeout limit 70

5.5. Schedule length generated by List Scheduling (x-axis) vs LDS (y-axis) for zero, one, and two discrepancies. 71

5.6. Schedule length generated by List Scheduling (x-axis) vs SW (y-axis) for one, two, and five iterations. 73

5.7. Schedule length generated by List Scheduling (x-axis) vs IFlat (y-axis) for one, five, and ten iterations. 73

5.8. Scheduling time use by List Scheduling (x-axis) vs LDS (y-axis) for zero, one, and two discrepancies. 76

5.9. Scheduling time use by List Scheduling (x-axis) vs SW (y-axis) for one, two, and five iterations. 77

5.10. Schedule time use by List Scheduling (x-axis) vs IFlat (y-axis) for one, five, and ten iterations. 77

5.11. Register pressure comparison of LDS-2, SW-5, and IFlat-10 against List Scheduling 81

5.12. Schedulers plotted by scheduling time vs schedule length 83

5.13. Register pressure (x-axis) vs schedule length (y-axis) 85

5.14. Two-Scheduler Scheduling Time vs Proportional Speedup 90

A.1. Dependency Graph Order distribution. 101

A.2. Dependency Graph Size distribution. 102

A.3. Dependency Graph Size vs Order scatter plot. 103

A.4. Execution Frequencies 103

LIST OF TABLES

Table Page

3.1. The five possible partial schedules for the 1RCS problem converted to IS2 with $p, q, r \in T$, $p, q \notin R$, and $r \in R$. Arrows indicate the define-use relationship between operations by special purpose register. 39

4.1. Summary of scheduling algorithms and their complexity. Note: $n = |G| = |V| + |\prec|$, D is the number of discrepancies, and $iters$ is the fixed number of iterations. Further, these results assume no backtracking by List Scheduling. 61

5.1. Scheduler Performances Proportional to List Scheduling. 86

5.2. Scheduler Performances Proportional to List Scheduling. 86

5.3. Simulated execution time relative to List Scheduling 87

CHAPTER I

INTRODUCTION

My first compiler was a pirated copy of Turbo C.[1] It was given to me, along with a copy of Kernighan and Ritchie's *The C Programming Language* by my boss at a local coffee shop. It was the 80s and I was still in high school. It was inspirational. Suddenly I had access to the full computational power of my 7.16 MHz Tandy computer. It was magic!

Oh, how things have changed. Today, compilers are omnipresent. Just about anyone who uses a computer uses a compiler. Software engineers use them to build their applications, computer scientists use them to probe at the edges of computability, and even causal users unknowingly invoke compilers by simply browsing websites, watching videos, or driving cars.

Due to their ubiquity, compilers have held the attention of researchers and software engineers since their conception. At the highest level, compilers are translators that read in a program in one format then write it out in another, typically converting the program from one humans can read to one machines can execute. This translation is not one-to-one; along the way, the program is typically modified to improve the new form of the program. Improving the target program has always been a compilation goal.

The first optimizing compiler was IBM's FORTRAN compiler released in 1957. At that time, most programmers were working with assembly. For FORTRAN to be accepted as an alternative, the generated program had to be efficient. Backus [4] described the situation as follows:

[1] The statue of limitations has long since expired.

> It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger.

The importance of compiler optimization has only increased since that time.

Translating a program from one form to another involves several steps. At the highest level, these steps are parsing, verification, and code generation. Parsing converts the input or source code representation of the program to a more manageable syntax tree. An abstract syntax tree (AST) contains all of the same control structures as the original program. Functional decomposition, if-statements, and arithmetic logic are all encoded in the syntax tree. The next phase of compilation is verification. This pass ensures that the syntax tree contains a meaningful program. For example, the compiler must verify the constraints imposed by the type system. Passing an integer to a function that takes a string may be syntactically correct but is invalid. Strict separation of these three steps is unnecessary. Often some verification is performed during parsing. However, the order is important as parsing generates the syntax tree on which verification operates and code generation produces the target program.

The source program is built up of high level, abstract components. Language constructs like if-statements, loops and functions rarely have hardware level equivalents. To realize the language-level semantics, the compiler must remove all of these abstractions, replacing them with the targeted machine's instructions. For example, function invocation is replaced by a number of instructions that move arguments into the correct locations followed by some type of jump instruction. The very structure of the program must change. High level languages provide a recursive

view of control structures. If-statements are nested within loops that are nested within subroutines. In the Algol family of languages, even subroutines nest. All of these abstractions must be removed.

The recursive control structures are replaced with a two-level structure. The outer structure, called a Control Flow Graph (CFG), captures the possible execution paths through the program. Each vertex of the CFG contains a portion of the program code. At one extreme, each CFG vertex could contain a single statement. More often each vertex is a larger block of code. In either case, the block is almost always represented as a directed acyclic graph called a Dependency Graph (DG).

The DG realizes the program block's semantics. The vertices correspond to primitive hardware instructions. The edges between vertices capture the flow of data from one instruction to others. Without optimization, the DG structure is nearly identical to the syntax tree. Optimization turns the tree structure into an arbitrary acyclic graph. For example, the following program fragment implements the quadratic formula:

```
double x1 = (-B + sqrt (d)) / (2 * A);
double x2 = (-B - sqrt (d)) / (2 * A);
```

The unoptimized DG for this fragment is shown in Figure 1.1. Except for literal values and variables, shown as boxes, the DG is two trees; one for each statement. The optimized DG, shown in Figure 1.2, lost the trees and is an acyclic graph. The DG structure is one of the last abstractions removed by the compiler.

Most physical hardware uses a primitive concept of execution. A program counter indicates which instruction to execute. The hardware executes that instruction and, barring a jump operation it moves to the next instruction. That is, the hardware executes a sequence of instructions. It does not execute a graph.

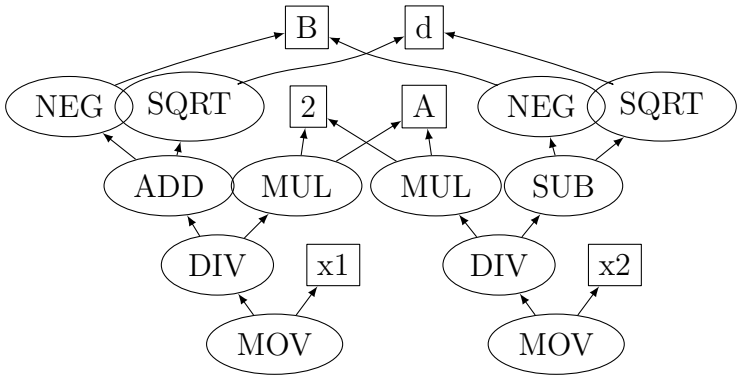


FIGURE 1.1. Unoptimized Dependency Graph for the Quadratic Formula.

The DG must be serialized. This serialization process is called Instruction Scheduling; it is the focus of this thesis. Typically, the algorithm used is a heuristic adaptation of the topological sorting algorithm called List Scheduling [19]. This algorithm is a one-pass algorithm that builds a single sequence quickly. It is simple to implement, executes quickly, and generates reasonably good sequences. Good in this sense means that the hardware can execute the sequence quickly.

Back in the day of FORTRAN I, processors executed each instruction to completion [38]. For these systems, all instruction sequences from the same DG were equally good. In 1961 IBM released the 7030. This STRETCH-based system introduced the Instruction Pipeline. Now executing different sequences from the same DG could take different amounts of time.

When viewed as a decision problem, Instruction Scheduling is an NP-Complete problem [1]. That is, instruction scheduling is as difficult as the Discrete Knapsack problem, the Traveling Salesman problem, and the Job Shop problem [17, 33]. In each case there is no known polynomial time algorithm.

The NP complexity class refers exclusively to yes/no decision problems. What we are really interested in is the optimization problem: generate a sequence with the shortest execution time. While List Scheduling is pretty good at creating schedules,

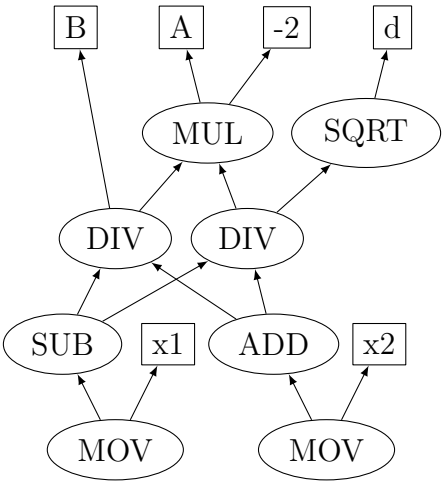


FIGURE 1.2. Optimized Dependency Graph for the Quadratic Formula.

it does not find optimal schedules. However, search-based optimization algorithms are frequently successful at generating optimum or near optimum solutions to other NP-Complete optimization problems.[2]

It is a little surprising, then, that most compilers use List Scheduling to sequence DGs. We are left to wonder why search-based scheduling is so rarely used in compilers. The central question address in this thesis is: Can search-based instruction scheduling produce better schedules than traditional schedulers without unacceptably increasing compile time?

To address this question, we formally state the scheduling problem as a specialization of resource constrained scheduling with precedences (Chapter III). The main contribution in the problem statement is identifying the way that CPU resources behave differently from traditionally studied Job Shop resources. Further, we show that instruction scheduling is NP-Complete even when constrained to a bounded number of conflicting resources.

[2] "NP-Complete optimization" is technically incorrect. We are interested in the generated schedule, not a yes/no answer. We will see that Instruction Scheduling is in the class $\mathsf{FP}^{\mathsf{NP}[\log n]}$ or in FP^{NP} depending on the problem formulation. These complexity classes deal with functions, FP, rather than decisions and they allow a limited number of calls to an NP Oracle.

We adapt three search-based optimization algorithms to our formulation of instruction scheduling (Chapter IV). Each algorithm depends differently on heuristic functions. The algorithm presentation includes an analysis of run-time complexity, the attribute contributing to compile time. We also discuss the extent to which each algorithm explores the range of possible solutions.

Using the SPEC CPU2006 benchmark suite, we compare the three search-based schedulers against the List Scheduling algorithm (Chapter V). We are primarily interested in the effect these techniques have on schedule quality and on scheduling run-time. Secondarily, we are interested in register allocation. We do not consider register allocation directly. Instead, we compare the number of registers needed to fully allocate each generated schedule.

In Chapter VI we discuss related scheduling research and possibilities for future work based on this thesis. We conclude with a summary remarks and directions for further research (Chapter VII). Before we can delve into search-based scheduling, we begin with background information necessary to begin discussing the details of instruction scheduling, in Chapter II.

CHAPTER II

BACKGROUND

Although a formal Instruction Scheduling problem statement is delayed until Chapter III, Section 2.1 presents a more extensive view of code generation than the introduction. Following this, Section 2.2 describes several existing approaches to Instruction Scheduling. Section 2.3 provides a brief overview of search-based optimization algorithms and the ways search differs from scheduling.

2.1. Code Generation

At the highest level, code generation comprises three tasks: Instruction Selection, Instruction Scheduling, and Register Allocation. Instruction selection chooses the appropriate hardware instructions that realize the program's desired behavior. Instruction Scheduling converts the compiler's graph representation of the program into an executable sequence. However, this sequence uses an unbounded number of so called virtual registers to store data. Register allocation assigns physical registers to the virtual registers used in the sequence. In the presence of typical optimizations like common subexpression elimination, each of these tasks is NP-Complete [1, 8, 10, 34].[1] Usually these three tasks are performed separately and they are usually performed in the described order [11].

This thesis is concerned with Instruction Scheduling. Instruction Scheduling involves two subtasks: block selection and scheduling. Block selection identifies the

[1]In addition to these references, we will prove our formulation of Instruction Scheduling NP-Complete with modest bounds on the parameters (e.g., number of functional units).

parts of the program to be scheduled together. Basic blocks are typically the smallest unit of scheduling. Formally,

> A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.[2]

Roughly speaking a basic block corresponds to the body of an if-statement provided there is no internal control structure. When the predicate expression is true the body of the if-statement is executed together starting at the top and exiting at the bottom.

The code associated with a basic block is represented by the compiler as a directed acyclic Dependency Graph (DG). Each vertex corresponds to a hardware instruction. Edges encode the dependencies between these instructions. Scheduling orders the instructions honoring these dependencies. That is, the DG represents a partially ordered set of instructions. Scheduling constructs a totally ordered extension of this partial order.

For example, consider the function QUADRATIC shown in Figure 2.1. This C++ program returns the two real quadratic roots if they exist and two NANs if they do not.[2] Parsing and verification succeed and the compiler generates the CFG shown in Figure 2.2. The vertices of this CFG contain fragments of the original C++ source code. This more clearly shows what portions of the program belong to each basic block. This program has no looping structures, so the CFG is acyclic. Control will flow from basic block BB1 to either BB2 or BB3. Finally control joins together at BB4 and the function returns its results.

[2]The symbol NAN is a IEEE 754 floating point constant that means that the value is not a number.

```
pair<double,double> Quadratic (double A, double B, double C)
{
  double x1,x2;
  double d = (B * B) - 4 * A * C;
  if (d >= 0) {
    x1 = (-B + sqrt (d)) / (2 * A);
    x2 = (-B - sqrt (d)) / (2 * A);
  }
  else {
    x1 = NAN;
    x2 = NAN;
  }
  return make_pair(x1,x2);
}
```

FIGURE 2.1. Quadratic Formula

Our interest lies in scheduling. If we select BB2 we find the DG shown in Figure 2.3 (also shown in Figure 1.2). We included data elements, shown in boxes, to help illustrate the DG's relationship to the original expressions. The compiler has optimized the two assignment statements in this block. Without optimization, the two statements in the block would be two largely independent trees that correspond almost directly to the parse tree. The optimizer has reduced the instruction count from about twelve to eight. Optimally scheduling trees is relatively easy[17]. Unfortunately, scheduling arbitrary graphs is NP-Complete [2].

Scheduling this DG involves choosing a valid topological sort of the graph. One possible ordering is the following:

```
v1 = MUL A, -2
v2 = SQRT d
v3 = DIV B, v1
v4 = DIV v2, v1
v5 = SUB v4, v3
v6 = ADD v4, v3
     MOV v5, x1
```

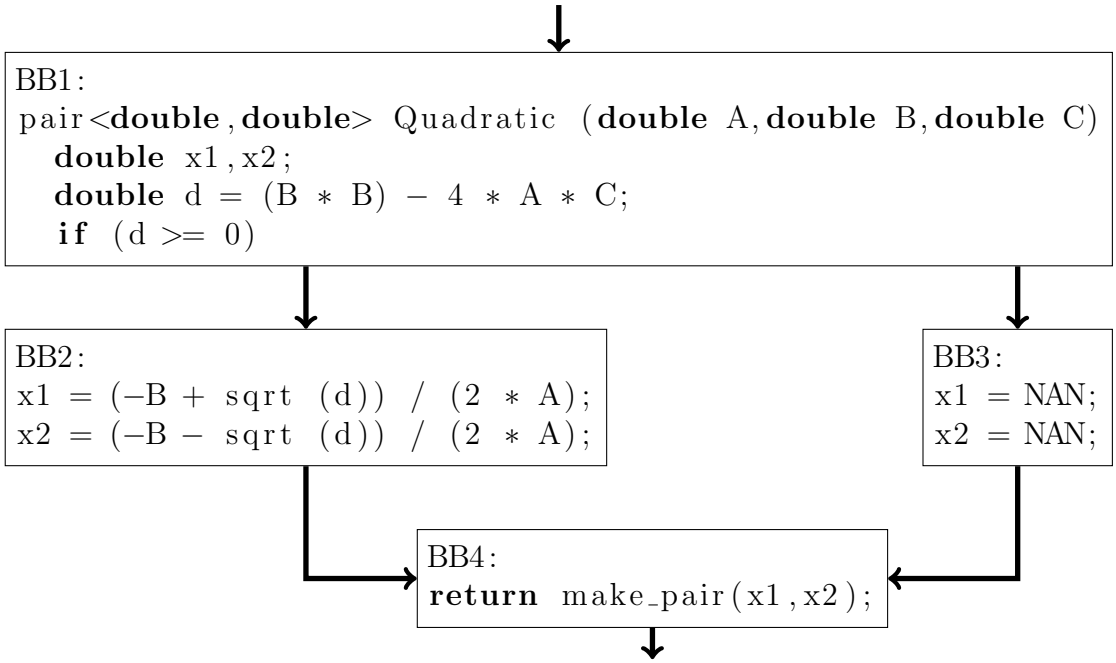


FIGURE 2.2. QUADRATIC's Control Flow Graph

```
MOV v6, x2
```

Variables *A*, *d*, and *v1* through *v6* are all virtual registers. In a later register allocation phase, these variables will be replaced with physical hardware registers. This sequence is valid but not optimal. Using Intel Core2 instruction latencies we find that floating point multiplication takes three clock cycles and square root takes 14. This means that the second divide cannot start until cycle 16.[3] A better schedule would start the square root operation first.

With only eight instructions and nine (meaningful) edges, optimally scheduling BB2 is easy. We could enumerate all possible sequences and choose the one with the shortest execution times. This is not generally possible. Other graphs will have many more valid sequences, too many to enumerate completely. Even counting the number

[3]The square root operation starts in clock cycle two and finishes in cycle 16.

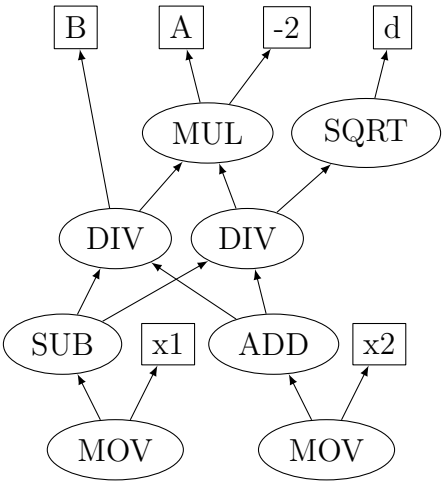


FIGURE 2.3. Dependency Graph for Quadratic's Basic Block BB2.

of valid sequences is hard; it is an #P-Complete problem[7].[4] Clearly we need a more elegant scheduling algorithm.

2.2. Scheduling Algorithms

Instruction Scheduling algorithms try to find a good sequence—that is, short execution times—without spending too much time scheduling. Most algorithms combine block selection with scheduling. The most basic instruction scheduler is a basic block List Scheduler.

A basic block scheduler places all of its focus on scheduling and none on block selection. The List Scheduling algorithm is the most commonly used scheduler. A full description is given in Chapter IV. At the highest level, List Scheduling is essentially Kahn's Topological sorting algorithm [26]. List Scheduling works by finding all of the instructions at the top of the DG. These instructions do not depend on the result of other instructions. In our example, initially the set of "ready" instructions are MUL

[4]#P is the class of enumeration problems that corresponds to NP-Complete decision problems. A #P problem asks how many linear extensions are there for a given partially ordered set. A polynomial-time solution to any of the #P-Complete problems implies that P=NP.

and SQRT. One of these instructions is selected. It is removed from the DG, and added to the partially constructed sequence. This processes repeats until the graph is empty.

Which instruction is selected from the ready list dictates the quality of the generated sequence. In our example sequence, we chose the MUL first. This choice led us to the suboptimal solution presented above.

List Scheduling uses a heuristic function to choose the next instruction to schedule. Given that the heuristic function has such profound impact on the quality of the generated schedule, many List Scheduling heuristics have been proposed. Smotherman et al. [39] provide an extensive survey of heuristics, their relative complexities, and effectiveness.

Dealing with just one basic block at a time is a little limiting. Consider our QUADRATIC function again. If BB1 and BB2 were scheduled together we could move the square root operation before the if-test. This of course would require a SQRT instruction that would not trigger a hardware exception when its operand is negative. This non-faulting instruction variant is actually common specifically for this kind of optimization. The question is which blocks should be combined?

Hwu et al. [24] combine basic blocks into Superblocks. Like basic blocks, superblocks have single entry points. However, they may have multiple exit points. Ideally, the last exit is the one most often taken. Figure 2.4 shows QUADRATIC's CFG with basic blocks BB1 and BB2 combined into a superblock. The combined DG contains instructions for both basic blocks. This enables the scheduling algorithm to move the square root instruction earlier in the sequence as discussed above.

If the superblock is constructed correctly for a given application, most invocations of QUADRATIC execute the block in its entirety. That is, the then-part of the if-

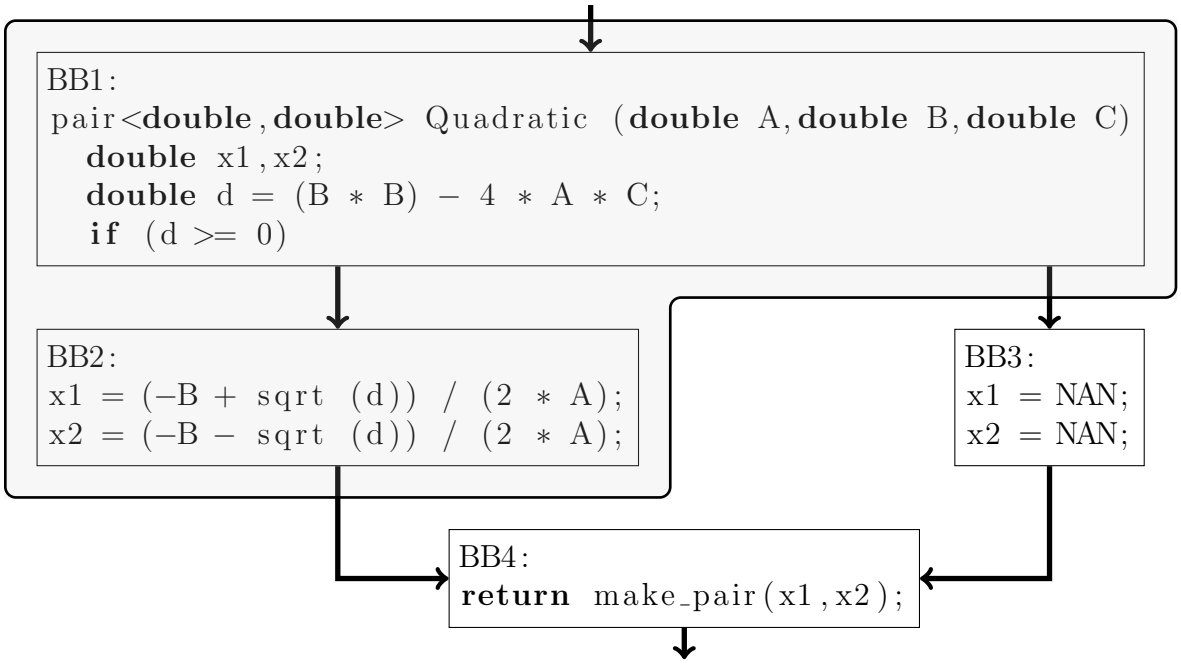


FIGURE 2.4. QUADRATIC's Control Flow Graph

statement is executed most often. If d is negative, then the side exit is taken and execution jumps to BB3. In this case, the SQRT executes but its result is unused. Generally, this is called speculative execution. They measure the degree of speculation and use it to judge the value of moving instructions between basic blocks.

Superblock construction depends on knowing the common execution path. Hwu et al. [24] use static code analysis to identify these common paths. Although it is effective, code analysis is fundamentally limited. QUADRATIC could be used in a larger program that uses QUADRATIC to computes many valid roots. Conversely, the larger program could be looking for a single set of parameters with valid roots. In this later case, QUADRATIC executes BB3 often but BB2 just once. Static analysis alone cannot always determine which behavior to expect.

Fisher [16] proposed using profiling data to capture execution traces. The program is compiled twice. First using a simple basic block scheduler with

instrumentation added. After running a few test cases, the instrumentation data is used to identify the common execution paths or traces. Trace scheduling then constructs and schedules trace-derived superblocks.

Many other block construction schemes have been proposed. Havanki et al. [23] construct Treegions that capture trees of basic blocks. In QUADRATIC, BB1, BB2, and BB3 could be collected into one treegion rooted at BB1. Using block duplication, two copies of BB4 could be generated and grafted on to BB2 and BB3 making the entire QUADRATIC program into one large treegion. Regions [21] combine basic blocks from other functions, essentially doing basic block level in-lining. Mahlke et al. [28] construct hyperblocks that, like treegions, can contain internal branching.

The goal of all of these block construction algorithms is to provide more flexibility to the scheduling algorithm. Allowing the scheduler to move instructions across basic block boundaries enables it to better handle long latency instructions and balance resource usage.

Given all of these different block selection and construction techniques, it might be surprising to note that they all use the same scheduling algorithm: List Scheduling. For example, Chang et al. [11] describe their six-step scheduling processes as follows:

1. Trace selection
2. Superblock formation
3. Superblock optimization
4. Dependence graph construction
5. Dependence graph optimization
6. List scheduling.

While it is ubiquitous, basic List Scheduling is not the only scheduling algorithm in use.

Pinter [35] approaches code generation more from a resource usage perspective than an instruction sequencing perspective. Starting with a DG, he builds a parallel graph called an Interference Graph that models the ways that instructions conflict with each other for CPU resources. In addition to limited functional units, physical registers are assigned. By considering register assignments, the Interference Graph includes more potential conflicts than traditional scheduling algorithms. The final sequence is generated by a modified List Scheduling algorithm that filters the set of ready instructions by the conflicts contained in the Interference Graph. This integrated scheduler/allocator approach is quite effective.

Goodman and Hsu [19] generate instruction sequences that are easier to allocate than those produced by a basic List Scheduler. Their List Scheduling adaptation uses two different heuristics: Code Scheduling for Pipeline processors (CSP) and Code Scheduling to minimize Register usage (CSR). CSP maximizes instruction level parallelism and leads to shorter schedules that are difficult to allocate. CSR, on the other hand, reduces the allocation work at the expense of longer schedules. While the List Scheduling algorithm builds the sequence, it keeps track of the number of physical registers used by the sequence. If the number of needed registers (or "register pressure") is low, the algorithm uses CSP. If the register pressure gets too high, the algorithm uses CSR. This two-focus composite heuristic is effective and led to several variations of Goodman's cooperative scheduler [6, 12, 31].

Goodman's approach still needs a separate register allocation pass. Typically, allocation recreates many of the same structures used by the scheduling algorithm. Leveraging this observation, Cutcutache and Wong [13] fold together List Scheduling with Linear Scan register allocation [36, 41]. Their focus was on decreasing compilation time of embedded compilation. By reusing the same structures, their

algorithm generates both an instruction sequence and a register allocation faster than as two separate tasks.

Like Goodman, Win and Wong [45] integrate scheduling with allocation. They use an iterative approach that builds a fully allocated sequence, then evaluates its resource usage. If some resource is overused the heuristic is adjusted via instruction weights and a new allocated sequence is created. This process continues until a valid schedule is constructed. This technique is very similar to Joslin and Clements [25] Squeaky Wheel Optimization, with one important distinction. Win and Wong iterate until a valid schedule is constructed, then stop. Joslin and Clements, on the other-hand, iterate a specified number of times keeping the best schedule as the result.

2.3. Search-Based Optimization

Win's integrated scheduler is an example of search-based scheduling. The other instruction schedulers built a single, valid sequence and stop. Win's approach builds several sequences and stops only when it finds a valid schedule. That is, it searches for a solution but it does not perform any optimization.

Search-based optimization considers many valid solutions. The best solution found is used as the final result. Unlike construction algorithms, search-based optimization needs an evaluation function.

Instruction Scheduling favors shorter valid schedules. With this evaluation function, a naive search-based optimization algorithm could try to enumerate every possible schedule favoring the valid schedule with the shortest execution time. On the plus side, this algorithm is guaranteed to find all optimum sequences. However, this approach is infeasible. There are $|V|!$ sequences where V is the set of instructions

in the DG. As discussed above, limiting the search to just the valid schedules is also infeasible. We must reduce or prune the search space further.

Malik et al. [29] generate optimum schedules using a pruning technique called branch-and-bound. The algorithm systematically generates each possible valid sequence. When a partially completed sequence is longer than the best sequences seen so far it gives up on the solution. Since the incomplete solution is already worse than the best seen, there is no point in completing it. Further, we know that any solution that shares this prefix will be suboptimal. This saves more than just the time to complete just the one candidate. It can prune large portions of solution space.

The main problem with branch-and-bound in this domain is that we must construct most of the instruction sequences before they can be shown suboptimal. That is, we save some but not a lot of time. Malik found that their scheduler took nearly two hours to schedule the Spec CPU200 benchmark suite. "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"[29].

Both exhaustive search and branch-and-bound are optimal search algorithms. During execution they construct a proof of optimality. The exhaustive search algorithm shows that all other solutions are inferior. Branch-and-bound is a bit more sophisticated. It shows that some early decisions lead to large numbers of poor solutions without actually constructing them. However, both of these techniques take too much time.

There are many search-based optimization algorithms, all of which share certain properties. For example, we can categorize algorithms based on the extent to which they explore the solution space. Another important property is completeness. Search

algorithms like those described above work through the solution space in a systematic fashion eventually finding the optimum solutions. An alternative approach would be to generate solutions randomly, possibly building the same solution many times. Given enough time, complete algorithms are guaranteed find the optimum solutions, but non-systematic algorithms tend to be simpler. They do not need to order the solution space nor keep track of what has been seen before.

In addition to the properties described above, different algorithms depend differently on heuristic functions. A heuristic is a "rule of thumb" that has proved useful in the past. Heuristics work well for domains where problems share some underlying similarities. In some cases, heuristic search algorithms can still offer proofs that their results are optimal. However, this is not possible in all cases. Often the best that can be claimed is that the solution is within some bound of an optimal solution.

Some algorithms search through heuristic space rather than solution space. These metaheuristic algorithms produce solutions indirectly. Instead of assuming some underlying similarity between problems, these algorithms develop an effective heuristic function for each problem individually. Each heuristic change is evaluated by constructing then evaluating the corresponding solution. The evaluation function further tunes the heuristic.

Not all algorithms are heuristically focused. Local search views solution space as a graph. Each valid solution is a vertex. Edges indicate which solutions are close to others. That is, edges define the local neighborhood of a solution. Morphing one solution into one of its neighbors is relatively easy. Local search starts at some solution, possibly heuristically generated, then moves to neighbors looking for a better solution.

Often local search algorithms employ some technique that keeps them from exploring the same few solutions repeatedly. For example, Tabu Search keeps a bounded list of "visited" solutions [18]. The local exploration cannot revisit these solutions again.

In this thesis we apply and evaluate three search-based optimization algorithms. A full description of each algorithm is provided in Chapter IV. Limited Discrepancy Search (LDS) is a heuristically directed complete search algorithm [22]. It uses the heuristic function to focus its search effort toward one area of solution space. Unlike List Scheduling, LDS is free to deviate from the heuristic to consider more than just one solution. To keep compilation time low, LDS explores a very small portion of the search space. However, when the problems are small enough, LDS's completeness allows it to search exhaustively and produce one optimum solution.

Squeaky Wheel Optimization (SWO) is a metaheuristic algorithm. It alternates between building an instruction sequence and constructing a new heuristic function. Each iteration attempts to tune the heuristic function toward better solutions. However, its nonsystematic approach means that even for very small problems there is no guarantee that an optimum solution will be generated.

Iterative Flattening (IFlat) [9] is a local search algorithm. IFlat is different from the algorithms we consider. It constructs a schedule and not just an instruction sequence. That is, IFlat assigns a start time to each instruction. Further, IFlat focuses on resource conflicts instead of DG dependencies. The initial solution is constructed by randomly resolving conflicts until a valid solution is constructed. Then, the local neighborhood is explored. Like the construction of the initial solution, neighborhood exploration is random. That is, IFlat is a nonsystematic local search algorithm that employs no techniques to prevent reconsideration of candidates.

2.4. Summary

This chapter has introduced and discussed compiler code generation. Generally speaking, all three major code generation tasks are NP-Complete problems. The Instruction Scheduling task is further divided into two subtasks: block selection and scheduling. Block selection identifies and collects a piece of the program to schedule together. The size and shape of scheduling blocks is an active area of research on its own. However, we are interested in the scheduling algorithm. The simplest effective block size is the basic block and this is the block selection scheme we will use throughout the remainder of this work.

Compilers have generally used a simple heuristically driven scheduling algorithm called List Scheduling. This one-pass algorithm is fast and, depending on the heuristic function, effective at generating efficient instruction sequences. However, Instruction Scheduling is an NP-Complete optimization problem. As such a simple one-pass algorithm cannot guarantee an optimal solution.[5]

We introduced search-based optimization and three ways to describe the algorithms. Further, we briefly introduced three algorithms that we will evaluate as scheduling algorithms for compiler code generation. That is, we propose to build and evaluate three basic block search-based Instruction Scheduling algorithms.

[5]This, of course, assumes $P \neq NP$. However, even if $P = NP$, a linear solution to Instruction Scheduling is unlikely.

CHAPTER III

THE SCHEDULING PROBLEM

This chapter introduces several formal problem statements that capture the details of Instruction Scheduling. These problems are similar to classic Job Shop scheduling. Job Shop scheduling is a source of initial intuition, but there are significant differences between Instruction Scheduling and Job Shop scheduling. These differences all relate to resources.

Job Shop resources fit nicely into two categories: capacity resources and reservoir resources. Capacity resources model tools and machines that are in use throughout the execution of a task. When the task is complete, these resources are immediately available. Reservoir resources, on the other hand, model work product and consumable items. Rivets and bolts are perfect examples. Some tasks produce these resources adding them to the reservoir or buffer. Other tasks consume these resources. As long as the buffer is nonempty, the consuming tasks can be scheduled. These models capture the resource behavior of a Job Shop, but fail to model CPU resources accurately.

The basic CPU resources are functional units and registers. Functional units contain computational logic like arithmetic units and floating-point units. Modern machines have several parallel functional units. At first glance, functional units appear like capacity resources but this is not the case. Functional units are pipelined. That is, each unit can start executing a new operation at each clock cycle. Rather than capturing the resource throughout the execution of an instruction, like capacity resources, the unit is only busy when the instruction starts or is dispatched.

Registers also behave unlike either of the Job Shop resources. Before an instruction can execute, all of its input data or operands must be available. These operands are usually the result of some other instructions. At first glance, it appears that the instruction that generates a value is the producer and the using instruction is the consumer of a reservoir resource. However, unlike rivets and bolts, which are interchangeable, the data defined by one instruction is very different from the data defined by another. The using instruction must have access to the right data, not just some data. Further, each datum takes storage space. Specifically, the defined data are stored in a limited set of registers that are busy until all using instruction have been dispatched.

The resource usage models are different from those in Job Shop scheduling. However, the two problem spaces share a great deal (e.g., parallel machines, latencies, and dependencies). Much of the computational complexity of Instruction Scheduling comes directly from these elements of Job Shop scheduling. In fact, we will reduce from classic scheduling problems to show that Instruction Scheduling is NP-Complete.

The remainder of this chapter is organized in the following manner. Section 3.1 presents the basic, register-unaware scheduling problem. This is the problem formulation addressed by most scheduling algorithms. Section 3.2 introduces special purpose registers and the corresponding scheduling problem. Section 3.3 considers the limited number of registers and the associated problem statement. Section 3.4 analyzes the complexity of this problem, showing it NP-Complete. Finally, Section 3.5 contains a summary and concluding remarks.

3.1. Basic Instruction Scheduling

Instruction scheduling is a variant of the resource constrained scheduling problem. As described above, the resources are functional units and registers. At the most basic level, the scheduler completely ignores the register limitations. Instead, it is assumed that the register set is unbounded. This assumption allows schedulers to focus on functional unit usage alone.

Modern processors are equipped with several parallel function units. Each functional unit is pipelined, allowing each to execute multiple instructions simultaneously. However, each pipeline can start at most one instruction in each clock cycle.

It is tempting to cast basic instruction scheduling as a special case of Job Shop scheduling where each task has unit latency. This approach does not work. Latencies are still important to the scheduling problem. While each operation uses a dispatching resource, the data it generates takes time. A dependent operation cannot start until the necessary data are available. For example, the SQRT instruction in Figure 2.3 takes 14 cycles to execute. Although it takes a single dispatch resource, the dependent DIV instruction cannot start until SQRT finishes 14 cycles later. This kind of data relationship is a kind of inter-instruction dependency.

There are two types of instruction dependencies. Primarily, instructions depend on the data produced by others. However, some dependencies enforce a particular execution order. For example, consider an increment instruction, INC, that modifies its single operand. That is, INC is destructive. If it shares its operand with another instruction, say ADD, then it is important that ADD is executed before INC. There is an order dependency between ADD and INC.

There is one additional complication that is worth noting. Superscalar processors, processors with more than one pipeline, honor scalar semantics. That is, the processor may execute two or more instructions in parallel but it will ensure that the effects are identical to serial execution. In the case where ADD and INC share an operand, both could be dispatched at the same time, but INC must appear second in instruction sequence. A schedule alone is not enough to generate a valid program.

Together the two types of dependencies make up one partial order, $\prec$. We will use the notation $\prec_d$ to denote a data dependency and $\prec_o$ to denote an order dependency. These two dependency types define two different problems: sequencing and scheduling. The scheduling problem is concerned with execution times and efficiencies. The sequencing problem focuses on program correctness. We use a combination of an instruction sequence and instruction schedule called a totally ordered schedule. That is, a solution to the scheduling problem is the pair $(\sigma, <)$ where $\sigma : V \to \mathbb{N}$ maps instructions to start times[1] and $<$ is a total order on V such that

$$x < y \Rightarrow \sigma(x) \leq \sigma(y).$$

With this notation we can present the following formal instruction scheduling problem statement:

[IS1] **Basic Instruction Scheduling**

INSTANCE: a set of operations V, two compatible partial orders $\prec_d$ and $\prec_o$, latencies $l : V \to \mathbb{N}^+$, dispatch limit (or number of parallel pipelines) P, and deadline D.

[1]We use $\mathbb{N}$ to denote the set of natural integers.

QUESTION: is there a totally ordered schedule $(\sigma, <)$, such that:

$$p \prec_o q \Rightarrow p < q \tag{3.1}$$

$$p \prec_d q \Rightarrow \sigma(p) + l(p) \leq \sigma(q) \tag{3.2}$$

$$0 \leq c < D \Rightarrow |\sigma^{-1}(c)| \leq P \tag{3.3}$$

$$\max_{v \in V} (\sigma(v) + l(v)) \leq D \tag{3.4}$$

Most of the constraints in IS1 follow directly from the above discussion: 3.1 ensures that all order dependencies are respected, 3.2 ensures enough time for operations to produce data, 3.3 enforces the functional unit dispatch limit, and 3.4 answers the fundamental question.

Most practical schedulers focus on effectively and efficiently solving IS1. In isolation, IS1 is missing important practical constraints defined by the hardware. We have already acknowledged that IS1 ignores the register limit: register management is left to a later compiler component. However, not all register assignments can be safely ignored; special purpose registers must be considered by the scheduler.

3.2. Special Purpose Registers

Most production hardware has some sort of special purpose registers. For example, MIPS has registers LO and HI that store the results of integer multiplication and division, Intel x86, ARM, and many other processors use one or more flags registers that store the results of comparisons. Of these special purpose registers, Intel's x86 flags register is the most difficult to schedule.

The flags register in the x86 instruction set stores the result of a comparison operations like CMP. CMP sets the flags to indicate the arithmetic relationship

between its two operands. For example, if comparing two equal operands, then CMP sets bits in the flags register to show this relationship. Usage of the flags register is illustrated in Figure 3.1. In the assembly fragment, the CMP operation sets the flags register and the JE jumps to the label *end_if* when the flags register shows that the last comparison was equal, skipping the body of the *if*-statement. There is no explicit link between the CMP instruction and the JE instruction in the assembly fragment. The flags register is an implied operand for these operations: an output operand for CMP and an input operand to JE. If the flags register were set by just a few operations, scheduling would be relatively simple. However, this is not always the case.

At first glance, the assembly fragment in Figure 3.1b appears equivalent to that of Figure 3.2a. However, the ADD instruction sets the flags register, overwriting the value set by CMP. ADD is not alone, many of the x86 integer operations set the flags register as a side effect. These operations implicitly compare their result with zero. In this fragment the results of the CMP instruction are unused. The actual behavior is like the source in Figure 3.2b, which is significantly different from Figure 3.1a.

Valid solutions must ensure that the data stored in each special purpose register is available to the correct using instruction. Before we can define the formal constraint we need a few definitions.

Each processor has a set of special purpose registers, F. For each register $f \in F$, let $\text{Define}_f \subseteq V$ denote the operations in V that define f and $\text{Use}_f : V \to 2^V$ the set of operations that use the value in f defined by each operation in V. Further, these

```
x += 1;
if (y != z) {
   ...
}
end_if:
```

(a) Source

```
x = ADD x, 1
    CMP y, z
    JE  end_if
     ...
end_if:
```

(b) Assembly

FIGURE 3.1. The C code fragment in (a) could be translated to the assembly fragment in (b). Note that the body of the if statement is elided in both fragments.

```
    CMP y, z
x = ADD x, 1
    JE  end_if
     ...
end_if:
```

(a) Assembly

```
y == z;
x += 1;
if (x != 0) {
   ...
}
end_if:
```

(b) Source

FIGURE 3.2. Incorrect schedule for Figure 3.1 and the actual corresponding source fragment.

sets satisfy the following constraints:

$$d \notin \text{Define}_f \Rightarrow \text{Use}_f(d) = \varnothing \tag{3.5}$$

$$d \in \text{Define}_f, u \in \text{Use}_f(d) \Rightarrow d \prec_d u \tag{3.6}$$

Constraint 3.5 prevents special purpose data dependencies on operations that do not set a special purpose register and 3.6 ensures that if a special purpose register dependency exists, it is encoded in the edges of the graph. We now have the support necessary to ensure that a totally ordered schedule satisfies the special purpose register constraints.

Given a total order $<$, we can keep track of instructions that need data stored in a special purpose register at the dispatch point of each instruction. Considering

an instruction $a \in V$ the set

$$\bigcup_{p<a} \text{Use}_f(p) \setminus \{p | p < a\}$$

contains all of the instructions that are sequenced at or after a that need the data stored in register f. If a sets f (i.e., $a \in \text{Define}_f$), then this set must either be empty or it must be only $\{a\}$. If it is empty, then a is safe to define f. If it is the set $\{a\}$, then a itself uses the value in f to compute the new value. This presentation considers just one instruction's effect on the special purpose register. If we verify that this is true for all instructions that set the register, then the ordering is valid with respect to this one spanning resource.

We can now generate the following formal problem statement.

[IS2] **Instruction Scheduling with Special Purpose Registers**

INSTANCE: in addition to the inputs to IS1, $(V, \prec_d, \prec_o, l, P, D)$, a set of special purpose registers F, special purpose register defining sets $\text{Define}_f \subseteq V$, and usage functions $\text{Use}_f : \text{Define}_f \to 2^V$.

QUESTION: is there a totally ordered schedule, $\sigma : V \to \mathbb{N}$ and $<$ that satisfies all of the constraints of IS1 and

$$f \in F, a \in \text{Define}_f \Rightarrow \left(\bigcup_{p<a} \text{Use}_f(p) \setminus \{p | p < a\} \right) \subseteq \{a\} \tag{3.7}$$

The new constraint, 3.7, ensures that data stored in each special purpose register is preserved until any operation that needs the data has been dispatched.

There are two intuitive interpretations of 3.7. The first is that a valid solution cannot order an operation in Define_f between any define-use pair of f, like CMP

and JE.[2] The alternate interpretation is instructions in Define_f are barriers. Any define-use pair must be scheduled both before, or both after, the barrier operation. In Figure 3.1b, ADD is the barrier operation. Since ADD cannot be scheduled after JE, otherwise it may not be executed, both CMP and JE must be sequenced after ADD. This barrier idea is essential for the proof that IS2 is NP-Hard (see Section 3.4).

IS2 is essential to correct compiler output. However, compilers that use IS2 as their problem definition still require a separate register allocation pass through the program. This separation of scheduling/sequencing from allocation simplifies the compiler implementation, but limits the effectiveness of both operations.

3.3. Limited Registers

The scheduler tries to schedule as many operations concurrently as possible, leveraging the parallel functional units. However, the instruction level parallelism requires the availability of more data; the operands of all of the executing instruction must be available. This makes register allocation more difficult. When the allocator runs out of registers, data is moved to memory. This temporarily frees a register for some other purpose. Moving data to memory is comparatively slow.

Consider a well optimized solution to IS2 that requires too many registers to effectively allocate. The register allocator will insert instructions that move data between registers and memory. The net effect of these moves may be a program that executes more slowly than a sub-optimal solution to IS2.

There are two strategies for allocation aware scheduling. More common is cooperative scheduling, which is register usage or register pressure sensitive. The second is an integrated scheduler/allocator that generates a schedule and register

[2]We use the term pair here rather loosely. There could be several users of that defined value. This is the case where $|\,\text{Use}_f(x)| > 1$.

allocation simultaneously. This section develops problem statements for both approaches.

3.3.1. Cooperative Scheduling

Cooperative schedulers monitor the number of registers needed by the schedule. Just like special purpose registers, the data stored in each register must be held until all operations that use the data have executed.

However, unlike special purpose registers that are defined within the scheduling problem or basic block, some registers hold data at basic-block entry. These are input data to the block. Similarly, some data are held in registers at basic-block exit; the block's results. Input and output data may be expressed as unschedulable, virtual operations. Let I define the set of input virtual operations, let O define the sets of output virtual operations, and let $\bar{V} = I \cup O \cup V$.

The set of data dependencies includes input data on the left and output data on the right. That is,

$$\prec_d \subseteq (I \cup V) \times (O \cup V) .$$

We can define Use as

$$\text{Use}(p) = \{q | p \prec_d q\} \subseteq (O \cup V) .$$

Given a sequence, $<$, we can measure the number of registers needed by the sequence. Using the above definitions:

$$\text{Pressure}(p) = |\{q | (q \leq p \vee q \in I) \wedge (\exists r \in \text{Use}(q), r \in O \vee p < r)\}| .$$

That is, register pressure at p is the number of data defining instructions that are separated from one or more using instructions by p.

With this definition of register pressure we can describe the cooperative scheduling problem formally.

[IS3] **Cooperative Instruction Scheduling**

INSTANCE: Same as IS2 except the set of operations is $\bar{V} = I \cup O \cup V$, and a register limit R.

QUESTION: is there a totally ordered schedule, $\sigma : V \to \mathbb{N}$ and $<$ that satisfies the constraint of IS2 and

$$\forall p \in V \Rightarrow \text{Pressure}(p) \leq R \tag{3.8}$$

Constraint 3.8 ensures that the number of registers needed to hold the produced data never exceeds the limit, R. This constraint is enough for cooperative scheduling. Generally, registers are interchangeable and the allocator is free to choose any physical assignment. However, simply measuring the total register pressure is insufficient for practical compilers and preassigned registers.

3.3.2. Integrated Scheduling

The formulation of IS3 assumes that all general purpose registers are interchangeable. However, this is not the case. Some instructions are tied to specific registers. ARM's branch-with-link instruction, BL, stores the return address in register R14 and Intel x86's integer divide uses registers EDX and EAX as implied input and output operands.

At first glance, preassigned and special purpose registers appear to be the same. Unlike special purpose registers, preassigned registers can be used for other purposes. For example, the EAX and EDX registers can be used as operands other instructions

(e.g., ADD and MOV). It is just a few instructions, like IDIV, that require the use of these specific registers.

Preassigned registers are not limited to instruction set idiosyncrasies, but also part of the execution environment. Traditionally, function arguments and return values are passed in memory. The calling code stores the values to their assigned location.[3] Accessing memory is relatively slow. In modern systems, arguments and return values are stored in registers. These locations are set by published standards. For example, the execution convention for 64-bit Intel x86 places the first three integer-sized arguments in RDI, RSI, and RDX and it places the first two return values in RAX and RDX. Notice that RDX is used for both input and output. This double-duty may require some register juggling.

An integrated scheduler and allocator generates a totally ordered schedule and a register assignment, $A : \bar{V} \to R$ where R is the set of hardware registers. The register assignment must honor all of the preassigned registers: hardware and compiler assigned. Additionally, the register assignment creates the same register availability problems as special purpose registers. We can extend Define and Use to include general purpose registers.

$$\text{Define}_r = A^{-1}(r)$$

$$\text{Use}_r(v) = \begin{cases} \text{Use}(v) & \text{if } A(v) = r \\ \varnothing & \text{otherwise.} \end{cases}$$

[3]In languages like C and Pascal, arguments are stored in main memory locations relative to the execution stack. FORTRAN stores arguments in compiler-assigned, fixed memory locations.

With these extensions, we can now formally describe the Integrated Instruction Scheduling problem.

[IS4] **Integrated Instruction Scheduling**

INSTANCE: In addition to the inputs to IS2, a set of general purpose registers R, an initial register assignment $A^* : V' \to R$ (where $V' \subseteq V$),

QUESTION: is there a totally ordered schedule, $\sigma : V \to \mathbb{N}$ and $<$, and total register assignment $A : V \to R$ that satisfies the constraints of IS2 and

$$A^*(v) = r \Rightarrow A(v) = r \tag{3.9}$$

$$r \in R, a \in \text{Define}_r \Rightarrow \left(\bigcup_{p<a} \text{Use}_r(p) \setminus \{p | p < a\} \right) \subseteq \{a\} \tag{3.10}$$

Constraint 3.9 ensures that A honors the pre-assigned registers. 3.10 places the same define-use limits on the general purpose registers that 3.8 places on special purpose registers. That is, it ensures that the data defined by one operation remains in the register until all operations that use that data are executed.

3.4. Problem Complexity

This chapter has introduced four variants of instruction scheduling. This section will show that each is an NP-Complete problem. The reduction strategy is illustrated in Figure 3.3. We begin with IS1.

Claim 1. *IS1 is NP-Hard.*

Proof. This proof is by reduction from the Precedence Constraint Scheduling presented by Garey and Johnson [17, pp. 239], which was proved NP-Complete by Ullman [42] using a reduction from SAT3.

[SS9] **Precedence Constrained Scheduling**

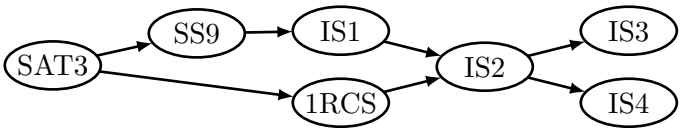


FIGURE 3.3. Reduction outline showing Instruction Scheduling NP-complete.

INSTANCE: a set T of tasks, each having length $l(t) = 1$, a number $m \in \mathbb{N}^+$ of processors, a partial order $\lessdot$ on T, and a deadline $D \in \mathbb{N}^+$.

QUESTION: is there an m-processor schedule σ for T such that:

$$p \lessdot q \Rightarrow \sigma(p) + l(p) \leq \sigma(q) \tag{3.11}$$

$$0 \leq c < D \Rightarrow |\sigma^{-1}(c)| \leq m \tag{3.12}$$

$$\forall t \in T. \quad \sigma(t) + l(t) \leq D \tag{3.13}$$

Precedences are guaranteed by 3.11, 3.12 ensures that at most m machines are utilized at any time, and the overall deadline is enforced by 3.13.

Given an instance of SS9, we can construct an instance of IS1 by setting $V = T$, $\prec_d \ = \lessdot$, $\prec_o \ = \varnothing$, $P = m$, with the same latencies and deadline. It remains to be shown that IS1 answers yes exactly when the answer to the SS9 problem is yes. If there is a schedule for the SS9 problem, then it supports 3.11, 3.12, and 3.13. These three constraints are equivalent to 3.2, 3.3, and 3.4 of IS1. Since $\prec_o \ = \varnothing$, 3.1 is always satisfied. Thus, IS1 will answer yes if SS9 answers yes. Conversely, if IS1 is yes, then the four constraints are satisfied, 3.1 trivially. By the same equivalences described above, the three constraints of SS9 are satisfied. Thus IS1 answer yes only for those problem instances that SS9 answers yes. Therefore, SS9 $\leq_p$ IS1 and, according to Ullman [42], SAT3 $\leq_p$ SS9, proving IS1 is NP-Hard. □

While it is sound, this proof only shows that the general form of IS1 is NP-Hard. Ullman's reduction from SAT3 to SS9 requires P proportional to the number of clauses in the SAT3 instance. However, practical compilers target specific hardware or a closely related family of hardware. These systems have a fixed number of pipelines, not an unbounded number. Garey and Johnson [17] noted this in their discussion of SS9, "[this problem] can be solved in polynomial time if m=2... Complexity remains open for all fixed $m \geq 3$ when $\preccurlyeq$ is arbitrary." This leaves the practical complexity of ISP with a fixed P in a dubious state.

Returning to Figure 3.3, this establishes the edges from SAT3 to SS9, thanks to Ullman, and from SS9 to IS1.

Proving the edge from IS1 to IS2 is straightforward, as IS1 is a sub-problem is IS2. That is, constraining IS2 with $F = \varnothing$, we are left with IS1 exactly (3.7 is trivially satisfied). However, this leaves the NP-Hardness of IS2 in the same dubious state as IS1. We can make a stronger claim about the complexity of IS2.

Claim 2. *IS2 is NP-Hard with bounded P, F, and l.*

Proof. The proof is by reduction from a special case of Resource Constrained Schedule called 1RCS (defined formally below). The 1RCS problem comes from the domain of Job Shop Scheduling. Like our Instruction Scheduling problems, 1RCS has a set of tasks with precedences. The special case has only two machines that can perform the tasks and a single shared resource that is used by some of the tasks. At most one task can use the shared resource at any one time; this property is what makes 1RCS difficult.

Formally, 1RCS is defined as follows:

[1RCS] **One-Resource Constraint Scheduling**

INSTANCE: a set T of tasks, a partial order $\lessdot$ on T, a set of resource use tasks $R \subseteq T$, and a dead line D.

QUESTION: is there a schedule τ for T such that:

$$p \lessdot q \Rightarrow \tau(p) < \tau(q) \tag{3.14}$$

$$0 \leq c < D \Rightarrow |\tau^{-1}(c)| \leq 2 \tag{3.15}$$

$$0 \leq c < D \Rightarrow |R \cap \tau^{-1}(c)| \leq 1 \tag{3.16}$$

$$\forall t \in T \quad \tau(t) < D \tag{3.17}$$

Much of this problem formulation is from Garey and Johnson [17]. The original problem is from Ullman [43] where it is proved NP-Complete.

Constraint 3.14 requires that the schedule, τ, honors the precedences contained in the partial order $\lessdot$. Constraint 3.15 prevents the schedule from needing more than two machines at any time point. Constraint 3.16 ensures that at most one task can use the shared resource at any one time. Finally, 3.17 requires that all tasks are scheduled before the deadline.

Notice that at any time in a valid schedule there are five possible task patterns:

1. No tasks are scheduled
2. One task is scheduled that uses the resource
3. One task is scheduled that does not use the resource
4. Two tasks are scheduled that do not use the resource

5. Two tasks are scheduled where one does and one does not use the resource

The missing sixth case is where two tasks are scheduled that both use the resource. This is not valid since there is only one shared resource and it is prevented by 3.16.

From an instance of 1RCS we will construct an instance of IS2 as follows:

- (V): For each $t \in T$, add three instructions, t_1, t_2, and t_3. Add $\{B_i | 0 \leq i \leq D\}$ (for the deadline D of the 1RCS problem).
- $(\prec_d)$: For $p, q \in T$ and $p \lessdot q$, then $p_3 \prec_d q_1$. For $t \in T$, $t_1 \prec_d t_2 \prec_d t_3$. For $0 \leq i < D$, $B_i \prec_d B_{i+1}$.
- $(\prec_o)$: Empty.
- (l): For $t \in T$, $l(t_1) = l(t_3) = 1$. If $t \in R$, then $l(t_2) = 2$. Otherwise $l(t_2) = 5$. For $0 \leq i < D$, $l(B_i) = 8$ and $l(B_D) = 1$.
- (P): The dispatch limit $P = 1$.
- (D): The deadline for the IS2 problem is $1 + 8D$.
- (F): Three special purpose registers x, y, z.
- (Define_x): For all $t \in T$, $t_1 \in \text{Define}_x$. For $0 \leq i \leq D$, $B_i \in \text{Define}_x$.
- (Define_y): For all $t \in T$, $t_2 \in \text{Define}_y$. For $0 \leq i \leq D$, $B_i \in \text{Define}_y$.
- (Define_z): For all $t \in T$, $t_2 \in \text{Define}_z$. For $0 \leq i \leq D$, $B_i \in \text{Define}_z$.
- (Use_x): For all $t \in T$, $\text{Use}_x(t_1) = t_2$).
- (Use_y): If $t \in R$, $\text{Use}_y(t_2) = t_3$.
- (Use_z): If $t \notin R$, $\text{Use}_z(t_2) = t_3$.

The B_is must be scheduled at $\sigma(B_i) = 8i$. Otherwise the latencies would cause the last B_D to be scheduled past the deadline. Since each B_i defines each of the three special purpose registers and the three operations for each task have a special purpose define-use relationship, all three operations must be scheduled between two B_is. For example, for any $t \in T$, if t_1 were scheduled before some B_i and t_2 were scheduled after B_i, t_2 would loose access to t_1's definition of x. This would violate IS2's Constraint 3.7.

Between any two B_i and B_{i+1} we are left with five scheduling patterns. Table 3.1 illustrates the five possible patterns.[4] Notice that these five patterns correspond exactly to the five patterns described above for 1RCS. If a schedule exists for IS2, we can construct a schedule if 1RCS by mapping the patterns in Table 3.1 to the patterns for 1RCS. Conversely, given a solution to 1RCS, we can construct a solution to IS2 by the inverse mapping.

To show that the dependencies in 1RCS are enforced correctly in IS2 consider any two tasks $p, q \in T$ where $p \lessdot q$. By inspection of Table 3.1 we see that $\sigma(p_3) > \sigma(q_1)$ when p and q are scheduled between the same barrier instructions, B_i and B_{i+1}. This ensures $\sigma(p) < \sigma(B_i) < \sigma(q) < \sigma(B_i)$. Thus the IS2 formulation honors the dependencies of 1RCS.

Finally, both construction of the IS2 instance and conversion of the answer back to 1RCS can be done in polynomial time. Ullman [43] reduced SAT3 to 1RCS. Hence, IS2 is NP-Hard with P, F, and l bounded. □

This proves the edges in Figure 3.3 left of IS2. The edges from IS2 to IS3 and IS4 are easily shown since IS2 is a sub-problem of both. That is, IS3 adds 3.8 to IS2. Given an instance of IS2, we can construct an instance of IS3 with $R = |V|$. In this

[4]The Type 3) pattern includes any right-shifting of the three operations.

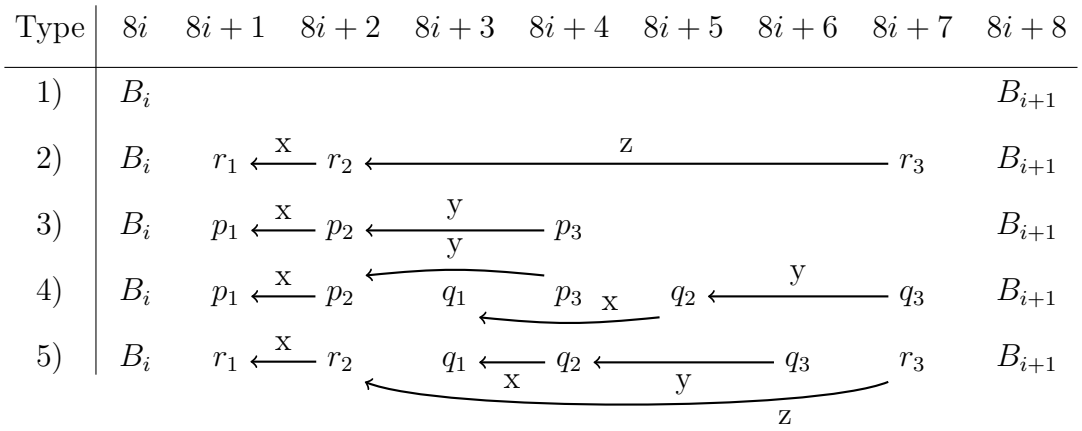


TABLE 3.1. The five possible partial schedules for the 1RCS problem converted to IS2 with $p, q, r \in T$, $p, q \notin R$, and $r \in R$. Arrows indicate the define-use relationship between operations by special purpose register.

case 3.8 is always satisfied and the solutions are identical. The same is true for IS4, let $A^* = \varnothing$, $R = V$, and $A(v) = v$. Both 3.9 and 3.10 are trivially satisfied.

We have established that all four scheduling problems are NP-Hard. There are several options for showing that the four problems are NP-Complete. The simplest is showing that a solution, σ and $<$, is verifiable in polynomial-time. By inspection, all of the solution constraints are easily verifiable in polynomial-time. Alternatively, basic blocks are linear. We could simply execute the schedule and verify the constraints.

Claim 3. *IS1, IS2, IS3, and IS4 are NP-Complete.*

3.5. Summary

This chapter has developed four formal problems statements that capture the complexities of instruction scheduling. At the most basic level, instruction scheduling and Job Shop scheduling are similar, differing primarily in resource usage models.

The problem statements presented above are each formulated as decision problems. With these formulations, we proved each NP-Complete. However, we

are more interested in the optimization problem. That is, what totally ordered schedule has the minimum deadline? Three of these problems, IS1, IS2, and IS4, are in $\mathsf{FP}^{\mathsf{NP}[\log n]}$.[5] That is, a program can produce an optimum totally ordered schedule in polynomial time by calling an NP-oracle a logarithmic number of times.[6] Problem IS3 is different in that it is a multi-objective optimization problem.

Rather than a single value, IS3 optimizes over both D and R. These two values compete with each other. Rather than a single optimum value, a solution to IS3 is a set of values, that are all Pareto optimal. Each solution is superior to all other members of the set by either D or P. That is,

$$\forall s \in S \Rightarrow \forall t \in S, t = s \vee D(s) < D(t) \vee P(s) < P(t).$$

This puts IS3 in the slightly broader complexity class of FP^{NP}. That is, polynomial time with a polynomial number of inquires to an NP-oracle.

Generally speaking, any of these complexity classes is too hard to solve to completion; unless, of course, $\mathsf{P} = \mathsf{NP}$. Given that exact solutions are sometimes impractical, we are justified in exploring a search-based approach to instruction scheduling.

[5]To truly be $\mathsf{FP}^{\mathsf{NP}[\log n]}$ we must bound instructions latencies by a polynomial of the input size.

[6]The algorithm to solve the optimization problem is a binary-search through deadlines until the NP-oracle answers no for $D - 1$ and yes to D.

CHAPTER IV

SEARCH-BASED OPTIMIZATION

Search-based optimization algorithms acknowledge a certain level of uncertainty or intractability of solution domains. Rather than investing in more complicated domain models, these algorithms probe through the set of possible solutions actively seeking better results. Some search algorithms depend heavily on a well tuned heuristic function. These algorithms leverage similarities between the problems to direct the search toward promising areas of the solution space. Other algorithms assume little similarity between problem instances. These algorithms probe around the search space developing a problem-specific heuristic function. Finally, some algorithms do not use heuristics at all. These algorithms assume very little about the solution space or, rather, they assume that any meaningful heuristic functions are themselves intractable. These techniques tend to construct many candidate solutions and employ efficient evaluation algorithms to find the best results.

In this chapter we will explore one search technique from each of the algorithm classifications described above. Limited Discrepancy Search (LDS) uses a well-tuned heuristic function to guide schedule construction. The heuristic pushes construction toward specific areas of the search space but the algorithm explores more than a single solution. Squeaky Wheel Optimization (SWO) focuses its attention on the difficult-to-schedule tasks, effectively discovering a heuristic function tuned for the specific scheduling problem. Finally, Iterative Flattening (IFlat) does not use a heuristic function at all. Instead, it focuses its attention on resource-heavy points within a schedule. By resolving resource violations, IFlat initially builds a valid solution. It then explores similar solutions by creating then resolving new resource conflicts.

While the focus of this thesis is Search-Based Instruction Scheduling, we begin by describing the ubiquitous List Scheduling algorithm.

4.1. List Scheduling

List Scheduling is the de facto standard Instruction Scheduling algorithm. Despite its name it is actually an instruction sequencer. That is, it generates a valid sequence but it does not schedule the instructions' start times.[1] List Scheduling is an adaptation of the topological sorting algorithm by Kahn [26]. The algorithm for list scheduling is shown in Figure 4.1.

List Scheduling gets its name from the ready list or set in this description. This set contains the instructions that are "ready" to be sequenced. The ready set invariant is

$$\forall r \in \mathit{ready}, p \in \mathit{preds}(r), r \notin S \wedge p \in S \,.$$

We abuse notation here by using the list S as an operand to the set operator $\in$. The invariant is initially established at line 2 by adding all instructions without predecessors. Since S is empty and these instructions have no predecessors, the invariant is trivially true. Lines 8 and 11 maintain the invariant. This implementation keeps a count of unsequenced predecessors for each instruction, P. When an instruction is added to the sequence, the P value is decremented for each of that instruction's successors. When the P count for some instruction, v, is exhausted, all of its predecessors are in S, and v is added to the ready set.

[1] List Scheduling could be extended to assign start times, but the affects these times have on the remainder of the schedule are generally not used.

```
LIST-SCHEDULE(G = (V, ≺), <)
 1  For each v ∈ V, P(v) ← | preds(v)|
 2  ready ← {v|v ∈ V ∧ preds(v) = ∅}
 3  S ← []
 4  while ready ≠ ∅
 5        do choose v from ready minimal by <
 6           S ← S : v
 7           ready ←ready\{v}
 8           for w ∈ succ(v)
 9                 do P(w) ← P(w) − 1
10                    if P(w) = 0
11                       then ready ← ready ∪{w}
12  return S
```

FIGURE 4.1. Naive List Scheduling Algorithm

The sequence, S, is built incrementally by moving one instruction from the ready set to S. The sequence invariant is the constraint

$$S = \langle\rangle] \vee \left(S = \langle T^\frown v\rangle \wedge \mathit{preds}(v) \subseteq T\right).$$

Again, we abuse set notation by using the list T as an operand to $\subseteq$. This invariant is maintained initially by setting S to the empty list. It is maintained at line 6 by appending an element of the ready set, v. Since v was in the ready set, the ready set invariant ensures S's invariant is maintained.

The main loop invariant is the combination of the ready set invariant and the partial sequence invariant. What remains is to show that when the main loop exits, S contains a valid and complete sequence of G. The loop terminates when the ready set is empty. If S were incomplete then there are some instructions that were never added to the ready set. These instructions have predecessors that were never sequenced. Since G is finite, this implies that there is a cycle in G. But, G is an acyclic directed

graph so this cannot be the case. Finally, since after each instruction is added to S the sequenced invariant holds, the completed S is valid and complete.

Topological Sort's runtime complexity is well known. The inner loop considers each edges exactly once. That is, The inner loop executes once for each edge. The outer loop executes exactly once for each vertex. Assuming that the rest of the operations within the loop are constant time, the sorting algorithm has complexity $O(|V| + | \prec |)$, linear in the size of the input. For shorthand, let $n = |V| + | \prec |$. This gives us $O(n)$. However, selecting the minimal element from the ready set is not necessarily constant time.

If the heuristic is static, meaning that it does not depend on the partially created sequences, then the ready set can be implemented as a priority queue. Operations on a priority queue generally take $O(\log n)$ time. Depending on implementation, it could be $O(\log n)$ insert, or $O(\log n)$ remove-min, or some combination of both. Since each instruction is inserted and removed exactly once from the priority queue, these details are unimportant. What is important is that, when using a priority queue, list scheduling has runtime $O(n \log |V|) = O(n \log n)$.

There are two details that must be addressed. First, not all heuristics functions are static. Some examine the partially constructed sequence when choosing an instruction from the ready set. For example, the dynamic critical path heuristic gives priority to instructions on the critical path given the timing of the partial sequence. In the worst case, these heuristics preclude a simple priority queue and a full scan of the ready set may be needed. Further, calculating the heuristic may be a complex task. For these types of heuristics, list scheduling takes time $O(n^2 h)$ where h is the time-complexity of the heuristic function.

The second problem is resource constraints. Since list scheduling is simply a sequencer, the dispatch limit is unaddressed. The actual schedule is determined by the CPU at execution time. Internally, the CPU uses register assignments to track dependencies. If the data needed by one instruction is still being computed, the instruction is delayed. While List Scheduling can ignore the instruction schedule, it must still manage special purpose registers.

We have established that sequencing with special purpose registers is an NP-Complete problem. A simple, one-pass, heuristically-greedy scheduler is insufficient. Schedulers in practical compilers get around this issue by altering the scheduling problem. Typically, this involves saving the results of some operations to main memory to be restored later or regenerating the data by duplicating the defining instruction. The second approach effectively de-optimizes the schedule.

Modifying the scheduling problem presents a whole new optimization problem. For this evaluation, we do not alter the scheduling problem. Instead, we have expanded list scheduling to track special-purpose register usage. A set $State_f$ is maintained for each special purpose register. When an instruction, v, that defines a special purpose register, f, is moved from *ready* to S, $State_f = \text{Use}_f(v)$. However, if the data stored in f is needed by an unsequenced instruction, then adding v to S is invalid. Stated another way, S extended with v is valid if either $v \notin \text{Define}_f$ or $State_f \subseteq \{v\}$.

Selecting an instruction from *ready* is now a little more complicated. Rather than simply choose the heuristically minimum instruction in *ready*, the algorithm chooses the heuristically minimum instruction in *ready*, v, such that $S^\frown v$ is valid. If there is no such instruction, the algorithm fails. We extend the Naive List Scheduler

to manage a single special purpose registers, shown in Figure 4.2. Extending the algorithm to include multiple special purpose registers is straightforward.

The runtime complexity of the special purpose register aware List Scheduler is different from the original algorithm. Choosing v is more difficult than a simple priority queue operation. In the worst case, the entire queue must be searched. This is an $O(n)$ operation that is invoked once for each instruction. The extended algorithm is now $O(n^2h)$ where h is complexity of the heuristic function. For our purposes we assume that the heuristic runs in constant time. For static heuristics, this is not an unreasonable constraint.

Before we move on to search-based instruction scheduling, we make one further extension to list scheduling: backtracking. Rather than simply failing when there is no valid candidate instruction in the ready set, backtracking enables the algorithm to retract a decision and try an alternative. Consider the scheduling problem shown in Figure 4.3. This very simple problem has three instructions with C depending on both A and B. Additionally, both A and B define a special purpose register but C depends only on A's definition; this is illustrated by the bold edge between C and A. The List Scheduling algorithm presented above fails to sequence this problem if it uses a heuristic that favors A over B. Initially the ready list contains A and B. Following the heuristic, A is moved from the ready list to S. Now only B is ready but B cannot be added to S. That would overwrite A's definition of the special purpose register. With the addition of backtracking the algorithm can move A back into the ready set and choose B. This decision leads to the only valid sequence B, A, C.

We extend List Scheduling with backtracking for two reasons. First, backtracking enables List Scheduling to sequence any solvable scheduling problem. This puts List Scheduling on an equal footing with the search-based schedulers described below.

```
LIST-SCHEDULE(G = (V, ≺), <)
 1  For each v ∈ V, P(v) ← | preds(v)|
 2  ready ← {v|v ∈ V ∧ preds(v) = ∅}
 3  S ← []
 4  State_f ←∅
 5  while ready ≠ ∅
 6        do choose v from ready minimal by < s.t. Define_f(v) ⇒ State_f ⊆ {v}
              If no such v, return FAIL
 7           S ← S : v
 8           State_f ←(State_f \{v}) ∪ Use_f(v)
 9           ready ←ready\{v}
10           for w ∈ succ(v)
11                 do P(w) ← P(w) − 1
12                    if P(w) = 0
13                       then ready ← ready ∪{w}
14  return S
```

FIGURE 4.2. Special Purpose Register Aware List Scheduling Algorithm

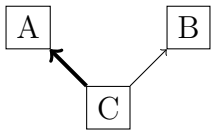


FIGURE 4.3. Scheduling Problem with Special Purposes Register conflict.

Second, it provides a gentle introduction to the algorithmic layout of the search-based schedulers.

Figure 4.4 illustrates the backtracking version of List Scheduling. Unlike the previous descriptions, this presentation is recursive rather than iterative. This allows us to save the backtracking information in the call stack rather than an explicit structure.

There are a few things to notice about this presentation. First, there are three returns: Lines 3, 9, and 10. The return on Line 3 corresponds to the final, successful return in the iterative presentation. That is, the ready set is empty and S is a

```
LIST-SCHEDULE-REC(G = (V, ≺), <, S, State_f)
 1  ready ← {v | v ∈ V ∧ preds(v) ⊂ S}
 2  if ready = ∅
 3     then return S
 4  for v ∈ ready increasing by <
 5        do if Define_f(v) ⇒ State_f ⊆ {v}
 6              then NewState_f ← (State_f \ {v}) ∪ Use_f(v)
 7                   Sln ← LIST-SCHEDULE-REC(G, <, (S : v), NewState_f)
 8                   if Sln ≠ FAIL
 9                      then return Sln
10  return FAIL

LIST-SCHEDULE(G = (V, ≺), <)
1  return LIST-SCHEDULE-REC(G, <, [], ∅)
```

FIGURE 4.4. List Scheduling Algorithm with Backtracking

valid sequence. The return on Line 10 corresponds to the failure return on Line 6 of Figure 4.2. The return on Line 9 is peerless in the iterative presentation. This return corresponds to the while-loop iterations. More accurately, it corresponds to leaving the while-loop and discarding the saved information related to backtracking.

Each recursive call to LIST-SCHEDULE-REC, Line 7, moves an instruction from the ready set to the sequences S. While heuristically preferred, adding the selected instruction to the sequence may cause the remaining problem to be invalid.

If there is a valid sequence, List Scheduling with Backtracking will eventually find it. This could take time exponential in the size of G in the wost case. This, however, is somewhat unsatisfying. In practice backtracking is only necessary in about 0.5% of the benchmark problems. That is to say, 99.5% of the time, this algorithm has runtime $O(n^2)$.

Adding backtracking to List Scheduling turns the one-pass construction algorithm into a systematic search algorithm. However, it performs no optimization.

Like Win and Wong's [45] scheduler, this algorithm simply runs until it finds one valid solution.

4.2. Limited Discrepancy Search

Good heuristic functions significantly improve the quality of the solutions generated by algorithms like List Scheduling. Well tuned heuristics are effective but they cannot overcome the inherent complexities of Instruction Scheduling.

Harvey and Ginsberg [22] found that, while heuristics may not lead directly to the optimum solution, the heuristic solution is often close to the optimum. Closeness, in this context, refers to how many times the heuristic selected the wrong instruction from the ready set. LDS uses this observation to direct exploration through the scheduling search space. Rather than simply following the heuristics, LDS will construct alternative schedules by occasionally not following the heuristic selection. Ignoring the heuristic is called a discrepancy. A parameter to LDS is the maximum number of discrepancies to use when generating alternative solutions.

Consider the search space illustrated in Figure 4.5. In the figure, each decision point is binary. That is, $|\,ready\,| = 2$ at each point except for the leaves of the tree. This binary branching is for illustrative purposes only; in practice the branching factor is larger than two. Say that at each decision point, the heuristic chooses the left-hand edge at each internal node. List Scheduling would follow the left-most path to leaf g and stop. Given a single discrepancy, LDS would explore leaves g, k, i, and h.

Harvey and Ginsberg's LDS uses discrepancies as early as possible. In this example, leaves are explored from k to g. The original LDS algorithm initially searches the tree with no discrepancies, finding leaf g. It then searches the tree

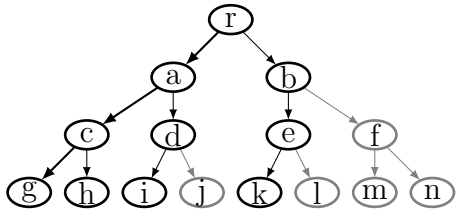


FIGURE 4.5. Limited Discrepancy Search search tree.

with one discrepancy, finding (k, i, h, g). It continues searching the tree with one more discrepancy than the previous search until the entire tree is explored.

The original presentation of LDS searches for a valid solution. In this domain we are looking for better schedules, not necessarily the best.[2] Rather than follow Harvey and Ginsberg's increasing discrepancy approach, we fix the number of discrepancies and search through the reachable portion of the tree in its entirety. In a production scheduler, coupling Harvey and Ginsberg's approach with a time-out may be effective.

The LDS scheduling algorithm is shown in Figure 4.6.[3] This algorithm is similar to List Scheduling with Backtracking. The differences begin at Line 8. This is where the List Scheduling algorithm finished and returned its solution. If there are no discrepancies available, $D = 0$, and LDS behaves the same. When discrepancies are available, LDS constructs a new heuristic function with the selected instruction, v, penalized (described below). This heuristic function is used to construct an alternative solution via a second recursive call to LDS-Rec. In addition to the new heuristic, this recursive call is given one fewer discrepancy and a new portion of the search space is explored.

[2]As we discussed in the summary of Chapter III, in practice Instruction Schedules is not an NP-Complete decision problem but an $\mathsf{FP}^{\mathsf{NP}[\log n]}$ optimization problem.

[3]We overload the meaning of min to return the preferred solution. For Instruction Scheduling, this is the solution with the shorter total execution time. As a tie breaker, solutions with lower register pressure are preferred.

```
LDS-REC(G = (V, ≺), <, D, S, State_f)
 1  ready ← {v|v ∈ V ∧ preds(v) ⊂ S}
 2  if ready = ∅
 3     then return S
 4  for v ∈ ready increasing by <
 5        do if Define_f(v) ⇒ State_f ⊆ {v}
 6              then NewState_f ← (State_f \{v}) ∪ Use_f(v)
 7                   Sln ← LDS-REC(G, <, D, (S : v), NewState_f)
 8                   if Sln ≠ FAIL ∧ D > 0
 9                      then ≪←< with v penalized
10                           Alt ← LDS-REC(G, ≪, D −1, S, State_f)
11                           if Alt ≠ FAIL
12                              then return min(Alt, Sln)
13                      else  return Sln
14  return FAIL

LDS(G = (V, ≺), <, D)
1   return LDS-REC(G, <, D, [], ∅)
```

FIGURE 4.6. Limited Discrepancy Search

There are many approaches to penalizing an instruction. We chose to adjust the heuristic ordering by moving penalized instructions after non-penalized instructions. Otherwise, the heuristic ordering is preserved. Effectively, we used several priority queues. One queue is for non-penalized instruction, one for penalized instruction, another for doubly-penalized instruction, and so on. Instructions are pulled from the non-penalized queue first.

Unlike List Scheduling, LDS can construct several alternative solutions. We must ask: how many solutions does LDS generate given D discrepancies? Returning to the search tree representation, each leaf denotes a full sequence and each interior node a decision point. For any scheduling problem graph, $G = (V, \prec)$, the height of the corresponding decision tree is $H = |V|$. Each node corresponds to extending

the partial solution with an additional instruction. For this analysis, we assume that every interior node in the tree has at least D choices; this is not always the case. So the question restated is, how many leaves are reachable by LDS with D discrepancies?

Each leaf has a unique path from the root. Any leaf explored by LDS has at most D discrepancies at any of the $H-1$ interior levels in the tree. That is, the number of generated sequences is at most

$$\sum_{d=0}^{D} \binom{(H-1)}{d} \leq (H)^D = O\left(|V|^D\right) = O(n^D). \tag{4.1}$$

In the degenerative case, $D = 0$, LDS produces just one sequence, the same solution as List Scheduling.

Creating those solutions is not without its costs. Considering the runtime complexity of LDS, the worst case answer is easy: because it is built on the backtracking variant of List Scheduling, LDS's runtime is exponential in the size of the input.[4] Just like List Scheduling with Backtracking, this answer is somewhat unsatisfying but we can make the same assumptions.

Assume that backtracking is rare and can be safely ignored. That is, there is an instruction v such that $\text{Define}_f(v) \implies \mathit{State}_f \subseteq \{v\}$ and the recursive call never returns *FAIL*. The runtime complexity of each individual call to LDS-Rec is $O(|V|)$ since scanning the entire ready set may be required (e.g., the only valid instruction is heuristically last in the ready set). Each recursive call to LDS-Rec corresponds to an edge in the search tree. So, the complexity of LDS is proportional to the number of edges in the search tree.

[4]Note that this exponential runtime is an artifact of the underlying algorithm and is not an attribute of LDS with a fixed number of discrepancies.

Except for the root, each node is reached by traversing an edge. So, we are equally interested in the number of unique nodes visited in the search. By the same arguments for Equation 4.1, we can enumerate the number of nodes in each level of search tree using the same summation. For the total tree, we have

$$\sum_{h=0}^{H-1}\sum_{d=0}^{D}\binom{h}{d} \leq \sum_{h=0}^{H-1}(h+1)^D = \sum_{h=0}^{H}(h)^D = O\left(H^{D+1}\right) = O\left(|V|^{D+1}\right). \tag{4.2}$$

On average, each edge of the search tree takes $O(|V|)$ time for a overall total of $O(|V|^{D+1}|V|) = O(n^{D+2})$ time.

Again, if we consider the degenerative case where $D = 0$, LDS takes time $O(n^2)$, the same as List Scheduling, as expected.

Limited Discrepancy Search is our first search-based scheduler. It relies on the provided heuristic. It searches around the heuristic path, bounded only by the specific discrepancy limit. Not all search algorithms rely quite so heavily on a heuristic as LDS.

4.3. Squeaky Wheel Optimization

Joslin and Clements [25] separated solution construction from heuristic selection. Following the adage, "the squeaky wheel gets the grease," Squeaky Wheel Optimization constructs a solution, analyzes the solution for problem tasks, and adjusts the heuristic ranking of the problem or "squeaky" tasks. It then builds a new solution and continues. The three major components are a simple constructor, solution analyzer, and a prioritizer.

Figure 4.7 illustrates the SWO view of scheduling. The left-hand space is the set of all solutions. (In this case, solution space is all legal instruction sequences.) The

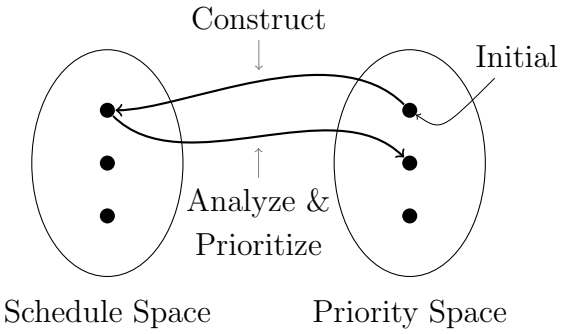


FIGURE 4.7. Squeaky Wheel Optimization's view of solution vs priority space.

right-hand space is the set of all instruction orders or List Scheduling heuristics.[5] The set of heuristics is larger than the set of solutions. There are $|V|!$ possible instruction orderings but, due to the dependencies, there are far fewer legal sequences. The three components of SWO map points from one space to points in the other.

The constructor maps an ordering from priority space to a solution in solution space. For Instruction Scheduling, orderings are List Scheduling heuristics specialized for a specific problem instance. The analyzer and prioritizer map a solution back to priority/heuristic space.

It is important to note that the constructor, List Scheduling, maps priorities *onto* solutions. That is, every solution is reachable from some priority. This is easily shown. Choose a valid solution. Since this is a total ordering of the instructions it corresponds directly to an element of priority space. Using that heuristic, List Scheduling will produce the original solution. The first instruction pulled from the ready set is the first instruction in the heuristic, the second instruction is the second, and so on. The main loop in List Scheduling chooses the next instruction in the priority.

[5]Strictly speaking, priority space is just the set of static heuristics. Heuristics like Dynamic Critical Path depend on the partially constructed solution to order tasks and comprise a much larger set.

A similar claim cannot be made with the analyzer. Since heuristic space is larger than solution space, analyze/prioritize cannot be onto.

Our SWO scheduling algorithm is shown in Figure 4.8. The prioritizer is a total order, $<$, on the instructions and a heuristic ordering. As described above, we use List Scheduling as the simple constructor. Once a full sequence is constructed, the BLAME function evaluates the sequence for problem instructions and generates a new prioritization. A new solution is generated and the search continues.

The BLAME function is responsible for evaluating a sequence and finding the squeaky wheels. It makes a single pass over the solution finding cases where one instruction is delayed by another. The delayed instruction's priority is decreased and the delaying instruction's priority is increased. This tries to move the predecessor earlier in the next sequence and the successor later.

There is some flexibility in the initial prioritization. A random order or any of the List Scheduling heuristics can be used. We evaluated both initialization approaches but found that a meaningful seed heuristic performed better.

We are left with the same questions for SWO that we had for LDS: how many sequences are considered and how long does it take to generate those solutions?

There is a simple answer to the first question. Each call to LIST-SCHEDULE generates a single sequence. This call is made *iters* times. However, there is no guarantee that each sequence is unique (consider G a linked list). So, SWO generates at most *iters* sequences.

Only the question of runtime complexity remains. Analyzing a sequence involves a single pass through the sequence and verifying latencies. This takes time $O(|V| + |\prec|) = O(n)$. Constructing a heuristic is little more than assigning a ranking to each operation, an $O(n)$ operation. As discussed above, a reasonable expected

```
SWO(G = (V, ≺), iters)
1  construct ≲ a total order on G.
2  sln ← List-Schedule(G, [], ≲)
3  for i ← 2 to iters
4       do sln ← min (sln, List-Schedule(G, [], Blame (sln))
5  return sln
```

FIGURE 4.8. Squeaky-Wheel

runtime complexity for List-Scheduling is $O(n^2)$. This clearly dominates the other two operations. Then SWO has runtime $O(iters \times n^2)$ when List Scheduling does not backtrack.

SWO builds upon List Scheduling by focusing its search in priority-space or rather heuristic space. Joslin and Clements [25] describes the behavior as:

> A point in the solution space represents a potential solution to the problem, and a corresponding point in priority space, derived by analyzing the solution, is an attempt to capture information about the structure of the search space in the vicinity of the solution.

The intuition is that small moves in priority space correspond to large, coordinated moves in solution space.

While not the view of Joslin and Clements, SWO can be thought of as a heuristic discovering algorithm where the heuristic is "tuned" for the specific problem. Contrast this to LDS which uses the heuristic to focus the search.

Not all search algorithms are built around a heuristic function or heuristic functions.

4.4. Iterative Flattening

The last search-based scheduler we consider is Iterative Flattening. It was developed by Cesta et al. [9] for multi-capacity scheduling problems, a similar but more difficult problem than Instruction Scheduling. IFlat focuses on resource conflicts rather than the heuristic or priorities of LDS and SWO. That is, rather than construct a schedule from start to finish, IFlat finds and resolves resource conflicts.

Figure 4.9 shows the IFLAT algorithm. First notice that IFLAT works with schedules and not sequences.[6] The supporting function FLATTEN initially schedules each node at its earliest start time. EST makes no attempt to satisfy resource constraints, it simply puts each instruction at the earliest point in the schedule that satisfies the dependencies in G. This will probably violate some resource constraints (i.e., dispatch limits and special purpose register usage). CONFLICTS scans the solution, collecting these conflicts.

A single conflict is selected randomly from among the set of conflicts that CONFLICTS identifies. From this conflict a Minimum Conflict Set (MCS) is chosen.

An MCS is a set of tasks, instruction in this case, that violates some resource limit. Further, the MCS is constructed such that an edge added between any two instructions resolves the conflict. For our formulation of Instruction Scheduling there are only two resources to consider: dispatch limit and special purpose registers.

The dispatch resource, P, limits the number of instructions that can be started at any point in the schedule. Figure 4.10a shows a partial schedule with five instructions scheduled in the second clock cycle. Assuming that $P = 3$, this schedule violates the dispatch limit. Choosing any four instructions from the set gives a MCS for this

[6] We overload min further to operate on schedules.

```
IFLAT(G = (V, ≺), iters)
1  σ, G = FLATTEN(G)
2  for i ← 2 to iters
3        do σ ← min (σ, FLATTEN(RELAX(G)))
4  return σ

FLATTEN(G = (V, ≺)))
1  σ ← EST(G)
2  cfl ← CONFLICTS(σ)
3  while cfl ≠ ∅
4        do choose c ∈ cfl
5           choose mcs from c
6           G ← RESOLVE(G, mcs)
7           σ ← EST(G)
8           cfl ← CONFLICTS(σ)
9  return σ, G
```

(Continued)

FIGURE 4.9. Iterative Flattening

violation. We arbitrarily picked the top four instructions. Adding an edge between any two of these instructions resolves the conflict.

For special purpose registers, the MCS is composed of two instructions that define the same special purpose register and one of the instructions scheduled between the other defining instruction and one or more of its using instructions. This is shown in Figure 4.10b. Here the last (right most) instruction uses the definition of the first but the middle instruction defines the same special purpose register. The MCS for this resource violation is simply the two defining instructions. There are two possible resolutions. First, we could add an edge from the first instruction to the second. This would ensure that the second instruction is moved early in the schedule. Second, we could add an edge from the second instruction to the last. This would move the

EST($G = (V, \prec)$)
1 For each $v \in V$, $\sigma(v) \leftarrow v$'s earliest start time.
2 **return** σ

RELAX($G = (V, \prec)$)
1 $crt \leftarrow \{e | e \in \prec$ and on a critical path$\}$
2 $art \leftarrow \{e | e \in \prec$ and added by FLATTEN$\}$
3 $rlx \leftarrow$ a few of $(crt \cap art)$
4 $G \leftarrow (V, \setminus rlx)$

CONFLICTS(σ)
1 $disp \leftarrow \{\sigma^{-1}(c) | \sigma^{-1}(c) > P\}$
2 $spr \leftarrow \{(p, q) | f \in F \quad p, q \in \text{Define}_f, p < q \wedge \exists r \in \text{Use}_f(p), q < r\}$
3 **return** $disp \cap spr$

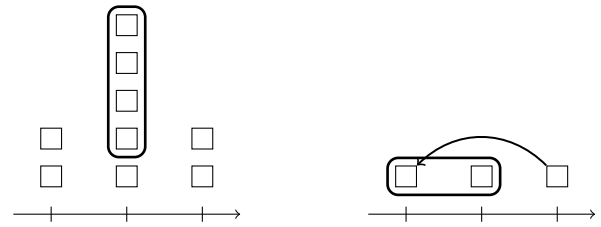
FIGURE 4.9. Iterative Flattening

middle instruction after the using instruction. Assuming that neither edge will create a cycle in the graph, either resolves the conflict.

Once the MCS is resolved, a new solution is generated by EST. More conflicts are created and resolved. Eventually, the schedule is conflict free and the solution is valid for both dependencies and resources. However, this is just a single schedule.

The search continues by removing some of the added edges. The procedure RELAX finds the critical paths in the modified G. Of the artificial edges added by FLATTEN, some lie on a critical path. RELAX removes a small number of these dependencies and FLATTEN is given another chance to generate a schedule.

Like SWO, IFlat is a nonsystematic algorithm. It generates at most *iters* candidate solutions. Since it is nonsystematic the same schedule can be generated repeatedly.



(a) An MCS for a Dispatch violation

(b) An MCS for a Special Purpose Register violation

FIGURE 4.10. Example Minimum Conflict Sets

The question of runtime complexity remains. We will begin by analyzing FLATTEN. There are four main subroutines: CONFLICTS, MCS construction, RESOLVE, and EST.

Finding conflicts is a linear pass over the proposed schedule that counts the number of instruction dispatched in each cycle and which instruction defined the special purpose registers last. CONFLICTS has runtime $O(|V|)$. MCS construction is, in general, NP-Complete [9]. However, in this case, both types of conflicts allow easy MCS construction; choosing $P + 1$ operations for dispatch limit violations or choosing two defining instructions for special purpose register conflicts. We will call MCS construction $O(P)$ or $O(1)$ since P is a fixed parameter of the targeted hardware. Finally, EST constructs an ordered schedule with each operation at its earliest start time. This is isomorphic to longest-path finding, an $O(n)$ operation. The real question is how many conflicts there can be?

FLATTEN's main loop continues until there are no more conflicts. There are two types of conflicts that interest us: dispatch limits, and special purpose registers. Resolving either type of conflict introduces a new edge into the graph. There can be at most $\binom{|V|}{2}$ edges inserted before a cycle is introduced. This puts an upper bound on the number of iterations of FLATTEN's loop at $O(|V|^2) = O(n^2)$. Given that

Algorithm	Schedules	Expected Complexity
List-Scheduling	1	$O\left(n^2\right)$
LDS	$O(\|V\|^D)$	$O\left(n^{D+2}\right)$
SWO	*iters*	$O\left(iters \times n^2\right)$
IFlat	*iters*	$O\left(iters \times n^3\right)$

TABLE 4.1. Summary of scheduling algorithms and their complexity. Note: $n = |G| = |V| + |\prec|$, D is the number of discrepancies, and *iters* is the fixed number of iterations. Further, these results assume no backtracking by List Scheduling.

the longest running component of the algorithm is $O(n)$, flatten has worst-case complexity $O(n^3)$.

The runtime complexity of IFlat is composed of Flatten and Relax. Relax must compute the critical path and remove a small number of edges from G. This time is dominated by the critical path calculation, with a known time-complexity of $O(n)$. Given that IFlat is dominated by Flatten, IFlat's runtime complexity is $O(iters \times n^3)$.

4.5. Summary

This chapter introduced and analyzed the four scheduling algorithms: List Scheduling, Limited Discrepancy Search, Squeaky Wheel Optimization, and Iterative Flattening. These results are summarized in Table 4.1. In each case, the search-based algorithms are more costly, from a runtime perspective, than List Scheduling. However, each algorithm evaluates more than the single solution that List Scheduling generates and should produce better solutions.

CHAPTER V

EXPERIMENTAL RESULTS

In the previous two chapters we formally introduced instruction scheduling as a resource constrained scheduling problem and we proved these problems NP-Complete. We described four different instruction scheduling algorithms and gave an analysis of both runtime complexity and number of solutions considered. This presentation is theoretically sound, but without a experimental evaluation, it would be incomplete.

In this chapter we evaluate the scheduling algorithms on the SPEC CPU2006 benchmark suite. The evaluation criteria that we use are schedule length, scheduling time, and register pressure. Additionally, we analyze instruction scheduling as a multi-objective optimization problem and we show that no one scheduler is universally superior to the others.

The following section introduces the evaluation metrics, benchmark problems, and experimental platform. An outline of algorithm performance is given in Section 5.1. Section 5.2 presents the success rates for the evaluated schedulers. Section 5.3 evaluates the lengths of the generated schedules. Section 5.4 compares the schedulers based on runtime. Section 5.5 presents a surprising comparison of register pressures of the generated schedules. Section 5.6 considers instruction scheduling as a multi-objective optimization problem and combines the results of the previous few sections. We consider simulated execution in Section 5.7. Section 5.8 explores a possible strategy that balances the costs and benefits of search-based scheduling in a just-in-time context. Finally, Section 5.9 provides concluding discussion and summarizes the material presented in this chapter.

5.1. Measuring Performance

The Standard Performance Evaluation Corporation (SPEC) maintains a collection of benchmark suites. Each suite targets a specific computer usage model (e.g., power usage and email servers). The SPEC CPU2006 benchmark suite is a collection of applications that represent typical, CPU-intensive computer usage. This suite includes common applications like GCC and Perl as well as more scientific programs like linear programming and laminar viscous flow modeling.

From this benchmark suite, we extracted 734,054 scheduling problems using LLVM 2.7. These scheduling problems target the Intel Core 2 CPU. That is, we use the Intel x86_64 instruction set and the Core 2 instruction latencies. We applied each of the four scheduling algorithms to these problems with various search parameters (i.e., iterations and discrepancies). These experiments were run on a 2.66 GHz Intel Core 2 Duo (6700) CPU running Linux 2.6.32.

The question of how to measure an instruction scheduler's performance is complicated. On the one hand, the length of the generated schedule is important; after all, we want efficient generated programs. On the other hand, we want the scheduler itself to execute quickly. A long running scheduler adds to the overall compilation time. However, these are not the only measures of scheduling performance.

Not all basic blocks are schedulable. The interactions between special purpose registers can create implied cycles in the dependency graph. Even when the problem does have a valid schedule, the constraints imposed by special purpose registers can make it difficult to construct a valid schedule. This is especially true for the simple, non-backtracking List Scheduling. So then, we must compare the algorithms based on the percentage of solved scheduling problems.

There are several reasons that an algorithm may fail to generate a schedule. Some problems have no valid schedules. Others are simply too difficult for a simple scheduler to find. The non-backtracking form of List Scheduling, for example, can get "stuck" with a ready set full of unschedulable instructions. Contrast this with the search-based approach which will eventually find a valid schedule but may, due to time constraints, give up.

Failing to generate a schedule for a basic block, then, can happen for one of three reasons: the problem has no valid solution, the algorithm reached a terminal state without a valid schedule, or the algorithm timed out.

The three measures described above focus on the effectiveness of a scheduling algorithm in isolation, but instruction scheduling is not the last compilation step. After a valid schedule is generated, a separate register allocation pass assigns physical, hardware registers to the data defined within the problem. These general purpose registers are another limited resources. Typically, the CPU has only a few general purpose registers. When the data demands of a schedule exceeded the available physical registers, some data must be moved to memory. The allocator will insert move instructions to copy data into memory to free up some registers for another data assignment.

The same scheduling problem can have schedules that demand many physical registers or few. The term register pressure describes the number of physical registers needed to successfully allocate the schedule without spilling data to memory. High pressure means the schedule needs more registers. We compare the scheduling algorithms by measuring and comparing the register pressure of each generated schedule.

5.2. Success Rates

An instruction scheduler can fail to create a schedule for three reasons. Some scheduling problems are simply not schedulable. Of the 734,054 basic blocks in the benchmark suite, 2,107 have cycles in their dependency graphs implied by special purpose usage.[1] The remaining 731,947 scheduling problems have valid solutions. These are the scheduling problems of interest for the remainder of this section.

The second failure mode is unique to the non-backtracking List Scheduling. List Scheduling builds solutions one instruction at a time. Without the ability to retract a decision, the algorithm can get caught with a non-empty ready set without any valid instruction to schedule. That is, each instruction in the ready set violates a special purpose register constraint. Of the 731,947 schedulable problems, the simple non-backtracking List Scheduling finds solutions for 720,306 of the problem and the non-backtracking List Scheduling fails to find a schedule for 11,641 basic blocks.[2]

The final failure condition is timeout. Scheduling problems range in difficulty from the easy to the very hard. The most challenging problem instances require several seconds to schedule. The scheduler is given a maximum time limit. Problems that take longer than this limit are unscheduled and the algorithm is considered to have failed.

Since timeout is a scheduling parameter, we are left without a single value, but a range for each algorithm. Figure 5.1 shows the success rate for List Scheduling for timeouts as high as 60 seconds. Increasing the timeout can only lead to more

[1] Unschedulable basic blocks are not a compiler error. In practical schedulers, the dependency graph is modified to break implied cycles. These modifications define a separate search problem and are outside of the scope of this work.

[2] Just like unschedulable basic blocks, this is not a compiler error. When trapped by early mistakes, practical schedulers modify the dependency graphs.

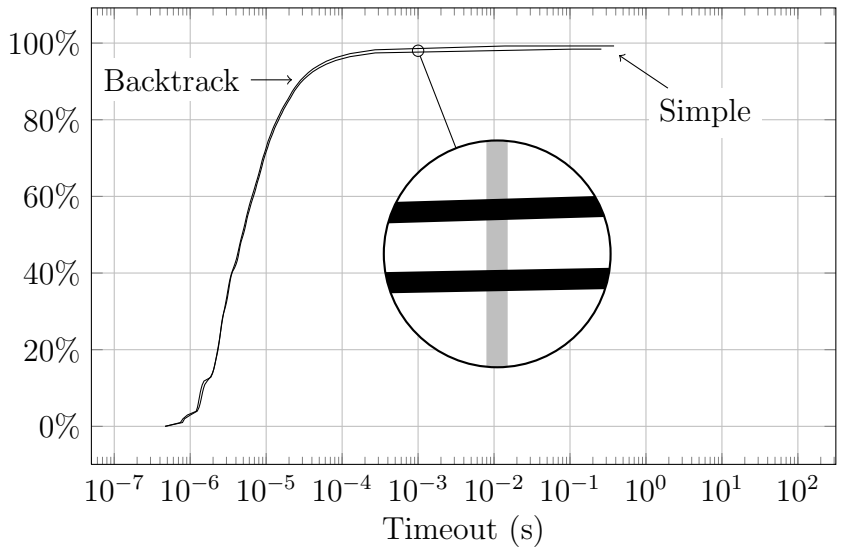


FIGURE 5.1. List Scheduling success rate by timeout limits

successfully scheduled problems—that is, the curves are non-decreasing. While there appears to be a single line, the plot actually contains both the simple non-backtracking List Scheduling as well as with backtracking. The addition of backtracking causes little impact on the scheduling time but succeeds 0.62% more often. In both cases, half of the scheduling problems succeed in less that five microseconds. At the upper extreme, List Scheduling succeeds in 98.7% with 0.3 second timeout for the non-backtracking variant and 99.4% with 0.5 seconds and backtracking.

List Scheduling constructs a single schedule. In this case, judging success or failure is well defined. Except for the degenerative cases (e.g., LDS with zero discrepancies) the meanings of success and failure for the search-based algorithms are less clear. Consider LDS with one discrepancy on a very large scheduling problem. If given enough time, LDS will construct $O(|V|)$ schedules. If the algorithm times out partway through the search, having considered fewer than this limit, did LDS fail? The most liberal definition of success would say that if LDS builds a single schedule,

then it succeeded. Conversely, the conservative definition would say that LDS failed unless it runs to completion.[3]

The three search-based scheduling algorithms have tuning parameters: discrepancies for LDS and iteration count for SW and IFlat. In each case, larger values of this parameter subsume the search performed by the lower value. For example, LDS with $D > 0$ discrepancies considers the same potential solutions as LDS with $D - 1$ discrepancies, then builds more schedules. The same is true for iteration counts for SW and IFlat. This subsumption property allows us to address both the conservative and liberal definitions of success.

Consider the results for LDS shown in Figure 5.2. The simple List Scheduling curve is plotted for comparison. Clearly LDS takes significantly more time that List Scheduling. For almost any time-out where List Scheduling succeeds, LDS fails even with no discrepancies (there is a small overlap in the 10^{-5} to 10^{-4} timeout range). Looking at a 10^{-3} second timeout, LDS-2, the two-discrepancy scheduler, conservatively succeeds on just 58% of the problems. If we use the liberal definition of success, then LDS-2 succeeds on 91% of the problems, the same as LDS-0. For the remainder of this chapter, we will use the conservative definition of success.

Given zero, one, and two discrepancies, LDS successfully schedules 99%, 95% and 78% of the problems (respectively) with a timeout of 60 seconds. Approximately half of the problems are scheduled with timeouts of 24, 25, and 450 microseconds for LDS-0, LDS-1, and LDS-2 respectively.

The success curves for LDS roughly match those of List Scheduling shifted down and right. Considering that LDS is an adaptation of List Scheduling, the similarities of these curves are expected.

[3]This is where a practical compiler would use the exploration order described by Harvey and Ginsberg [22] (see the discussion on page 51).

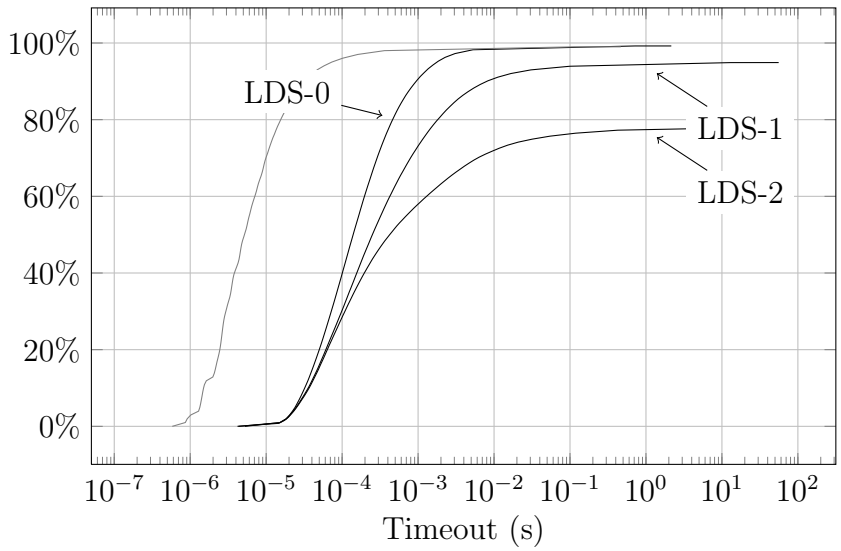


FIGURE 5.2. LDS scheduling success rate by timeout limit

Like LDS, SW is built upon List Scheduling and the success curves for SW, shown in Figure 5.3, have the same, log-sigmoid shape. The success curves shown are for SW with one to five iterations. Like the curves for LDS, the List Scheduling curve is added to ease comparison. SW succeeds for roughly half of the scheduling problems in 20 to 70 microseconds. With the full 60 second timeout, SW eventually succeeds in 99.3% of the problems for all five iteration counts.

Success rates for IFlat are a bit more interesting. Figure 5.4 shows the success rates for IFlat with one to ten iteration counts. The one-iteration curve is similar to List Scheduling and SW. However, once relaxations are applied, the problems divide into two sets. A little more than half of the problems succeed with the same timeout regardless of iterations. This implies that relaxation and flattening are fast and expose few if any new conflicts. The problems that remain take more time to relax and re-flatten. The assumption here is that the relaxation step exposes many new conflicts. In either case, IFlat successfully schedules half of the scheduling problems with a 14 microsecond timeout for one iteration and 17 microseconds for two to ten

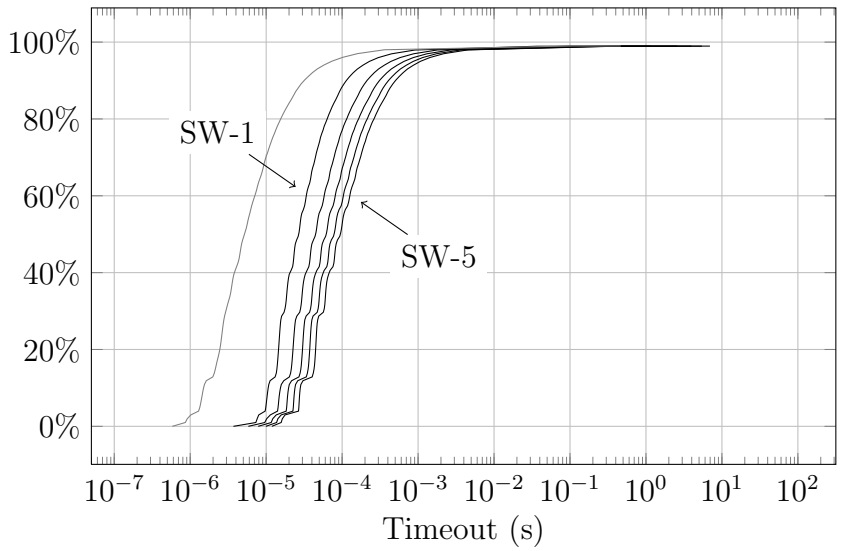


FIGURE 5.3. SW scheduling success rate by timeout limit

iterations. With the full 60 second timeout, IFlat successfully schedules 97% of the scheduling problems. It achieves all of this in the first second of runtime.

With the possible exception of LDS and depending on the definition of success, all of the search-based schedulers successfully schedule nearly all of the benchmark scheduling problems. However, simply generating a schedule is not enough. We are also interested in the quality of the generated schedules.

5.3. Schedule Length

Ultimately, the goal of search-based instruction scheduling is to generate shorter schedules. We measured the length of schedules generated by the various algorithms and compare the results against the schedules generated by List Scheduling. With a 60-second timeout, we compared the length of the generated schedules by each search-based algorithm with various search parameters (e.g., iterations and discrepancy counts) against the schedules generated by the simple List Scheduler.

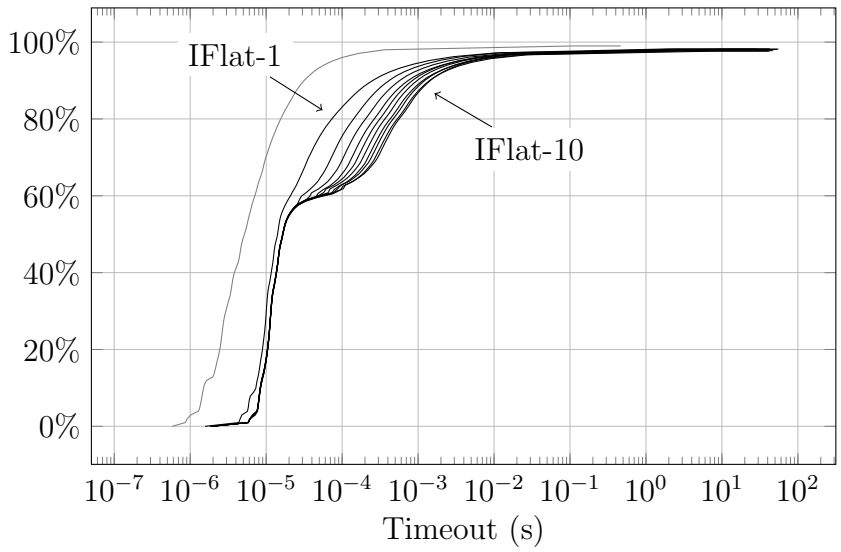


FIGURE 5.4. IFlat scheduling success rate by timeout limit

We limit the reported results to scheduling problems for which the search-based algorithm and List Scheduling succeeded in generating valid schedules. We measure schedule lengths by computing the makespan or total execution time of each problem assuming x86_64 Core2 instruction latencies, special purpose register behavior and number of pipelines.

Figure 5.5 compares the schedule lengths generated by LDS against those generated by the simple List Scheduling algorithm. We added the identity function ($y = x$) as a dashed line to aid in the comparison. Each problem is plotted with the List Scheduling schedule length as the x-coordinate and the LDS schedule length as the y-coordinate. Points below the diagonal indicate that LDS generates shorter schedules. Points above the line indicate the LDS's schedulers are longer.

Unsurprisingly, LDS-0 is essentially the same scheduler as List Scheduling and the results in the first plot are generally on top of the diagonal line. The differences shown in the first plot are simply the variation in heuristic tie-breaking. As the number of discrepancies increases, more of the scheduling problems move below the

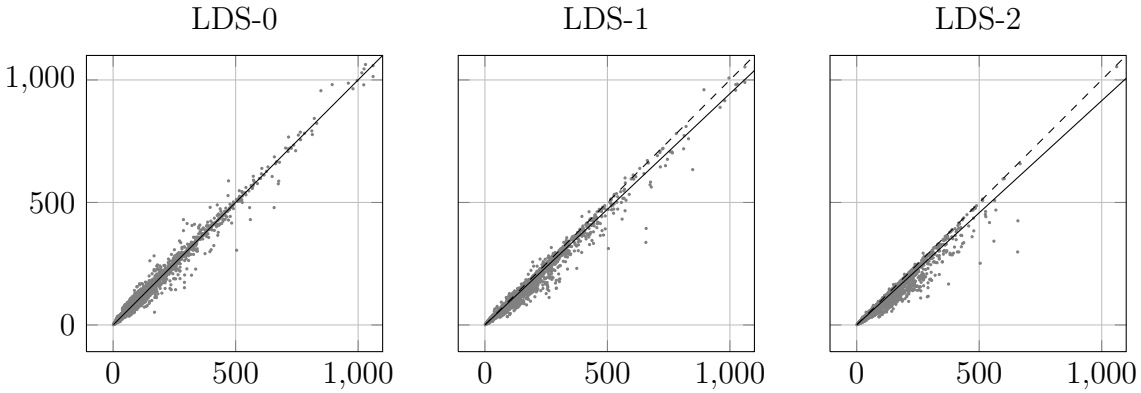


FIGURE 5.5. Schedule length generated by List Scheduling (x-axis) vs LDS (y-axis) for zero, one, and two discrepancies.

identity function, represented by the dashed-diagonal line. In other words, LDS-1 shows an improvement over LDS-0, and LDS-2 shows even greater improvement.

The solid lines in Figure 5.5 are the best-fit, Deming regression lines. For the three plots, these lines are

$$f_0(x) = (0.014 \pm 0.017) + x\,(0.998 \pm 0.003)$$

$$f_1(x) = (0.177 \pm 0.016) + x\,(0.944 \pm 0.003)$$

$$f_2(x) = (0.282 \pm 0.017) + x\,(0.915 \pm 0.003)$$

for LDS-0, LDS-1, and LDS-2 respectively. In all three cases, slope and intercept ranges are for the 99.9% confidence interval. By these results, LDS-0 is roughly equivalent to List Scheduling. This is an unsurprising result. Except for heuristic tie-breaking, LDS-0 follows the same scheduling steps as List Scheduling.

The slope of the regression lines indicate that LDS-1 and LDS-2 both perform better on larger scheduling problems than smaller problems. Schedule length is roughly proportional to the number of instructions and LDS explores approximately $O(|V|^D)$ schedules. With larger scheduling problems, LDS generates more schedules

and chooses the best schedule from among the larger candidate pool. The consequence is that LDS produces better schedules. LDS is the only search-based scheduler evaluated with this non-linear exploration of scheduling space. Both SW and IFlat generate one schedule per iteration.

Figure 5.6 shows the schedule length comparison plot for SW. The best-fit regression lines added. The regression lines are

$$f_1(x) = (0.030 \pm 0.031) + x\,(0.995 \pm 0.005)$$
$$f_2(x) = (0.021 \pm 0.024) + x\,(0.993 \pm 0.004)$$
$$f_5(x) = (0.002 \pm 0.017) + x\,(0.996 \pm 0.003)$$

for one, two, and five iterations respectively. Since SW-1 constructs a single schedule using the seed heuristic ordering, we expect the regression line to approximate $y = x$. This is what we see given the 99.9% confidence. Once the blame-prioritize-rebuild cycle executes SW-2 and above, we see a slight decrease in the slope of the regression line. However, the slopes for both SW-2 and SW-5 are below the diagonal but just slightly.

Both LDS and SW construct schedules similarly to List Scheduling. This is apparent from the previous two figures. IFlat builds schedules in a completely different way. This is clearly shown in Figure 5.7 by the larger schedule length variation. The best-fit regression lines are

$$f_1(x) = (1.337 \pm 0.198) + x\,(0.739 \pm 0.031)$$
$$f_5(x) = (1.313 \pm 0.196) + x\,(0.731 \pm 0.031)$$
$$f_{10}(x) = (1.298 \pm 0.187) + x\,(0.730 \pm 0.030)\,.$$

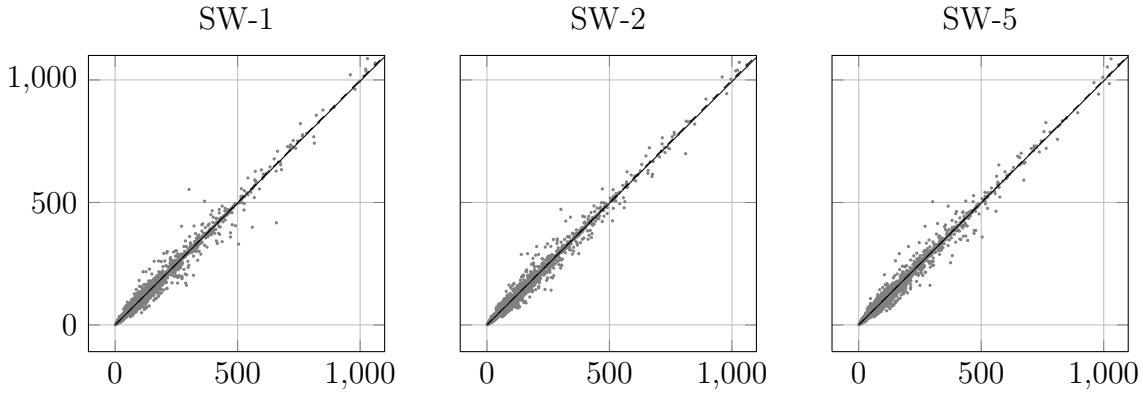


FIGURE 5.6. Schedule length generated by List Scheduling (x-axis) vs SW (y-axis) for one, two, and five iterations.

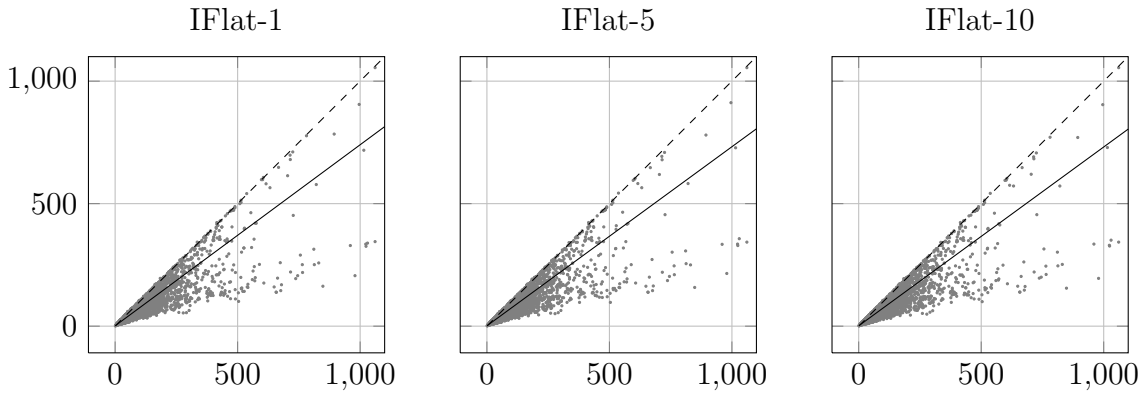


FIGURE 5.7. Schedule length generated by List Scheduling (x-axis) vs IFlat (y-axis) for one, five, and ten iterations.

Again, these are the 99.9% confidence intervals. Note that the confidence intervals for IFlat are ten times larger than the other schedulers.

Clearly IFlat performs better than List Scheduling, LDS, and SW when considering just schedule length. It is also interesting to note that the slope of the regression lines (except LDS-0 and SW-1) are less than one. These slopes indicate that the search-based scheduling algorithms perform better on larger problems. These larger problems may provide more opportunities for the search-based algorithms to improve the generated schedules.

Schedule length is critical, but not the only metric by which a scheduling algorithm is judged. We must consider the time it takes to schedule each problem as well.

5.4. Scheduling Time

While schedule length is ultimately the goal of search-based scheduling, it is not the only criterion by which scheduling algorithms are measured. Total compilation time is also a concern, since scheduling time contributes to compilation time. In this section we compare the scheduling time of three search-based schedulers against the simple, non-backtracking List Schedule.

Scheduling time is measured against the system's monotonic clock. This timer reports the wall-clock time, not the process's CPU usage. CPU utilization timers proved to be unstable when measuring instruction scheduling. CPU usage timers are updated at the end of the OS's quantum or CPU burst. Easy scheduling problems are scheduled entirely within a single CPU burst. This causes the OS to estimate the CPU usage for the entire problem and this estimate is too rough for these experiments. To mitigate system load effects on the monotonic clock, these experiments were executed on a dedicated multi-core, system and no unnecessary services were running.

For each experiment, we scheduled each problem using the simple List Scheduling time as the x-coordinate and the search-based algorithm as the y-coordinate. Except for the discrepancy count, nothing in the development of these algorithms involves exponential relationships. Despite this, we have plotted these values on a log-scale to provide a greater range of values and better visualization of the relationships between the timing data.

Figure 5.8 compares LDS against List Scheduling. We added a best-fit regression line in the logarithmic domain. These regression lines translate to the following curves:

$$f_0(x) = 0.841 \times x^{0.829}$$

$$f_1(x) = 3670 \times x^{1.469}$$

$$f_2(x) = 5\,070\,000 \times x^{2.039}$$

for LDS-0, LDS-1, and LDS-2 respectively.

These plots show that in all three cases, LDS takes more time than List Scheduling. The curve for LDS-0 is nearly linear. Since LDS-0 is essentially List Scheduling with the search overhead, this nearly linear relationship is expected.

Adding a single discrepancy increases the number of generated schedules from 1 to $O(|V|)$ and the runtime complexity increases form $O(n^2)$ to $O(n^3)$. We expect the exponent for f_1 to be at most 1.829, one more than the exponent of f_0. We find 1.469, a little less than expected. This suggests that while LDS-1 could explore $|V|$ schedules, dependencies within the graph prevent LDS-1 from reaching this number.

With two discrepancies, LDS explores up to $O(|V|^2)$ schedules. Just like the step from LDS-0 to LDS-1, we expect a similar increase in the exponent in f_2. We find 2.039, which is a bit less than expected.

Like LDS, SW is built on List Scheduling and we expect the runtime behavior of SW to be similar. Figure 5.9 contains the scheduling time comparison. The three

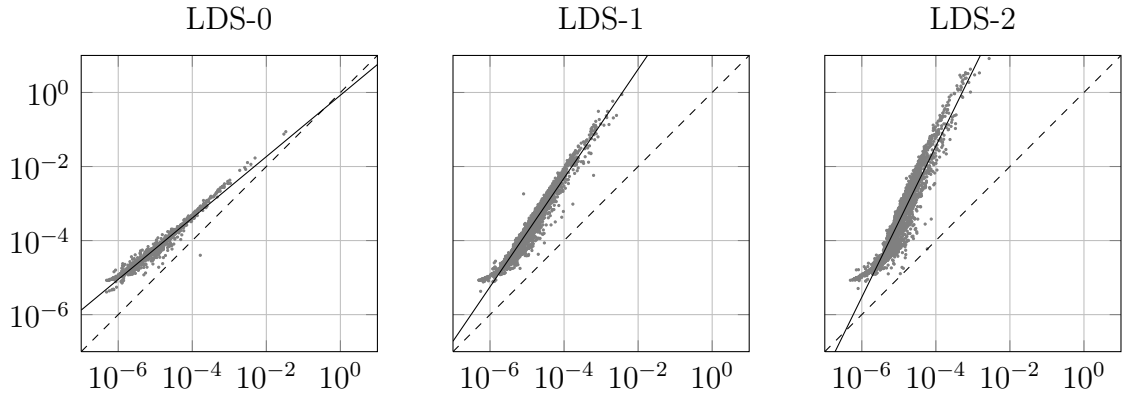


FIGURE 5.8. Scheduling time use by List Scheduling (x-axis) vs LDS (y-axis) for zero, one, and two discrepancies.

best-fit regression curves are

$$f_1(x) = 0.427 \times x^{0.791}$$

$$f_2(x) = 1.657 \times x^{0.861}$$

$$f_5(x) = 8.224 \times x^{0.932}$$

for one, two, and five iterations.

In all three cases, these curves will eventually cross the diagonal line. Since SW uses List Scheduling to construct each candidate schedules, this would certainly not be the case and the runtime of SW will always be above that of List Scheduling. These curves show that the blame-prioritize-rebuild cycle overhead is mitigated when applied to longer and more difficult to schedule problems.

Figure 5.10 shows the scheduling time comparison for IFlat against List Scheduling for one, five, and ten iterations. The best-fit regression curves for these

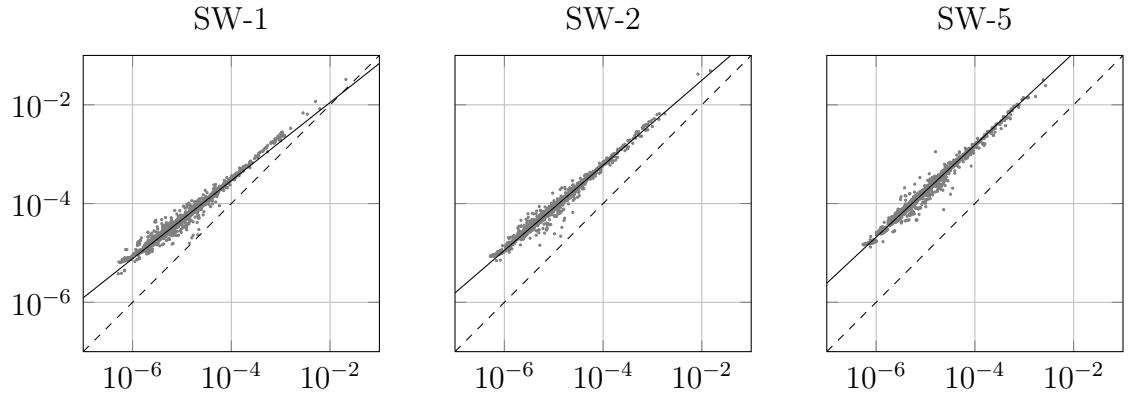


FIGURE 5.9. Scheduling time use by List Scheduling (x-axis) vs SW (y-axis) for one, two, and five iterations.

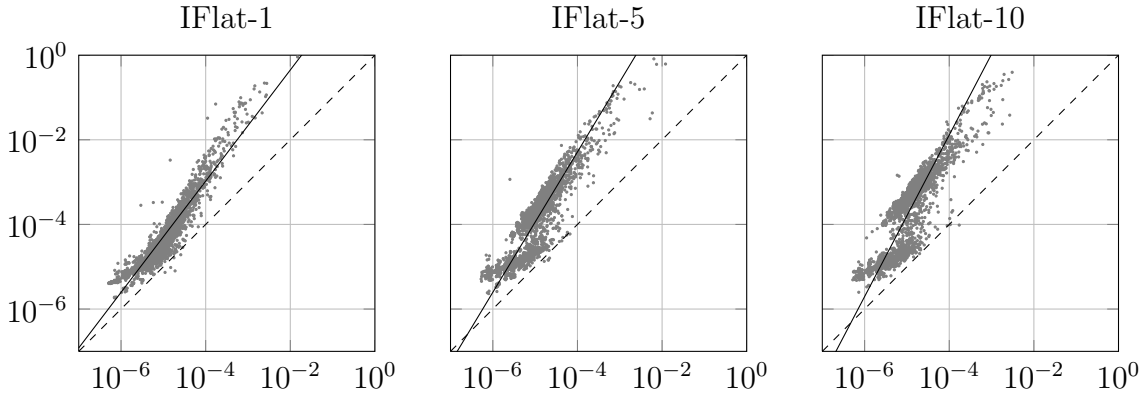


FIGURE 5.10. Schedule time use by List Scheduling (x-axis) vs IFlat (y-axis) for one, five, and ten iterations.

plots are

$$f_1(x) = 189 * x^{1.314}$$

$$f_5(x) = 22140 * x^{1.658}$$

$$f_{10}(x) = 565908 * x^{1.910}$$

for the one, five, and ten-iteration experiments respectively. IFlat takes more time to schedule than List Scheduling and generally follows the same increases. Unlike LDS and SW, IFlat appears to behave in two different ways.

The one-iteration case appears to partition the scheduling problems into two clusters. This trend continues with five iterations. With ten iterations, the two clusters are clear. DBSCAN, a clustering algorithm from data mining [15], finds these two major and several smaller clusters in the IFlat-10 data. The lower cluster represents about two-thirds of the experiments and the upper cluster about one-third.

We fit curves to the two clusters that DBSCAN identified:

$$f_{lower}(x) = 0.00811 * x^{0.512}$$

$$f_{upper}(x) = 1054 * x^{1.299}.$$

The exponent value of the lower cluster approaches the diagonal in much the same way as the curves for SW-0 and SW-2. From this we draw the same conclusion: the search overhead is mitigated by larger problems. The upper cluster is a different story.

The upper curve's exponent is greater than one and the regression moves away from the diagonal as the problems get harder. This supports the assumptions that we derived from the curves in Figure 5.4 that, for some problems, relaxation exposes several new conflicts.

These scheduling-times support the runtime complexity analysis of the previous chapter. The search-based algorithms take uniformly more time that list scheduling, but tend to produce shorter schedules. Scheduling success is measured by more than just length and time. How the schedules interact with the rest of the compilation processes must also be considered.

5.5. Register Pressure

While register allocation is typically a separate compiler task usually performed after scheduling, the two tasks are not fully separable. Shorter schedules tend to require more registers. These schedules tend to exploit more instruction level parallelism—that is, have more instructions executing simultaneously. More simultaneous operations mean more data on which to operate.

Register pressure is a function of the instruction sequences, $<$. At any point in the sequence, the instantaneous register pressure is the number of instructions sequenced before that point that define data used by instructions after that point. Formally, we can partition V at $v \in V$ into two sets:

$$P_v = \{p \in V | p < v\}$$
$$S_v = \{s \in V | s = v \vee v < s\}.$$

The instantaneous register pressures at this point is

$$R(v) = \Big| \{d | d \in P_v \wedge \exists s \in S_v \text{s.t.} (d, s) \in E \wedge l(d, s) > 0\} \Big|,$$

and the register pressure for the entire scheduling problem is

$$R(V) = \max_{v \in V} R(v).$$

Strictly speaking, this is a lower-bound on the actual register pressure. We exclude instructions that produce two values. For example, the Intel x86 integer-divide instruction calculates the integer quotient and the remainder simultaneously.

For allocation purposes, this under-estimate would be significant; for scheduling method comparison, this measure is sufficient.

When comparing two different schedules, our search-based implementation always chooses the schedule with the shorter length. However, given two schedules of the same length, the algorithms prefer the schedule with the lowest register pressure. In other words, there is a slight bias toward lower register pressures but not at the expense of longer schedules.

The register pressure comparison of search-based schedules against List Scheduling is shown in Figure 5.11. We focus solely on the more extreme versions of the schedulers: LDS-2, SW-5, and IFlat-10. The following Deming regressions lines are added to the plots:

$$f_{LDS}(x) = -0.300 + x * 0.959$$

$$f_{SW}(x) = 0.385 + x * 0.993$$

$$f_{IFlat}(x) = 5.118 + x * 1.112$$

These curves show that the register pressure of LDS tends to be lower than that of List Scheduling. Further, the slope of the regression line is slightly less than unity. This suggests that LDS can make better use of the available registers as the scheduling problems become more complicated. However, LDS failed to schedule any of the problems with register pressures above 80.

While LDS makes some headway with register pressure, SW performs about the same as List Scheduling and IFlat performs rather poorly. In nearly every problem, schedules generated by IFlat require significantly more registers than List Scheduling.

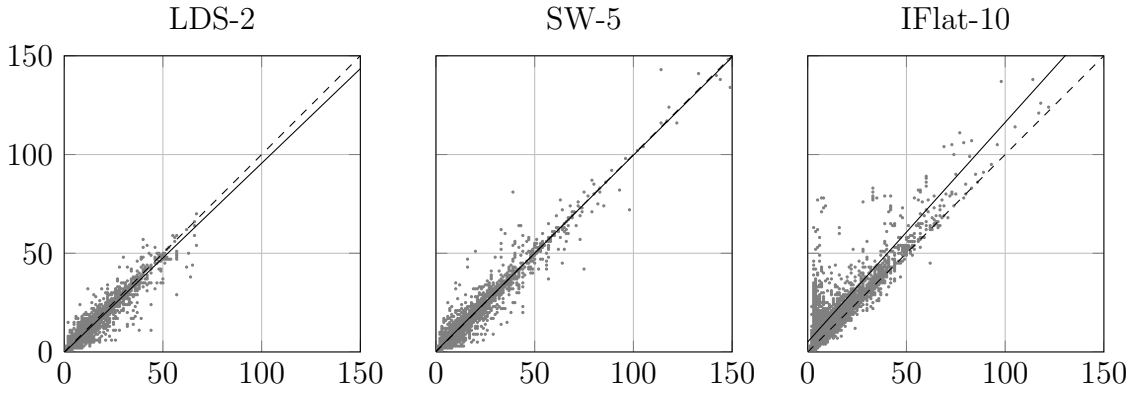


FIGURE 5.11. Register pressure comparison of LDS-2, SW-5, and IFlat-10 against List Scheduling

So far we have considered the three major metrics (time, length, and pressure) independently. This is a myopic view of instruction scheduling. If the domain demands fast scheduling, List Scheduling should be used; if it demands short schedules, IFlat is preferred. A more pragmatic approach balances the strengths and weaknesses of the various algorithms.

5.6. Multi-Objective Comparison

Schedule performance is more complicated than a one-dimensional comparison. Comparing generated schedule lengths without considering the scheduling time or register pressure only tells part of the story. Fundamentally, instruction scheduling is a multi-objective optimization problem.

Multi-objective optimization accepts that there may not be one optimum solution, but an entire set of optimal solutions. This set is called Pareto optimal. Members of the Pareto optimal set are superior to all other solutions on at least one objective function.

For Instruction Scheduling, there are three objectives: schedule length, scheduling time, and register pressure. This creates a 3-D objective space with

the origin at the ideal point (short schedules, generated quickly, with low register pressure). Rather than work in three dimensions, we will analyze the results using two objectives at a time staring with length and time.

In order to compare the three algorithms, we limited the results to the 685,600 scheduling problems that were successfully scheduled by all experimental runs. This represents a little more than 93% of the schedulable problems.

For each scheduling algorithm, we calculated the mean schedule length and mean scheduling time. These results are shown in Figure 5.12. The y-axis is plotted on a log-scale to compress the scheduling times vertically. Lines were added to visually connect the same scheduling algorithm with different parameters; this should not suggest that embedded continua exists.

As a two-objective problem, List Scheduling, SW-1, SW-2, and all of the IFlat schedulers are among the Pareto optimal solutions. The LDS solutions are excluded from the set because they are above and to the right of some other solutions. For example, IFlat-1 produces shorter schedules in less time than either LDS-1 or LDS-2, thus LDS-1 and LDS-2 are excluded.

Scheduling time and schedule length are the two primary objects to consider, but register pressure does play a significant roll in code generation. This is especially true for platforms with few general purpose registers. This interaction between scheduling and register allocation is well studied. Bradlee et al. [6] found the relationship between length and pressure follows

$$c + d/x^2$$

where c and d are constants set for the specific scheduling problem. To compare scheduling algorithms, we consider the trend of the mean register pressures.

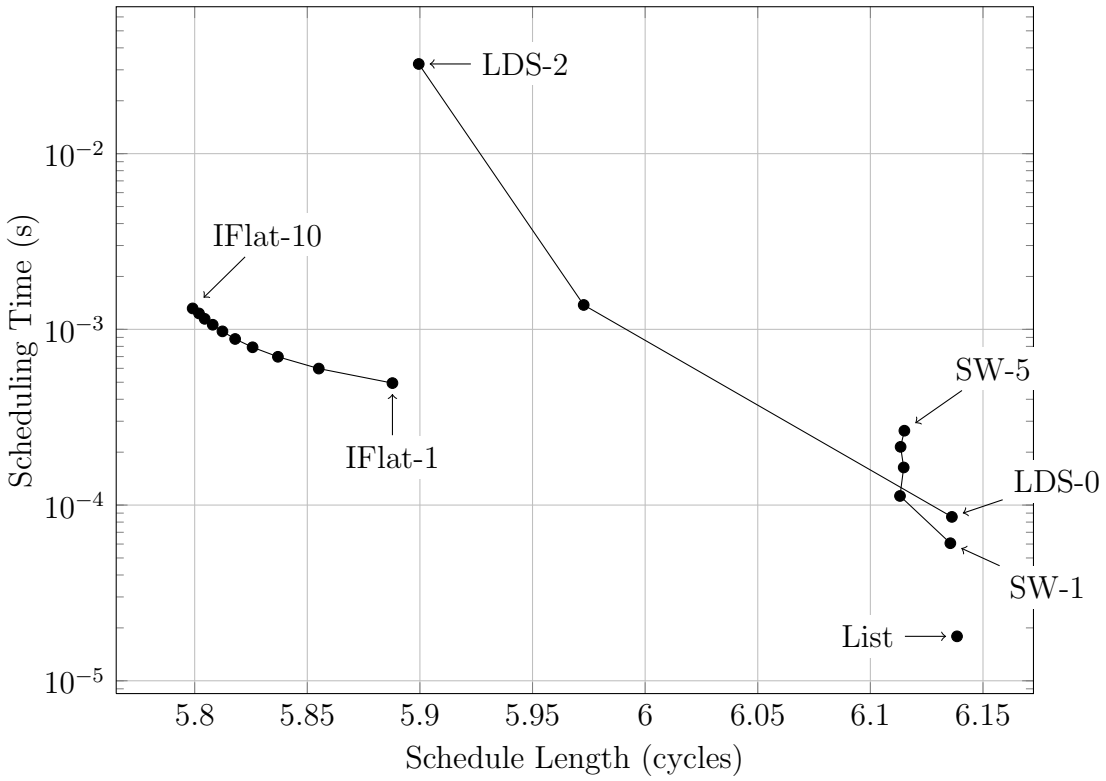


FIGURE 5.12. Schedulers plotted by scheduling time vs schedule length

We compared mean schedule length to mean register pressure and the results are shown in Figure 5.13. A new Pareto optimal set is created when just focusing on these two objectives. Here the optimal set includes LDS-2, and all of the IFlat schedulers.

It is interesting to notice that the pressure trends for LDS and SW are contrary to the results of Bradlee et al. [6]. Rather than tending to increase with shorter schedules, the register pressure decreases. One possible explanation is that the comparison between two candidate schedules uses register pressure as a secondary component. In other words, the search algorithms always favor shorter schedules, but between two schedules of the same length, the one with the lower pressure is chosen. Considering that LDS-2 considers many candidate schedules, this slight preference in selection appears to make a significant difference in the results.

When considering all three objective functions, we are left with a rather large optimal set that includes all of the IFlat schedulers, List Scheduler, SW-1, SW-2, LDS-1, and LDS-2.

In the end, we are comparing search-based scheduling to List Scheduling. These data are shown in Table 5.1 as ratios compared to List Scheduling. We see that, on average, LDS-2 takes 1810 times the scheduling time compared with List Scheduling, but LDS-2 produces schedules that are nearly 4% shorter than List Scheduling and that require 4% fewer registers.

These results summarize all of the scheduling problems on which all of the schedulers succeeded. In Section 5.3 we found that the search-based schedulers perform better on more difficult problems. Table 5.2 contains the same results for the 337,840 problems with more than 5 instructions. Here we see that IFlat-10 generates schedules that are bout 7.2% shorter than List Scheduling.

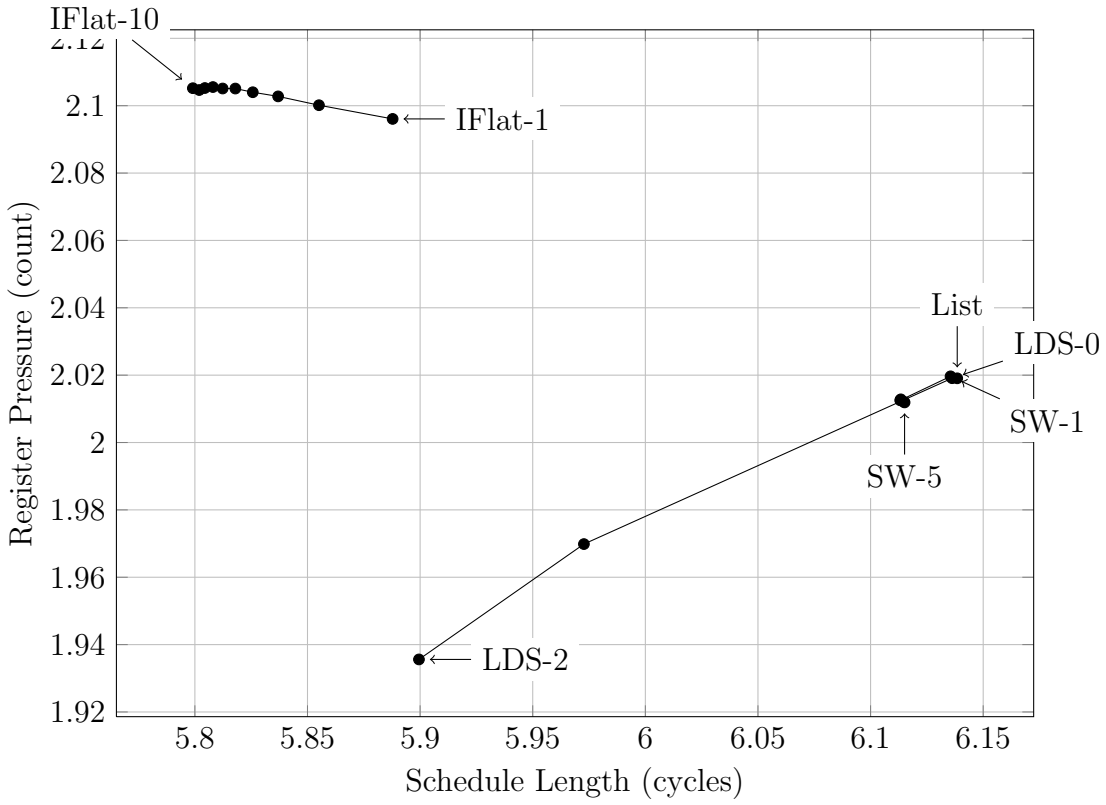


FIGURE 5.13. Register pressure (x-axis) vs schedule length (y-axis)

5.7. Simulated Execution

Comparing scheduling algorithms based on per-problem results is valuable, but incomplete. As users of compiler technology, we are more interested in the effect these schedulers have on the generated program. Stated another way, the presentation above assumes that basic blocks are executed with equal probability. This is certainly not the case.

We modified LLVM to insert instrumentation that logs invocation counts in to the scheduling problems . This resulted in 3.4 trillion execution events covering about 64 thousand basic blocks.

Method	Time/List	Length/List	Registers/List
IFlat-1	27.6	0.959	1.038
IFlat-5	49.2	0.948	1.043
IFlat-10	73.5	0.945	1.043
LDS-0	4.78	1.000	1.000
LDS-1	76.9	0.973	0.976
LDS-2	1810	0.961	0.959
SW-1	3.39	1.000	1.000
SW-2	6.29	0.996	0.997
SW-5	14.8	0.996	0.996

TABLE 5.1. Scheduler Performances Proportional to List Scheduling.

Method	Time/List	Length/List	Registers/List
IFlat-1	29.6	0.947	1.049
IFlat-5	52.8	0.932	1.054
IFlat-10	78.9	0.928	1.054
LDS-0	4.57	1.000	1.000
LDS-1	82.4	0.965	0.972
LDS-2	1950	0.950	0.952
SW-1	3.17	0.999	1.000
SW-2	6.03	0.995	0.996
SW-5	14.5	0.995	0.996

TABLE 5.2. Scheduler Performances Proportional to List Scheduling.

We simulated execution by multiplying the invocation counts of each basic block by the schedule length generated by the various schedulers. These results are shown in Table 5.3 relative to List Scheduling. The 99.9% confidence interval is ± 0.01.

Here we see that the larger, more difficult to schedule basic blocks tend to be executed more frequently than smaller blocks. This magnifies the effectiveness of the search-based schedulers. Except for SW and LDS-0, the search based schedulers significantly improve the performance of the generated program. However, these blocks have not been allocated. The 15% speedup indicated for IFlat-10 would only be realized on hardware with many general purpose registers.

Method	Time / List
IFlat-1	0.890
IFlat-5	0.860
IFlat-10	0.858
LDS-0	1.010
LDS-1	0.948
LDS-2	0.926
SW-1	0.998
SW-2	1.010
SW-5	1.009

TABLE 5.3. Simulated execution time relative to List Scheduling

Actually running these schedules would be the ultimate test. However, many details are masked by the processor. For example, most modern hardware contains out of order execution circuitry. This hardware feature reschedules the program at execution time, essentially correcting or improving poorly constructed schedules.

Why are we interested in better schedules in the presence of out of order circuitry? Out of order execution requires a large number of transistors and die real estate. In some cases, like small embedded processors, the chip simply cannot support this feature. More generally, if compilers generate schedules that depend less on out of order execution, then those transistors and die real estate could be used for different purposes (e.g., an additional pipeline or more L1 cache).

The purpose of this thesis is to answer the question: can search based scheduling improve compiler code generation? At this point, evaluating these schedulers in a practical compiler would add little to answering that question.

5.8. Just-In-Time Compilation

The results presented so far are most applicable to traditional, off-line compilation. Separating compile-time from run-time makes compilation speed less

critical. Even significant increases in compilation time can be justified by modest performance improvements in the generated program when that program is executed many times by many users. However, embedded or just-in-time compilers do not have this luxury.

Dynamically compiled languages like Java, Python, and JavaScript compile during execution time. Unlike off-line compilation, embedded compilers have access to invocation specific profiling data. These data include block execution frequencies. Initially the program is interpreted by the execution environment or virtual machine (VM). While interpreting the program, the VM keeps a count of each block's executions. When this count passes a threshold, the VM compiles the block. All future invocations execute the much faster compiled form. The VM amortizes the compilation costs against the performance improvement of all future invocations of the block. Ideally the overall performance effect is positive.

With the possible exception of very long-lived programs, replacing List Scheduling with any of the search-based algorithms is probably a poor choice. For example, using IFlat-10 increases compile time by 80 times while decreasing execution time by 15%. To fully amortize the increases scheduling time the program's execution time needs to increase by about 535 times. This is a poor choice For short-lived programs. However, this all-or-nothing approach is unnecessary.

Rather than use one algorithm for every block consider using two. List Scheduling is used for the less active blocks and a search-based scheduler is used for the more active blocks. Embedded compilers have access to block execution count data like those shown in Figure A.4. We must ask: can this two-scheduler approach can achieve some of the performance improvements of the pure search-based schedulers with scheduling times closer to that of List Scheduling?

To answer this question we combined List Scheduling and IFlat-10. Using the data shown in Figure A.4 we schedule the most frequently executed blocks with IFlat and the rest with List Scheduling. Scheduling time is reported proportionally to a pure List Scheduling approach. Execution is measured by multiplying the length of the generated schedule with block execution count. Like scheduling time, execution time is reported proportionally. The results are shown in Figure 5.14.

Using IFlat-10 to schedule the top 3% most frequently executed blocks increases scheduling time by a factor of five and reduces overall execution time by about 14%. Reserving IFlat for the most frequent 1% increases scheduling time by just 20% and increases performance by about 5%.

It is difficult to indicate exactly when a VM would choose to employ List Scheduling and when it should opt for the more aggressive IFlat or other search-based scheduler. However, using more than one scheduling algorithm seems effective at increasing overall performance when using an embedded compiler. Further, by providing an additional compilation parameter the VM gains additional control over the expense and effectiveness of just-in-time compilation.

5.9. Summary

Each of the search-based instruction schedulers produces shorter schedules than List Scheduling. This improvement is at the cost of significant increases in scheduling time. Turning this extra scheduling time into better schedules, however, differs greatly between the different schedulers.

LDS explores a large portion of the solution space. This contributes to the large scheduling time and to the large scheduling improvements that it makes. Harvey and Ginsberg's [22] assumption that the heuristic is good, but not perfect at directing

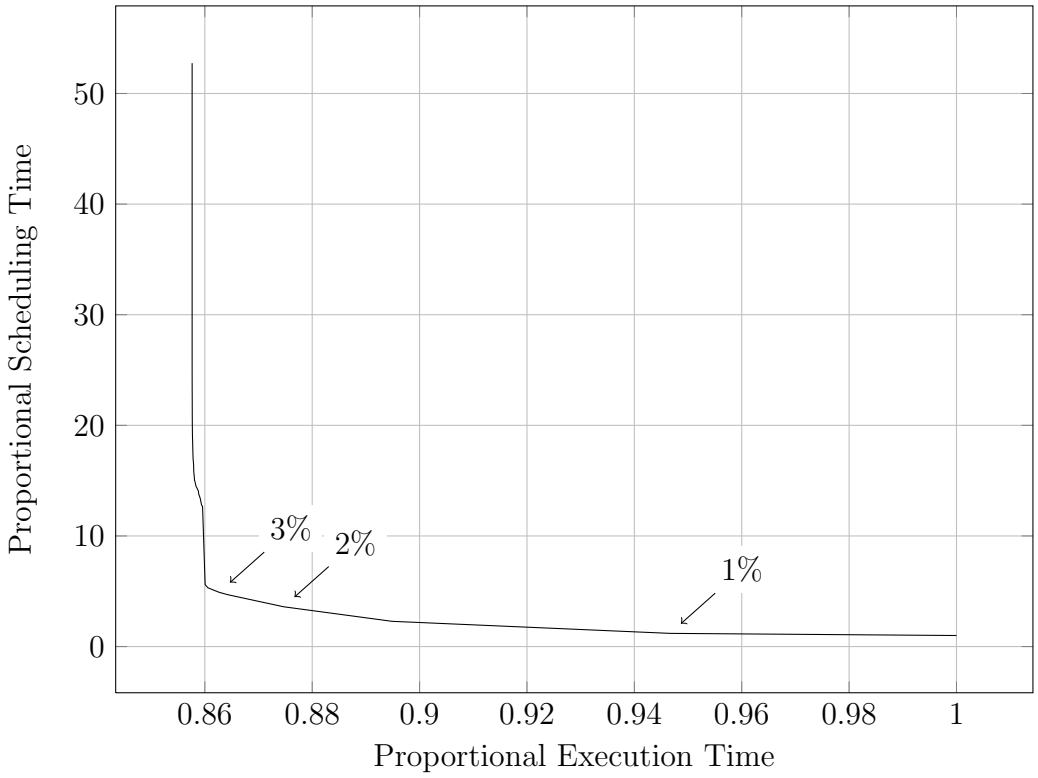


FIGURE 5.14. Two-Scheduler Scheduling Time vs Proportional Speedup

construction is clearly valid in the Instruction Scheduling domain. However, the amount of work required to widen the search area makes LDS unattractive for any application that requires fast compilation. In domains where register usage is more important than compilation speed, LDS performs very well. This is despite the fact that the experimental heuristics we used do not consider register pressure at all. It is just the slight bias toward lower register pressure and the extent to which LDS explores the solution space that reduces the register pressure of the solutions.

SW is much less computationally expensive than LDS, but it does not make the same improvements to the generated schedules. This would imply that directing instruction scheduling from priority or heuristic space is more difficult than expected. SW's non-systematic approach allows it to generate the same solution over and over.

Generally, SW is applied to domains where priority space is smaller than solution space. For task scheduling, where each task is given a specific start time, this is usually the case. This allows the constructor to be one-to-one or it is nearly one-to-one. That is, each prioritization leads to a unique solution. In this domain, however, solution space is much smaller than priority space. We believe that this is what causes SW to make a modest improvement in the first full cycle and then get stuck.

The success of IFlat is a bit surprising. Aside from conflict identification and resolution, it has no embedded instruction scheduling knowledge. Conflicts are solved randomly. Despite this uninformed approach, IFlat produces significantly shorter solutions than any of the other algorithms explored. This is true when IFlat does no search but just constructs a single solution. It is difficult to draw any other conclusion than that reasoning about conflicts is superior to reasoning about dependencies. This is not to say that IFlat's successes are only because of its conflict resolution focus. Each additional iteration builds upon the success of the previous solution. So the local searching that IFlat does is advantageous, but it clearly starts with a very good solution.

There is no one clear "winner" among the search-based schedulers. Each has its advantages and disadvantages. Our multi-objective interpretation of instruction scheduling let us identify the suite of schedulers that may be ideal for some domain. This Pareto Optimal set includes instances of each of the implemented schedulers. Only some of the SW instances are excluded from the set. Which scheduler to use depends on the demands and expectations of the specific domain.

CHAPTER VI

RELATED AND FUTURE WORK

Our problem description of Instruction Scheduling with Special Purpose Registers appears to be unique. This is a bit surprising since including special purpose registers in the problem description moves Instruction Scheduling from NP-Complete with unbounded parameters (e.g., arbitrary latencies and number of pipelines) to NP-Complete with tight bounds on the parameters. Usually, special purpose registers are treated as an implementation detail.

A noteworthy exception is early work by Muchnick and Gibbons [32]. Their scheduling algorithm is a post-compilation optimizer. Rather than compiling to a binary executable, the compiler generates an assembly file. This file is read and the Dependency Graph is recreated. Muchnick and Gibbons reschedule the basic blocks using heuristics that are specifically designed for their target hardware, an early PA-RISC. Since their input is a complete but unassembled program, physical registers have already been assigned. The rescheduled program uses the same registers. Further, the authors mention a PA-RISC special purpose register: the carry/borrow bit. Muchnick and Gibbons [32] described their solution as:

> Carry/borrow dependencies are handled specially in constructing the dags, since carries and borrows are very frequently defined but only rarely used. Serializing all carry/borrow definitions against each other would be unduly constraining. Instead, a special subgraph is generated within the dag for each instruction which *uses* a carry or borrow; the subgraph includes all the instructions which must appear between the use and

the corresponding definition (or the beginning of the basic block if no definition is found in it).

This subgraph approach is similar to how our IFlat implementation handles SPRs.

For the basic Instruction Scheduling problem, List Scheduling reigns supreme. Research based on List Scheduling generally falls into two categories. The first group uses List Scheduling as a component in a larger scheduling algorithm. Generally these algorithms manipulate the heuristic function or they annotate the dependency graph that List Scheduling uses.

These algorithms try to learn the "correct" heuristic function for the specific scheduling problem. Auyeung et al. [3] used several heuristics with List Scheduling. They applied their approach to task scheduling, not instruction scheduling. However, the two domains are closely related. They actively tune the relative weights of these heuristics with a Genetic Algorithm (GA). Grajcar [20] use GA to develop the heuristic ordering directly. Their approach is very similar to our SW scheduler.

Terada et al. [40] described SW as a form of GA with a population of one that uses genetic engineering rather than cross-over and mutation as genetic operators. They incorporated SW into a GA-based search algorithm. They found that GA+SW performs better than GA alone after many generations. From their perspective, our SW scheduler is a primitive GA algorithm. However, a full GA approach with a large population and several genetic operations would significantly increase scheduling time.

Wang et al. [44] used an Ant System Optimization (AS) approach to heuristic discovery [14]. Unlike GA, which considers a few samples from the population, the agents in AS communicate with each other via a "pheromone" trail. This communication channel enables AS to converge on a common solution more quickly than a more general GA approach.

Russell et al. [37] applied Decision Tree Induction (DT) to learning List Scheduling heuristics. Unlike the schedulers described above, DT develops a heuristic off-line using training data. The learned heuristic is used directly by a List Scheduling compiler. Since the training is done beforehand, compile time is greatly reduced compared to the other AI schedulers.

The second type of List Scheduling research extends the basic algorithm with additional functionality. For example, Goodman and Hsu [19] added a leader set to List Scheduling. The leader set are ready instructions that have resource conflicts with the partially generated schedule. That is, leaders are almost ready. Goodman and Hsu use this approach to manage register usage while scheduling. Their scheduler solves the Cooperative Scheduling problem. A second register allocation phase was still needed. Others have built upon this basic design to solve the Integrated Scheduling problem that simultaneously schedules instructions and assigns physical registers [6, 12, 13].

Moon and Ebcioğlu [30] adapt the core List Scheduling algorithm to maximize Instruction Level Parallelism (ILP) on superscalar and VLIW systems with predicated instructions. Predicated instructions only execute if the corresponding predicate expression is true. For example, the x86 CMOV instructions move data but only if the correct condition bits are set in the flags register. VLIW systems bundle several instructions together. With predicated instructions, VLIW systems can effectively execute both sides of an if-statement without branching. Moon and Ebcioğlu [30] developed Selection Scheduling that behaves much like List Scheduling. The major difference is that Selection Scheduling is a global scheduler. It schedules an entire subroutine not just a single basic block.

Our LDS scheduler fits nicely with these schedulers. LDS is an extension to the basic List Scheduling algorithm. Unlike Moon and Ebcioğlu [30], our LDS scheduler is a local scheduler since it operates on a single basic block.

Not all compilers use List Scheduling. IFlat's approach is unlike that of List Scheduling or any of the algorithms described above. Similarly, Convergent Scheduling makes several passes over the scheduling problem [27]. Each pass can modify the solution addressing a particular aspect of the target hardware. This is similar to IFlat's CONFLICTS function. However, IFlat resolves a single conflict at a time. A Convergent Scheduling pass finds and solves a particular kind of conflict throughout the potential solution. This would be analogous to IFlat resolving all dispatch conflicts in one pass and resolving all of the conflicts for one particular special purpose register in another. Convergent Scheduling applies these heuristics repeatedly until the schedule converges.

Win and Wong [45] combine Convergent Schedule with Linear-Scan register allocation [36]. Their Integrated Scheduler uses the same DG for scheduling and allocation. What is more interesting about their solution is that it uses a blame-prioritize-rebuild cycle that is very similar to SW. However, their algorithm does not search through scheduling space for a better solution. It searches through priority space for a weighted heuristics that generate a valid schedule and allocation.

Measuring performance is something of a challenge. Win and Wong [45] were concerned with embedded JIT compilation and focus on scheduling time and register pressure. They ignore schedule length entirely. Contrast this approach with Bebenita et al. [5] who focus on schedule length or, more accurately, the execution time of the generated program.

The most common approach is to report scheduling time and schedule length over some appropriate benchmark suite. Moon and Ebcioğlu [30] use SPEC 89, Russell et al. [37] use SPEC 2000, and Bebenita et al. [5] use SunSpider benchmark suites.[1] Following this pattern, we chose to use the latest SPEC CPU 2006 benchmark suite for our experiments.

Our work fits nicely with the body of Instruction Scheduling research. The three search-based scheduling algorithms that we developed are similar to other schedulers and our experiments use the latest benchmark suite and cover at least the metrics used in the literature. But where do we go from here?

6.1. Future Work

This thesis presented three significant Instruction Scheduling problems. Our scheduling algorithms only address one of the three, Instruction Scheduling with Special Purpose Registers. The effect on register pressure was measured but we did not address register usage directly. These algorithms can be extended to Cooperative and possibly Integrated scheduling.

IFlat is successful without any embedded knowledge of the scheduling domain. Adding domain specific knowledge should improve its results. For example, when resolving special purpose register violations, IFlat orders the two defining instructions randomly. This is the case even if one of the instructions is already scheduled before the other. Further, no consideration is given to instructions on the critical path when resolving dispatch constraint violations. IFlat may delay a critical instruction, thus

[1]SPEC Benchmark suites are collections of FORTRAN, C, and C++ programs that are considered "typical"computationally intensive applications (see `http://www.spec.org`). The SunSpider is a benchmark suite for JavaScript (see `http://www.webkit.org/perf/sunspider/sunspider.html`).

extending the length of the entire schedule. Adding even these simple extensions to IFlat should make significant improvements.

Further, our experimental results were based on basic block scheduling length. These algorithms should be applied to larger blocks like Traces or Regions. The same performance improvements should be available to the search-based schedulers that were seen by List and Convergent Schedulers.

Finally, these schedulers should be implemented in a production compiler like LLVM. We explored the effect that search-based scheduling has on the schedule length without allocating registers and without directly executing the schedules. Measuring the schedule lengths in isolation is the best way to gauge the effectiveness of the scheduling algorithm. As a multi-objective optimization problem, it is difficult to judge the tradeoffs between shorter schedules and longer compilation times without truly understanding the effect that these choices have on the performance of the generated program.

CHAPTER VII

CONCLUSION

The central question that this thesis addresses is whether search-based optimization algorithms can be applied successfully to compiler instruction scheduling.

In answering this question we established a formal definition of instruction scheduling. At the highest level, instruction scheduling is an instance of general resource constrained scheduling. However, the resources of a CPU are different from the resources models used in general scheduling. Spanning resources, the model that describes preassigned and implied registers, are enough to show instruction scheduling NP-Complete with modest assumptions about instruction interactions.

Considering that search-based optimization is successful in other NP-Complete domains including resource constrained scheduling, the fact that search-based schedulers generated shorter schedules than the standard, construction-based List scheduler is not a surprise. However, success in this domain is not simply a question of shorter schedules, but what is the cost to generate these schedules?

Instruction scheduling considers the quality of the generated schedule in addition to the time spent scheduling and number of hardware registers required by the generated schedule. This is fundamentally a multi-objective optimization problem as different compilers will place different values on these goals. The relative importance of these objectives may change within the same compiler.

Consider a just-in-time compiler embedded within a Java Virtual Machine or Python interpreter. Initially, the compiler favors compile time over all other factors. After all, embedded compilers consider compile-time run-time. In this mode, List

Scheduling is clearly preferred. If the program spends most of its time executing just a few rather large blocks, the compiler could reschedule those blocks with more focus on schedule length.

Contrast this model with a static, off-line compiler for small embedded hardware. These systems have modest performance and very few registers. In this model, the focus may shift to register usage and schedule length with little regard for compilation time. Here, LDS is the preferred scheduler.

For commercial-off-the-shelf software, the customer places no value on compile-time. The software is purchased in binary, compiled form and simply executed. This could easily justify the increased compilation of a more aggressive search-based scheduler.

The answer to our central question is yes, search-based scheduling can be successfully applied to instruction scheduling. Search algorithms are not a compiler's panacea. There are domains where IFlat, LDS, and SW are inappropriate.

In the end, we have shown that there are different tools for different jobs.

APPENDIX

BENCHMARK STATISTICS

We use the SPEC CPU 2006 benchmark suite to gather the experimental results shown in Chapter V. The benchmark suite is a collection of typical CPU intensive applications that are written in C, C++, and FORTRAN. From these programs we extracted the scheduling problems used in the experimental evaluation of our scheduling algorithms. This appendix provides summary statistics of these benchmark problems.

Graph order measures the number of vertexes in the graph. For DGs, graph order indicated the number of instructions. Figure A.1 shows the number of scheduling problems of orders up to 100. The largest graphs have several hundred to thousands of instructions. However, there are very few of these problems. Less than 3000 problems have more than 100 instructions. Problem order nearly fits a log-normal distribution with $\mu = 1.61$ and $\sigma = 0.69$. We can see from these statistics that scheduling problems tend to be very small. The peak at three instructions represents almost 20% of the scheduling problems alone.

The size of a graph refers to the number of edges. For instruction scheduling, this includes both data dependencies and order dependencies. The distribution of DG sizes is shown in Figure A.2 for graph sizes up to 100 edges. The data actually extend to about $15,000$ edges. Only about 100 problems have more than 1000 edges. Like graph order, size follows a log-normal distribution with $\mu = 2.15$ and $\sigma = 1.19$.

Figure A.3 shows these two distributions as a scatter plot. The linear best fit has slope 0.43, which represents about 2.3 dependencies per instruction. Further, we

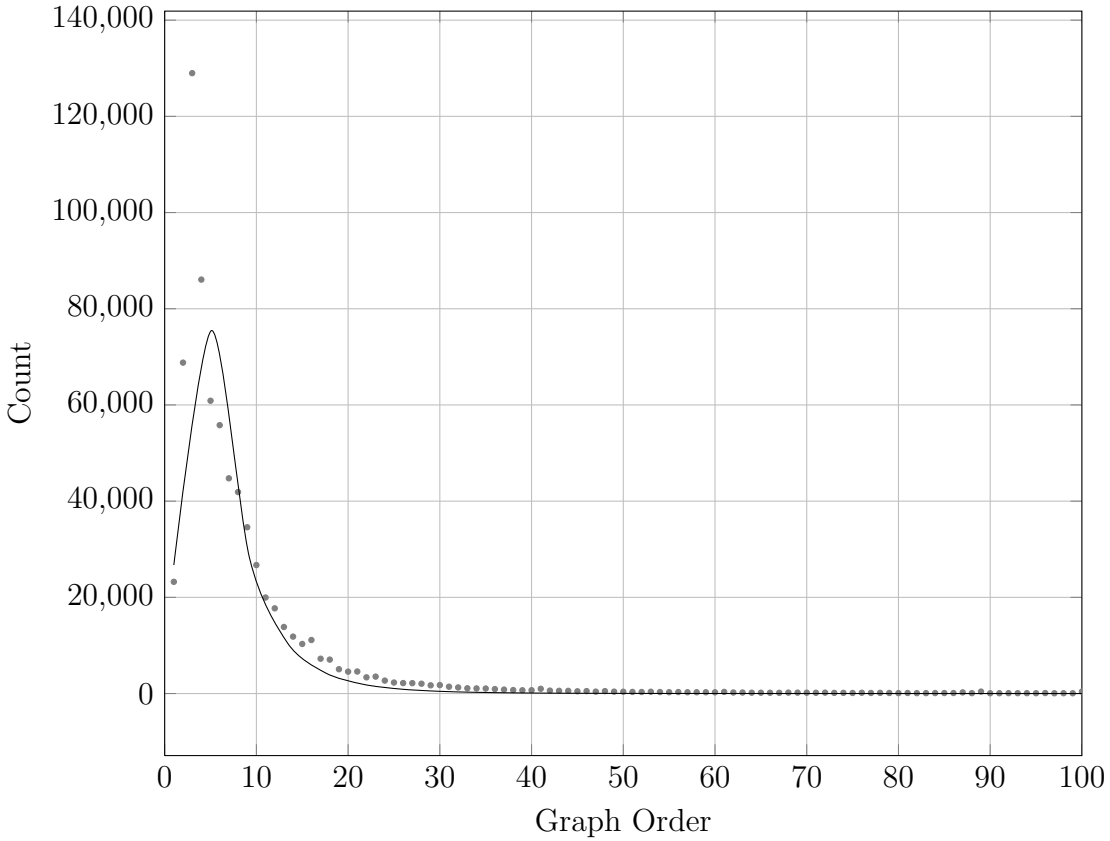


FIGURE A.1. Dependency Graph Order distribution.

see that most of the problems are clustered near the origin indicating that the DG are relatively small and easy to schedule.

Finally, Figure A.4 shows the execution frequency distribution. This distribution shows that roughly 80% of the scheduling problems are executed fewer than one million times. Further, this curve supports the 90-10 rule as 10% of the scheduling problems are executed 90% of the time. Further, there is no strong correlation between execution frequencies and problem size. That is, large basic blocks are executed about as often as smaller blocks.

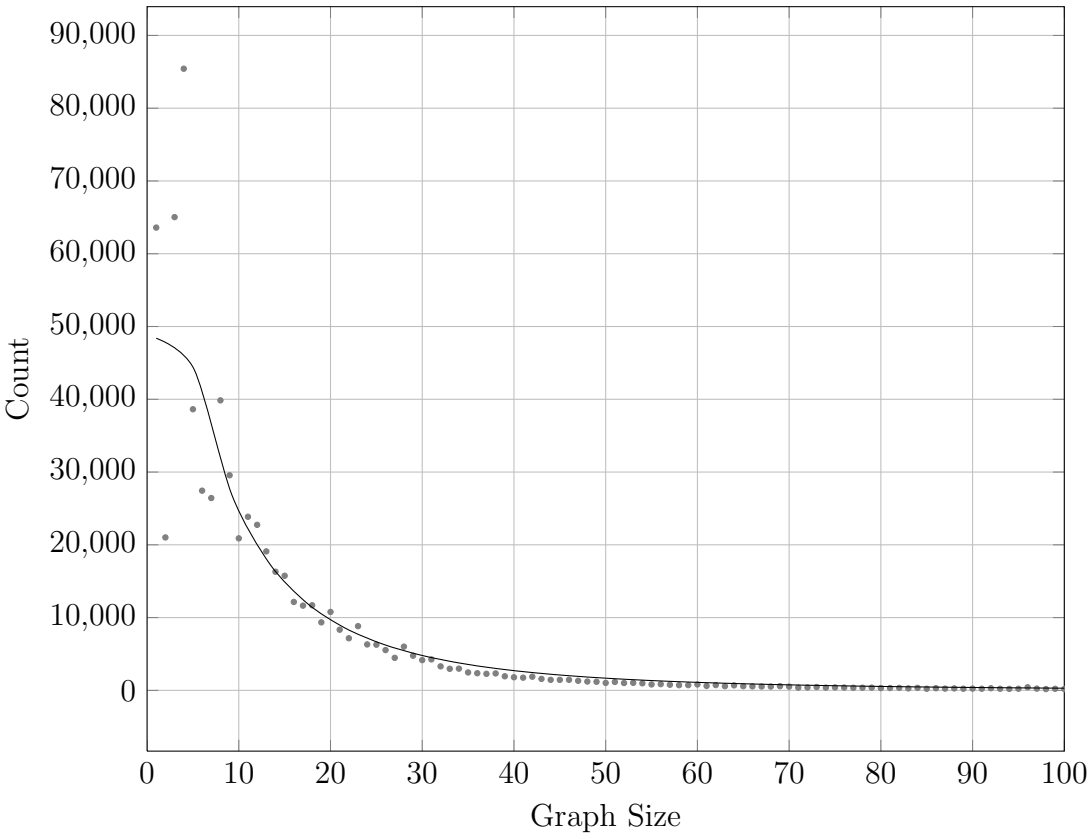


FIGURE A.2. Dependency Graph Size distribution.

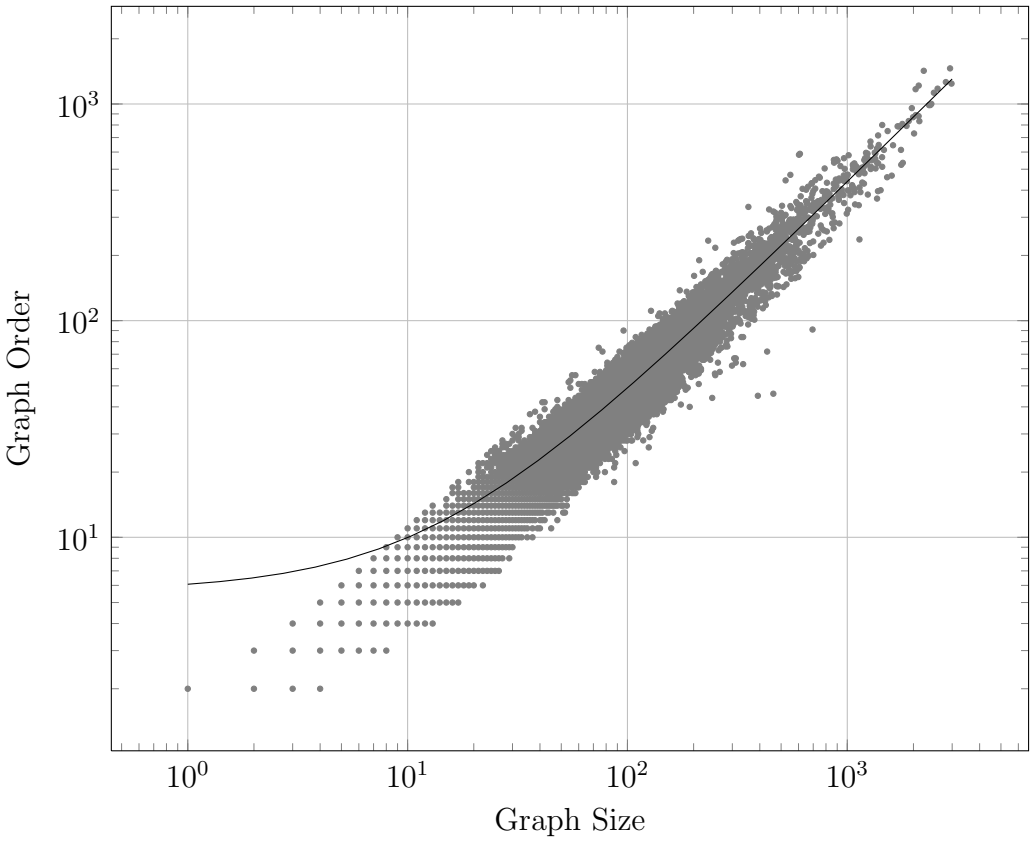


FIGURE A.3. Dependency Graph Size vs Order scatter plot.

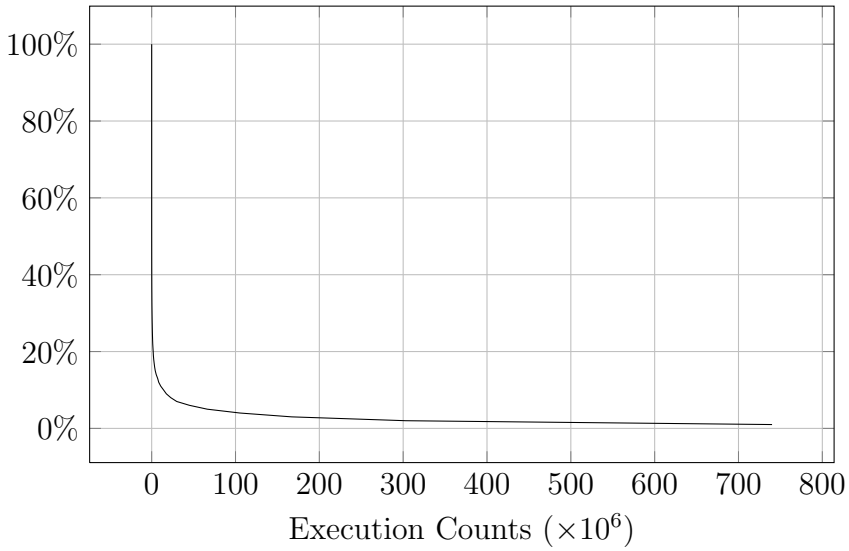


FIGURE A.4. Execution Frequencies

REFERENCES CITED

[1] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code Generation for Expressions with Common Subexpressions. *J. ACM*, 24(1):146–160, January 1977. ISSN 0004-5411. doi: 10.1145/321992.322001.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[3] Andy Auyeung, Iker Gondra, and H. K. Dai. Multi-Heuristic List Scheduling Genetic Algorithm for Task Scheduling. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 721–724, New York, NY, USA, 2003. ACM. ISBN 1-58113-624-2. doi: 10.1145/952532.952673.

[4] John Backus. The History of FORTRAN I, II and III. *IEEE Ann. Hist. Comput.*, 1:21–37, July 1979. ISSN 1058-6180. doi: 10.1109/MAHC.1979.10013.

[5] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: a Trace-Based JIT Compiler for CIL. *SIGPLAN Not.*, 45(10):708–725, October 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869517.

[6] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 122–131, New York, NY, USA, 1991. ACM. ISBN 0-89791-380-9. doi: 10.1145/106972.106986.

[7] Graham Brightwell and Peter Winkler. Counting Linear Extensions is #P-complete. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 175–181, New York, NY, USA, 1991. ACM. ISBN 0-89791-397-3. doi: 10.1145/103418.103441.

[8] John Bruno and Ravi Sethi. Code Generation for a One-Register Machine. *J. ACM*, 23(3):502–510, July 1976. ISSN 0004-5411. doi: 10.1145/321958.321971.

[9] Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems. In *In AAAI/IAAI*, pages 742–747, 2000.

[10] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation via Coloring. *Computer Languages*, 6(1):47–57, 1981. ISSN 0096-0551. doi: 10.1016/0096-0551(81)90048-5.

[11] Pohua P. Chang, Daniel M. Lavery, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors. *IEEE Transactions on Computers*, 44:353–370, 1994.

[12] Gang Chen. *Effective Instruction Scheduling with Limited Registers*. PhD thesis, Harvard University, Cambridge, MA, USA, 2001. Adviser-Michael D. Smith.

[13] Ioana Cutcutache and Weng-Fai Wong. Fast, Frequency-Based, Integrated Register Allocation and Instruction Scheduling. *Softw. Pract. Exper.*, 38(11): 1105–1126, 2008. ISSN 0038-0644. doi: 10.1002/spe.v38:11.

[14] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System: Optimization by a Colony of Cooperating Agents. *Trans. Sys. Man Cyber. Part B*, 26(1):29–41, February 1996. ISSN 1083-4419. doi: 10.1109/3477.484436.

[15] Martin Ester, Hans P. Kriegel, Jorg Sander, and Xiaowei Xu. A Density-Based algorithm for discovering clusters in large spatial databases with noise. In Evangelos Simoudis, Jiawei Han, and Usama Fayyad, editors, *Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, Portland, Oregon, 1996. AAAI Press. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.2930`.

[16] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. volume 30, pages 478–490. IEEE Computer Society, Washington, DC, USA, July 1981. doi: 10.1109/TC.1981.1675827.

[17] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

[18] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. ISBN 079239965X.

[19] J. R. Goodman and W. C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 442–452, New York, NY, USA, 1988. ACM. ISBN 0-89791-272-1. doi: 10.1145/55364.55407.

[20] Martin Grajcar. Conditional Scheduling for Embedded Systems using Genetic List Scheduling. In *Proceedings of the 13th international symposium on System synthesis*, ISSS '00, pages 123–128, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 1-58113-267-0. doi: 10.1145/501790.501817.

[21] Richard E. Hank, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Region-Based Compilation: an Introduction and Motivation. In *MICRO 28: Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 158–168, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press. ISBN 0-8186-7349-4.

[22] William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 607–613, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8, 978-1-558-60363-9. URL `http://dl.acm.org/citation.cfm?id=1625855.1625935`.

[23] W. Havanki, S. Banerjia, and T. Conte. Treegion Scheduling for Wide Issue Processors. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 266, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8323-6.

[24] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *THE JOURNAL OF SUPERCOMPUTING*, 7:229–248, 1993.

[25] David E. Joslin and David P. Clements. "Squeaky Wheel" Optimization. *J. Artif. Int. Res.*, 10(1):353–373, 1999. ISSN 1076-9757.

[26] A. B. Kahn. Topological Sorting of Large Networks. *Commun. ACM*, 5(11): 558–562, November 1962. ISSN 0001-0782. doi: 10.1145/368996.369025.

[27] Walter Lee, Diego Puppin, Shane Swenson, and Saman Amarasinghe. Convergent Scheduling. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 111–122, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. ISBN 0-7695-1859-1.

[28] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. *SIGMICRO Newsl.*, 23:45–54, December 1992. ISSN 1050-916X. doi: 10.1145/144965.144998.

[29] Abid M. Malik, Jim McInnes, and Peter van Beek. Optimal Basic Block Instruction Scheduling for Multiple-Issue Processors Using Constraing Programming. In *ICTAI '06: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, pages 279–287, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2728-0. doi: 10.1109/ICTAI.2006.92.

[30] Soo-Mook Moon and Kemal Ebcioğlu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. *ACM Trans. Program. Lang. Syst.*, 19(6):853–898, November 1997. ISSN 0164-0925. doi: 10.1145/267959.269966.

[31] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining Register Allocation and Instruction Scheduling. Technical report, Stanford University, Stanford, CA, USA, 1995.

[32] Steven S. Muchnick and Phillip B. Gibbons. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Not.*, 39(4):167–174, 2004. ISSN 0362-1340. doi: 10.1145/989393.989413.

[33] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982. ISBN 0-13-152462-3.

[34] Fernando Magno Quintão Pereira and Jens Palsberg. *Register Allocation After Classical SSA Elimination is NP-Complete*, volume 3921/2006 of *Lecture Notes in Computer Science*, pages 79–93. Springer Berlin, Heidelberg, March 2006. ISBN 978-3-540-33045-5.

[35] Shlomit S. Pinter. Register Allocation with Instruction Scheduling. *SIGPLAN Not.*, 28(6):248–257, 1993. ISSN 0362-1340. doi: 10.1145/173262.155114.

[36] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999. ISSN 0164-0925. doi: 10.1145/330249.330250.

[37] Tyrel Russell, Abid M. Malik, Michael Chase, and Peter van Beek. Learning Basic Block Scheduling Heuristics from Optimal Data. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '05, pages 242–253. IBM Press, 2005.

[38] Mark Smotherman and Dag Spicer. IBM's Single-Processor Supercomputer Efforts. *Commun. ACM*, 53(12):28–30, December 2010. ISSN 0001-0782. doi: 10.1145/1859204.1859216.

[39] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling. In *Proceedings of the 24th annual international symposium on Microarchitecture*, MICRO 24, pages 93–102, New York, NY, USA, 1991. ACM. ISBN 0-89791-460-0. doi: 10.1145/123465.123482.

[40] Justin Terada, Hoa Vo, and David Joslin. Combining genetic algorithms with squeaky-wheel optimization. In *GECCO '06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 1329–1336, New York, NY, USA, 2006. ACM. ISBN 1-59593-186-4. doi: 10.1145/1143997.1144203.

[41] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and Speed in Linear-Scan Register Allocation. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 142–151, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277714.

[42] J. D. Ullman. NP-Complete Scheduling Problems. *J. Comput. Syst. Sci.*, 10(3): 384–393, June 1975. ISSN 0022-0000. doi: 10.1016/S0022-0000(75)80008-0.

[43] J. D. Ullman. *Computer and Job-Shop Scheduling Theory*, chapter Complexity of Sequencing Problems, pages 139–164. New York, NY, USA, 1976. John Wiley & Sons, Inc.

[44] Gang Wang, Wenrui Gong, and Ryan Kastner. Instruction Scheduling using MAX-MIN Ant System Optimization. In *Proceedings of the 15th ACM Great Lakes Symposium on VLSI*, GLSVLSI '05, pages 44–49, New York, NY, USA, 2005. ACM. ISBN 1-59593-057-4. doi: 10.1145/1057661.1057674.

[45] Khaing Khaing Kyi Win and Weng-Fai Wong. Cooperative Instruction Scheduling with Linear Scan Register Allocation. In *Proceedings of the 12th international conference on High Performance Computing*, HiPC'05, pages 528–537, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-30936-5, 978-3-540-30936-9. doi: 10.1007/11602569_54.