Software Pipelining and Superblock Scheduling: Compilation Techniques for VLIW Machines

Meng Lee, Partha Tirumalai, Tin-Fook Ngai
Computer Systems Laboratory
HPL-92-78
June, 1992

Compilers for VLIW and superscalar processors have to expose instruction-level parallelism to effectively utilize the hardware. Software pipelining is a scheduling technique to overlap successive iterations of loops, while superblock scheduling extracts ILP from frequently executed traces. This paper describes an effort to employ both software pipelining and superblock scheduling techniques in a VLIW compiler. Our results show that the two techniques used together are effective for accelerating a variety of programs.
1 Introduction

Many recently introduced processors have the ability to exploit instruction-level parallelism (ILP). For instance, a RISC machine with an $n$-stage pipeline can keep up to $n$ operations in flight. Hence, a VLIW [1, 2, 3] or superscalar [4, 5] machine that issues $m$ operations per cycle can sustain up to $m \times n$ operations in flight. To fully exploit such hardware, a compiler must find as much ILP as to saturate the machine's parallelism. Experience has shown that a single basic block typically has insufficient ILP to fully exploit a VLIW machine; hence the search for techniques that find ILP across successive basic blocks. Two techniques for extracting ILP are software pipelining [6, 7] and superblock scheduling [8, 9, 10].

Software pipelining overlaps successive basic blocks from successive iterations of an innermost loop. Although this technique greatly increases performance by exposing ILP within loops, it is not applicable outside of loops. Another technique, called superblock scheduling, has demonstrated substantial performance improvement on a set of C benchmarks [11]. Superblock scheduling uses profile information gathered from previous executions of a program to construct larger blocks with potentially higher ILP. It is in fact a restricted version of trace scheduling [12].

This report presents results derived by simultaneously applying both techniques. Ten Fortran benchmarks are analyzed, and upon close examination, it appears that both techniques are required to obtain the highest performance across a variety of programs. Section 2 of this report provides a brief description of software pipelining and superblock scheduling. Section 3 describes superblock scheduling implementation in detail. Section 4 shows the improvements resulting from these techniques. Section 5 states the bottlenecks observed in this study, and suggests future work. Section 6 concludes the report.

2 Software Pipelining & Superblock Scheduling

2.1 Overview of software pipelining

Software pipelining parallelizes loops by starting new iterations before previous iterations complete. Successive iterations start every $\Pi$ (initiation interval) cycles. A single iteration's schedule can be divided into $SC$ (stage count) stages, each consisting of $\Pi$ cycles. Fig. 1 shows the execution of five iterations of a four-stage software-pipelined loop. The three phases during execution of the loop are ramp up, steady state, and ramp down. The first $(SC-1)\times \Pi$ cycles, when not all stages of the software pipeline execute, constitute the ramp-up phase. The steady-state portion begins with the last stage of the first iteration. During the steady-state phase, one iteration completes for every one that starts. The steady-state phase ends when the first stage of the last iteration has completed. For the final $(SC-1)\times \Pi$ cycles, known as the ramp-down phase, one iteration completes every $\Pi$ cycles. The execution of the code generated for the loop is divided into three parts: prologue, kernel, and epilogue.

Assume that without software pipelining every iteration takes $L$ cycles to complete. With software pipelining, after the prologue, every new iteration takes $\Pi$ cycles to complete. The speed up is a factor of $L/\Pi$, when ignoring the overhead of the prologue. Scheduling the prologue and epilogue code in parallel with the code outside the loop would minimize this overhead. The overhead is insignificant for loops with large trip counts. The compiler used in this study has achieved nearly optimal $\Pi$ over more than 1300 loops in various benchmarks. A detailed description of the implementation is presented in [13].
2.2 Speculative execution --- hardware support for superblock scheduling

Speculative execution refers to the issuing of an operation before it is known that its execution is required. Speculative execution occurs when the scheduler places an operation above a preceding conditional branch. Operations that are data ready but not control ready (that is, operations for which the input operands have been computed but to which control flow has not yet reached) are candidates for speculative execution. It is desirable for the compiler to make use of speculative operations to reduce the time required for a computation. However, operations that can cause exceptions cannot be executed speculatively (to avoid spurious exceptions). An example of speculative execution is shown in Fig. 2.

```
Original Program
a = expression-1;
b = expression-2;
if (b != 0) {
y = a/b;
}

Program with Speculative Execution
a = expression-1;
b = expression-2;
t = a/b; /* speculative divide must not cause an exception */
if (b != 0) {
y = t;
}
```

Figure 2. Example of speculative execution.
On most commercial processors, exceptions can be disallowed on simple integer operations. To derive maximum benefit from speculative scheduling, an architecture should permit the speculative execution of nearly all operations. One way to achieve this is to simply turn off exception processing for most operations by setting a mode bit in the processor. The Trace [2] and Cydra 5 [3] compilers adopt this approach at the highest levels of optimization. However, this significantly changes a program's error reporting behavior because operations that would have caused errors will cause no errors after optimization. A second option is to provide non-trapping versions of those operations that can cause exceptions. These non-trapping operations can be used in speculative execution. With this approach, the program's error reporting behavior is improved because errors caused by non-speculative operations will halt execution when they occur. With additional architectural support, a program can precisely report errors even in the presence of speculative execution [14].

A couple of issues remain regarding speculative execution. For instance, although speculative execution can speed up the execution of a program from the scheduler's point of view, a single page fault caused by a speculative load may result in the loss of thousands of processor cycles. To alleviate this difficulty, a policy could be conceived where page faults from speculative loads are not serviced immediately; rather, they are postponed and serviced only when the data is truly required. A similar policy could be used with TLB and cache misses. A second concern is regarding programs that rely on precise error reporting to trap and recover gracefully from unexpected or illegal situations. In such situations, the use of speculative execution either requires extra care or must be avoided.

This study permits the most general type of speculative execution (referred to as the general percolation model in [14]). Operations other than branches and memory stores are allowed to be executed speculatively across branches.

2.3 Description of superblock scheduling

A superblock is a block of operations which has only one entry but one or more exit points. Superblocks are formed such that all operations in a superblock are likely to be executed; early exits are infrequently taken. Within a superblock, the compiler schedules operations to generate denser code by speculatively moving operations up past branches.

A superblock is built by selecting a trace (a set of basic blocks that are frequently executed in sequence), and fusing the blocks in the trace into a single block. Traces are limited in length due to several restrictions like procedure boundaries, loop boundaries, etc. Fig. 3 shows a segment of a program as a control flow graph annotated with profiling information. Each basic block is annotated with its execution count. Each control flow arc is annotated with the branch probability. A possible trace is constructed as blocks 1, 2, 3 and 4 together. But a side entrance into block 3 precludes the formation of one large superblock. Instead, blocks 1 & 2 and blocks 3 & 4 are fused to form separate superblocks.

Intuitively, the longer a trace the more opportunity the scheduler has to generate denser code. However, large superblocks are rare in common program graphs. Two simple program transformation techniques can be used to form larger superblocks: tail duplication and loop unrolling. By duplicating appropriate basic blocks (for example, blocks 3 and 4 in the above figure), tail duplication removes side entrances and yields larger superblocks. Loop unrolling replicates the body of a loop to give a larger block. Sections 3.1 and 3.2 describe in detail how tail duplication and loop unrolling are implemented in this study.
3 Implementation of Superblock Scheduling

The compiler used in this study is based on the Fortran77 production compiler for the Cydra 5 mini-supercomputer. The numeric processor in the Cydra 5 is a VLIW processor. The Fortran compiler has a state-of-the-art implementation of software pipelining for innermost DO loops [13].

The compiler performs syntactic and semantic analyses of the input program, followed by machine-independent local and global optimization [15, 16]. Then loop analysis is applied to each innermost DO loop. It performs vector data dependence analysis, attempts to transform the loop and produces an overlapped schedule. After loop analysis, the compiler performs global register allocation and data dependence analysis for code other than overlapped loops. Finally, for each basic block, scheduling and local register allocation take place before code emission. This compiler was not designed for flexible phase ordering among transformations and optimizations. Many implementation decisions made in this study had to accommodate the original structure; however, the effect on the code quality is not significant.

One task that precedes superblock formation is profiling. Extra instructions that accumulate visit counts are inserted into the program when compiled to generate profile information. When the instrumented program executes, it dumps the basic block visit counts and branch taken/not taken counts. This profile information is later used during compilation if superblock optimization is invoked. The compiler updates the profile counts as it applies program transformations and optimizations. The performance numbers presented in this report are calculated using the compiler's updated counts and scheduled block lengths in cycles.
3.1 Superblock identification and tail duplication

Superblocks are formed from the most frequently executed traces based on profile information. The compiler processes one procedure at a time. It first locates the most frequently executed basic block in a procedure as the beginning of the trace. Then the trace is grown forward by following branch edges with probability more than 50%. The trace is terminated when one of the following conditions is met.

1. The next block is already in a previously marked trace. If the next block is in the current trace and is not the leading block, the trace is retracted and terminated before the next block.

2. The next block is not likely to be executed more than once per ten executions of the entry block of the procedure.

3. The estimated code increase due to tail duplication, that is the amount of code after the first side entry into the trace, is more than 50% of the code in the trace.

4. If tail duplication is disallowed in the compilation, the trace is terminated before any side entrances.

Conditions 2 and 3 limit the amount of code explosion due to tail duplication, and ensure that the trace is likely to be executed from its beginning block to its last block. The trace is a candidate for superblock formation. The above procedure is repeatedly applied to the remaining basic blocks until no more traces can be identified.

Figure 4. Removing side entrance by tail duplication.
If a trace contains any side entrances, the side entrances are removed by tail duplication. Fig. 4 illustrates how tail duplication produces a longer trace resulting in a larger superblock. Blocks 1, 2, 3 and 4 are identified as a trace. Blocks following the side entrance, i.e. blocks 3 and 4, are removed from the trace and replaced by the duplicated copies, i.e. blocks 3' and 4'. The side entrance from block 6 remains intact. The old profile information is updated proportionally for the new control flow graph. The new counts may not be accurate because the proportionality assumption can be invalid.

In case profile information is not available, a heuristic can be used to predict the most frequently executed trace. The predicted trace is either inside a loop or along the longest path to an exit/return block. It is expected that a longer trace will form a larger superblock yielding higher performance. Before any superblock identification and tail duplication, all blocks in the procedure are ranked by the length of their longest non-cyclic paths to an exit/return block. The superblock trace identification is as follows: the block with the longest path length (the highest rank) to an exit/return block is selected as the leading block of the trace. When growing the trace, if there are any back edges, the back edge whose target block has the lowest rank is traversed. If there are no back edges, the branch target block with the highest rank is chosen.

3.2 Loop unrolling & superblock formation

Loop unrolling is a special form of tail duplication, since the tail can be viewed as the loop body itself. When a trace constitutes a frequently executed loop, it can be expanded many times by simply unrolling the loop body. Innermost DO loops are not considered for unrolling since they may be eligible for software pipelining. Fig. 5 shows how a loop is unrolled to give a larger superblock.

![Figure 5. Loop unrolling.](image-url)
Loop unrolling is performed after tail duplication is applied to a trace which has been identified as a loop. The loop execution count and the branch back probability for each loop iteration are first determined. The number of times the loop is unrolled is the smaller of the average trip count and five. This ensures that the loop is likely to execute the entire unrolled body at least once per procedure invocation and also limits code explosion. When profile information is not available, every loop is always unrolled five times.

To take the advantage of global optimization on the new flow structure, tail duplication and loop unrolling are done before the global optimization phase. After global optimization and loop analysis, superblocks are formed. During the formation, the basic blocks in the superblock are repositioned consecutively and grouped into one large sequential block. From this point onwards, a superblock is treated as one unit and interfaced with the rest of the program the same way as a basic block.

3.3 Dependence analysis

Dependence analysis associates pairs of operations that must be scheduled in a fixed relative order to preserve the semantics of the program. A directed acyclic graph (DAG) is constructed with operations as nodes and dependencies as edges between nodes. Data-, output-, anti- and miscellaneous dependencies are reflected in the DAG [17]. Since there can be branch nodes inside superblocks, extra dependencies are added as follows:

1. Operations with side effects, which cannot be speculatively executed, are not allowed to move across any branches.

2. Since global register allocation has already been done, and since compensation code is disallowed in superblock scheduling, operations that modify global registers cannot move (either upward or downward) across branches if the registers carry live-in values for the branch target blocks.

Since there is no inter-procedural analysis, operations are not allowed to move across subroutine calls.

3.4 Superblock and meld scheduling

The input to the scheduler is a list of superblocks (and ordinary blocks). For blocks that are not software pipelined, the scheduler places operations to minimize program execution time subject to machine resource and dependency constraints. This involves scheduling individual superblocks as well as inter-superblock transitions, which allows operations issued in one block to complete in a succeeding block (meld scheduling). The machine has no interlocks and has to be completely scheduled statically to produce the correct answer. The only exception is the non-deterministic duration of a memory read. If a memory read does not complete by the expected time (i.e. read latency), the processor will preserve correctness by stalling until the data arrives.

Nodes in the DAG for a block may be native machine operations, generic operations (which can be cast on more than one functional unit) or macros (which can be expanded into a list of native machine operations). A block is scheduled as follows. Two special nodes, START and STOP, are added as the predecessor and successor of all nodes in the DAG. Operations are ordered bottom up from the STOP node, with priority given to operations on the critical path. List scheduling [18] is employed to place operations in time. In the process, the scheduler also selects the best-suited native machine instructions for nodes that are either generic or macro operations. For ordinary blocks, the scheduler attempts to minimize the distance between the START and STOP nodes. For superblocks, it attempts to simultaneously minimize the distance from the
START node to the exits inside the block as well as to the STOP node. If a conflict arises, the heuristic favours minimizing the total length of the block. Branches have delay slots that the scheduler tries to fill with useful operations. After all operations have been scheduled, local register allocation is attempted. If it fails, the block is rescheduled with operations prioritized by register lifetimes, then local register allocation is attempted. If it fails again, the block is scheduled after inserting spill code.

Meld scheduling takes care of dependency and machine resource constraints at transitions between blocks. As dependency information between operations in different blocks is not available, the scheduler itself does simple dependence analysis to derive data-, output- and anti-dependencies between operations in a block and those in its successors and predecessors. Since meld scheduling allows an operation issued in one block to complete in a successor block, operations dangle across more than one block. With superblocks the dangles can go through the early exits as well. Cycles can cause a dangle from a block back into itself. Multiple cycles imply that any linear combination of the elementary cycles can also occur. Software-pipelined loop blocks present the possibility of operations dangling through a block of varying length based on the trip count. The scheduler takes all possible dangling situations into account in deriving the top and bottom boundary conditions for a block. These conditions determine the spacing, if any, that is needed in transitions between blocks.

3.5 Register allocation

Graph coloring [19, 20] is used for register allocation. For this study, there was no modification to global register allocation, which is done before the superblocks are formed. A concern about the early global register allocation is that it might place significant restrictions on the scheduler. Upon closer examination, however, it was found that the extra restrictions described in section 3.3, lower performance by less than 1\% when compared with no restrictions. This occurs because only a few allocated global registers are live out on the early exits of a superblock. They place no new data dependencies between operations, and the extra dependencies upon branches are likely to be absorbed by other critical paths in the DAG.

Local register allocation for superblocks does not have to be different from that for ordinary blocks. However, the register pressure is higher in superblocks due to higher instruction-level parallelism. Tuning the register allocator might yield higher performance.

Table I. The machine model.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Number</th>
<th>Latency</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>Integer and floating point adds, compares, and logical</td>
</tr>
<tr>
<td>Multiply</td>
<td>1</td>
<td>2</td>
<td>Integer and floating point multiply</td>
</tr>
<tr>
<td>Memory</td>
<td>2</td>
<td>3 (read)</td>
<td>Memory reads and writes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 (write)</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>2</td>
<td>1</td>
<td>Address calculation</td>
</tr>
<tr>
<td>Control</td>
<td>1</td>
<td>2</td>
<td>Branches</td>
</tr>
</tbody>
</table>

4 Performance

The target machine model is described in Table I. It is similar to the Cydra 5 [3] but with shorter latencies and multiple ports to the general register file. Up to seven operations can be issued in each clock cycle. The data path and the functional units can handle double precision data. Speculative (non-trapping) execution of nearly all operations, except memory writes and branches,
is permitted. The machine has architectural support for the software pipelining of innermost loops, namely, rotating register files and predicates.

In general, there is a complex interaction between the code produced by the compiler and the cache stalls that occur when a program is run. It is possible that the application of a technique such as software pipelining can increase or decrease cache misses. The impact of superblock scheduling on cache performance is also not obvious. Superblocks group frequently executed basic blocks together, thus improving local instruction locality. But there may be considerable code increase due to tail duplication and loop unrolling which can worsen cache performance. In this study, the primary interest is in investigating the increased instruction level parallelism exposed by the application of both superblock scheduling and software pipelining in the compiler. Therefore, the study is simplified by assuming a perfect cache.

4.1 Performance with software pipelining and superblock scheduling

Fig. 6 shows the effect of applying software pipelining and superblock scheduling to ten Fortran benchmark programs. These benchmark programs were selected from the Linpack, SPEC and Perfect Club suites. Five of the programs, linpack, srs, ocs, tis and aps, are loop intensive; the remaining programs, lws, spice, lgs, css and doduc, are not. The reference base is the execution time when at most one operation is issued per cycle (a RISC-like machine); it corresponds to 100.0 in Fig. 6. The figure shows the execution times normalized to the base reference with the following optimizations applied: (1) neither software pipelining nor superblock scheduling, (2) superblock scheduling only, (3) software pipelining only, and (4) both superblock scheduling and software pipelining. In all four cases, more than one operation can be issued per cycle up to the width of the machine. For loop intensive programs, software pipelining yields benefit and additional superblock scheduling provides little improvement. For scalar programs, superblock scheduling gives significant benefit.

![Figure 6. Execution times with software pipelining and superblock scheduling.](image-url)
The benefits of superblock scheduling and software pipelining are not independent. Superblock scheduling, if applied in the absence of software pipelining, would result in innermost loops being unrolled, and thus provide improvement on loop code. Fig. 6 also shows how well superblock scheduling does with innermost loops unrolled (SB) and with innermost loops software pipelined (Both). It appears that superblock scheduling with loop unrolling does not perform as well as software pipelining on the innermost loops except for css. Two possible reasons are: (1) For loops with small trip counts, the overhead associated with the setting up and tearing down of the software pipeline is too high; (2) Some loops fail to software pipeline (due to reasons such as early loop exits or function calls in the loop body), and they fall back to traditional (single iteration) scheduling. In these cases, unrolling provides more improvement.

4.2 Effect of superblock scheduling

Fig. 7 shows the ratio of the scheduled lengths of 311 superblocks to the sum of the lengths of the constituent blocks when they are scheduled individually. These blocks were taken from the most highly executed routines in the five benchmarks that are not loop intensive (lgs, css, lws, doduc, spice). For the vast majority of the blocks there is improvement with superblock scheduling. The average ratio is 0.85 (15% decrease in scheduled length). The decrease in the lengths of the blocks indicates an increase in instruction level parallelism. In a few cases, the scheduling heuristic produces slightly poorer results. This can occur because constraints from minimizing the distance to the early exits can conflict with the minimization of the overall length of the block. In addition, register spills are more likely to be introduced into larger blocks.

![Figure 7. Effect of superblock scheduling on block length.](image)

On the average, without applying tail duplication and loop unrolling, there are 2.49 basic blocks in a superblock; with tail duplication and loop unrolling, the average is 3.89. The average number of operations issued per cycle increased from 1.36 for ordinary blocks to 1.62 for superblocks. It should be noted that because of the speculative execution of operations inside superblocks, not all operations issued inside superblocks are useful. Nevertheless, the decrease in a schedule's length is indicative of performance improvement.
4.3 Overall machine utilization

Table II shows some additional characteristics and performance data for the ten benchmarks. The execution time of each program has been broken up into three parts. The percentage under the SWP loop column represents the fraction of time spent inside software-pipelined loops. The percentage under the superblock column represents the fraction of time spent inside blocks that later become part of superblocks.

Table II also shows the number of operations issued per cycle (the upper number), and the number of operations in flight per cycle (the lower number) with the five different optimizations applied to each benchmark. The latter number represents the amount of instruction level parallelism exploited by the compiler for the machine. The width of the machine and the latency of the operations both affect these numbers. Linpack, which is ideally suited to the machine model, shows the best performance. It spends a significant amount of its execution time in a loop whose operations almost match the ratio of the functional units in the machine (2 memory, 2 address, 1 adder and 1 multiplier), and keeps the machine saturated. The other programs show a range of performance depending upon the code content and the match between the application and the machine.

Table II. Performance data for the ten benchmarks.

<table>
<thead>
<tr>
<th>Program name</th>
<th>SWP Loop %</th>
<th>Super-block %</th>
<th>Base</th>
<th>None</th>
<th>SB</th>
<th>SWP</th>
<th>Both</th>
</tr>
</thead>
<tbody>
<tr>
<td>linpack</td>
<td>85.77</td>
<td>4.72</td>
<td>0.99</td>
<td>1.08</td>
<td>1.16</td>
<td>4.16</td>
<td>4.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.61</td>
<td>1.76</td>
<td>1.89</td>
<td>6.85</td>
<td>6.91</td>
</tr>
<tr>
<td>srs</td>
<td>97.54</td>
<td>0.18</td>
<td>0.90</td>
<td>1.20</td>
<td>1.40</td>
<td>2.46</td>
<td>2.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.74</td>
<td>2.31</td>
<td>2.69</td>
<td>4.76</td>
<td>4.76</td>
</tr>
<tr>
<td>ocs</td>
<td>89.63</td>
<td>0.96</td>
<td>0.96</td>
<td>1.27</td>
<td>1.80</td>
<td>1.97</td>
<td>1.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.54</td>
<td>2.03</td>
<td>2.89</td>
<td>3.16</td>
<td>3.17</td>
</tr>
<tr>
<td>aps</td>
<td>60.71</td>
<td>12.13</td>
<td>0.86</td>
<td>1.24</td>
<td>1.43</td>
<td>1.89</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.65</td>
<td>2.38</td>
<td>2.78</td>
<td>3.39</td>
<td>3.42</td>
</tr>
<tr>
<td>lgs</td>
<td>7.07</td>
<td>30.83</td>
<td>0.88</td>
<td>1.29</td>
<td>1.47</td>
<td>1.55</td>
<td>1.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.65</td>
<td>2.41</td>
<td>2.75</td>
<td>2.89</td>
<td>2.99</td>
</tr>
<tr>
<td>tis</td>
<td>84.92</td>
<td>5.49</td>
<td>1.00</td>
<td>1.19</td>
<td>1.48</td>
<td>1.88</td>
<td>1.88</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.64</td>
<td>1.96</td>
<td>2.46</td>
<td>3.20</td>
<td>3.20</td>
</tr>
<tr>
<td>lws</td>
<td>35.68</td>
<td>26.66</td>
<td>0.85</td>
<td>1.09</td>
<td>1.26</td>
<td>1.63</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.56</td>
<td>1.99</td>
<td>2.37</td>
<td>2.86</td>
<td>2.94</td>
</tr>
<tr>
<td>doduc</td>
<td>10.60</td>
<td>55.01</td>
<td>0.81</td>
<td>1.10</td>
<td>1.28</td>
<td>1.24</td>
<td>1.42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.69</td>
<td>2.31</td>
<td>2.68</td>
<td>2.53</td>
<td>2.89</td>
</tr>
<tr>
<td>css</td>
<td>7.76</td>
<td>61.52</td>
<td>0.84</td>
<td>1.15</td>
<td>1.35</td>
<td>1.19</td>
<td>1.39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.63</td>
<td>2.24</td>
<td>2.60</td>
<td>2.31</td>
<td>2.66</td>
</tr>
<tr>
<td>spice</td>
<td>2.48</td>
<td>87.18</td>
<td>0.78</td>
<td>0.87</td>
<td>1.11</td>
<td>0.89</td>
<td>1.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.31</td>
<td>1.45</td>
<td>1.91</td>
<td>1.49</td>
<td>1.93</td>
</tr>
</tbody>
</table>

When applying both techniques, Table III presents the percentage of execution time broken down by the number of operations issued per cycle. Although it seems that these programs spent most of the time issuing 0 to 3 ops/cycle, the execution time when more than 4 operations are issued per cycle is not negligible.
Table III. Histogram of percentage of (0-7)-operations issued per cycle.

<table>
<thead>
<tr>
<th>name</th>
<th>0-issue</th>
<th>1-issue</th>
<th>2-issue</th>
<th>3-issue</th>
<th>4-issue</th>
<th>5-issue</th>
<th>6-issue</th>
<th>7-issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>linpack</td>
<td>1.6</td>
<td>5.6</td>
<td>5.8</td>
<td>24.1</td>
<td>19.8</td>
<td>7.2</td>
<td>35.9</td>
<td>0.0</td>
</tr>
<tr>
<td>srs</td>
<td>24.9</td>
<td>20.9</td>
<td>12.5</td>
<td>6.8</td>
<td>10.8</td>
<td>9.8</td>
<td>12.0</td>
<td>2.7</td>
</tr>
<tr>
<td>ocs</td>
<td>12.3</td>
<td>30.5</td>
<td>30.8</td>
<td>12.6</td>
<td>2.7</td>
<td>10.6</td>
<td>0.5</td>
<td>0.0</td>
</tr>
<tr>
<td>aps</td>
<td>25.7</td>
<td>24.5</td>
<td>21.3</td>
<td>9.1</td>
<td>6.6</td>
<td>7.4</td>
<td>5.3</td>
<td>0.1</td>
</tr>
<tr>
<td>lgs</td>
<td>24.4</td>
<td>28.7</td>
<td>25.7</td>
<td>10.3</td>
<td>3.6</td>
<td>7.1</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>tis</td>
<td>8.7</td>
<td>56.9</td>
<td>3.0</td>
<td>16.2</td>
<td>0.1</td>
<td>14.9</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td>lws</td>
<td>18.9</td>
<td>34.6</td>
<td>22.6</td>
<td>16.7</td>
<td>1.7</td>
<td>4.1</td>
<td>1.4</td>
<td>0.3</td>
</tr>
<tr>
<td>css</td>
<td>23.9</td>
<td>38.4</td>
<td>21.5</td>
<td>9.3</td>
<td>4.4</td>
<td>2.1</td>
<td>0.3</td>
<td>0.0</td>
</tr>
<tr>
<td>spice</td>
<td>20.7</td>
<td>57.5</td>
<td>14.9</td>
<td>4.4</td>
<td>1.1</td>
<td>1.5</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

4.4 Effect of tail duplication

Superblock optimization involves superblock formation followed by scheduling. In superblock formation, tail duplication is used to remove branches and form larger superblocks. Tail duplication can also be used to form larger ordinary basic blocks (rather than superblocks) to increase ILP. Fig. 8 shows a comparison of three cases: tail duplication alone to form larger ordinary blocks (no superblock scheduling), superblock scheduling without tail duplication, and superblock scheduling with tail duplication. In all three cases, software pipelining is applied to innermost loops. As expected, the loop intensive benchmarks show little change with or without tail duplication and superblock scheduling. For the other benchmarks, the best performance is obtained when both techniques are applied; neither by itself is sufficient.

In the machine model, the hardware support for rotating registers and predicated execution permits the generation of extremely compact (kernel only) code for software-pipelined innermost loops.
Hence, the code increase is mostly due to tail duplication and unrolling of loops that are not software pipelinable. For the ten benchmarks, the increase ranged from 3% to 20%, with an average of 12%.

5 Discussion

There are some legitimate concerns regarding profile-based compiler optimizations and scheduling. First, how well does the profile collected from one or a few input data sets represent a program's behavior on other input data sets? Second, even if a program can be well represented by a profile, is the extra effort to collect a profile and recompile justified by the improvement from these techniques? If reasonable superblocks can be formed using heuristic branch predictions, it would save the extra effort. Fig. 9 shows the results without superblock optimization, with superblock optimization using the heuristics described in 3.1 & 3.2 in the absence of profile data, and in the presence of profile data. The profile data for these experiments is the same as the input data. This is the best scenario for profiled based optimizations. From the figure, it can be seen that some performance improvement occurs even in the absence of profile information.

![Figure 9. Performance improvement w/ and w/o profile information.](image)

It is intuitive that larger blocks should provide higher ILP. Therefore, eliminating branches and procedure calls could help. Inlining of function calls, code replication [21], and versioning (i.e. duplicating code by replacing invariants through constant propagation) [22, 23] can also be used to increase the block size. For instance, inlining the most frequently executed routine in doduc resulted in a 15% performance improvement.

Some shortcomings inherited from Cydra 5 compiler hurt the overall performance. The Cydra 5 architecture also imposes a set of restrictions that reduce performance. For instance, code alignment considerations occasionally dictate generating uniop code (one operation per cycle) instead of multiop code (multiple operations per cycle).

To obtain better results, a global optimizer which handles superblocks, powerful block expansion techniques, and a simpler machine model are important. A powerful memory disambiguator can
shorten critical paths in blocks to yield a better schedule. Blending the prologue and epilogue code from software pipelining with the code outside loops reduces overhead. Recognizing general loop structures in the program and transforming them into while-loops provide the opportunity to software pipeline them as described in [24]. More wisely choosing between software pipelining and loop unrolling can improve the performance of short trip count loops.

6 Conclusion

This report presents results on the application of two techniques, software pipelining and superblock scheduling, for exploiting instruction level parallelism. These techniques are useful on processors that possess parallelism through pipelined operations, or multiple functional units, or both. Architectural support for speculative execution is required to obtain maximum benefit from superblock scheduling. Data from ten benchmarks show that both techniques have to be applied together to derive the highest performance. For loop intensive programs, software pipelining provides impressive performance gains ranging from 50% to 300%. For programs that are not loop intensive, superblock scheduling provides significant improvement ranging from 10% to 25%. This indicates that such programs present a challenge for architectures and compilers that rely on instruction level parallelism.

Acknowledgements

Profiling was implemented by Joseph Park and Vinod Kathail. W. W. Hwu provided a lot of insights to implementing superblock scheduling. And Richard Huff gave valuable comments on an early draft of this paper.

References


