5-1-1990

Optimal Code Scheduling for Multiple Pipeline Processors

Ashar Nisar  
*Purdue University*

Hank Dietz  
*Purdue University*

Follow this and additional works at: [https://docs.lib.purdue.edu/ecetr](https://docs.lib.purdue.edu/ecetr)
Optimal Code Scheduling for Multiple Pipeline Processors

Ashar Nisar
Hank Dietz

TR-EE 90-42
May 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907
OPTIMAL CODE SCHEDULING FOR
MULTIPLE PIPELINE PROCESSORS

A Thesis
Submitted to the Faculty

of

Purdue University

by

Ashar Nisar

In Partial Fulfillment of the
Requirements for the Degree

of

Master of Science in Electrical Engineering

August 1990
this is dedicated
to my mom and dad
ACKNOWLEDGMENTS

This page of thesis is usually reserved for we degree candidates, to eulogize, cajole or beg (not necessarily in that order) the members of our advisory comit-tees. In a glimmer of hope that they will not pronounce our theses and claims to be baseless or "It has already been done!" So here goes.

Hank Dietz. It is an honor to know this person. A genius with a profound research insight, brilliant ideas, an amiable personality and, most importantly, a sparkling sense of humor. I have learned a lot from him — everything from Chimpminkpunk parodies to the secrets of preparing palatable fruit-punch. What can I possibly say in return except, perhaps, "Live long and prosper."

I also offer my thanks to Professor Robert Fujii and Professor Shaheen Ahmad for serving on my advisory committee. My special thanks to Imran and Carol for their help and support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>x</td>
</tr>
<tr>
<td>CHAPTER 1 - INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2. Pipeline Characteristics</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1. Compiler's View</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2. Architecture's View</td>
<td>5</td>
</tr>
<tr>
<td>1.3. NOPs and Delay Slots</td>
<td>6</td>
</tr>
<tr>
<td>1.4. An Overview of This Document</td>
<td>7</td>
</tr>
<tr>
<td>CHAPTER 2 - BACKGROUND AND SURVEY OF RELATED LITERATURE</td>
<td>9</td>
</tr>
<tr>
<td>2.1. Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2.2. An Example of Code Scheduling</td>
<td>9</td>
</tr>
<tr>
<td>2.3. The Complexity of Finding An Optimal Schedule</td>
<td>11</td>
</tr>
<tr>
<td>2.4. PostPass Code Optimization</td>
<td>15</td>
</tr>
<tr>
<td>2.4.1. Proof of NP-Completeness</td>
<td>17</td>
</tr>
<tr>
<td>2.4.2. The Algorithm</td>
<td>18</td>
</tr>
<tr>
<td>2.4.2.1. Reordering Constraints</td>
<td>19</td>
</tr>
<tr>
<td>2.4.2.2. Heuristics</td>
<td>21</td>
</tr>
<tr>
<td>2.4.2.3. Results</td>
<td>22</td>
</tr>
<tr>
<td>2.5. Improved Approximation Algorithm by David Bernstein</td>
<td>23</td>
</tr>
<tr>
<td>2.5.1. Background</td>
<td>23</td>
</tr>
<tr>
<td>2.5.2. Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>2.6. Reorganizer for a Variable-Length Pipelined Microprocessor</td>
<td>25</td>
</tr>
<tr>
<td>2.6.1. Introduction</td>
<td>25</td>
</tr>
<tr>
<td>2.6.2. The Algorithm</td>
<td>25</td>
</tr>
<tr>
<td>2.7. Micro-Optimization of Floating-Point Operations</td>
<td>27</td>
</tr>
<tr>
<td>2.8. Scheduling Trees in Pipelined Environments</td>
<td>27</td>
</tr>
<tr>
<td>2.9. Summary</td>
<td>28</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.4. Simulation of the General Behavior</td>
<td>64</td>
</tr>
<tr>
<td>5.4.1. Procedure</td>
<td>64</td>
</tr>
<tr>
<td>5.4.2. Pipeline Constraints for Simulations</td>
<td>64</td>
</tr>
<tr>
<td>5.4.3. Results</td>
<td>65</td>
</tr>
<tr>
<td>5.4.4. Discussion</td>
<td>66</td>
</tr>
<tr>
<td>5.5. Variations in the Curtail Point</td>
<td>70</td>
</tr>
<tr>
<td>5.5.1. Procedure</td>
<td>70</td>
</tr>
<tr>
<td>5.5.2. Results</td>
<td>70</td>
</tr>
<tr>
<td>5.5.3. Discussion</td>
<td>72</td>
</tr>
<tr>
<td>5.6. Varying the Number of Variables</td>
<td>72</td>
</tr>
<tr>
<td>5.6.1. Procedure</td>
<td>72</td>
</tr>
<tr>
<td>5.6.2. Results</td>
<td>72</td>
</tr>
<tr>
<td>5.6.3. Discussion</td>
<td>74</td>
</tr>
<tr>
<td>5.7. Variation in the Pipeline Structure</td>
<td>74</td>
</tr>
<tr>
<td>5.7.1. Procedure</td>
<td>74</td>
</tr>
<tr>
<td>5.7.2. Results</td>
<td>74</td>
</tr>
<tr>
<td>5.7.3. Discussion</td>
<td>74</td>
</tr>
<tr>
<td>5.8. Sub-Optimal Solutions</td>
<td>75</td>
</tr>
<tr>
<td>5.8.1. Procedure</td>
<td>76</td>
</tr>
<tr>
<td>5.8.2. Results</td>
<td>77</td>
</tr>
<tr>
<td>5.8.3. Discussion</td>
<td>77</td>
</tr>
<tr>
<td>5.9. Summary</td>
<td>78</td>
</tr>
<tr>
<td>CHAPTER 6 - CONCLUSIONS</td>
<td>80</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td>81</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Search Space for Exhaustive Search</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Search Space for Representative Examples</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Percent of NOPs Required for Different Heuristics</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Sample Pipeline Description Table</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Sample Operation-to-Pipeline Mapping</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Sample Pipeline Description Table</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Sample Operation-to-Pipeline Mapping</td>
<td>60</td>
</tr>
<tr>
<td>5.1</td>
<td>Synthetic Benchmark Instruction Frequencies</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Pipeline Description for Simulations</td>
<td>66</td>
</tr>
<tr>
<td>5.3</td>
<td>Operation-to-Pipeline Mapping for Simulations</td>
<td>66</td>
</tr>
<tr>
<td>5.4</td>
<td>Statistics for Scheduling 16,000 Blocks</td>
<td>67</td>
</tr>
<tr>
<td>5.5</td>
<td>Variations in the Pipeline Structures</td>
<td>75</td>
</tr>
<tr>
<td>5.6</td>
<td>Final NOPs in suboptimal Solutions</td>
<td>78</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>56</td>
</tr>
<tr>
<td>5.1</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>68</td>
</tr>
<tr>
<td>5.4</td>
<td>68</td>
</tr>
<tr>
<td>5.5</td>
<td>69</td>
</tr>
<tr>
<td>5.6</td>
<td>71</td>
</tr>
<tr>
<td>5.7</td>
<td>71</td>
</tr>
<tr>
<td>5.8</td>
<td>73</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.9.1 Average Runtime Vs. Variables</td>
<td>73</td>
</tr>
<tr>
<td>5.10. Average Percentage Run To Completion Vs. Pipeline Structure</td>
<td>76</td>
</tr>
<tr>
<td>5.11. Fraction of Initial NOPs Vs. Runtime</td>
<td>77</td>
</tr>
</tbody>
</table>
ABSTRACT

Ashar Nisar, M.S.E.E., Purdue University. August 1990. Optimal Code Scheduling for Multiple Pipeline Processors. Major Professor: Dr. Henry Dietz.

Pipelining the functional units and memory interface of processors can result in shorter cycle times and dramatic increases in performance, but only if the pipeline delays can be hidden by other useful operations. The portion of pipeline delays which is not hidden results in an extension of the total execution time, either implemented by hardware interlocks or by compile-time insertion of NOPs (Null OPERations). By rearranging instructions, it is possible to minimize the total pipelined execution time, but the problem of finding this optimal code schedule is well known to be NP-complete.

In this thesis, we describe a code scheduler for multiple pipeline processors where each pipeline may have a different latency and enqueue time. Previous approaches simplify the search for a good schedule by arbitrarily imposing constraints which sacrifice optimality; the technique given in this paper uses a new set of pruning criteria which preserves optimality. Although, in the interest of reducing compile time, the new technique permits the search to be truncated, this truncation only rarely (in less than 2% of the cases examined) sacrifices optimality.
1.1. Introduction

Most modern processors, especially RISC designs like Motorolla's 88000 [Mel88], MIPS R3000 [Rio88], SPARC [Muc88], etc., attempt to achieve a peak performance of one instruction completing execution with every clock tick. However, this does not imply that execution of a single instruction always happens within a single clock tick; rather, pipelined hardware is used to overlap execution of multiple instructions to achieve this throughput.

For example, if each instruction requires 5 clock ticks to execute, throughput of one instruction per clock tick can be obtained by allowing 5 instructions to overlap execution within a 5-stage pipeline. In order to obtain one instruction per clock tick throughput, one simply needs to have one instruction ready to enter the pipeline at every clock tick. The problem is that if code is generated from a high-level language in the most obvious way, many instruction sequences will require that a delay be introduced before the next instruction can be issued.

The problem of compiling code so as to minimize the total delay which must be introduced is nearly as old as the concept of pipelining hardware, and appears to have been considered as early as the 1950s. In the 1960s, as circuitry became inexpensive enough to make the hardware cost-effective, machines with multiple functional units became common: typically, independent adders and multipliers which could operate in pipelined overlap with other instructions. Most of the compiler research centered on the development of heuristics which could be used to "generate" code so that total delay would be reduced for such machines; a reasonable overview appears in [CoS70].
Although the compiler techniques used to generate low-delay code were reasonably effective, they generally assumed that the code-generation process was relatively straightforward; in other words, these techniques become awkward when other compiler optimizations are also being performed. For this reason, the emphasis has shifted from heuristics for generating code to heuristics for re-organizing, or scheduling, code after it has been generated using whatever other optimizations were appropriate.

Probably the best known work in instruction scheduling for pipelined processors is by Gross, detailed in [Gro83]. Gross proposed a heuristic algorithm for reordering instructions and showed that, although his heuristic typically does not result in the minimum delay (optimal schedule), the algorithm executes quickly and generally yields good results. By applying his algorithm to the optimized assembly language output of a compiler, he also avoids the complexity of integrating scheduling with the other optimizations within the compiler. It appears that this is a reasonable approach, except in that the compiler has performed register allocation. Hence, the register assignment can impose unnecessary restrictions on the schedule, resulting in unnecessary execution delays.

Bernstein presented an improved scheduling algorithm, but his work considers only pipelines having a fixed delay [Ber88]. Abraham et. al. [AbP88] permitted variable delay pipelines, but resorted to a greedy heuristic algorithm, instead of searching for the optimal schedules.

The algorithm we propose differs from previous work in several ways:

[1] We apply our algorithm to an intermediate form of code which does not have specific registers assigned, hence register allocation happens after scheduling and the scheduler is not unnecessarily constrained.

[2] Although our algorithm is also heuristic, none of the heuristics applied sacrifices optimality. In other words, the search space is pruned dramatically, but the optimal solution will never be pruned. In cases where the pruned search space is still too large, the search may be terminated after an arbitrary number of cases have been examined, but this happens only rarely and still generally results in very good schedules.
The target pipeline architecture model supported is significantly more general than that typically used, permitting multiple pipelines, each with its own latency and enqueue time, to be specified. In particular, we believe our proposal is the first to consider the pipeline enqueue time as a key pipeline parameter (relating to conflict-induced delays, described in section 1.2.1).

Using reasonable compile-time time limits, the algorithm we propose was found to generate provably optimal schedules for 15,812 of the 16,000 synthetic benchmark programs examined (over 98%).

1.2. Pipeline Characteristics

In describing the basic characteristics of pipelined computer systems, it is useful to consider the compiler and architecture aspects separately. Naturally, this work is more concerned with the compiler's view, however, the discussion of the architectural structures clarifies how the proposed scheduling model applies to various real machines.

1.2.1. Compiler's View

As a compiler views a pipelined machine, the main concern is simply that the order in which instructions are executed must be sensitive to various pipeline-related timing constraints. It is convenient to think in terms of the incremental task of trying to generate code for the next in a sequence of instructions.

There are two primary reasons for which execution of an instruction might need to be delayed:

- **Dependence.** A dependence occurs when this instruction uses a result computed by an earlier instruction, but the earlier instruction has not yet completed pipelined execution. Violating a dependence generally results in incorrect results being computed.

- **Conflict.** A conflict occurs when this instruction requires access to a hardware structure which is still being used by the pipelined execution of an earlier instruction. An unresolved conflict results in a pipeline hazard and unpredictable behavior.

Dependence is the most common reason for requiring delays. For example, loading a datum from memory into a register might be an instruction which takes
4 clock ticks to execute, but the very next instruction might depend on the value being loaded. Consider typical code implementing the addition of \( X \) to register \( R0 \):

\[
\begin{align*}
\text{Load} & \ R1, X \quad ; \text{make register} \ R1 = \text{memory}[X] \\
\text{Add} & \ R0, R1 \quad ; \text{make register} \ R0 = R0 + R1
\end{align*}
\]

If the hardware were simply to enqueue the load in the pipeline and, in the very next cycle, attempt to use the register, the wrong value would be obtained; hence, some technique must be used to prevent the second instruction from executing until after the first has completed. This would introduce a delay of 3 clock ticks between the \text{Load} and \text{Add} instructions.

Notice that traditional compiler code generation techniques tend to load values on demand, resulting in code sequences which have many such dependences.

Modifying the above example, a conflict would arise instead of a dependence if the second instruction is another \text{Load} instruction and, for example, the hardware required the memory address register (MAR) to hold the memory address being accessed for the first 2 clock ticks of the \text{Load} operation. Consider:

\[
\begin{align*}
\text{Load} & \ R1, X \quad ; \text{make register} \ R1 = \text{memory}[X] \\
\text{Load} & \ R2, Y \quad ; \text{make register} \ R2 = \text{memory}[Y]
\end{align*}
\]

In this case, the second \text{Load} would have to be delayed until the first \text{Load} had finished using the MAR — a delay of 1 clock tick would have to be placed between the two \text{Load} operations.

Hence, there is a significant difference between dependence-induced and conflict-induced delays: beside the semantic differences, they generally do not imply the same amount of delay. For each pipeline, the compiler needs to be aware of two separate parameters corresponding to the delay times seen for dependence and conflict resolution, respectively:

- **Latency.** The pipeline latency is the number of clock ticks which must occur between enqueuing an operation in a pipeline and the result of that operation becoming available. In other words, it is the minimum time between issuing an instruction and issuing a second instruction which has a
dependence on the first; the "depth" of the pipeline measured in units of time.

- **Enqueue time.** The pipeline enqueue time is the minimum number of clock ticks which must occur between enqueuing one operation in a particular pipeline and enqueuing a second operation in that pipeline. In other words, it is the minimum time between items in a pipeline.

For a classical pipeline, the latency is a few clock ticks and the enqueue time is 1 clock tick (since each stage of the pipeline uses functional units independent from those of other stages). However, it is not uncommon to find hardware being shared by a few pipeline stages (or, equivalently, to find each stage taking a few cycles). Further, machines which have functional units that can operate in parallel with other functional units but are not internally pipelined are easily modeled by making each functional unit appear as a pipeline where the enqueue time \(=\) latency.

The fact that some architectures have multiple pipelines raises yet another issue in the compiler's management of pipelined systems: the compiler may have to decide which of several viable pipelines to use for each operation. For example, in a machine with two pipelined multipliers, which multiplier should be used for each operation?

### 1.2.2. Architecture's View

In the compiler's view we identified the causes of execution delays, but we did not define their architectural implementation. When a dependence or conflict would otherwise cause improper execution, the architecture must have some mechanism for introducing the appropriate delay. In discussions of pipelined hardware, these delays are sometimes referred to as "pipeline bubbles" [Pat85]. There are three basic approaches to forcing a delay:

- **Implicit interlock.** In this technique, the hardware checks each instruction just before execution to make sure that it does not depend on the results of any operations which are currently in the pipeline. If there is such a conflict, the hardware simply delays issuing the instruction until the conflicting operation in the pipeline has completed.
The implicit interlock approach has long been the standard approach. It continues to be used in most modern processors, including RISC-style architectures such as the IBM 801 [Rad83], RISC II, and SPARC [Gar88] architectures.

- Explicit interlock (explicit waiting). In this technique, the compiler marks each instruction with a tag indicating whether it must wait for a particular pipelined operation to complete before this instruction can begin executing. This technique is very similar to an implicit interlock, however, the hardware is simpler since it does not need to detect which operations interfere.

  The machine being developed by Tera [Smi88] uses an explicit interlock based on the compiler tagging instructions with a count field which gives the number of instructions since the last instruction that this instruction depends on or conflicts with. Another example of explicit interlock is the proposed CARP machine [DiS89]; CARP uses a bit mask in each instruction to indicate which variable-latency resources (e.g., global memory accesses using an interconnection network) each instruction must wait for.

- NOP insertion (padding). In this technique, the compiler takes full responsibility for the management of the pipeline by simply placing NOP (Null Operations — instructions known to be non-interfering with any type of pipeline activity) between instructions which would otherwise result in pipeline conflicts. The hardware is the simplest of the three techniques, but the compiler must perform analysis of the pipeline activity implied by the code.

  The best known example of NOP padding for introducing delays is probably the MIPS processor [Hen81], although this seems to be becoming more popular as a general approach. For example, much of the work toward GaAs processors uses NOP padding. Further, pipelines with fixed latency are handled in this way in the CARP machine [DiS89].

1.3. NOPs and Delay Slots

In code scheduling, for a pipeline processor, the best solution is to never have the next instruction interfere with the instructions currently in the pipeline. By pipeline analysis and rearrangement — scheduling — of the code, a compiler can
effectively eliminate the need for inserting delays. But when no instruction can be found to replace a delay slot then it becomes necessary to "execute" these delays. It should be pointed out here that we can implement these delay slots in a variety of ways such as Implicit interlock, explicit interlock or NOP padding as described above.

The current popularity of the NOP insertion technique is, probably to a great extent, the result of the realization that this scheduling is important enough that every compiler should do it, in which case the compiler technology for NOP insertion is free, whereas the hardware implementing an interlock is not.

In this thesis, for convenience, we shall consistently refer to delays in terms of inserting NOPs. However, the approach is not sensitive to which hardware mechanism is being employed. This is a key reason for discussing the architecture's view — to show that it is in fact orthogonal to the compiler's view. Hence, the scheduling techniques discussed in this thesis apply equally well to any architectural implementation of delays. In fact our algorithm, that is presented in Chapter 3, is based on the general notion of delay slots and is not specific to NOP padding. The choice of the method used to make those delay slots visible to the processor is up to the person implementing this algorithm. Our implementation uses the NOP insertion technique, for reasons noted above.

1.4. An Overview of This Document

Chapter 1 provides an introduction to the problem of code scheduling for multiple pipeline processors. We have presented the problem from the Compiler perspective and from the Architecture point of view and concluded that these issues are orthogonal and that the code scheduling should be incorporated in every compiler even if other methods are used to resolve pipeline conflict and dependency problems.

The background material and a survey of related research in open literature is presented in Chapter 2. This chapter begins with an overview of the complexity of the code scheduling problem viewed as an exhaustive search problem. This is followed by a compendium of various algorithms proposed by other researchers and how they differ from our work.
Chapter 3 presents a concrete illustration of the concepts and rationale behind our proposed algorithm. Later in the chapter, a detailed problem statement is defined and is followed by a description of our algorithm. The chapter ends with a formal proof about the optimality of the solutions obtained by our algorithm.

Chapter 4 addresses various issues pertaining to the implementation of our algorithm and its integration with existing compilers. The structure of our prototype compiler and implementation of the algorithm are discussed and the basic characteristics of pipelined systems are reviewed with examples.

Performance analysis of our algorithm (through its implementation) is carried out in Chapter 5. Interaction between various system parameters is explored and compared with the expected behavior.

Finally, Chapter 6 presents conclusions and directions for further research.
CHAPTER 2

BACKGROUND AND SURVEY OF RELATED LITERATURE

2.1. Introduction

In this chapter we discuss some of the work done by other researchers in this area. Probably the best known example is the work of Thomas Gross, which is discussed in Section 2.2. In Section 2.3, we investigate the work done by David Bernstein. The contributions of Abraham and Padmanabhan are reviewed in Sections 2.4 and 2.5. Differences between their work and our approach, and other general conclusions, can be found in section 2.6.

2.2. An Example of Code Scheduling

In the previous chapter, we reviewed the concepts of pipeline conflicts and instruction dependence issues. Figure 2.1 gives a concrete example of code scheduling to resolve these problems. Suppose that this code is to be run on a processor that has a memory load delay of one machine instruction. In other words, the result of a memory load operation becomes valid two machine instructions after its initiation. Assume that all other instructions take one machine instruction. Then clearly, this piece of code will produce incorrect result. This is because the value of R2 used by the Add instruction will not be what is intended in the program.

We need proper delay before the execution of Add instruction. One easy way to implement this is by placing NOP instructions to fill the delay slots. Now the code sequence will produce a correct result on this processor. This sequence is shown in Figure 2.2.
The example code sequence requires one delay slot for proper execution. The time wasted by this delay slot can be utilized in executing some other instruction at that spot. So effectively, we can “fill” the delay slots in the code with other instructions in the code sequence. When no instruction is found that can move to the position of the delay slot without violating the legal order of execution or pipeline usage (conflict), we simply place a NOP there. Note that the legal order of execution implies an ordering of instructions such that no consumer of a value comes before the producer of that value. A code schedule that eliminates the delay slot before the Add instruction is shown in Figure 2.3.

An optimal code schedule would be the one with the minimum possible number of delay slots in it. To efficiently find such schedules is our goal in this thesis. In the next section, we throw some light on the complexity of finding an optimal schedule for pipeline processors.
Ld R2, [Z]
Ld R1, #5 ; replace the delay slot
Add R3, R1, R2
St [X], R3 ; X = Z + 5

Figure 2.3. Instruction Sequence without any Delay Slots

2.3. The Complexity of Finding An Optimal Schedule

The problem of finding an optimal code schedule for pipeline machines is well known to be NP-complete [Gro83a]. The problem of instruction scheduling for a program, given set of pipeline constraints, is typically handled by compiling the program into assembly language instructions. These instructions are then grouped into basic blocks [AhS86] and each basic block is independently scheduled for the given pipeline constraints.

Without employing any pruning, as is clear intuitively, finding the optimal schedule for a block of $n$ instructions requires an exhaustive search of all $n!$ possible schedules. It is convenient to think of this as requiring $n!$ invocations of an $O(n)$ procedure, which we call $\Omega$, that generates a schedule of the $n$ instructions and computes the number of NOPs required by that schedule.

As discouraging as these complexity measures sound, we continued to determine the approximate time one might expect for a compiler to schedule a typical block containing about 15 instructions. A reasonably efficient C implementation of the procedure $\Omega$ was created and its approximate runtime determined on a variety of machines. The average time for one application of $\Omega$, including the call overhead, was 0.12 milliseconds on a heavily-loaded Gould NP1. For a Sun 3/50 workstation the average time was about 0.3 milliseconds. Given a block containing 15 instructions, $\Omega$ would be applied 15!, or 1,307,674,368,000, times. Hence, our typical 15-instruction block could be scheduled on an NP1 in a mere 158,920,924 seconds — just under 5 years! Worse still, most programs

---

1 Interactions between adjacent blocks can be managed without major modification of the basic block schedules, essentially by modifying the initial conditions in the analysis for each block.
contain many such blocks. An interpolation of the average runtimes for different sized basic block is shown in Table 2.1. Column one in this table shows the size of basic block in terms of the number of instructions (after other classical optimization and dead code removal has been done). The second column gives the number of search calls for an exhaustive search algorithm. Obviously, this number is the factorial of the size of basic block. The third column shows the approximate time required to execute these many calls to find an optimal solution.

Table 2.1. Search Space for Exhaustive Search

<table>
<thead>
<tr>
<th>Instructions In Block</th>
<th>Exhaustive Search Ω Calls</th>
<th>Approximate Runtime for Ω Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5,040</td>
<td>0.6 seconds</td>
</tr>
<tr>
<td>8</td>
<td>40,320</td>
<td>4.8 seconds</td>
</tr>
<tr>
<td>9</td>
<td>362,880</td>
<td>43.5 seconds</td>
</tr>
<tr>
<td>10</td>
<td>3,628,800</td>
<td>7.2 minutes</td>
</tr>
<tr>
<td>11</td>
<td>39,916,800</td>
<td>79.8 minutes</td>
</tr>
<tr>
<td>12</td>
<td>4.8×10^6</td>
<td>15.9 hours</td>
</tr>
<tr>
<td>13</td>
<td>6.2×10^6</td>
<td>8.6 days</td>
</tr>
<tr>
<td>14</td>
<td>8.7×10^10</td>
<td>121.1 days</td>
</tr>
<tr>
<td>15</td>
<td>1.3×10^12</td>
<td>5.0 years</td>
</tr>
<tr>
<td>16</td>
<td>2.1×10^13</td>
<td>79.6 years</td>
</tr>
<tr>
<td>17</td>
<td>3.6×10^14</td>
<td>1353.5 years</td>
</tr>
</tbody>
</table>

No doubt, it is this type of analysis which led researchers to sacrifice optimality and investigate heuristic scheduling techniques. However, all is not as bleak as it seems because many of the schedules can be pruned from the search. Our approach was simply to prune the search as much as possible without sacrificing optimality. The most obvious pruning of the schedule search space is to avoid consideration of any orderings which would result in incorrect execution
due to violating a dependence (i.e., making the consumer of a value execute before the producer of that value).

One question arises at this point that why other researchers did not use this approach to find the optimal solutions? Probably, they realized that the worst-case time complexity of this approach is still exponential, therefore this idea was deemed to be useless for practical compilers. An excerpt from [HeG83] summarizes this — “Since we have shown that the reorganization problem is NP-complete even for the case where [pipeline] interlocks are only one or two instructions long, we need to consider heuristic solutions [foregoing the optimal solution].” On the other hand, we investigated the average runtime for a refined exhaustive search algorithm and also studied the frequency of the occurrence of its worst-case performances. Moreover, we also formulated and implemented a number of other heuristics which pruned the search space significantly without sacrificing optimality. From this empirical study we found that for typical inputs (similar to what occur in real programs), nearly all of the inputs resulted in optimal schedules within very reasonable runtimes.

Table 2.2 presents a sample of how well we were able to prune the search space for schedules for typical blocks. All these examples are representatives of original test samples. The nature of these sample inputs is described in Section 5.3.

Note that for the same block size there can be great variations in the number of calls required to perform optimal scheduling. This is true because the search space is proportional to the nature of inter-dependencies within a basic block, and is independent of the basic block size. However, the search space in general increases with the size of basic blocks. This is because of the fact that the range in which instructions can move and still have a legal evaluation order, depends on the inter-dependencies and the size of the basic block. In Table 2.2, some basic block sizes appear more than once to illustrate the variations in the runtime for the same block size. Note that this Table is presented here only to highlight the remarkable difference between the number of calls (to procedure Ω) required to schedule various basic blocks using our pruning techniques. An extensive set of

---

2 Although they did use similar exhaustive search methods to compare the results of their heuristics with the optimal solutions.
results is given in Chapter 5. From those results it follows that the same typical 15-instruction block that would have taken 5 years to schedule optimally can be scheduled optimally in an average of about 0.01 seconds using the proposed pruning techniques.

Of course, despite the fact that our pruning works very well on average, it has an exponential worst-case performance. To limit the worst-case runtime for our algorithm, the concept of a curtai point $\lambda$ is used. This is a user-supplied parameter specifying the maximum number of schedules to be considered. The proposed scheduling algorithm terminates when either:

[1] All possibly-optimal schedules have been examined\(^3\). In this case, the best

\(^3\) Our search algorithm will sometimes prune optimal schedules from the search, but only if they are provably equivalent to a schedule which was not pruned.
schedule found is an optimal schedule.

[2] A total of λ schedules have been examined (i.e., λ calls have been made to Ω). Because some possibly-optimal schedules have not been examined, the best schedule found might or might not be an optimal schedule.

Fortunately, our results show that the vast majority of all blocks will terminate on case [1] if λ is on the order of 1,000. In fact, for most blocks of fewer than 20 instructions, a λ value of about 50 would suffice. Using the algorithms and synthetic benchmarks described in detail later in this paper, the search for 15,812 of the 16,000 blocks terminated on condition [1]: the number of schedules searched for each of these trials is plotted in Figure 2.4.

In the case that a reasonable λ is exceeded and the search is truncated by rule [2], a sub-optimal solution might result. We were generally unable to determine how often the schedule resulting from a truncated search is actually optimal despite the fact that some schedules were not considered. This is due to the fact that when a reasonable value of λ was exceeded, the search space tended to be very large, so that even increasing the λ value by a factor of fifty did not cause the search to run to completion — however, neither did the best schedule change. For this reason, we suspect that many of the truncated searches also found optimal or nearly optimal solutions, but we cannot yet prove this.

Note that the total number of legal schedules which must be searched derives primarily from the dependence and conflict properties of instructions within the block rather than from the block size.

Having presented the basics of our work, we compare our research with the relevant work done previously in the literature.

2.4. PostPass Code Optimization

The Stanford University Microprocessor without Interlocked Pipeline Stages (SU-MIPS) was one of the first projects to integrate VLSI computer design and compiler design. Migration from hardware to software (e.g., compilers) was sought whenever possible without any performance degradation. Pipeline synchronization using interlocks is performed by the compiler, thus no hardware interlocks exits.
Figure 2.4. Schedules Searched Vs. Block Size Vs. Distribution of Inputs
There were many issues explored in that project, like instruction packaging, delayed branches, instruction set design etc ([GrH82], [GiG83] [Gro83] and [GrH88]). But we will discuss here only the code scheduling algorithm [Gro83a] for the pipeline interlocks (terms code scheduling and reorganizing are use interchangeably in this section). This algorithm works on the assembly level instructions that have been generated by an earlier phase of the compiler. First we take a brief look at how the problem of code scheduling for pipeline processor is proved to be NP-complete, then we summarize the algorithm along with the various heuristics and how it differs with our work.

2.4.1. Proof of NP-Completeness

In his Ph.D. dissertation [Gro83a], Thomas Gross has shown that the problem of optimal reorganization of machine-level instructions at compile time is NP-complete. This is done by first showing that the problem is NP-complete when an unbounded pipeline interlock length is a parameter to the problem. And then to show the strong NP-completeness of the problem, it is proved to be NP-complete even when the interlock length is limited to one or two, and only one register is used in the original schedule. And finally it is stated that the problem is in NP, since an optimal solution can be found non-deterministically by trying all possible solutions.

The problem of pipeline scheduling with unbounded interlock length is equivalent to a precedence-constrained multiprocessor scheduling problem. A sequencing problem computes an optimal single-processor execution sequence for a series of tasks under certain constraints. A sequencing problem is NP-complete only under a well-defined set of restrictions. Scheduling problems for multiprocessors are NP-complete even with fairly simple restrictions. A multiprocessor scheduling problem deals with finding a schedule of a set of tasks using more than one processors. Gross has shown that the pipeline interlock restriction effectively makes the reorganization problem equivalent to a multiprocessor scheduling problem.

For a real processor, there is always a bound on the interlock length and the resulting reorganization problem would not necessarily be NP-complete. The strong NP-completeness of the problem is claimed by deriving an equivalence between the reorganization problem with interlock length one and two and at
least one register, and a resource scheduling problem that is already known to be
NP-complete. The reorganization problem could be constructed from the
resource scheduling problem in polynomial time, therefore the reorganization
problem is NP-complete. And since the optimal solution can be obtained by
guessing at every possible sequence, which can be evaluated for legality and cost
in polynomial time, thus the problem is in NP.

Although the algorithm presented in this thesis works on an intermediate
form of code, instead of the assembly level code for which the scheduling
problem was shown to be NP-complete, the same proof applies. This is because
the only difference that is visible in problem formulation is that of machine-level
registers. In the intermediate form memory-variables can be thought of as
registers without any loss of generality. In fact this increases the runtime of the
reorganization algorithm because it removes precedence constraints that are
present when a limited set of registers are allocated to a code sequence. Hence,
the problem of code scheduling (reorganization) for intermediate level code is also
NP-complete.

2.4.2. The Algorithm

Thomas Gross implemented a postpass code reorganizer that resolves the
problem of pipeline interlocking by inserting NOPs (Null OPerations) in the code
to fill load and branch delay slots, reordering the resulting code to eliminate
pipeline dependencies, removing as many NOP instructions as possible, and
packing. Reordering is done on code within a basic block. The branch instruction
at the end of the block can not be moved; it has to remain the last instruction of
the reordered block. In a subsequent phase, which we will not discuss here,
instructions around the branch instruction may be moved to fill the branch
delays.

He proposed a heuristic algorithm for the code scheduling:

[1] Read in a basic block and create a machine-level DAG.
[2] At any point, determine the set of instructions that can be generated.
The first step shows that this algorithm works on machine level basic blocks, one at a time. Steps [2] and [3] determine valid instructions that can be scheduled next according to the constraints given in Section 2.4.2.1. In step [4] heuristics to select a set of instructions and partial scheduling of that set of instructions are considered. This is described in Section 2.4.2.2.

2.4.2.1. Reordering Constraints

The following reordering constraints are applied to compute the set of legal instructions:

[1] All children⁴ are evaluated before their parents.

[2] All uses of a register or memory value are completed before that value is altered.

[3] Loads and stores to memory are maintained in their original order whenever they could refer to the same address. This information easily can be determined, or can be provided by the preceding phase of the code generator.

[4] If a node stores into a register and that value is used in another basic block, then that store must be the last store to the register. This may be alternatively stated as: if a register value is live at the end of the original basic block, all legal evaluation orders must leave it live.

One observation is immediately obvious. That is, the postpass scheduling is limited by the existing register assignments which are fixed before the scheduling starts. The scope of reorganization done at this level is limited because the assembly code (in general) reflects the assignment of values to a limited number of registers based on the initial ordering of the instructions in the source program.

Hence, in the constraints on instruction reordering given above, constraints 2 through 4 are trivially met if reordering is done on the intermediate code before register allocation phase. Also, any aliased memory references are not seen by the reorganizer, and therefore cannot be exploited. In intermediate code, the

⁴ The sense of direction for DAGs in our work is opposite to this. A parent node comes before children nodes in our representation of DAGs. Though, the difference is in terminology only.
compiler can be made to use analysis and renaming so that these complications need not hinder scheduling [Die87].

The constraints given above greatly reduce the freedom with which the individual instruction is a code stream can move and thus the reorganized code, in general, is not as good (in terms of the number of NOPs inserted) as it could be. This is illustrated in the example code sequence given in Figure 2.5, which is the best schedule given this register allocation. However, the same sequence with a new register assignment eliminates all NOPs, as shown in Figure 2.6. Hence, by using an intermediate code which allows any register assignment, better code sequences can often be found.

```
Ld R0, [ B ]
Ld R1, #5
Add R1, R0, R1
St [ A ], R1 ; A = B + 5
Ld R1, [ D ]
NOP ; delay slot
Add R1, R0, R1
St [ C ], R1 ; C = B + D
```

Figure 2.5. Best Code Sequence for a Given Register Assignment

The code scheduler suggested and implemented by Gross works on the assembly level instructions that are already generated by other phases of compiler. Instruction scheduling at code-generation time (before register allocation phase) was considered inappropriate for the following reasons:

1. Instruction Scheduling tends to increase register lifetime, making it more difficult to obtain a "good" register allocation. The cost of spilling a register may easily exceed the cost of an interlock or inserted NOP.

2. The scheduling may be difficult to perform prior to register allocation and final instruction selection. In machines with multiple addressing modes and instruction formats, the exact instruction to be used to
Ld  R0, [ B ]
Ld  R1, #5
Add R1, R0, R1
St  [ A ], R1 ; A = B + 5
Ld  R2, [ D ]
Add R1, R0, R2
St  [ C ], R1 ; C = B + D

Figure 2.6. Improved Code Sequence

implement a particular function and the interlock properties of that instruction may not be determined until after the register allocation is known (thus after scheduling).

3. The code generator can not be readily applied to assembly-language programs.

In our view, the choice of register allocation before scheduling is not a good one and the reasons summarized above are not necessarily true in all cases. For example, with reference to the statement number 1 above, there is no reason why register allocation and pipeline scheduling can not be mixed in a single scheduler that can perform cost comparisons between register spill and pipeline delays. Similarly, the second statement is not applicable to modern RISC style machines.

2.4.2.2. Heuristics

In this section, we take a closer look at the different heuristics Gross devised for the scheduling problem and how they compare with optimal solution.

The algorithm is based on the idea of safe paths. A safe path for a resource \( r \) with a starting node \( i \) in a given DAG \( D \) with a set of generated (covered) node (instruction) \( d \) is either \( i \) or a minimum set of unscheduled instructions such that the set contains all unscheduled descendants of \( i \) and we obtain a safe position with \( d \). A safe position for resource \( r \) is a set of instructions \( S \) in the DAG such that, once code has been generated for all the nodes in \( S \), the nodes in \( S \) do not effect the generation of code with respect to resource \( r \) for the remaining instructions in the DAG that are not in \( S \). Detailed definition and explanation of
safe paths may be found in [Gro83a].

The proposed reorganization algorithm is a constraint algorithm with heuristics added to choose between conflicting safe paths. These heuristics do not in general return an optimal solution. Recall that, in step [4] of the algorithm described earlier, there is a choice to be made between different candidates for scheduling. Once a choice has been made there is no backtracking, and thus the quality of the solution heavily rely on the criteria for choosing among the candidate sets. Three different heuristics were proposed:

1. Choose the largest safe path. The assumption is that the number of pipeline conflicts is proportional to the number of nodes in the safe path.
2. Choose the safe path that has the highest interlock penalty. This cost can be evaluated by counting the number of instructions that can interlock.
3. Choose the safe path that starts with the node farthest from the root. This strategy uses a simple criterion that is known to work well for other scheduling problems.

When there are several safe paths with the same weight (as defined by one of the three heuristic strategies described above), the safe path whose start node appears first is chosen.

2.4.2.3. Results

Clearly, none of the heuristics attempt to find an optimal solution. The empirical results reported in [Gro83a] are reproduced in Table 2.3. These results are obtained by the application of postpass reorganizer to a set of different programs. Only the average values are shown in Table 2.3. Strategy 3 was chosen for the final version of the reorganizer. The reorganizer produces “good” results based on these heuristics. We conclude this section with a note that the work of Thomas Gross is fairly good in terms of the integration of compiler and architecture concepts and the implementation is quite reasonable.
Table 2.3. Percent of NOPs Required for Different Heuristics

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Strategy 1</th>
<th>Strategy 2</th>
<th>Strategy 3</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>4.3</td>
<td>3.9</td>
<td>4.0</td>
<td>3.4</td>
</tr>
</tbody>
</table>

2.5. Improved Approximation Algorithm by David Bernstein

David Bernstein proposed an improved approximation algorithm for scheduling instructions for pipeline machines [Ber88].

2.5.1. Background

A class of scheduling algorithms, called leveling algorithms, is defined and analyzed. The basic leveling algorithm has been improved so that the worst case ratio of the length of a schedule generated by the algorithm over the length of an optimal schedule is better than what is achieved by general list scheduling algorithms. This upper bound for this ratio is $2 - 1/(d+1)$ for list schedules and is refined to $2 - 2/(d+1)$. In these expressions $d$ is the amount of delay, for an instruction using a pipeline, after which the result become valid.

The time complexity of the refined leveling algorithm is $O(n\alpha(n) + e \log n)$ where $n$ is the number of instructions, $e$ is the number of dependencies among the instructions, and $\alpha(n)$ is a very slow-growing function.

In his research, approximation heuristics are used and the worst case behavior of the algorithm is analyzed.

2.5.2. Algorithm

The schedule model considered in this research work consists of a single processor $P$ and a job system $T = (J, D, G)$. $T$ is a set of unit time execution tasks (or instructions) $J = \{J_1, \cdots, J_n\}$, a set of delays (which model the pipeline structure) $D = \{D_1, \cdots, D_n\}$, where $D_i \in \{0, \cdots, d\}$ for some fixed integer $d$, and a directed graph $G = (J, E)$ of precedence constraints. Let $IS_i$ be the set of immediate successors of $J_i$. 
Define level $l(J_i)$ of a task $J_i$ as:

$$l(J_i) = \begin{cases} 
0 & \text{if } J_i \text{ has no immediate successors} \\
D_i + \max_{X \in I_s} l(X) & \text{otherwise}
\end{cases}$$

A priority list $L$ of the tasks (instructions) is constructed in a non-increasing order of their levels. A schedule $S$ corresponding to such an $L$ is called a leveled schedule. A refined leveling algorithm that improves on the upper bound of list schedule is then introduced.

Let refined level of task $J_i$ be denoted by $rl(J_i)$ and let $M_i = rl(J_{i_1}), ..., rl(J_{i_{|s_i|}})$ be a sequence on non-negative integers constructed from the refined levels of the immediate successors of $J_i$ in a way that $rl(J_{i_1}) \geq \cdots \geq rl(J_{i_{|s_i|}})$. Then, $rl(J_{i})$ is defined recursively as follows:

$$rl(J_{i}) = \begin{cases} 
0 & \text{if } J_i \text{ has no immediate successors} \\
D_i + Q_i & \text{otherwise}
\end{cases}$$

where $Q_i = \max(rl(J_{i_1}), rl(J_{i_2}) + 1, ..., rl(J_{i_{|s_i|}}) + |S_i| - 1)$

The refined level schedule is generated according to the following algorithm:

1. Compute the levels $l(J_i)$ for all $i$.
2. Compute the refined levels $rl(J_i)$ for all $i$.
3. Create a priority list $L$ by first ordering $J_i$ in a non-increasing order of $l$, and then ordering the jobs with the same value of $l$ in a non-increasing order of $rl$. The order among the jobs of the same level of $l$ and $rl$ is arbitrary.

Again, we note that this is an approximation algorithm designed to obtain solutions with an upper bound of $2 - 2/(d+1)$ on the worst case ratio of the length of a schedule generated by the algorithm over the length of an optimal schedule. As mentioned earlier $d$ is the amount of delay, for an instruction using a pipeline, after which the result becomes valid.

One restriction that this research imposed is to limit the instruction delay $d$ to be the same for all instructions. Therefore the algorithm and results presented do not apply to multiple pipeline machines. Our approach, on the other hand, also takes into account different pipeline delays for different instructions. In the next section, we discuss another approach that consider multiple pipeline
machines with variable delays.

2.6. Reorganizer for a Variable-Length Pipelined Microprocessor

The implementation of an instruction reorganizer for a floating point microprocessor with variable-length pipeline is described in [AbP88]. This work was done by Seth Abraham and Krishnan Padmanabhan. Some Benchmark results are presented by these authors using BLAS and Livermore Loops. The presence of variable length pipelines is described as a key feature in this work.

2.6.1. Introduction

The reorganizer is designed to work with compiler generated or hand written assembly language code. A greedy heuristic algorithm is used to reorder instructions inside basic blocks.

The reorganizer, that works at the assembly language level, can accept guidelines or directives from either the compiler or the assembly language programmer about data dependencies and memory aliasing.

2.6.2. The Algorithm

The input to the reorganizer is a sequence of assembly language instructions which are broken down into a set of basic blocks. Then the reorganizer schedules the instructions within a basic block as the first phase. In the second phase the dependencies between the blocks are resolved. Although the reorganizer resolves data dependencies between basic blocks, but information is used only to add NOPs to the end of the ancestor basic block or at the beginning of the descendent basic block. But the instruction ordering is driven essentially by instruction dependencies within a basic block and that order is not changed by the inter-basic block analysis. That phase exists only to prevent any pipeline conflict when a stream of basic blocks is run in succession.

A set of four lists is maintained for performing this algorithm. These lists are:

AII. Active Instruction List. At any point in time, the active instruction list contains a window of instructions that have been reorganized and sequenced., along with the maximum and minimum completion
times of the instructions. This list corresponds to a window of instructions that could exist inside the pipeline at run time.

**RAL** This is the Resource Allocation List associated with AIL. Determining whether one instruction can safely follow another at a certain distance requires a test for dependencies; thus each instruction in the AIL or DIL has a list of resources that it will use as a source or destination. These are RAL and RRL for lists AIL and DIL respectively.

**DIL** Deferred instruction List. This contains an ordered list of instructions that can not be scheduled safely (at some point in time).

**RRL** The Resource Requirement List that is associated with DIL.

A greedy algorithm is used to order the instructions in each basic block. Again, the goal is not to find an optimal solution, instead the algorithm uses greedy heuristics to find an "approximate" solution in a reasonable run time. The starting point is an empty AIL, and a DIL containing the entire block. Then apply steps 1 and 2 until DIL is empty.

1. Sequentially go down up to $k$ instructions in the DIL and get the first instruction which may be safely scheduled at this point. If an instruction is found, insert it into the AIL and also into the reorganizing sequence for the basic block. If no such instruction is found and the DIL is not empty then insert a NOP. If DIL is empty, exit the algorithm. $k$ is the lookahead distance.

2. Now cycle the AIL. This involves removing a completed instruction, if any, from the AIL. Every time an instruction is scheduled, it is necessary to cycle the AIL in order to free resources and prevent detection of outdated dependencies. This is done as follows:
   a. For all instructions in the AIL, decrement both minimum and maximum completion times. For all destination resources in the RAL, decrement both maximum and minimum use times.
   b. All items with maximum completion times decremented to zero can be removed from the list. At most one memory store instruction will have this condition true at this point.
c. If no memory store instruction was found in the last step, then from the set of all items with minimum completion time less than or equal to zero (if any), remove the one with the smallest value of this time.

The branch delays are handled in a fashion similar to [Gro83a]. Intra-block dependencies are resolved by the addition of NOPs without altering the instruction ordering in the basic block achieved by the algorithm.

Although satisfactory results are presented, this work also suffers from the artificial constraints introduced by the assembly level code generation that we discussed in Section 2.4.2.1.

Two more examples of pipeline code scheduling implementations are given in the next sections, without going into the details of the respective algorithms.

2.7. Micro-Optimization of Floating-Point Operations

William Dally described a technique in [Dal89] for reducing the operations count and time required to perform floating-point calculations on pipeline floating-point function units. This work is an effort to integrate floating-point arithmetic into RISC computer architecture. Micro Floating-Point function units are proposed that break down the floating-point tasks. These are pipelined and hence the original task can be divided into micro-operations which can be scheduled allowing for overlapping between instructions depending upon the interdependencies.

A greedy heuristic algorithm is presented that schedules instructions and try to fill pipeline delayed slots with other instructions. In addition to that, redundant re-normalizations are eliminated by the scheduler.

2.8. Scheduling Trees in Pipelined Environments

Scheduling task trees to be executed in parallel and/or pipelined processing systems are examined under individual situations in [LIH77]. Simple optimal algorithms are presented for special cases for task tree structures. Some simple techniques for binary trees for parallel pipeline models are also discussed.

The author has shown that for simple precedence structures in the form of a tree, scheduling for pipeline and/or parallel systems is a NP-Complete problem. Heuristic approaches are favored and the search for optimal solutions is
discouraged by presenting counterexamples of exponential complexity for optimal solutions.

2.9. Summary

As discussed earlier, the few pipeline scheduling algorithms presented in the literature act as postpass reorganizers, and work on the assembly level produced by the compiler. Doing so imposes unnecessary constraints that sacrifice optimality of the solution. Moreover, all techniques rely on heuristics to obtain solutions and do not attempt to find solutions which are optimal (even given a fixed register allocation). In contrast, our approach which is discussed in the next chapter works at an intermediate code level and uses a new set of pruning criteria which preserve optimality.

The important point to note is that the advantage of our algorithm is that it finds optimal solutions for typical inputs. For a very small percentage of the inputs, our algorithm does not guarantee the optimality of the solution, but we have found those solutions to be close to optimal. Hence, although we did not benchmark the other algorithms, we suspect that typical performance of our technique is superior.
3.1. Introduction

As discussed in the previous chapter, most of the algorithms in the open literature perform code scheduling for pipelined machines using heuristics that do not preserve the optimality of solution. Our approach, which is described in this chapter, is different from other related works in several ways:

[1] We apply our algorithm to an intermediate form of code instead of the final assembly code.

[2] Our algorithm employs pruning techniques which preserve the optimality of solution. Unless the search is truncated (which happens rarely) the solutions are guaranteed to be optimal.

[3] We allow for a pipeline architecture model that is significantly more general than that of other algorithms.

The problem of finding an optimal code schedule for pipeline machines is well known to be NP-complete. We studied the complexity of the problem to investigate if it is possible to find optimal solution (for most cases) in a reasonable time. The results of this investigation, which are given in the next sub-section encouraged us to further explore the search for optimal solution. Finally, we came up with an algorithm that finds optimal schedules (for most cases) in a very reasonable time. Moreover the solutions that it finds that are not guaranteed to be optimal are very good solutions (comparable to the optimal solutions themselves) are certainly at least as good as the solutions obtained using other heuristic algorithms in the literature. In Chapter 5, we see that over 98% of times our algorithm is successful in finding optimal code schedules.
The basics of the exhaustive search are described in the next section, which is augmented by a discussion of various refinement techniques and how they effect the search space for a scheduling problem. Although our strategy in this thesis is to schedule an intermediate form of code, for the purpose of illustration most of the examples in this chapter are restricted to machine level instructions. The intermediate form code in terms of instruction tuples is introduced in the next chapter.

In Sections 3.3 we present a formal description of the problem studied in this thesis. Various definitions involving the development of our algorithm are formulated. A brief description of the compiler model, which is described in detail in the next chapter, appears in Section 3.4. And finally, our proposed algorithm is presented in Section 3.5. We conclude this chapter with a set of proofs on the quality of the solutions obtained through our algorithm.

3.2. Refined Exhaustive Search

An optimal schedule for the problem of code scheduling for pipeline constraints can be found by examining all possible orderings. Clearly, the obvious implementation of this approach would be impractical because of the factorial nature of the complexity of the problem. However we have found that this problem is amenable to solution by a refined backtracking search over a tree of all possibilities. The basic idea in refined backtracking is to reduce the size of the search tree as much as possible such that the resulting minimal tree is guaranteed to contain at least one optimal solution. This is substantiated by the results obtained by our proposed refined backtracking algorithm which are given in Chapter 5.

General backtracking works by continually trying to extend a partial solution. At each stage of the search, if an extension of the current partial solution is not possible, we "backtrack" to a shorter partial solution and try again. Since we are only interested in code sequences with orderings that result in a legal schedule, as mentioned in the last chapter, we need not consider the solutions comprised of illegal schedules. Therefore, we can say that a backtracking search is equivalent to an exhaustive search of all orderings that result in a possible solution.
Consider a search tree for exhaustive search of all possible orderings for \( n \) instructions. The nodes of the tree can be thought of as sets of configurations, and the children of a node \( n \) each represent a subset of the configurations that \( n \) represents. Finally, the leaves each represent single configurations, or solutions to the problem. We may evaluate each such configuration to see if it is the best (or optimal) solution. Figure 3.1 depicts a search tree for a code sequence consisting of three instructions labeled \( \{1, 2, 3\} \). We want to find a solution \( \{a_1, a_2, a_3\} \) that minimizes the cost of pipeline induced delays. Note that there are a total of six possibilities:

\[
\begin{align*}
\{1, 2, 3\} \\
\{1, 3, 2\} \\
\{2, 1, 3\} \\
\{2, 3, 1\} \\
\{3, 1, 2\} \\
\{1, 2, 1\}
\end{align*}
\]

These solutions are obtained from the search tree by traversing all paths from the root to each leaf, picking up the labels of the nodes that are encountered. The time complexity of such an approach is \( O((n+1)!/n!) \) on an \( n \) instruction code
sequence, since we must consider $n!$ different leaves and each traversal takes $O(n)$ time. In the next section we consider a series of refinements to general backtracking technique to obtain an algorithm that is no better than the above in the worst case, but on average produces optimal results very rapidly.

### 3.2.1. Refinements

Now we will examine techniques to greatly reduce the number of possibilities tried in an exhaustive search. All these techniques involve adding tests to a simple backtracking algorithm to discover that subtrees should not be made for certain nodes. This corresponds to *pruning* the exhaustive search tree — cutting certain branches and deleting all subtrees beneath.

#### 3.2.1.1. Preclusion

One important pruning technique is to cut off the search as soon as it is determined that it can not possibly lead to a possible solution. Remember, a possible solution is a legal schedule in which precedence constraints between the instructions are maintained. For example, consider the code sequence of Figure 3.2.

```
1: Ld R0, #5
2: Ld R1, [Mem1]
3: Add R2, R0, R1
4: St [Mem2], R2
```

**Figure 3.2. Code Sequence for the Preclusion Example**

While trying all different orderings for this code sequence, it becomes apparent that the choice of instruction `Add R2, R0, R1` as the first instruction, or as a node on the first level of the corresponding search tree, *precludes* the placement of all other instructions on any of its descendant nodes because none of these instructions can be executed after the `Add R2, R0, R1` instruction. Therefore this node along with its subtrees is excluded from the search tree.
3.2.1.2. Pruning Based on Cost

This pruning rule is applied to cut off branches in a search tree whenever we can prove that pursuing the subtrees of a node will not result in a better solution than the one that can be obtained without examining the descendants of that node. We are interested in a minimum cost (in terms of pipeline delays) path in the search tree.

The basic technique is applicable to the pipeline scheduling problem because of the existence of partial solutions and also because adding more instructions (nodes) to the search path will never decrease the total cost associated with the partial solution. See Section 3.3.2 for a more formal description of this property.

In the exhaustive search without pruning, if we find that the cost of some solution is less than the cost of the minimum cost solution found so far, then we save the new solution as the best solution so far, and record its cost as the minimum cost so far for any solution. We make use of this minimum cost to exclude nodes and their descendants from the search tree. This technique can be implemented by making no search to the descendants of a node if the cost of the current partial path is greater than or equal to the best full path found so far.

3.2.1.3. Tree Rearrangement

The refinement technique discussed in the previous section is more effective if a low-cost path is found early in the search. Since the search tree arrangement depends on the initial order of the code sequence, this implies that we can rearrange the search tree by changing the order of instructions the input code sequence before starting the exhaustive search.

For example, if the paths from root to leaves are examined from left to right in the search tree, then having low-cost solutions towards the left will increase the effectiveness of the pruning technique described in the previous section. In other words, if near optimal solutions are found early in the search, then more subtrees can be cut off from the search tree based on the minimum cost function.

In the next chapter, we discuss how we obtain a good lower bound on the cost of the solutions by applying pre-scheduling that effectively rearranges the search tree.
3.2.1.4. Equivalence Test

If two or more schedules can be shown to be equivalent then we can arbitrarily choose to consider just one of them without sacrificing the optimality of the final solution. An example of this is the equivalence between two schedules that differ only in the value of constants in some instruction. For example, the following two schedules are equivalent in terms of the pipeline delays:

\[
\begin{align*}
1 &: \text{Load}(c) & 1 &: \text{Load}(c) \\
0 &: \text{Const}(3) & 0 &: \text{Const}(4) \\
4 &: \text{Const}(4) & 4 &: \text{Const}(3) \\
2 &: \text{Add}(0, 1) & 2 &: \text{Add}(0, 1) \\
5 &: \text{Add}(4, 2) & 5 &: \text{Add}(4, 2) \\
3 &: \text{Store}(a, 2) & 3 &: \text{Store}(a, 2) \\
6 &: \text{Store}(d, 5) & 6 &: \text{Store}(d, 5)
\end{align*}
\]

The equivalence between these two schedules stems from the fact that instructions (such as loading constants) which do not require any pipeline resource and are not dependent on any other instructions can swap positions with other such instructions without having any effect on pipeline conflicts and delays associated with other instructions in the schedule.

3.2.2. Combining All Refinements

Each time that we cut off the search tree at a node, we avoid searching the entire subtree below that node. For very large trees, this is a very substantial savings. In fact, the savings is so significant that it is worth while to do as much as possible when examining a node to avoid examining its children. As mentioned above, a cutoff early in the tree can lead to truly significant savings; and missing an obvious cutoff can lead to very significant waste.

In the next section we define the problem statement and other definitions that are used later to describe our algorithm, and to prove its optimality.
3.3. Problem Statement And Definitions

The scheduling model that we consider is an extension of the models used in [Ber88], [BrJ80], or [Gro83]. The key differences between their and our models are:

- Our model is general enough to allow for both the latency and the enqueue time of multiple pipeline resources. These parameters can vary from one pipeline to another, and are not limited to a single value as in [Ber88].
- This model is not specific to NOP insertion, and deals with delay slots that can be filled either by software (for example, a NOP padding compiler), by hardware instruction-waiting, or any other technique depending upon the architecture of system.

3.3.1. The Scheduling Model

Consider a task scheduling system \( T \) on a processor \( P \) having a pipeline model given by \( \Sigma \). The scheduling problem consists of a finite set of tasks (or instructions) \( Z = \{\zeta_1, \zeta_2, \cdots , \zeta_n\} \), where \( \zeta_1, \cdots , \zeta_n \) are tasks (or instructions) that are to be executed successively by the processor \( P \), and there is some precedence constraint given by a partial ordering \( \prec \) on the elements of \( Z \), and a cost function \( c_\pi(i) \); cost associated with completing task \( \zeta_i \) as the \( \pi_i \)th task in a schedule \( \pi \). We want to find an (optimal) schedule \( \pi \), representing a complete ordering \( \{\zeta_i \prec \zeta_j : i, j \in \pi\} = \{\zeta_{\pi_1}, \zeta_{\pi_2}, \cdots , \zeta_{\pi_n}\} \), that minimizes \( \sum_{k=1}^{n} c_\pi(k) \) such that no instruction is scheduled before its immediate predecessor as given by the partial ordering \( \prec \).

Our model assumes that each instruction that does not use any pipeline resource takes unit time for execution. Pipelined instructions can take any length of time for execution and it is incorporated in the Latency and Enqueue time of the pipeline employed and is described later. It should be pointed out, however, that it is trivial to introduce variable time for the non-pipelined instructions. But since most modern machines are designed to execute one or more instructions per cycle, we shall assume one instruction per cycle.

The partial ordering \( \prec \) can be defined by a Directed Acyclic Graph (DAG) \( G = (Z, E) \) with vertex set \( Z \) and \( \prec \) corresponding to the edges \( E \) in the graph due
to chronological inter-dependencies. The tuple $T = (Z, <, P)$ is referred to as a task system.

At this point it is befitting to elaborate on the pipeline resource model of the target architecture with respect to its instruction set. This determines the cost function mentioned above. Evaluation of this cost function is explained in Section 3.3.2.

The pipeline model is described by $\Sigma = (PI, LT, EN)$, where $PI$ is a set of integers from zero to $m \{0,1,...,m\}$ corresponding to $m$ pipeline resources in processor $P$. $LT$ and $EN$ are the Latency and Enqueue delay times of the pipelines and are discussed later. The cardinality of set $PI$ is equal to the total number of unique pipeline functional units that may be employed for the execution of any instruction in the complete instruction set of Processor $P$.

**Definition 1: $\sigma(\zeta)$**

$\sigma(\zeta)$ is a set of pipeline resources such that each member of this set, denoted by an integer $0...m$, represents a pipeline that may be used for the execution of instruction $\zeta$. Let $U_\zeta$ be the universe of all unique instructions (in other words, the instruction set of processor $P$). Then the function $\sigma: U_\zeta \rightarrow IP$ is an into mapping, i.e., more than one instructions may use a single pipeline resource (one at a time), or a single instruction may be executed on any one of more than one pipeline resources.

**Example:** The Add and Sub instructions in one processor may use the same pipeline resource, and in another processor there might be more than one pipeline function units just for the Add instruction (implying that more than one such instructions may overlap execution).

Now we can relate the total number of pipelines $PI$ (specific to different instructions) in a processor to the instruction set as $PI = \{ x : x \in \sigma(\zeta), \zeta \in U_\zeta \}$.

Finally, we describe the Latency and Enqueue Time characteristics of pipeline resource model. $LT$ is a set of Latency delays $LT = \{LT_1, \cdots, LT_n\}$ where $LT_i \in \{0...lt\}$ such that $lt$ is the maximum latency delay in any pipeline resource. Similarly, $EN$ is a set of Enqueue time delays $EN = \{EN_1, \cdots, EN_n\}$ where $EN_i \in \{0...en\}$, and $en$ is the maximum enqueue time for any pipelines in the system. As a convention, an instruction that does not make use of any pipeline will return a $\sigma()$ value of zero. Therefore, we set $LT_0 = EN_0 = 0$ for this "pipeline".
This completes the definition of our scheduling model. The concrete examples that follow in the next sub-section will elucidate these models.

3.3.1.1. An Example of the Pipeline Resource Model

For each hardware pipeline, the function, latency, and enqueue time must be specified. Further, so that the compiler can know which pipelines, if any, may be used to execute each type of operation, each hardware pipeline is given a unique identifier and operation types are associated with sets of pipelines. This is done using two tables.

Consider a processor with the following pipelined resources: two memory access pipelines (loaders), two adders, and one multiplier. These hardware resources are described in Table 3.1.

Table 3.1. Sample Pipeline Description Table

<table>
<thead>
<tr>
<th>Pipeline Function</th>
<th>Pipeline Identifier</th>
<th>Latency</th>
<th>Enqueue Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>loader</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>loader</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>adder</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>adder</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>multiplier</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

The second table used to describe the scheduling problem for our compiler is Table 3.2, the operation-to-pipeline mapping table. Given these tables, for example, the Add instruction has two independent pipelines available to it (namely, numbers 3 and 4), and thus can be scheduled for either pipeline. In this example, Add and Sub operations share two independent pipelines; likewise, Mul and Div share a single pipeline.
Table 3.2. Sample Operation-to-Pipeline Mapping

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>Add</td>
<td>{3, 4}</td>
</tr>
<tr>
<td>Sub</td>
<td>{3, 4}</td>
</tr>
<tr>
<td>Mul</td>
<td>{5}</td>
</tr>
<tr>
<td>Div</td>
<td>{5}</td>
</tr>
</tbody>
</table>

Notice that changing the pipeline structure changes only the entries in these tables, not the structure of the scheduling algorithm.

The pipeline structure sketched above will be described as a tuple $\Sigma = (PI, LT, EN)$, where $PI = \{0,1,2,3,4,5\}$, and $LT = \{0,2,2,4,4,4\}$, and $EN = \{0,1,1,3,3,2\}$. For this pipeline structure the formulation for $\sigma()$ is:

$$
\sigma(\gamma) = \begin{cases} 
\{1,2\} & \text{if } \text{OPCODE}(\gamma) = \text{Load} \\
\{3,4\} & \text{if } \text{OPCODE}(\gamma) = \text{Add}|\text{Sub} \\
\{5\} & \text{if } \text{OPCODE}(\gamma) = \text{Mul}|\text{Div}
\end{cases}
$$

3.3.1.2. An Example of the Task System

The following sequence of instruction tuples are to be scheduled for minimum pipeline delay on a processor with a pipeline structure given in the previous section.

1: Const(#5)
2: Ld(M)
3: Add(1, 2)
4: St(X, 3)
At this stage the reader is expected to derive a meaning of this code sequence intuitively. Any discussion about the definition of these tuples is deferred until next chapter.

The corresponding task system is \( T = \{ Z, <, P \} \) where \( Z = \{ 1 : \text{Const}(#5), 2 : \text{Ld}(a), 3 : \text{Add}(1, 2), 4 : \text{St}(X, 3) \} \). The partial ordering \(<\) due to the precedence constraints is \( (1 < 3, 1 < 4, 2 < 3, 2 < 4, 3 < 4) \). We illustrated the pipeline structure for the target processor in the previous section. A graphical representation of the precedence constraints for our example appears in Figure 3.3.

![Figure 3.3. An Example of the Task System](image)

3.3.2. The Cost Criteria in Scheduling

We mentioned a cost function in our task system model, and said that our goal is to find an optimal schedule that minimizes the total accumulated cost. In this section we describe this cost function in detail, particularly its interaction with the linear ordering of instructions and pipeline resources.
Recall that \( c_\pi(i) \) is the cost associated with completing task \( \zeta_i \) as the \( \pi_i^{th} \) instruction in the schedule. This is actually the time (or delay) an instruction must wait until all its source operands have become valid. In other words, an instruction can not be executed until all its immediate predecessors have finished their execution.

**Definition 2: \( \rho(\zeta) \)**

\( \rho(\zeta) \) is the set of all instructions \( \delta \in \pi \) such that \( \zeta \) has an immediate dependence on \( \delta \). Equivalently, \( \rho(\zeta) \) is the set of all immediate predecessors of \( \zeta \) in the DAG \( G(Z, E) \).

**Definition 3: \( \sigma(\zeta) \)**

\( \sigma(\zeta) \) is the pipeline resource that is utilized for the execution of instruction \( \zeta \). This function associates an instruction to a pipeline resource that was actually chosen among all the available alternatives. Obviously, \( \sigma(\zeta) \in \sigma(\zeta) \).

The cost function is expressed in terms of instructions and pipeline resources in the following equations. This has been split into two parts, namely \( D1 \) and \( D2 \), for the purpose of clarity. In Equation 1 and 2, \( D1 \) and \( D2 \) are defined for some \( i, j \).

\[
D1 = \max(EN_\delta): q = \sigma(\zeta_i), q \in \{ \pi_{i-r} : 1 \leq r \leq EN_\delta \} \tag{1}
\]

This represents the number of delay slots required to resolve the conflict between instruction \( \zeta_i \) and those that use the same pipeline resource (if any).

\[
D2 = \max_{j \in \rho(\zeta_i)} \left( \max(\pi_i - \pi_j - 1, LT_\sigma) - (\pi_i - \pi_j - 1) : q = \sigma(\zeta_j), \sigma(\zeta_j) \neq \emptyset \right) \tag{2}
\]

\( D2 \) is the number of delay slots required such that all operands in instruction \( \zeta_i \) become available. The expression in Equation 2 examines all parents of \( \zeta_i \) and, depending upon the relative position of a parent instruction, finds the number of delay slots to fill the latency delay of parent's pipeline (if any). \( D2 \) becomes the maximum number of delay slots computed for any parent. As pipeline conflicts and latency delays are resolved simultaneously, the cost function picks either \( D1 \) or \( D2 \), whichever is greater.

\[
c_\pi(i) = \max(D1, D2) \tag{3}
\]

It is important to note that this cost function for an instruction at some position within a schedule is computed by looking only at the values associated with
the instructions that occur before that instruction, and does not use any information about instructions that follow it. Two important properties follow from this observation.

1. This function is applicable to partial schedules.
2. \( c_\pi(i) \) can be applied incrementally to instructions in a schedule.

These properties are instrumental in the development of our algorithm for finding an optimal schedule. But before discussing that, we need to say a few words about the accumulated cost function, \( AC(i) \).

**Definition 4:** \( AC_\pi(i) \) Accumulated Cost

The accumulated cost for a (possibly partial) schedule \( \pi \) for the first \( i \) instructions is:

\[
AC_\pi(i) = \sum_{k=1}^{k-i} c_\pi(k); i, j \in \pi
\]

This is the total number of delay slots required in schedule \( \pi \) up to position \( k \).

### 3.3.3. Delay Slots and Optimal Schedule

**Definition 5:** Legal Schedule

A legal schedule (or feasible schedule) is defined as a one-to-one and automapping \( \Theta \) from the elements of \( Z \) into the set \( N \) of positive integers, 1 to \( |\pi| \), (that is, relative positions within a stream of instructions) such that

\[
\Theta(\zeta_i) - \Theta(\zeta_j) > 0 \mid \zeta_i, \zeta_j \in E, j \in \rho(i) \text{ for all } i,j.
\]

Hence a more precise statement of the scheduling problem is that we want to find a legal schedule (or a mapping \( \Theta \)) such that the accumulated cost function \( AC_\pi(|\pi|) \) is minimum. Recall that the accumulated cost function is comprised of the summation of cost functions \( c_\pi(.) \) for all instructions in a scheduled stream. As was pointed out earlier, \( c_\pi(i) \) is the amount of delay, in multiples of unit time, that should be added after instruction \( i \). We can view these wait periods as delay slots inserted within an instruction stream. Each delay slot takes unit time to elapse. Therefore our scheduling problem can be described in terms of minimizing the number of delay slots.

**Definition 6:** Optimal Schedule

An optimal schedule \( \pi_{opt} \) is a legal schedule for which the number of delay
slots are minimum. Equivalently, an optimal schedule $\pi_{opt}$ is a legal schedule $\pi$ for which the accumulated cost function $AC_{\pi}(|\pi|)$ is minimum.

A few remarks are appropriate at this point. First, the accumulated cost function $AC_{\pi}(|\pi|)$ represents the total delay time that must be expended on a pipelined processor to ensure correct results. This is equivalent to the difference between the time taken to execute on the pipelined processor and the time taken to execute on a similar processor for which every instruction executes in unit time. The delay slots after instruction $i$, given by $c_{\pi}(i)$, can be implemented in various ways, and our model is not specific to an implementation. Some common methods, as discussed in Section 1.2.2, are insertion of NOPs and hardware interlocks. Occasionally, we will refer to unit delays interchangeably with NOPs, but this does not imply that NOP insertion must be used instead of interlocks.

We present our algorithm in Section 3.5. Since this algorithm can be implemented using a variety of approaches, Section 3.4 gives an overall picture of the various trade-offs that should be considered when implementing the algorithm. In Section 3.6, we show that in the absence of a curtail point our algorithm indeed finds optimal solutions for all cases. A curtail point is a user-specified limit on search-space.

3.4. Compiler Model

Our scheduling algorithm works as compiler back end. Different phases of compiler are discussed briefly here. Details can be found in the next chapter.

As discussed earlier, the few pipeline scheduling algorithms presented in the literature act as postpass reorganizers, and work on the assembly level code produced by the compiler. The scope of reorganization done at this level is limited, because the assembly code reflects the assignment of values to a limited number of registers based on the initial ordering of the instructions in the source program.

Our algorithm works on an intermediate form, that is expounded in the next chapter. Traditional code optimizations are performed before scheduling the code for the pipeline machines. This is necessary because, if the optimization is performed after scheduling then some pipeline constraints might be violated, and the code has to be rescheduled.
Figure 3.4. Compiler Model with respect to the Scheduling Algorithm

It is assumed that the compiler front end has done appropriate analysis for memory reference aliasing, and has done renaming so that all references to variables in the tuple code are unambiguous and mutually exclusive.

At this stage, it might be necessary to do some initial register allocation analysis of live values to estimate register spill code. This is followed by the pipeline scheduler itself. An optional heuristic scheduling might be performed before the optimal pipeline scheduler to increase the pruning of the search space. A type of initial (list) scheduling is discussed in Chapter 4.
The approach presented here is not constrained by "artificial" conflicts resulting from coincidental reuse of a register name. Only at this stage, after scheduling has completed, are values assigned to specific registers. Further, it is at this time that the tuple form is converted into the notation for the target machine instruction set. It is assumed that the tuple operations are defined so that each tuple corresponds directly to one target machine instruction, hence this transformation is easily accomplished.

3.5. Scheduling Algorithm

The input to the pipeline scheduling algorithm is an initial (list) schedule and the DAG (Direct Acyclic Graph) [AhS86] it embeds. From this, all needed dependence information is derived. The pipeline scheduling algorithm is a heavily-pruned search algorithm that works on one basic block at a time and finds a schedule for which the number of delay slots required is minimum.

Section 3.5.1 defines a few terms and functions that are used in describing the algorithm. The algorithm itself is presented in two parts: the algorithm to determine Pipeline Delay Cost for different instructions in a schedule appears in in Section 3.5.2 and the complete search procedure in Section 3.5.3.

3.5.1. Definitions

In addition to the terms defined earlier in this chapter, the following terms and functions are used in the algorithms which follow:

Definition 7: $\pi$

$\pi$ is the current complete ordering of all instructions within this basic block. The $i^{th}$ instruction in $\pi$ will be denoted as $\pi(i)$; likewise, $\pi^{-1}(i)$ returns the position of instruction $i$ within $\pi$. Instructions within $\pi$ are labeled $1, 2, 3, ..., |\pi|$.

Definition 8: $earliest(\zeta)$

$earliest(\zeta)$ is the minimum number of instructions in $Z$ which must be executed before $\zeta$ in order to preserve the dependence structure given by the DAG. In other words, it is the number of instructions in a slice rooted at $\zeta$.

Definition 9: $latest(\zeta)$

$latest(\zeta)$ is the maximum number of instructions in $Z$ which could be
executed before \( z \) in order to preserve the dependence structure given by the DAG. In other words, it is \( |Z| \) - the number of instructions which transitively or directly depend on \( z \).

**3.5.2. Algorithm D — Compute \( c_\pi(i) \)**

The following algorithm is used to determine the amount of delay that would need to be inserted in the schedule \( \pi \) immediately before the \( i \)th instruction, \( z \). It is assumed that for each instruction scheduled in a position \( j < i \), \( c_\pi(j) \) has previously been computed. Recall that, in Section 3.3.2, \( c_\pi(i) \) is defined as

\[
c_\pi(i) = \max(D_1, D_2)
\]

where \( D_1 \) and \( D_2 \) are,

\[
D_1 = \max(EN_q; q = \delta(j), q \in \{\pi_i - r; 1 \leq r \leq EN_q\})
\]

\[
D_2 = \max(\max(\pi_i - \pi_j - 1, LT_q) - (\pi_i - \pi_j - 1); q = \delta(j), \sigma(j) \neq \emptyset)
\]

The algorithm to compute \( c_\pi(i) \) is therefore comprised of the following steps:

1. \( c_\pi(i) = 0 \). If \( i = 1 \), then done. Otherwise, go to step [2].

2. If \( \sigma(z) = \emptyset \), goto step [4].

3. (Check for conflict.) Let \( \tau(j) = c_\pi(i) + \sum_{k=j+1}^{i-1} c_\pi(j) + 1 \), the execution time between the start of the \( j \)th instruction and the \( i \)th instruction. Search backward from the \( j=1 \)th instruction until \( \tau(j) > EN_{\delta}(i) \cup \delta(j) = \delta(i) \cup j = 1 \). If \( \delta(j) = \delta(i) \cup \tau(j) < EN_{\delta}(i) \), then \( c_\pi(i) = EN_{\delta}(i) - \tau(j) \).

4. If \( \rho(z) = \emptyset \), then done.

5. (Check for dependence.) Perform step [6] for each instruction \( \delta \in \rho(z) \), then done.

6. Let \( z = LT_{\delta}(a^{-1}(\delta)) - \tau(\pi^{-1}(\delta)) \). If \( z > 0 \), then \( c_\pi(i) = c_\pi(i) + z \). Note that \( LT_{\delta}(a^{-1}(\delta)) \) is the latency of pipeline used (if any) by a parent of instruction \( \delta \).
3.5.3. Algorithm A — Find Optimal Schedule

The following is the schedule search algorithm which forms the core of our approach. This finds an optimal schedule (unless the search is truncated) for a basic block of instructions in the set $Z$. The initial schedule to this algorithm is passed in $\pi$. Algorithm D from Section 3.5.2 is used repeatedly to evaluate schedules being considered. Algorithm A consists of these steps:

1. For $i = 1$ to $|\pi|$, invoke the above algorithm to insert the correct number of delay slots before instruction $\pi(i)$. Call the resulting schedule $\pi_{\text{best}}$, the best schedule found thus far. Then $AC_{\pi_{\text{best}}}(|\pi_{\text{best}}|) = \sum_{k=1}^{\pi} c_{\pi}(k)$.

2. Partition $\pi$ into $\Phi$ and $\Psi$, where $\Phi$ represents the partial schedule being considered and $\Psi$ represents the list of instructions to be added to schedule $\Phi$. Initially, $\Phi = \emptyset$ and $\Psi = \pi$. Let $i = 1$. Let $\Lambda = 0$.

3. If $\Psi \neq \emptyset$ then the schedule is not yet complete and search continues with step [4]. If $AC_{\pi}(|\pi|) < AC_{\pi_{\text{best}}}(|\pi_{\text{best}}|)$ then $\pi_{\text{best}} = \pi$. Goto step [8]. Otherwise consider swapping instruction $\kappa = \pi(i) | \kappa \in \Phi$ with an instruction $\xi \in \Psi$. Let $\delta(\xi) = \hat{\xi}, \hat{\xi} \in \sigma(\xi)$. Repeat for all available choices one by one.

4. (Get next schedule pruned by legality.) The swap should be performed only if both of [4a] and [4b] are true:

   [4a] (Quick approximate check for legality.)
   \[ \text{latest}(\kappa) \geq \pi(\xi)(\xi) \cap \text{earliest}(\xi) \leq i \]

   [4b] (Real test for legality.) $\rho(\xi) \subseteq \Phi$

   If no legal swap was found, goto step [9].

5. (Check for equivalence.) Goto step [9] if the following condition is not true, else proceed to the next step.
   \[ \delta(\xi) \neq \emptyset \cup \rho(\xi) \neq \emptyset \cup \delta(\kappa) \neq \emptyset \cup \rho(\kappa) \neq \emptyset \]

6. (Apply Excessive Cost Pruning.) Let $\Phi'$ be a partial schedule formed by substituting instruction $\xi (\xi \in \Psi)$ for $\kappa (\kappa \in \Phi)$.
   If $AC_{\Phi'}(|\Phi'|) < AC_{\pi_{\text{best}}}(|\pi_{\text{best}}|)$ then go to the next step, otherwise, continue with step [9].

7. (Apply curtail point search truncation.) Let $\Lambda = \Lambda + 1$. If $\Lambda \geq \lambda$ then $\text{abort}$, with a possibly suboptimal best schedule $\pi_{\text{best}}$. Otherwise, continue
with step [9].

[8] Now actually perform that swap which was considered in step [3]. Interchanging $\xi$ with $\kappa$ alters $\pi$, $\Phi$, and $\Psi$. Move the partition between $\Phi$ and $\Psi$ to reduce $\Psi$ by one instruction and goto step [3].

[9] Restore the previous values of $\pi$, $\Phi$ and $\Psi$. This done by "undoing" the most recent changes made in these sets. For example, the set $\pi$ is restored to its previous contents by swapping the most recently swapped instruction back to its original position.

[10] If $i < |\Psi|$ then $i \leftarrow i+1$ and goto step [3]. Otherwise, done, with an optimal solution in $\pi_{\text{best}}$. Then $\pi_{\text{opt}} = \pi_{\text{best}}$.

The pruning techniques in the algorithm cut the search time by $(|\pi| - k)!$ when pruning occurs at position $k$. Note that, because condition [5] filters-out equivalent schedules, the algorithm presented finds an optimal schedule, but might not examine all optimal schedules when the optimal schedule is not unique.

3.6. Proof of Optimality

In this section, we prove that our algorithm produces results which are guaranteed to be optimal if the search is not truncated. This is done by first proving the optimality of a non-truncating algorithm, described in the next section, and then by showing its equivalence to our algorithm for the inputs for which the search is not aborted.

Recall the problem statement from Section 3.3.1 that for a given task system $T = (Z, <, P)$, we want to find an optimal schedule $\pi$, for processor $P$, representing a complete ordering $\{s_i : i \in \pi\} = \{s_{\pi_1}, s_{\pi_2}, \cdots, s_{\pi_n}\}$, that minimizes $\sum_{k=1}^{n} c_\pi(k)$ such that no instruction is scheduled before its immediate predecessor as given by the partial ordering $<$. $Z = \{s_1, s_2, \cdots, s_n\}$, where $s_1, \cdots, s_n$ are tasks (instructions) that are to be executed successively by the processor $P$, $c_\pi(i)$ is the cost associated with completing task $s_i$ as the $\pi_i^{th}$ task in a schedule $\pi$. 
3.6.1. Non-truncated Algorithm NT

The non-truncating algorithm NT is obtained from A by deleting step [7]. We derive a series of algorithms that preserve the optimality of the solution to the code scheduling problem for pipeline constraints, and show their equivalence (except for algorithm ALL) in terms of the quality of the solutions obtained from them. The objective is to prove that NT finds an optimal solution.

3.6.1.1. Algorithm ALL

Construct an algorithm ALL from algorithm A by deleting steps [4] to [7] from it. Let \[ T_{\text{all}} \] be a search tree of all possible orderings of instructions in \( Z \).

**Lemma 1:** The search tree generated by algorithm ALL is \( T_{\text{all}} \).

**Proof:** Because of steps [3], [8] and [9] it is clear that algorithm ALL generates all possible permutations of the set of instructions in \( Z \). And we have defined \( T_{\text{all}} \), as the search tree of all possible orderings of instructions in \( Z \), therefore it is the search tree that is traversed by ALL. It should be noted that we are not interested in the solution found by this algorithm, because it might violate the DAG precedence constraints. Rather, we use it to construct the algorithm of interest by adding back steps to restrict the search.

3.6.1.2. Algorithm LG

Let LG be an algorithm constructed from ALL by adding step [4] of A. Define \( S_{\text{legal}} \) as the set of all possible legal schedules and \( T_{\text{legal}} \) as the search tree corresponding to \( S_{\text{legal}} \).

**Lemma 2:** Algorithm LG examines all solutions in the set \( S_{\text{legal}} \).

**Proof:** Step [4] adds the preclusion refinement, described in Section 3.2.1.1, to the exhaustive search algorithm ALL. Let \( S_1 \) be the search tree examined by LG. Then \( S_1 \subseteq T_{\text{ALL}} \) (\( \subseteq \) means subtree here). Let \( k \) be a node in \( T_{\text{ALL}} \) that is not present in \( S_1 \). And let \( j \) be any node in a path from the root to some leaf in \( T_{\text{ALL}} \) such that it violates the precedence constraints in the order corresponding to such path. Then from step [4] of A it follows that \( k \) must be a descendant of \( j \). Since each \( k \) corresponds to one or more illegal solutions, all paths from root to leaves in \( T_{\text{ALL}} \) that are not present in \( S_1 \) are exactly the illegal solutions. Therefore \( S_1 = S_{\text{legal}} \).
**Theorem 3:** An optimal solution always exists.

*Proof:* Since the code as initially generated is correct, there will always be at least one legal schedule that can satisfy the precedence constraints. Since the pipeline interlock length and conflict parameters are finite, the total delay for each schedule can be computed. Hence, an optimal schedule, which is the minimum cost taken over the set of all possible legal schedules, exists.

**Lemma 4:** Algorithm LG always finds an optimal solution $\pi_{opt}$.

*Proof:* From Lemma 3, $\pi_{opt}$ exists. Algorithm D computes the minimum amount of delay for a given order that resolves the pipeline constraints. Since $\pi_{opt} \in S_{legal}$, and from Lemma 2 and LG computes delay cost for each solution in $S_{legal}$, step [3] ensures that the final solution $\pi_{best} = \pi_{opt}$.

### 3.6.1.3. Algorithm B

Construct algorithm B from LG by adding step [6] of algorithm A. Note that this corresponds to the minimum cost pruning in Section 3.2.1.3. Now we prove that B always finds an optimal solution. To prove this, we first need to introduce the following:

**Lemma 5:** Optimal schedule, $\pi_{opt}$, is not unique (in general).

*Proof:* Proof by contradiction. We construct a counter example that will have more than one optimal schedule. Consider an optimal schedule $\pi_{opt}$ for a task system similar to one described in Section 3.3.1. Suppose that there was only one delay slot to be filled after instruction $\zeta$ and it is replaced by an instruction $\delta$. Hence $AC_{\pi_{opt}}(\pi_{opt})=0$. Let $\pi_2$ be another schedule with an instruction $\xi$ following instruction $\zeta$ and $\xi \neq \delta$. Then $AC_{\pi_2}(\pi_2)=0$, only if $\delta(\delta) \neq \delta(\zeta), \delta \notin \rho(\zeta)$. This condition depends only on the precedence constraints in the code block and pipeline structure. Therefore, for an arbitrary input this may be true and hence $\pi_2$ is also an optimal solution. But $\pi_2 \neq \pi_{opt}$. Therefore optimal solution is not unique for an arbitrary task system.

**Lemma 6:** $AC_x(.)$ is a monotonic increasing function.

*Proof:* From equation (3) we note that $c_x(i) \geq 0$ for all $i$ and $\pi$. $AC_x(.)$ is a monotonic increasing function if and only if, for some $k_1$ and $k_2$, condition $k_1 > k_2$ implies $AC_x(k_1) \geq AC_x(k_2)$. Now, for $k_1 > k_2$

$$AC_x(k_1) = \sum_{j=k_1}^{\pi} c_x(j)$$
\[ j=k_2 \]
\[ = \sum_{j=1}^{j=k_1} c_\pi(j) + \sum_{j=k_2+1}^{j=k_1} c_\pi(j) \]
\[ = AC_\pi(k_2) + \sum_{j=k_2+1}^{j=k_1} c_\pi(j) \]

Since \( c_\pi(i) \geq 0 \) for all \( i \) and \( \pi \), hence \( AC_\pi(k_1) \geq AC_\pi(k_2) \). Proof of the converse is obvious and is not given here.

**Lemma 7:** Cost \( AC_\pi(k) \) of partial solution \( (k < |\pi|) \) exists.

**Proof:** Since \( AC_\pi(k) = \sum_{j=1}^{j=k} c_\pi(j) \), no information about the schedule after \( k \) is required. Therefore, this cost is computable for a partial solution.

**Theorem 8:** Algorithm B always finds an optimal solution \( \pi_{opt} \).

**Proof:** Lemma 8 allows us to compute cost of partial schedules. Step [6] of B states that do not examine any descendant of a node \( \xi \) at level \( k \) in the search tree if:

\[ AC_\pi(k) \geq AC_{\pi_{best}}(|\pi_{best}|) \] (5)

Let \( \delta \) be a descendant of \( \xi \) at level \( j = |\pi| \). Since \( j \geq k \), then from Lemma 6,

\[ AC_\pi(j) \geq AC_\pi(k) \] (6)

From (5) and (6),

\[ AC_\pi(j) \geq AC_{\pi_{best}}(|\pi_{best}|) \] (7)

This means that the cost of any complete solution that contains node \( \xi \) will have delay cost greater or equal to the cost of the best solution found so far. Obviously, if this cost is greater, then we can ignore these nodes, or prune them from the search tree. What if the two costs are equal? From Lemma 5, we know that an optimal schedule is not unique, therefore if two or more schedules have the same minimum cost we can arbitrarily choose one of them and that will be our optimal schedule. This proves our theorem.
3.6.1.4. Algorithm C

Derive algorithm C from B by adding step [5] of algorithm A. This corresponds to the equivalence check pruning of Section 3.2.1.5. We develop some definitions and Lemma to prove the optimality of this algorithm.

Define orthogonal instructions as those that do not use any pipeline resource, do not have any parent instructions and their execution does not have any side-effects (e.g. no I/O). An example of such instructions is loading of a constant value in a register (on virtually all machines).

Step [5] mentioned above states that no two orthogonal instructions should be swapped. We show that this is equivalent to arbitrarily picking one of the schedules that are guaranteed to be equivalent in terms of pipeline delays.

**Lemma 9**: Any ordering (schedule) within a set of orthogonal instructions is an optimal schedule for that set of instructions.

**Proof**: By definition, orthogonal instructions do not use any pipeline resource. Therefore, no delay slots are required between instructions because all instructions in the schedule are orthogonal. Moreover, any schedule is legal due to the absence of precedence constraints among these orthogonal instructions. All schedules are legal and require zero delay slots — they are optimal schedules.

**Lemma 10**: Accumulated cost function associated with a basic block that contains one or more clusters of orthogonal instructions remains unchanged if the order of orthogonal instructions within clusters is changed. (A cluster of orthogonal instructions denotes a contiguous sub-block of these instructions).

**Proof**: Any change in the order of orthogonal instructions within a cluster does not effect ordering constraints for any of their dependent instructions because these instructions remain within the original cluster range. The order of these instructions within the cluster boundaries does not effect pipeline delays and conflicts. In other words, all such permutations of orderings are equivalent as far as the pipeline delays are concerned. Hence, the total number of delay slots required for the basic block remain unaffected. An example of this type of equivalence is given in Section 3.2.1.4.
Theorem 11: Algorithm C always finds an optimal solution.

Proof: Algorithm B examines all possible legal schedules except for those that lead to non-optimal schedules. Adding step [5] in this algorithm gives us a new algorithm, Algorithm C, which prunes the search space by discarding equivalent solutions. If all instructions in a basic block are orthogonal then by Lemma 9 all schedules are equivalent and optimal. Therefore this algorithm returns the first such schedule and ignores all other possibilities. On the other hand, if there are other instructions present which are not orthogonal then, by Lemma 10, we do not need to examine all possibilities within clusters of orthogonal instructions.

Some points are not intuitively clear from the above discussion. We give an example to illustrate how Algorithm C still manages to examine all schedules that are not equivalent. For example, what about the instructions that are dependent on orthogonal instructions and (say) two of them are separated by many other instructions? An example of this sequence is \((a, b, C, d, e, f, G, h, i, j)\), where \(C\) and \(H\) are two orthogonal instructions and \(i\) and \(j\) are dependents of \(H\). Apparently, if we do not swap \(C\) and \(H\) then \(i\) and \(j\) will not be able to appear before position 7 in this example code sequence. But a careful analysis of step [5] shows that this is not the case, because there is no restriction on swapping orthogonal instructions with other instructions which are not orthogonal. Hence \(H\) can go anywhere except at position 3, and hence, its dependent can also move freely within the constraints of legal ordering. From Lemma 10, instruction \(H\) at position 2 or 4 with \(C\) at position 3 is equivalent to \(H\) being at position 3. Therefore step [5] does not prevent us from considering any possibility that might lead to an optimal schedule.

Theorem 12: Algorithm NT always finds an optimal solution.

Proof: Algorithm NT is the same as algorithm C because both are derived from A by adding steps [4] to [6]. In Theorem 11, we showed that C finds the optimal solution, therefore NT will also find the optimal solution.

3.6.2. Truncating Algorithm

We extend NT by adding step [7] given in Section 3.5.3. This is our original algorithm A now. The step [7] adds a curtail point \(\lambda\) in NT. We shall prove here that for the inputs for which the search is not truncated this algorithm
returns an optimal solution.

**Theorem 13:** Any schedule found by Algorithm A is guaranteed to be optimal if the search is not truncated.

**Proof:** Let $S_1$ be the solution found by A for an input $I$, and $S_2$ be the solution obtained from NT. The two algorithms differ in terms of step [7]. Since the search was not curtailed in A for input $I$ therefore we can say that it is equivalent to running A for $I$ without step [7]. But an algorithm A without step [7] is the algorithm NT. Hence $S_1 = S_2$ for $I$. From theorem 12 we know that $S_2$ is an optimal solution. Therefore $S_1$ is also an optimal solution. Hence proved.

This concludes our proof for the optimality of our algorithm (when the search is not truncated).

### 3.7. Summary

In this chapter we have presented our research work in a more formal manner. In the next chapter we describe issues relevant to the implementation of this algorithm and its integration with existing compilers and other optimizations.
4.1. Introduction

In Chapter 3, we discussed the basic algorithm for scheduling code for pipeline processors. In this chapter we present an implementation of the optimal code scheduler.

First, in Section 4.2, we describe the complete picture of our compiler system that incorporates the pipeline scheduler. The implementation consist of a compiler system that accepts a small subset of C allowing input of basic blocks and generates an intermediate form code as instruction tuples. This is detailed in Section 4.2.1 Classical optimizations are done incrementally, a backward pass removes any dead or redundant code found. The resulting tuple code is then reorganized using a heuristic initial scheduling algorithm followed by the main pipeline scheduling algorithm, as discussed in Sections 4.2.2 and 4.2.3 respectively. Finally the the tuple code is converted to the target machines instructions and register allocation is performed. Register allocation is covered in Section 4.2.4. We conclude this chapter by presenting a summary in Section 4.4.

4.2. Structure of the Scheduler

In this section, we outline the general structure of a prototype implementation of the proposed optimal pipeline scheduling technique. The construction of the compiler front end does not impact the scheduling technique, hence only the back end of the compiler is discussed. Figure 4.1 shows the organization of the compiler back end in the prototype implementation. The phases of the compiler back end are discussed in the following sections.
4.2.1. Optimized Tuple Generation

The compiler front end is responsible for parsing the source program, performing traditional optimizations, and emitting an appropriate intermediate form representation of the program.

Optimization of the code is not strictly necessary in order to perform pipeline scheduling; in fact, if traditional optimizations are applied, the general effect is that finding good schedules becomes more difficult. Hence, in the interest of obtaining accurate results, the prototype compiler performs most traditional optimizations. These include constant folding with value propagation, common subexpression elimination, dead code elimination, and various peephole optimizations. The resulting code, which is usually substantially smaller than the unoptimized code, is then represented as a DAG (directed acyclic graph) [AhS86]
embedded in a linear notation.

The notation we use for each instruction is that of a tuple of the form $\Gamma_{i,o,\alpha,\beta}$ where $i$ is the reference number of the tuple, $O$ is the operation type, and $\alpha$ and $\beta$ are two operands. Each operand can be a variable, the result of another tuple (the reference number of another tuple), or $\emptyset$. An example of tuple code, corresponding to a very simple basic block is given in Figure 4.2.

{  
  b = 15;
  a = b * a;
}

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Const 15</td>
<td>$\Gamma_{1,\text{Const},&quot;15&quot;}$</td>
<td></td>
</tr>
<tr>
<td>2: Store #b, 1</td>
<td>$\Gamma_{2,\text{Store},&quot;b&quot;,1}$</td>
<td></td>
</tr>
<tr>
<td>3: Load #a</td>
<td>$\Gamma_{3,\text{Load},&quot;a&quot;}$</td>
<td></td>
</tr>
<tr>
<td>4: Mul 1, 3</td>
<td>$\Gamma_{4,\text{Mul},1,3}$</td>
<td></td>
</tr>
<tr>
<td>5: Store #a, 4</td>
<td>$\Gamma_{5,\text{Store},&quot;a&quot;,4}$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2. Sample of Intermediate Form

At the level of the tuple code, all references to variables are assumed to be unambiguous and mutually exclusive, i.e., no two variable names refer to the same object. Since this is not true of some high-level language program references to array elements or objects accessed through indirection on pointers, it is assumed that the compiler front end has done appropriate analysis and renaming so that these ambiguities need not be seen in the tuple code [Die87]. Since the prototype compiler was used solely for synthetic benchmarks whose properties could be controlled directly, the prototype compiler simply assumes that all variable names appearing in tuples are unambiguous and mutually exclusive.

At this stage, it is also important that a portion of the register allocation analysis be performed — the creation of register spill code. Since values are not
allocated to particular registers, the concept is simply that if there are more live values than registers in the target machine, then all values beyond the number of registers will be explicitly re-loaded. In other words, we insure that when registers are actually allocated later, there will be no need to introduce new spill instructions, since these could invalidate the optimality of the schedule. Note that inserting spill instructions after scheduling would usually result in a valid schedule, since \texttt{Store} instructions typically do not interfere with any pipelined operations.

In the simulations presented here, the prototype implementation simply assumed that there were always enough registers so that spilling would be unnecessary.

4.2.2. List Scheduler

As tuple code is emitted by the front end, the code is grouped into basic blocks [AhS86] and each block is processed independently. The purpose of the initial scheduling phase is to apply heuristics to generate a reasonable schedule of the current block. This is important because the search is pruned, in part, by a branch and bound technique which makes the total number of schedules searched sensitive to the quality of schedules searched early in the process.

The heuristic used is described in depth in [ZaD90], where it was applied to generate an order for incrementally scheduling tuples across multiple processors in barrier MIMD machines. In essence, the heuristic arranges the tuples into a sequential order (schedule) so that the distance between each instruction and the instructions that depend on it is as large as possible. Because of the branch and bound pruning, the time taken in applying the initial scheduling heuristic is more than recovered by the fact that the search for an optimal pipeline schedule will converge more quickly.

Alternatively, any other scheduling technique proposed in the literature, e.g. Gross [Gro83], etc., could be applied to find this initial schedule. It is unclear whether the extra complexity of those techniques would be justifiable for use in place of our list scheduling heuristic.
4.2.3. Pipeline Scheduler

Having obtained a "reasonable" initial schedule, the pipeline schedule search algorithm is applied to find the optimal schedule. This algorithm, discussed in Section 3.5, represents the prime contribution of this research. The output is simply a schedule of the tuples within each block.

4.2.4. Register Allocation and Code Generation

As discussed in Chapter 2, the few pipeline scheduling algorithms presented in the literature act as postpass reorganizers, and work on the assembly level produced by the compiler. The scope of reorganization done at this level is limited, because the assembly code (in general) reflects the assignment of values to a limited number of registers based on the initial ordering of the instructions in the source program.

The approach presented here is not constrained by "artificial" conflicts resulting from coincidental reuse of a register name. Only at this stage, after scheduling has completed, are values assigned to specific registers. Further, it is at this time that the tuple form is converted into the notation for the target machine instruction set. It is assumed that the tuple operations are defined so that each tuple corresponds directly to one target machine instruction, hence this transformation is easily accomplished.

4.3. Pipeline Configuration Information

In this section we describe the pipeline structure model that is an input to the pipeline scheduler. For each hardware pipeline, the function, latency, and enqueue time must be specified. Further, so that the compiler can know which pipelines, if any, may be used to execute each type of operation, each hardware pipeline is given a unique identifier and operation types are associated with sets of pipelines. This is done using two tables.

Consider a processor with the following pipelined resources: two memory access pipelines (loaders), two adders, and one multiplier. These hardware resources are described in Table 4.1.
The second table used to describe the scheduling problem for our compiler is Table 4.2, the operation-to-pipeline mapping table. Given these tables, for example, the add instruction has two independent pipelines available to it (namely, numbers 3 and 4), and thus can be scheduled for either pipeline. In this example, Add and Sub operations share two independent pipelines; likewise, Mul and Div share a single pipeline.

The results presented in this paper were obtained using a more conservative, single pipeline unit per function, the tables for which appear in Section 5.2. Notice that changing the pipeline structure changes only the entries in these tables, not the structure of the scheduling algorithm. Further, note that the list scheduler does not examine these tables, hence, the initial schedule is independent of the target pipeline structure.

4.4. Summary

The search for optimal code schedule for a given pipeline target model is done by making calls to an \(O(n)\) routine that incorporates pruning techniques discussed in the previous chapter. The algorithm is very easy to implement and can be readily modified to include other solution-cost evaluation criteria when performing pruning. One example would be to consider both pipeline delays and

\[1\] The current implementation does not support this feature.
Table 4.2. Sample Operation-to-Pipeline Mapping

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>Add</td>
<td>{3, 4}</td>
</tr>
<tr>
<td>Sub</td>
<td>{3, 4}</td>
</tr>
<tr>
<td>Mul</td>
<td>{5}</td>
</tr>
<tr>
<td>Div</td>
<td>{5}</td>
</tr>
</tbody>
</table>

delays resulting from register spills and minimize both simultaneously. Other criteria for optimality can also be taken into account.

Although the upper bound for the worst case is still exponential, our tests indicate that our algorithm is usually able to find optimal solutions in a very reasonable time. These results are presented in Chapter 5.
CHAPTER 5

PERFORMANCE ANALYSIS

5.1. Introduction

In the previous chapters we have made claims that our algorithm finds optimal solutions for "most" blocks. In this chapter we justify these claims and explain how we evaluated the performance and merits of our technique. In Chapter 2 we annotated the fact that the problem of optimal code scheduling for pipeline constraints is NP-complete. This implies that there is no known method, with a worst-case polynomial time complexity, for finding an optimal schedule. Although we can devise heuristics to reduce the upper bound time complexity, the resulting solutions will not be optimal for some blocks. Since our proposed algorithm focuses on the optimality of solutions, its worst case time complexity is exponential. We have guarded against such cases by halting the search algorithm after some specified number of search calls and forcing it to return the best solution found so far in the search.

Having stated the worst case time complexity of our algorithm, we now demonstrate its superiority (relatively speaking) and usefulness for real applications. It should be kept in mind that the runtime for finding an optimal schedule for input basic blocks varies significantly due to the type of instructions present, interdependencies, pipeline structure used, and the number of instructions. Therefore, it is not feasible to formulate a closed-form expression for the performance of this algorithm. Rather, an empirical study was carried out and the results presented here are indicative of what to expect when this algorithm will be used for real application programs on machines with typical pipeline structures.

A prototype compiler implementing the algorithms given in Chapter 4, was tested with carefully generated benchmark programs. These programs were
synthesized according to statistics obtained from "real" programs. The construction of the synthetic benchmark programs is discussed in Section 5.3. Section 5.4 describes a general simulation procedure and results. The effects of variations in curtail point, the number of memory references and pipeline parameters on the performance are studied in Sections 5.5, 5.6 and 5.7 respectively. Suboptimal solutions are reviewed in Section 5.8 and finally all the analysis is summarized in Section 5.9.

5.2. Performance Metrics and Parameters

We are mainly concerned with the percentage of cases for which our algorithm is successful in finding optimal solutions for a given set of input basic blocks, and the average runtime\(^1\) associated with it. Typically there are many basic block in real programs and the average runtime will dictate the time overhead in compiling those programs. The number of NOPs removed is of secondary importance to us. This is because an optimal solution itself implies the minimum possible number of NOPs in the schedule. Moreover, in Chapter 2, we demonstrated that our algorithm will eliminate more NOPs than previous algorithms simply because it operates prior to register assignment.

An important parameter in this study is the \textit{curtail point} \(X\). Obviously, given a large enough \(X\), our algorithm will always find an optimal solution. However, we must to choose a value of \(X\) that will result in an acceptable average runtime (compile time overhead).

For a given value of \(X\), our algorithm will terminate with a suboptimal solution if the number of search calls exceed \(X\). The number of search calls required to find optimal solutions varies with the type of input basic block (i.e., its size, dependencies etc.) and the pipeline configuration. Therefore, we should expect the percentage of optimal solutions for a given value of curtail point to vary with basic block size and pipeline structure. This is indeed the case, as shown by the various result graphs in the following sections.

\(^1\) In this chapter we denote runtime in terms of the number of calls to search procedure \(\Omega\) that was explained in Chapter 2.
5.3. **Construction of Synthetic Benchmarks**

A C program was developed to randomly generate basic blocks according to the statistics described below. This program requires as input the number of statements, variables, and constants desired in the generated code. It then generates a random sequence of assignment statements satisfying the desired conditions. The frequency of the types of assignment statements corresponds loosely to the instruction frequency distributions found in [AIW75]. These frequency distributions reflect the statistics obtained from real programs. The frequency distributions are shown in Table 5.1.

We preferred synthetic basic blocks over real programs for testing our algorithm for the following reasons:

- The performance of our algorithm depends on the nature of inputs. For real programs (applications), the structure of basic blocks is not under our control. This makes it impossible to vary them in order to study performance as block structure changes.

- Typical block size for real programs is very small (fewer than ten instructions). We have found that our algorithm works extremely well for basic block sizes of up to twenty instructions. We did not want to be overly optimistic and wanted to study performance on large basic blocks that might occur using techniques such as trace scheduling [Ell85]. Such large blocks are readily attained using synthetic generation of basic blocks.

For very large basic blocks, it might be useful to split the basic blocks into smaller sections (containing, say, twenty instructions or less each) and find solutions which are locally optimal. A good heuristic for the split might be to simply partition the list schedule, however, we have not yet examined such techniques.

Note that Table 5.1 does not give the frequencies for **Load** and **Store** instructions. These instructions are provided as necessary during code generation and optimization: the first reference to a variable causes a load for that variable to be generated, and a store is generated when a variable is assigned a value. In Section 5.6 we vary the frequencies of **Load** and **Store** instructions and study the outcome.
Table 5.1. Synthetic Benchmark Instruction Frequencies

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>—</td>
</tr>
<tr>
<td>Store</td>
<td>—</td>
</tr>
<tr>
<td>Add</td>
<td>45.8%</td>
</tr>
<tr>
<td>Sub</td>
<td>33.9%</td>
</tr>
<tr>
<td>And</td>
<td>8.8%</td>
</tr>
<tr>
<td>Or</td>
<td>5.2%</td>
</tr>
<tr>
<td>Mul</td>
<td>2.9%</td>
</tr>
<tr>
<td>Div</td>
<td>2.2%</td>
</tr>
<tr>
<td>Mod</td>
<td>1.2%</td>
</tr>
</tbody>
</table>

5.4. Simulation of the General Behavior

Results obtained in this section are based on realistic pipeline parameters and input basic blocks. Therefore, the performance of our algorithm gives a good measure of what can be expected in real benchmarks.

5.4.1. Procedure

A set of 3200 basic blocks was generated with varying number of constants, variables and instruction count. Frequency distribution of these basic blocks with respect to their sizes is shown in Figure 5.1. These inputs were compiled and scheduled for the pipeline constraints given in the next section. Curtail point for these runs was set to 10000, 20000, 50000, 100000 and 200000 successively. Hence, we obtain a total of 18,000 run samples.

5.4.2. Pipeline Constraints for Simulations

The results shown in this and some subsequent sections were obtained using a pipeline design given in Tables 5.2 and 5.3. Again, this is close to what might
be expected in a "real" machine. However, this pipeline structure is still more complex than SU-MIPS [GrH88] and RCA-MIPS. In Section 5.7, we examine performance on more varied and complex pipeline structures.

5.4.3. Results

The results presented in this section reflect a total of 16,000 runs with basic blocks containing various numbers of statements, variables, and constants. The curtail point was also varied, but was always large relative to the number of search calls made for an optimal search on average. A very brief summary of the results appears in Table 5.4.

Figure 5.2 shows the final number of NOPs after optimization versus the initial number of NOPs. Figure 5.3 shows the average runtime over all 16,000 sample blocks. Figure 5.4 shows the percentage of all runs which found optimal schedules, i.e., which were not truncated by λ.
Table 5.2. Pipeline Description for Simulations

<table>
<thead>
<tr>
<th>Pipeline Function</th>
<th>Pipeline Identifier</th>
<th>Latency</th>
<th>Enqueue Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>loader</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>multiplier</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.3. Operation-to-Pipeline Mapping for Simulations

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>{1}</td>
</tr>
<tr>
<td>Mul</td>
<td>{2}</td>
</tr>
<tr>
<td>Div</td>
<td>{2}</td>
</tr>
</tbody>
</table>

5.4.4. Discussion

Notice that the average number of instructions per block for all these inputs was 20.6, which implies that the typical search, without pruning, would have required searching on the order of $10^{19}$ schedules, whereas only about $10^3$ were searched for the average block in our sample.

Figure 5.1 shows the frequency distribution of the number of instructions per basic block for our sample. Studies have shown that on average a basic block in real programs has fewer than ten instructions, however, our average sample block had 20.6; this yields overly conservative results, since for basic blocks with fewer than 20 instructions the algorithm nearly always produces optimal solutions. Though programs with basic blocks that have more than forty instructions are very rare, we have included even such blocks in our study to show the worst-case effectiveness of our algorithm.
Table 5.4. Statistics for Scheduling 16,000 Blocks

<table>
<thead>
<tr>
<th></th>
<th>Search Completed (Optimal)</th>
<th>Search Truncated (Suboptimal?)</th>
<th>Totals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Runs</td>
<td>15,812</td>
<td>188</td>
<td>16,000</td>
</tr>
<tr>
<td>Percentage of Runs</td>
<td>98.83%</td>
<td>1.17%</td>
<td>100%</td>
</tr>
<tr>
<td>Avg. Instructions/Block</td>
<td>20.50</td>
<td>32.28</td>
<td>20.6</td>
</tr>
<tr>
<td>Avg. Initial NOPs</td>
<td>9.50</td>
<td>14.34</td>
<td>9.6</td>
</tr>
<tr>
<td>Avg. Final NOPs</td>
<td>0.67</td>
<td>4.03</td>
<td>0.7</td>
</tr>
<tr>
<td>Avg. $\Omega$ Calls</td>
<td>427.4</td>
<td>54,150</td>
<td>1,060</td>
</tr>
</tbody>
</table>

Figure 5.2. Initial and Final NOPs Vs. Block Size
Figure 5.3. Runtime (log scale) Vs. Block Size

Figure 5.4. Percentage Run To Completion Vs. Block Size
The percentage of optimal solutions decreases as the size of basic blocks increases. It is just in accordance with what we expected. The number of possible (legal) schedules that are to searched without any pruning increases as a factorial function of the block size, but our pruning technique works exceptionally well and even for large basic blocks we are able to find optimal solutions for most of the cases within a small value of the curtail point.

We selected a set of high curtail points for these runs, i.e., 10000, 20000, 50000, 100000 and 200000. From the results we obtained it is clear that this seemingly high curtail point results in an average of just 1000 search calls. This is due to the introduction of artificially produced large basic blocks. — with typical blocks, there would be even fewer calls. The average number of search calls for all basic blocks with less than twenty instructions was about 75. In Figure 5.5, we have plotted the average number of search calls versus the maximum basic block size in the sample.

![Figure 5.5. Average Search Calls Vs. Maximum Block Size](image-url)
Figure 5.2 shows the final number of NOPs after optimization versus the initial number of NOPs. Note that the initial number of NOPs grows linearly with the number of instructions, but the final number of NOPs remains nearly constant. Obviously, for larger basic blocks there are more instructions that use pipeline function units — and hence more initial NOPs. The bottom curve, the final number of NOPs, indicates that the number of removeable NOPs increases with the number of instructions in a basic block. This is quite understandable, since more instructions are available to fill the delay slots in a larger basic block.

Our results show that for a very small percentage of the inputs (less than 1.2% overall) the outputs were possibly not optimal. Further study of these inputs revealed that the optimal solutions for most of these inputs were not found even by increasing the runtime curtail point fifty fold. Moreover, the number of final NOPs found (in general) after that was not much different from what was found in the runtime allowed in the sample runs. This suggests that the algorithm quickly converges to a near-optimal solution. This is further explored in Section 5.8.

5.5. Variations in the Curtail Point

In this section we investigate how the performance of our algorithm is effected by varying the curtail point λ.

5.5.1. Procedure

We separate the results obtained in the previous section for various curtail points and plot these results against the values of λ. For each λ, there was a sample of 3200 basic blocks.

5.5.2. Results

In Figure 5.6 we have plotted the average percentage run to completion versus the value of the curtail point. The average runtime for each value of λ is shown in Figure 5.7.
Figure 5.6. Average Percentage Run To Completion Vs. Curtail Point

Figure 5.7. Average Runtime Vs. Curtail Point
5.5.3. Discussion

The above results bring forth an important feature of our algorithm. For $\lambda=10000$ the percentage of optimal solutions obtained is 97.87%, and for $\lambda=200000$ this increases to 99.31%. The fact that an increase of twenty fold in the curtail point improves the performance by only 1.47% shows that nearly all optimal solutions are obtained quickly by our algorithm. And, as we discussed earlier, a value of $\lambda$ about 100 will be sufficient for most blocks.

5.6. Varying the Number of Variables

In our discussion about the synthetic benchmark instruction frequencies, we mentioned that the Load and Store instructions are provided as necessary during code generation and optimization: the first reference to a variable causes a load for that variable to be generated, and a store is generated when a variable is assigned a value. One parameter for the synthetic basic block generation program is the maximum number of variables allowed in a basic block. Then program statements are generated randomly using different variables from this set of variables. Although code optimizations like dead-code removal and value propagation eliminate some of the instructions referencing these variables, we indirectly vary the number of memory references in a basic block by specifying the maximum number of variables. In this section, we vary the number of variables in basic blocks and study the corresponding results.

5.6.1. Procedure

We generated a sample of basic blocks for different number of variables from 2 to 15. There are about 360 basic block inputs for each value of the maximum number of variables. Thus the total number of samples for this experiment is 3600 basic blocks. These inputs are run for the same pipeline constraints and curtail point parameters as in Section 5.4.

5.6.2. Results

The percentage run to completion versus the maximum number of of variables in any basic block is shown in Figure 5.8. And the average runtime for the various values of the maximum number of of variables in any basic block is drawn in Figure 5.9.
Figure 5.8. Average Percentage Run To Completion Vs. Variables

Figure 5.9. Average Runtime Vs. Variables
5.6.3. Discussion

When the synthetic program generator has a small number of variables to choose from, it tends to reuse the same variable names in a basic block more often. And when that basic block is compiled and optimized, most of the instructions are removed as dead-code, resulting in a low average runtime. The runtime increases as the number of variables increases. The average runtime has its peak over five variables. After that the runtime begins to decrease as the number of variables increases further. This is because when the number of variables is comparable to the number of statements then more rigid dependencies between instructions begin to appear, which limit the freedom with which the instructions can move within the basic block. In any case, these variations are not very significant.

5.7. Variation in the Pipeline Structure

All the results that we have discussed up till now were obtained by using the pipeline structure of Section 5.4.2. Here we show the effect of scheduling the same code for different pipeline hardware.

5.7.1. Procedure

Recall that the pipeline configuration, i.e., the latency of pipelines, their enqueue time and association of instructions with different pipelines, is an input to our scheduling algorithm. We collect a sample of inputs with various block sizes and variables similar to the sample taken in Section 5.4.1. These inputs are run with various settings of curtail points for the six pipelines structures shown in Table 5.5. Here \( d \) and \( en \) denote pipeline delay and enqueue time respectively.

5.7.2. Results

Figure 5.10 summarizes the effect of variation in the pipeline structure on the percentage of optimal solutions.

5.7.3. Discussion

We note that as we progressively make the pipeline structure more complex the percentage of optimal solutions, for the same curtail points, decreases. This is because the number of delay slots associated with different instructions increases.
Table 5.5. Variations in the Pipeline Structures

<table>
<thead>
<tr>
<th>Pipeline Structure</th>
<th>Pipelined Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load</td>
</tr>
<tr>
<td></td>
<td>$d$</td>
</tr>
<tr>
<td>#1</td>
<td>1</td>
</tr>
<tr>
<td>#2</td>
<td>2</td>
</tr>
<tr>
<td>#3</td>
<td>2</td>
</tr>
<tr>
<td>#4</td>
<td>2</td>
</tr>
<tr>
<td>#5</td>
<td>2</td>
</tr>
<tr>
<td>#6</td>
<td>2</td>
</tr>
</tbody>
</table>

Therefore, for more complex pipeline structures, our algorithm has to go down deeper in the search tree before the pruning based on the minimum cost (see Section 3.2.1.2) can be done. This in turn increases the number of search calls required to find optimal solutions, and for a fixed set of curtail points the percentage of optimal solutions decreases.

Perhaps a different tree rearrangement criteria (that was discussed in Section 3.2.1.3) based on pipeline structure would be helpful in reducing the runtime.

5.8. Sub-Optimal Solutions

Throughout this chapter, we have talked about the percentage of solutions that were guaranteed to be optimal by our scheduling algorithm. We also have shown that this percentage is very high for all the cases that we have tested. A natural question that arises is “what about the cases which are not optimal?” How bad are they compared to the optimal solutions? In this section, we attempt
Figure 5.10. Average Percentage Run To Completion Vs. Pipeline Structure

to answer this question and show some interesting properties of suboptimal solutions found in our experiments.

5.8.1. Procedure

The suboptimal solutions were isolated and studied using the following procedure.

[1] We started with a large sample of basic blocks, similar to the one described in section 5.4.1.

[2] These basic blocks were run with a curtail point of 1000.

[3] All those basic blocks that our algorithm was able to schedule optimally were discarded from the sample.

[4] The value of the curtail point was doubled and the remaining basic blocks were run with the new value of curtail point.

[5] Steps [3] and [4] were repeated until the curtail point was as high as 512,000.

The set of basic blocks obtained by applying these steps represents a sample that produces 0% optimal solutions for a curtail point of 512,000. The number of
final NOPs for each curtail point for these inputs was recorded.

5.8.2. Results

Figure 5.11 shows the number of final NOPs as a fraction of the initial (without any code scheduling) NOPs versus the number of search calls made.

![Figure 5.11. Fraction of Initial NOPs Vs. Runtime](image)

The average number of initial and final NOPs that are obtained after each curtain point truncation, for this sample of basic blocks, is given in Table 5.6.

5.8.3. Discussion

The most remarkable feature of our algorithm depicted by Figure 5.11 is how quickly it converges to a near-optimal solution. We call it near-optimal because increasing the number of search calls by over five hundred fold we could
Table 5.6. Final NOPs in suboptimal Solutions

<table>
<thead>
<tr>
<th>Calls</th>
<th>Initial NOPs</th>
<th>Fraction of Initial NOPs</th>
<th>Final NOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>13</td>
<td>0.417</td>
<td>5.92</td>
</tr>
<tr>
<td>2000</td>
<td>13</td>
<td>0.417</td>
<td>5.92</td>
</tr>
<tr>
<td>4000</td>
<td>13</td>
<td>0.402</td>
<td>5.69</td>
</tr>
<tr>
<td>8000</td>
<td>13</td>
<td>0.392</td>
<td>5.54</td>
</tr>
<tr>
<td>16000</td>
<td>13</td>
<td>0.386</td>
<td>5.46</td>
</tr>
<tr>
<td>32000</td>
<td>13</td>
<td>0.382</td>
<td>5.38</td>
</tr>
<tr>
<td>64000</td>
<td>13</td>
<td>0.367</td>
<td>5.15</td>
</tr>
<tr>
<td>128000</td>
<td>13</td>
<td>0.362</td>
<td>5.08</td>
</tr>
<tr>
<td>256000</td>
<td>13</td>
<td>0.352</td>
<td>4.92</td>
</tr>
<tr>
<td>512000</td>
<td>13</td>
<td>0.332</td>
<td>4.62</td>
</tr>
</tbody>
</table>

improve the suboptimal solution by only a small fraction. Also note that while the average number for the final number of NOPs for optimal solutions is about one, the same average for suboptimal solutions is around five. Therefore we conclude that those basic blocks which have a high count of NOPs in their optimal solutions will generally result in suboptimal solutions when scheduled by our algorithm with reasonable values of the curtail point. This also follows from the pruning based on the minimum cost that our algorithm uses.

From this discussion we can assume that if we can not find an optimal schedule with a low enough value of curtail point then it is worthless to continue search because the quality of the solution is going to improve only marginally (if at all).

5.9. Summary

The huge search space for optimal (minimal NOP) code schedules has long discouraged researchers from attempting to find optimal code schedules. However, we have presented a search algorithm which has demonstrated that for over
98% of our realistic synthetic benchmark blocks it is possible to dramatically reduce the size of this search space without sacrificing optimality. For the fewer than 2% in which the search space cannot be completely searched, good results were obtained by simply truncating the search, although this may result in suboptimal schedules. A prototype compiler using our algorithm, running on workstation-class machines, schedules about 100 typical blocks per second (>10K source LPM).

For very large basic blocks, it might be useful to split the basic blocks into smaller sections (containing, say, twenty instructions or less each) and find solutions which are locally optimal. A good heuristic for the split might be to simply partition the list schedule, however, we have not yet examined such techniques.
CHAPTER 8

CONCLUSIONS

We have presented an algorithm that searches for an optimal schedule for multiple pipeline processors that dramatically reduce the size of search space without sacrificing optimality.

Previous approaches simplify the search for a good schedule by arbitrarily imposing constraints which sacrifices optimality. Our algorithm uses techniques that ensure that the optimality is preserved. For the fewer than 2% of the cases (in our test runs) in which the search space cannot be completely searched, near-optimal good results were obtained by simply truncating the search, although this may result in suboptimal schedules.

In addition to demonstrating the feasibility of optimal code scheduling, we have defined our algorithm to use a more general model of pipeline structure than previous work. Our model allows multiple pipelines, each with its own latency and enqueue time, to be specified. Further, the set of pipelines which may be used for each type of instruction can be independently specified.

Ongoing work examines performance using various (more complex) pipeline structures than the work presented here. Future work will extend the proposed pipeline scheduling algorithm to more general code structures including very large blocks (as might be generated by trace scheduling [Ell85]) and arbitrary control flow. As presented here, the algorithm applies best to scheduling individual basic blocks averaging about 20 or fewer instructions each.
LIST OF REFERENCES


B. Smith, from numerous personal communications. B. Smith is currently at Tera Computer Company, Seattle, WA 98103.