1-1-1990

Optimal Code Scheduling for Multiple-Pipeline Processors

Ashar Nisar  
*Purdue University*

Henry G. Dietz  
*Purdue University*

Follow this and additional works at: https://docs.lib.purdue.edu/ecetr

https://docs.lib.purdue.edu/ecetr/703

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
Optimal Code Scheduling for Multiple-Pipeline Processors

Ashar Nisar
Henry G. Dietz

TR-EE 90-11
January, 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907
Optimal Code Scheduling for Multiple-Pipeline Processors

Ashar Nisar and Henry Dietz
School of Electrical Engineering
Purdue University
West Lafayette, IN 47906

ABSTRACT

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 latency can be hidden by other useful operations. The portion of pipeline latency 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 paper, 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.

Keywords: pipelines, code/instruction scheduling, optimizing compilers, pipeline latency, pipeline enqueue time.
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 reorganizing, 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.

3. 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 2).

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%).

The basic characteristics of pipelined systems are reviewed and the terminology to be used in the remainder of this paper is given in section 2. Section 3 presents an overview of the complexity of the code scheduling problem viewed as an exhaustive search problem. The structure of our prototype compiler and algorithm are discussed in section 4; performance of our approach is summarized in section 5. Finally, section 6 presents conclusions and directions for further research.

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 paper 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.
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 incre­
mental 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 com­
  puted by an earlier instruction, but the earlier instruction has not yet com­
  pleted 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

\[
\text{R0:} \quad \text{Load R1,X ;make register R1 = memory[X]} \\
\text{Add R0,R1 ;make register R0 = R0 + R1}
\]

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 **Load** and **Add** instructions.

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

Modifying the above example, a conflict would arise instead of a depen­
dence if the second instruction is another **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 **Load** operation. Con­
sider:

\[
\text{Load R1,X ;make register R1 = memory[X]} \\
\text{Load R2,Y ;make register R2 = memory[Y]}
\]
In this case, the second Load would have to be delayed until the first Load had finished using the MAR — a delay of 1 clock tick would have to be placed between the two 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?

### 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].

Of course, 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. The current popularity of the NOP insertion technique is, 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 paper, 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 paper apply equally well to any architectural implementation of delays.

2.3. The Complexity Of The Problem

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, 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, called Q, which 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 Q was created and its approximate runtime determined on a variety of machines. The average time for one application of Q, 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, Q 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 156,920,924 seconds — just under 5 years! Worse still, most programs contain many such blocks.

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

---

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. However, detailed discussion of block interactions is beyond the scope of this paper.
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). This was implemented, however, we also formulated and implemented a number of other heuristics which pruned the search space significantly without sacrificing optimality. Table 1 presents a sample of how well we were able to prune the search space for schedules for typical blocks. 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.

<table>
<thead>
<tr>
<th>Instructions In Block</th>
<th>Exhaustive Search Calls</th>
<th>Pruning Illegal Calls</th>
<th>Proposed Pruning Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>40,320</td>
<td>163</td>
<td>76</td>
</tr>
<tr>
<td>11</td>
<td>39,916,800</td>
<td>9,039</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>6.2x10^9</td>
<td>65,105</td>
<td>394</td>
</tr>
<tr>
<td>13</td>
<td>6.2x10^9</td>
<td>40,240</td>
<td>21</td>
</tr>
<tr>
<td>14</td>
<td>8.7x10^10</td>
<td>175,384</td>
<td>1,676</td>
</tr>
<tr>
<td>16</td>
<td>2.1x10^13</td>
<td>27,487</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>2.1x10^13</td>
<td>5,800,000</td>
<td>66,890</td>
</tr>
<tr>
<td>16</td>
<td>2.1x10^13</td>
<td>92,228,324</td>
<td>5,434</td>
</tr>
<tr>
<td>20</td>
<td>2.4x10^18</td>
<td>12,872</td>
<td>334</td>
</tr>
<tr>
<td>21</td>
<td>5.1x10^19</td>
<td>58,581</td>
<td>202</td>
</tr>
<tr>
<td>22</td>
<td>1.1x10^21</td>
<td>&gt;9,999,000</td>
<td>119</td>
</tr>
</tbody>
</table>

Table 1: Search Space for Representative Examples

Of course, despite the fact that our pruning works very well on average, it has terrible worst-case performance. To limit the worst-case runtime for our algorithm, the concept of a curtail 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\(^2\). In this case, the best

\(^2\) 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 \( \lambda \) schedules have been examined (i.e., \( \lambda \) calls have been made to \( \Omega \)). 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 \( \lambda \) is on the order of 1,000. In fact, for most blocks of fewer than 20 instructions, a \( \lambda \) 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 1.

![Figure 1: Schedules Searched Vs. Block Size for 15,812 Complete Runs](image)

Unfortunately, in the case that a reasonable \( \lambda \) is exceeded and the search is truncated by rule [2], we were generally unable to determine how often the schedule is actually optimal despite the fact that some schedules were not considered. This is due to the fact that when a reasonable value of \( \lambda \) was exceeded, the search space tended to be very large, so that even increasing the \( \lambda \) 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.
3. 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 2 shows the organization of the compiler back end in the prototype implementation.

![Figure 2: Organization of Prototype Scheduling Compiler](image)

Each phase is discussed briefly below. Section 3.1 discusses optimized tuple generation. The main contributions of this paper are discussed in sections 3.2 and 3.3: respectively, the list scheduler and the pipeline scheduler. Finally, register allocation and code generation are reviewed in section 3.4.

3.1. 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 $i, O, \alpha, \beta$ 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 3.

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

```
1: Const 15 $\Gamma_{1,\text{Const},"15"}$
2: Store #b, 1 $\Gamma_{2,\text{Store},"b",1}$
3: Load #a $\Gamma_{3,\text{Load},"a"}$
4: Mul 1, 3 $\Gamma_{4,\text{Mul},1,3}$
5: Store #a, 4 $\Gamma_{5,\text{Store},"a",4}$
```

**Figure 3: 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 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.

3.2. 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 list 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 an \( \alpha-\beta \) 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 \( \alpha-\beta \) pruning, the time taken in applying the list 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.

3.3. Pipeline Scheduler

Having obtained a "reasonable" initial schedule, the pipeline schedule search algorithm is applied to find the optimal schedule. This algorithm, given in section 4.2, represents the prime contribution of this paper. The output is simply a schedule of the tuples within each block.

3.4. Register Allocation and Code Generation

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. 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. Scheduling Algorithm

Before presenting the scheduling algorithm, it is useful to define the information which will be used as input to the pipeline scheduler. In the previous section, an overview was given of the tuple form representing each basic block to be scheduled. Section 4.1 presents a similar overview of the pipeline configuration information the search procedure needs in order to determine the optimality of a schedule. The following section, section 4.2, presents the scheduling algorithm itself.

4.1. Pipeline Configuration Information

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 2.

<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>

Table 2: Sample Pipeline Description Table

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

<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>

Table 3: Sample Operation-to-Pipeline Mapping

The results presented in this paper were obtained using a more conservative, single pipeline unit per function, the tables for which appear in section 5.1. 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 \textit{not} examine these tables, hence, the initial schedule is independent of the target pipeline structure.

4.2. Pipeline Scheduling Algorithm

The input to the pipeline scheduling algorithm is an initial (list) schedule and the DAG (Direct Acyclic Graph) \cite{AhS86} it embeds. From this, all needed dependence information is derived. The pipeline scheduling algorithm is a heavily-pruned search algorithm in which the minimum valid number of NOPs are inserted before each instruction is added to each partial schedule. The schedule with the fewest NOPs inserted is the best schedule.

Section 4.2.1 defines a few terms and functions used in the algorithm. The algorithm itself is presented in two parts: the NOP insertion algorithm in section 4.2.2 and the complete search procedure in section 4.2.3.

4.2.1. Definitions

The following terms and functions are used in the algorithms which follow:

\textbf{Definition 1:} \(\Pi\)

\(\Pi\) is the current complete ordering of all instructions within this basic

\(^3\) The algorithm presented in section 4.2 does not support this feature.
block. The $i^{th}$ instruction in $\Pi$ will be denoted as $\Pi(i)$; likewise, $\Pi^{-1}(\delta)$ returns the position of instruction $\delta$ within $\Pi$. Instructions within $\Pi$ are labeled 1, 2, 3, ..., $|\Pi|$.

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

$\rho(\zeta)$ is the set of all instructions $\delta \in \Pi$ if $\zeta$ has an immediate dependence on $\delta$. Equivalently, $\rho(\zeta)$ is the set of all immediate predecessors of $\zeta$ in the DAG described above.

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

$\sigma(\zeta)$ is the pipeline resource used by instruction $\zeta$.

**Definition 4:** $\eta(i)$

$\eta(i)$ is the number of NOPs inserted immediately before the $i^{th}$ instruction within $\Pi$.

**Definition 5:** $\mu(\Pi)$

$\mu(\Pi) = \sum_{j=1}^{\Pi} \eta(j)$, the total number of NOPs required by the schedule $\Pi$.

**Definition 6:** $earliest(\zeta)$

$earliest(\zeta)$ is the minimum number of instructions in $\Pi$ 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 7:** $latest(\zeta)$

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

### 4.2.2. NOP Insertion Algorithm

The following algorithm is used to determine the number of NOPs which would need to be inserted in the schedule $\Pi$ immediately before the $i^{th}$ instruction, $\zeta$. It is assumed that for each instruction scheduled in a position $j < i$, $\eta(j)$ has previously been set to the number of NOPs which must be inserted immediately before that instruction. The algorithm is:

1. $\eta(i) = 0$. If $i = 1$, then done. Otherwise, go to step [2].
2. If $\sigma(\zeta) = \emptyset$, goto step [4].
3. (Check for conflict.) Let $\tau(j) = \eta(i) + \sum_{k=j+1}^{i-1} \eta(k) + 1$, the execution time between the start of the $j^{th}$ instruction and the $i^{th}$ instruction. Search backward from the $j=i-1^{th}$ instruction until $\tau(j) > enqueue time of \sigma(i) \cup \sigma(j) = \sigma(i) \cup j=1$. If $\sigma(j) = \sigma(i) \cup \tau(j) < enqueue time of \sigma(i)$, then $\eta(i) = enqueue time of \sigma(i) - \tau(j)$. 


Optimal Scheduling

[4] If $\rho(\zeta) = \emptyset$, then done.

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

[6] Let $x = \text{latency of pipeline } \eta(\Pi^{-1}(\delta)) - \tau(\Pi^{-1}(\delta)).$ If $x > 0$, then $\eta(i) = \eta(i) + x.$

4.2.3. The Search Procedure

The following is the schedule search algorithm which forms the core of our approach. It uses the NOP insertion algorithm given above and the initial list schedule as, $\Pi$, the current block to schedule.

[1] For $i = 1$ to $|\Pi|$, invoke the above algorithm to insert the correct number of NOPs before instruction $\Pi(i)$. Call the resulting schedule $\pi$, the best schedule found thus far.

[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 $\mu(\Pi) < \mu(\pi)$, then $\pi = \Pi$. Goto step [7].

[4] (Apply curtail point search truncation.) Let $\Lambda = \Lambda + 1$. If $\Lambda \geq \lambda$ then done, with a possibly suboptimal best schedule $\pi$. Otherwise, continue with step [5].

[5] (Get next schedule pruned by legality and equivalence checks.) Consider swapping instruction $\kappa = \Pi(i) \mid \kappa \in \Phi$ with an instruction $\xi \in \Psi$. The swap should be performed only if all of [5a], [5b], and [5c] are true:

[5a] (Quick approximate check for legality.)

\[
\text{latest}(\kappa) \geq \Pi^{-1}(\xi) \cap \text{earliest}(\xi) \leq i
\]

[5b] (Real test for legality.) $\rho(\xi) \subseteq \Phi$

[5c] (Check for equivalence.)

\[
\sigma(\xi) \neq \emptyset \cup \rho(\xi) \neq \emptyset \cup \sigma(\kappa) \neq \emptyset \cup \rho(\kappa) \neq \emptyset
\]

If no legal swap was found, goto step [7]. Otherwise, interchange $\xi$ with $\kappa$ (which alters $\Pi$, $\Phi$, and $\Psi$) and invoke the above algorithm to insert NOPs for this last instruction.

[6] (Apply $\alpha$-$\beta$ pruning.) If $\mu(\Phi) < \mu(\pi)$, then move the partition between $\Phi$ and $\Psi$ to reduce $\Psi$ by one instruction and goto step [3]. Otherwise, continue with step [7].

[7] 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.

[8] If $i < |\Psi|$ then $i = i+1$ and goto step [3]. Otherwise, done, with an optimal solution in $\pi$.

The $\alpha$-$\beta$ and other pruning cuts the search time by $(|\Pi|-k)!$ when pruning occurs at position $k$. Note that, because condition [5c] 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.

5. Results

A prototype compiler implementing the algorithms given in section 4.2 was tested with carefully generated benchmark programs. These programs were synthesized according to statistics obtained from “real” programs.

Section 5.1 gives the pipeline descriptions used. The construction of the synthetic benchmark programs is given in 5.2. Finally, the results are summarized in section 5.3.

5.1. Pipeline Constraints for Simulations

All the results shown in this paper were obtained using a very straightforward pipeline design. These pipeline constraints appear in tables 4 and 5. Later studies will examine performance on more varied and complex pipeline structures; the purpose of this paper is to demonstrate that optimal code scheduling is possible, not to study variations in performance associated with different pipeline structures.

<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 4: Pipeline Description for Simulations
5.2. 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].

Note Table 6 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.
5.3. Simulation Results

The results presented in this paper 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 items searched for an optimal search of an “average” block of that size. A very brief summary of the results appears in Table 7.

<table>
<thead>
<tr>
<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>16,000</td>
</tr>
<tr>
<td>Percentage of Runs</td>
<td>98.83%</td>
<td>1.17%</td>
</tr>
<tr>
<td>Avg. Instructions/Block</td>
<td>20.50</td>
<td>32.28</td>
</tr>
<tr>
<td>Avg. Initial NOPs</td>
<td>9.50</td>
<td>14.34</td>
</tr>
<tr>
<td>Avg. Final NOPs</td>
<td>0.67</td>
<td>4.03</td>
</tr>
<tr>
<td>Avg. Ω Calls</td>
<td>427.4</td>
<td>54,150</td>
</tr>
<tr>
<td>Avg. Search Time (Sun 3/50)</td>
<td>~0.1s</td>
<td>~15s</td>
</tr>
</tbody>
</table>

Table 7: Statistics for Scheduling 16,000 Blocks

Notice that the average number of instructions per block 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 4 shows the final number of NOPs after optimization versus the initial number of NOPs. Note that the initial number of NOPs grow linearly with the number of instructions, but the final number of NOPs remains nearly constant.

Figure 5 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 less 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 even included such blocks in our study to show the worst-case effectiveness of our algorithm.
Figure 4: Initial and Final NOPs Vs. Block Size

Figure 5: Distribution of Sample Block Sizes

Figure 6 shows the average runtime over all 16,000 sample blocks. Figure 7 shows the percentage of all runs which found optimal schedules, i.e., which were not pruned by \( \lambda \). From these two graphs, it can easily be seen that common block sizes are easily scheduled within a reasonable compile time, and usually can be optimally scheduled within that time.
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 by 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 indicates that the algorithm quickly converges to a near-optimal solution.

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.

6. Conclusions

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).

Figure 6: Runtime Vs. Block Size

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 by 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 indicates that the algorithm quickly converges to a near-optimal solution.

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.

6. Conclusions

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).
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.

7. References


Parallel Processing 1988, pp. 430-433.


[Smi88] B. Smith, from numerous personal communications. B. Smith is currently at Tera Computer Company, Seattle, WA 98103.