Modulo Scheduling with Cache Reuse Information*

Chen Ding\(^1\) and Steve Carr\(^2\) and Phil Sweany\(^2\)

\(^1\) Dept. of Computer Science, Rice University, Houston TX 77005-1892
\(^2\) Dept. of Computer Science, Michigan Technological University, Houghton MI 49931

Abstract. Software pipelining for instruction-level parallel computers with non-blocking caches usually assigns memory access latency by assuming either all accesses are cache hits or all are cache misses. We contend setting memory latencies by cache reuse analysis leads to better software pipelining than either an all-hit or all-miss assumption. Using a simple cache-reuse model, our software pipelining optimization achieved 10\% improved execution performance over assuming all-cache-hits and used 18\% fewer registers than required by an all-cache-miss assumption. We conclude that software pipelining for architectures with non-blocking cache should incorporate a memory-reuse model.

1 Introduction

In modern processors, main-memory access time is at least an order of magnitude slower than processor speed. To tolerate main memory latency, hardware designers use non-blocking caches which allow cache accesses to continue when misses occur \(^2\). This, in turn, allows an instruction scheduler to overlap more operations with memory accesses, possibly hiding main-memory latency. Thus, a significant increase in instruction-level parallelism (ILP) can be achieved.

Even though memory latency is variable with non-blocking caches, instruction schedulers typically either assume that all memory accesses are cache hits or that all are cache misses. Assuming all hits reduces register lifetimes keeping register pressure to a minimum. However, significant penalties are incurred when a cache miss occurs. Assuming all cache misses tolerates the latency of a cache miss better, but may increase register pressure significantly. Knowing the latency of a memory operation allows the instruction scheduler to get the best of both schemes. Previous work has used profiling to determine memory latencies for architectures with non-blocking caches \(^1\) or has used software prefetching instructions to hide main-memory latency \(^6\). We report here on an experimental evaluation of modulo scheduling using cache-reuse analysis \(^5\) to choose the proper latency for loads and stores on machines with non-blocking caches and no prefetching instructions.

\(^*\) This work was partially supported by the National Science Foundation under grants CCR-9409341 and CCR-9308348, as well as a grant from Digital Equipment.
2 Experiment

To evaluate our contention that taking advantage of memory reuse information can improve software pipelining's efficiency, we compiled and simulated 120 Fortran loops in which our software pipelining implementation [7] used one of three different memory latency policies, namely 1) all loads are cache hits, 2) all loads are cache misses, 3) each load is either always a cache hit or always a cache miss, as determined by memory reuse analysis [5]. When reuse analysis determines that an array reference has any reuse, we assume the reference is always a cache hit; otherwise, we assume it is always a cache miss. Our hypothesis is that software pipelining in which the load latency is determined using this model yields better execution performance than pipelining with an all-hit latency policy, and though it should lead to slightly poorer execution performance than pipelining with an all-miss latency policy (assuming an infinite number of registers) the reuse policy leads to significantly fewer registers required for the loop.

2.1 Machine Model

The hypothetical superscalar architecture we chose has two integer and two floating point functional units; each may issue an instruction in each cycle should data dependences allow. Latency for integer instructions is two cycles; latency for floating point instructions is four cycles. Only one load or store can be issued per cycle. All loads and stores use an integer unit and the cache hit latency is two cycles.

Because one parameter we wished to investigate was register usage, we could have chosen a fixed, relatively small number of registers similar to current ILP machines and “measured” register pressure as part of execution time, since spilling would necessarily degrade loop performance. However, to separate register concerns and loop performance concerns, we assume 256 integer and 256 floating point registers, thus ensuring no spill code and allowing evaluation of register pressure effects by a direct measurement of registers required to generate software pipelined code for the loop.

Our cache model is an 8K, direct-mapped cache with 32-byte lines. The cache is non-blocking and allows up to 6 outstanding misses to occur in parallel. The penalty for a cache miss is an additional 25 cycles. On each miss, two consecutive 32-byte lines are brought into the cache.

We simulated loop behavior by resetting the simulator for each outermost loop construct. Thus, although we only pipelined innermost loops we counted all nested loops in our simulation results. However, we did not simulate non-loop code.

2.2 Test Programs

We software pipelined 107 Fortran innermost loops from three SPEC programs (hydro2d, su2cor, swm256) and an additional 13 loops from Fortran kernels, yielding a total of 120 innermost loops. For 45 loops there was no difference
in any of the pipelined schedules. We used iterative modulo scheduling [7] to pipeline the loops, and restricted our attention to loops with no control flow or function calls. Thus, we pipelined only single-block loops in this study.

2.3 Results

Table 1 summarizes results for the performance, in terms of execution cycles, of the 75 loops tested. Column one shows “normalized” execution time for code compiled with an all-hit latency policy. Column two shows the same computation for the all-miss latency policy and column three lists results of code compiled with reuse information. We normalized execution cycles so that whichever of the three compiled codes (hit, miss, reuse) required the fewest cycles was set to 100, and the other two were normalized with that value. Table 1’s values represent the unweighted average of these normalized execution values for all 75 loops. As expected, loops pipelined with latencies set by reuse required fewer cycles, on average, than those compiled with latencies set by a cache-hit assumption (roughly 10% less). Based only on execution cycles, we also expected reuse to perform slightly worse than cache-miss, due in part to our over-simplified model of cache behavior that assumes each static load is either always a hit or always a miss. We anticipated that this would lead to a small performance penalty, but, in fact, virtually all of the roughly 8% degradation in performance between cache miss and reuse policies can be attributed to our over-simplified reuse model, as discussed in Section 2.4. Finally, the summary data shows that miss was not always the best performance policy. Several loops showed better performance with reuse than cache miss. We discuss this unexpected result as well in Section 2.4.

<table>
<thead>
<tr>
<th>Cache Hit</th>
<th>Cache Miss</th>
<th>Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>104</td>
<td>112</td>
</tr>
</tbody>
</table>

Table 1. Summary Performance Numbers — Normalized

In our experiment, the reuse version of a loop improved on the hit version for 17 of the 75 loops tested, while hit never did better than reuse. Thus, all of the roughly 10% average performance improvement was found in less than one fourth of the loops. In fact, for 13 loops, the reuse-compiled code was more than 20% faster than the code compiled with hit latencies. The largest difference was a factor of 2.61. The other 58 loops all produced the same results when compiled with hit latencies or reuse latencies. In contrast, while miss resulted in better schedules 34 times out of 75, reuse outperformed miss 19 times, by as much as 41% in one instance. To obtain the roughly 8% average improvement of cache miss to reuse then, cache miss had to be significantly better than reuse for some loops and, in fact, this is what we found. Although reuse outperformed miss by at least 20% only twice, miss was more than 20% faster than reuse for 19 of the 75 loops. The maximum penalty of reuse for any loop was 69%.

Table 2 shows register requirements of the pipelined loops. While compiling with reuse required about one register more on average than compiling with hit latencies, it took 6 fewer registers than were required by assuming miss latency.
This represents a 17.9% register savings over schedules that assume miss latency. For architectures with moderate numbers of registers this can be a considerable factor in deciding between using miss latencies and reuse information. When we restrict ourselves to those loops in which miss provided at least 20% better execution performance the difference is even greater. For those loops, reuse required an average of 31.6 registers while miss required 40.9, a savings of 22.8%.

<table>
<thead>
<tr>
<th>Cache Hit</th>
<th>Cache Miss</th>
<th>Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.7</td>
<td>40.8</td>
<td>34.6</td>
</tr>
</tbody>
</table>

Table 2. Summary of Registers Required

2.4 Discussion

Our basic premise was that compiling with reuse information would allow both more efficient pipelined loops than would compiling with hit latency and fewer registers required than would compiling with miss latency. Our experimental evidence certainly suggests that this is true.

To understand the reason for the degradation of reuse compared to miss, consider the definition of self-spatial reuse. Self-spatial reuse occurs because an entire cache line is brought into the cache on a single miss. If we assume stride-1 access of data (accessing adjacent data items on successive loop iterations) then self-spatial reuse leads to one miss every N loop iterations, where N is the number of adjacent data elements brought into the cache at once. This is quite different from our compiler's assumption that every access is a hit when we have self-spatial reuse.

Investigation of the 34 loops for which miss led to more effective pipelined schedules showed that each exhibited self-spatial reuse. Many included several self-spatial reuse loads. This means that, in our machine model, each self-spatial reuse load incurs a 25-cycle penalty every 8 loop iterations (since we bring 8 data items into the cache for each miss,) resulting in the observed performance degradation of reuse.

Perhaps more puzzling is the fact that for 19 loops the schedule generated with reuse information required fewer cycles than were needed using miss latency. Our intuition suggested that miss should never yield a worse schedule, but it did. Closer investigation suggested that miss should never yield a worse schedule, but it did. Closer investigation of the loops in question showed that, for each, software pipelining "overhead" of prologue and postlude as well as pre-conditioning (used when a loop is unrolled for modulo variable expansion [4]) was significantly greater for the miss schedule than the reuse schedule. Thus, performance improvement is not an advantage of the reuse model but, rather, a side effect of code generation strategy.

2.5 Refinements

The above experimental data indicates that scheduling with reuse information can achieve performance equivalent to all-cache-miss with lower register pressure
if we properly handle references exhibiting self-spatial reuse. To overcome this
problem, we suggest two possible hardware modifications. First, hardware could
be modified to prefetch the next cache line on a hit or miss if that line were
not already in the cache. Alternatively, load instructions could be modified to
stream a specified number of consecutive cache lines into the cache to reduce the
negative effects of self-spatial accesses.

3 Conclusion

In this paper, we have demonstrated experimentally that using reuse information
while software pipelining is effective. On our benchmark suite we produce on
average 10% better schedules than an all-cache-hit assumption (a factor of 2.61
better on one loop) and on average we use 18% fewer registers than an all-cache-
miss assumption. Even though all-cache-miss sometimes out performs reuse, it
does so at the cost of 23% more registers. We refer the reader to the extended
version of this paper for more details on this experiment [3].

Given that the cycle time of cache-miss latencies is increasing, software pipe-
lining methods must eliminate performance degradation caused by these laten-
cies. The methods presented here are an important step in eliminating the la-
tency problem.

References

1. Abraham, S., Sugumar, R., Windheuser, D., Rau, B., and Gupta, R. Predict-
dability of load/store instruction latencies. In Proceedings of the 26th Interna-
tional Symposium on Microarchitecture (MICRO-26) (Austin, TX, December 1993),
pp. 139–152.
2. Chen, T.-F., and Baer, J.-L. Reducing memory latency via non-blocking and
prefetching caches. In Proceedings of the Fifth International Conference on Ar-
chitectural Support for Programming Languages and Operating Systems (Boston,
3. Deng, C., Carr, S., and Sweany, P. Software pipelining with cache-reuse
4. Lam, M. Software pipelining: An effective scheduling technique for vliw
'88 Conference on Programming Language Design and Implementation.
5. McKinley, K. S., Carr, S., and Tseeng, C.-W. Improving data locality with
loop transformations. ACM Transactions on Programming Languages and Systems
6. Mowry, T. C., Lam, M. S., and Gupta, A. Design and evaluation of a compiler
algorithm for prefetching. In Proceedings of the Fifth International Conference on
Architectural Support for Programming Languages and Operating Systems (Boston,
Massachusetts, 1992), pp. 62–75.
7. Rau, B. Iterative modulo scheduling. In Proceedings of the 27th Interna-
tional Symposium on Microarchitecture (MICRO-27) (San Jose, CA, December 1994), pp. 63–
74.

This article was processed using the \LaTeX\ macro package with LLNCS style