TWO CASE STUDIES IN LATENCY TOLERANT ARCHITECTURES

James E. Bennett
Michael J. Flynn

Technical Report No. CSL-TR-94-639

October 1994

The research described herein has been supported by NASA-Ames under grants NAG2-248 and NAGW 419, using equipment supplied by Silicon Graphics, Inc.

TWO CASE STUDIES IN LATENCY TOLERANT ARCHITECTURES

by

James E. Bennett

Michael J. Flynn

Technical Report No. CSL-TR-94-639

October 1994

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-4055

Abstract

Researchers have proposed a variety of techniques for dealing with memory latency, such as dynamic scheduling, hardware prefetching, software prefetching, and multiple contexts. This paper presents the results of two case studies on the usefulness of some simple techniques for latency tolerance. These techniques are nonblocking caches, reordering of loads and stores, and basic block scheduling for the expected latency of loads. The effectiveness of these techniques was found to vary according to the type of application. While nonblocking caches and load/store reordering consistently improved performance, scheduling based on expected latency was found to decrease performance in most cases. This result shows that the assumption of a uniform miss rate used by the scheduler is incorrect, and suggests that techniques for estimating the miss rates of individual loads are needed. These results were obtained using a new simulation environment, MXS, currently under development.

Key Words and Phrases: Code scheduling, CPU simulation, Memory latency, Nonblocking cache, Prefetching.

Contents

List of Figures

List of Tables

1 Introduction

Processor cycle times are currently much faster than memory cycle times, and the trend has been for this gap to increase over time. The problem of increasing memory latency, relative to processor speed, has been dealt with by adding high speed cache memory. However, depending on the miss rate, memory latency can still have a significant performance impact. Since the trend of increasing memory latency is expected to continue, the performance impact will become even more significant with time.

Researchers have proposed a variety of techniques for dealing with memory latency, many of which have been implemented. These techniques fall into the categories of dynamic scheduling, hardware prefetching, software prefetching, or supporting multiple contexts [GGH92, CB92, MLG92, GHG+91]. Various combinations of techniques for latency tolerance are possible as well. This paper presents the results of two case studies on the usefulness of some simple techniques for latency tolerance. These techniques are nonblocking caches, reordering of loads and stores, and basic block scheduling for the expected latency of loads.

These results were obtained using a new simulation environment, MXS, currently under development. MXS stands for "Memory coupled eXecution based Simulation environment" and consists of a compiler and a simulator. The compiler takes as input a binary file compiled for execution on the MIPS R3000 processor [Kan88], and produces a file that the simulator can run. The compiler is responsible for rescheduling the code as appropriate for the processor model being simulated and for register re-allocation. The simulator then executes the file, simulating both the CPU and the memory subsystem.

The rest of the paper is organized as follows. Section 2 describes the simulation environment and section 3 describes the processor models used in the studies. Then two studies are described, one study involving nonblocking caches and load/store reordering (section 4), and another study on the impact of basic block scheduling for the expected latency of loads (section 5). In section 6 the results of these studies are discussed as well as our plans for enhancing the simulation environment.

2 The Simulation Framework

Figure 1 shows the simulation framework. The benchmark is first compiled and then it is executed by the simulator. During execution, the simulator maintains counts of various statistics of interest, such as number of loads, stores, cache misses, etc. After the execution is completed, the statistics are written out to a results file. Both the compiler and the simulator are configured by means of a parameter file which describes the processor and memory subsystem being modeled. While MXS can simulate a wide variety of benchmarks, it is currently limited to simulating a single process only, so benchmarks involving multiple processes cannot be simulated.

The parameter file is implemented as a C language header file. When the configuration needs to be changed, this file is modified and the MXS compiler and simulator are rebuilt (compile time configuration). The idea behind compile time configuration is that the time to build a new version

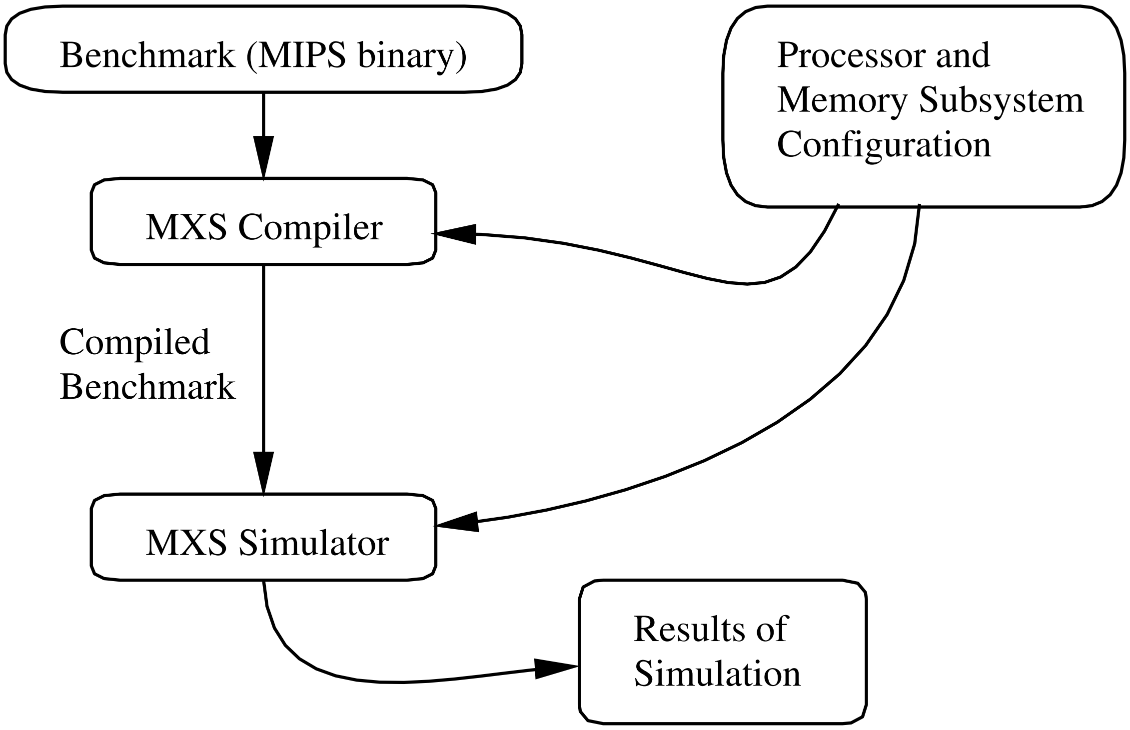

Figure 1: The Simulation Process

of the compiler and the simulator is small compared to the time to run the benchmarks in the simulator.

2.1 The MXS Compiler

The MXS compiler performs the following operations on its binary input file. It builds a list of procedures by reading the symbol table, then passes over the code segment building a basic block graph for each procedure (more procedures may be discovered during this phase). Then the R3000 instructions are translated into the format that is interpreted by the MXS simulator. During this translation the compiler can insert special opcodes that increment counters during simulation. This feature currently is used to count the number of cycles lost to no-ops in the branch delay slot.

The compiler then constructs a call graph of the benchmark, and traverses the call graph from the leaves to the root, performing a dataflow analysis on each procedure [ASU88]. After dataflow analysis, the registers can be renamed, to simulate an architecture with a different number of registers than an R3000. Then each basic block is rescheduled. A dependency graph is built for each basic block and the edges of the graph are annotated with the corresponding operation latencies. A list scheduling algorithm processes the basic block, at each step scheduling the ready operation whose path to the final node of the graph is the longest [Fis81, GM86].

The compiler includes a parameter file which defines the particular CPU and memory system being configured. The parameters defined in this file include instruction latency (which can vary depending on the instruction), size of the cache, cache line size and degree of associativity, cache

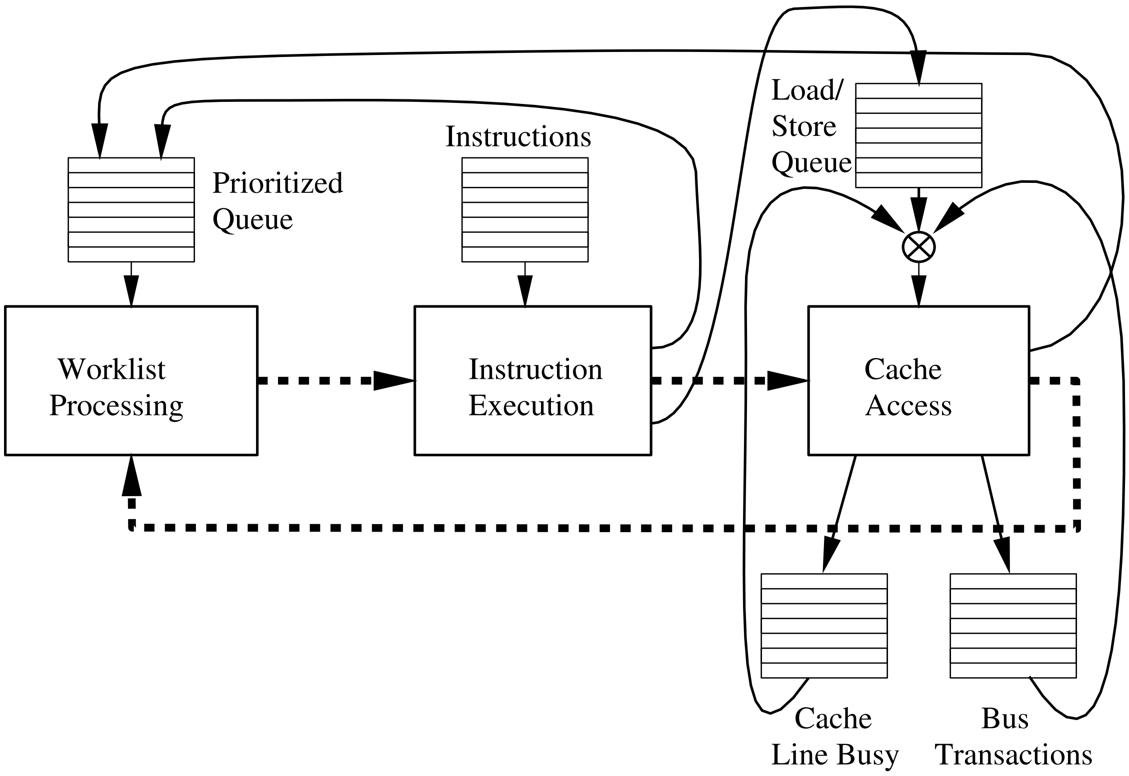

Figure 2: MXS Simulator

read and write miss penalties, and bus bandwidth. The compiler currently makes use of only the instruction latencies and the cache miss penalties. The other parameters are included because the parameter file is shared between the compiler and the simulator.

2.2 The MXS Simulator

In order to achieve reasonable performance, the MXS simulator does not completely simulate the operation of the processor. Instead, it simulates each instruction in terms of its initiation and completion. Each cycle the simulator executes one instruction (assuming that its operands are ready), checks to see if any pending operations complete this cycle, and processes the load/store queue (see figure 2). If the instruction being executed completes in one cycle, then we are through with that instruction.

If the instruction takes multiple cycles to complete, then after the operation is performed, the destination register for that instruction is locked and the register unlock operation added to the work list queue. When the cycle counter advances to the point where the instruction completes execution, the register unlock is pulled from the work list queue and executed. While the register is locked, instructions which use that register are stalled. In this way the exact latencies of instructions can be modeled, without modeling the details of processor operation.

For load and store instructions, the operation is added to the load/store queue, and the subsequent

Operation	*Latency*
Load	2 cycles
Branch	2 cycles
Int. Multiply	12 cycles
Int. Divide	35 cycles
Other int. op	1 cycle
FP Add	2 cycles
FP Multiply	4/5 cycles
FP Divide	12/19 cycles

Table 1: Operation Latencies

processing of the operation depends on the memory subsystem model. In the case of cache misses, where the result may not be available for several cycles, the work list mechanism is again used. The cache miss is translated into a bus request and placed in a list of outstanding bus transactions. The completion of the bus transaction is then added to the work list queue, and at the appropriate cycle it is pulled from this queue and executed.

3 Processor Models

For both of the following studies, the processor being modeled was a statically scheduled scalar processor, with operation latencies taken from the R3000 (see table 1). Where two latencies are given, the smaller number is for single precision and the larger latency is for double precision. The latency for the load is the latency assuming a hit in the cache. The number of registers in the processor was 32 integer and 16 floating point registers for most of the following results. The register renaming option of the second study allowed an unlimited number of registers.

Only a single level of cache was modeled, in this case a 16K byte direct mapped cache with a 16 byte line size. The cache miss penalty was taken to be 10 cycles for both read misses and write misses.

The processor model labeled "blocking" (see figure 3) blocks on both cache read and cache write misses. It remains stalled until the time specified for handling the cache miss has passed. In all other models, the processor continues execution past the load or store up to the point when a register is referenced whose value is not yet available. This limited level of dynamic behaviour in the processor can be implemented using register interlocks.

The processor models labeled "one miss" and "unlimited" have the ability to reorder loads and stores, via a set of queues called the *cache line busy queues*. In these models, loads and stores are allowed to proceed as long as they don't conflict with prior memory operations which are waiting on a cache miss. Conflict detection is handled during cache access. If a memory operation accesses a cache line that is busy (handling a cache miss), then that operation is placed on a cache line busy queue for that cache line.

Subsequent memory operations can then be processed, as long as they don't access a cache line

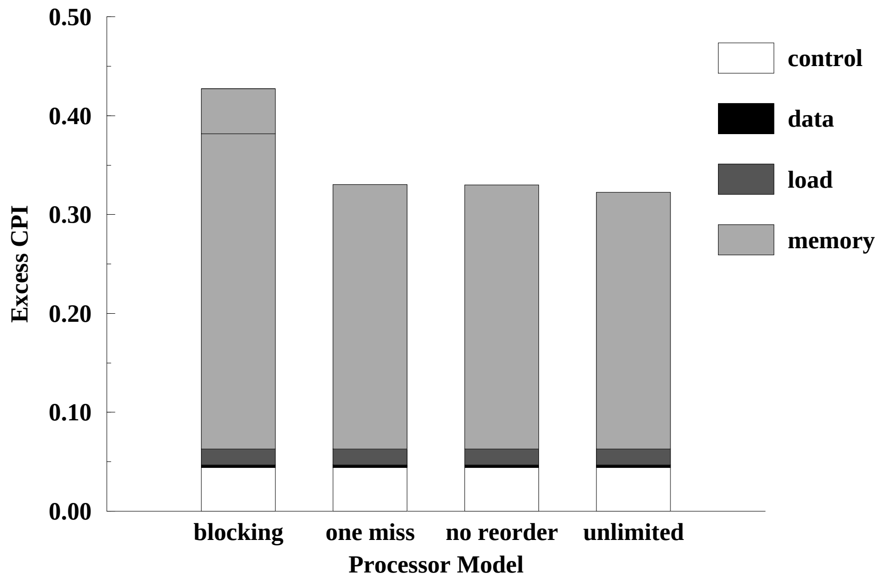

Figure 3: Compress

that is busy. This is safe since if a memory operation accesses a cache line that isn't busy, this means that its address doesn't match the address of any pending memory operation.

If a subsequent memory operation does access a busy cache line, then it is also added to the cache line busy queue for that cache line. When the cache miss handling is completed, the operations in the cache line busy queue are processed. They are given a higher priority than other memory operations. In this way memory operations with addresses that might match are processed in order.

4 Study 1: Nonblocking Cache

One of the simplest techniques to provide for latency tolerance is the nonblocking (or lockup-free) cache [Kro81]. The nonblocking cache allows multiple memory transactions to be in progress, limited by the number of registers allocated to track these transactions (called miss information/status holding registers in [Kro81]). Techniques for latency tolerance, such as those listed in section 1, rely on some mechanism to support multiple outstanding memory transactions. In this sense, the nonblocking cache is the lowest common denominator of techniques for latency tolerance. Studies of nonblocking caches include [CB92, FJ94].

In this study we look at varying the bus bandwidth assumed to be available to handle the outstanding cache misses and varying the ability of the processor to reorder loads and stores in conjunction with a nonblocking cache. Four different system models were used in this study and their per-

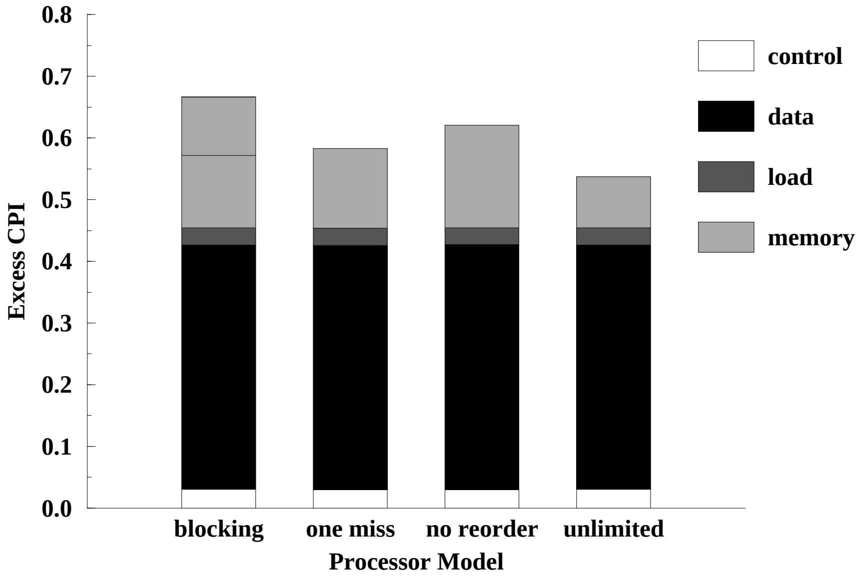

Figure 4: Doduc

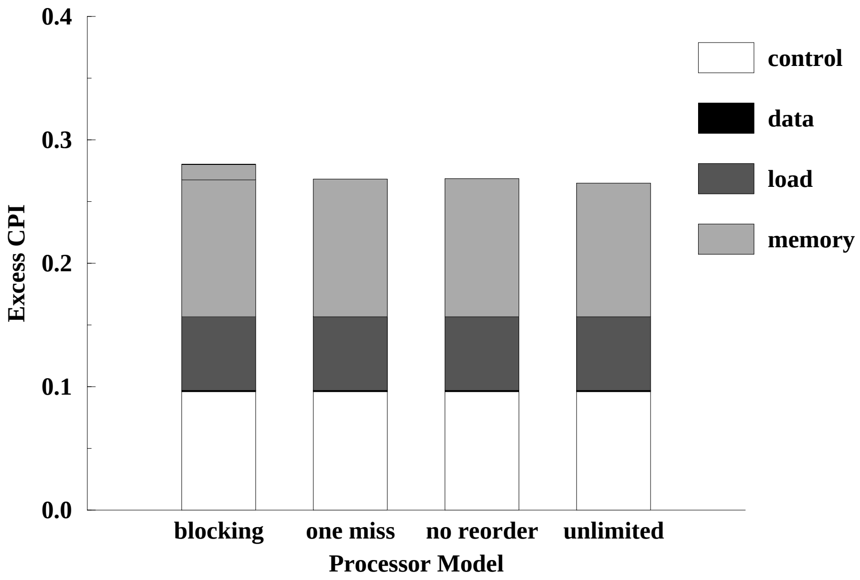

Figure 5: Espresso

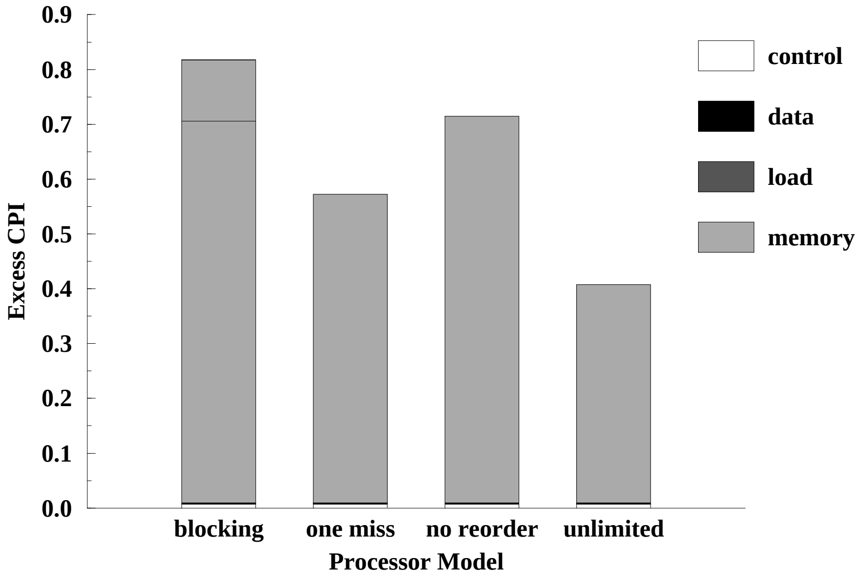

Figure 6: Tomcatv

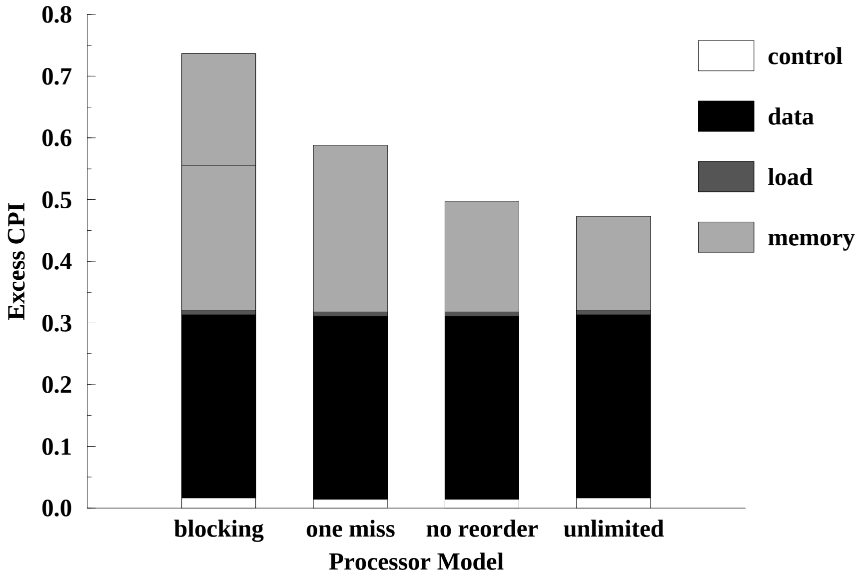

Figure 7: Wave5

formance was compared on five different benchmarks taken from the SPEC92 benchmark suite (see figures 3 through 7). The benchmarks were chosen to represent a variety of different types of applications. All benchmarks were run to completion.

These graphs plot excess CPI versus processor model for the five benchmarks. Excess CPI is the number of stall cycles divided by the number of instructions executed. The excess CPI is broken down according to the source of the delay: control hazard (i.e. an unfilled branch delay slot), data hazard (or data dependency), load delay, or memory latency.

The first model (blocking) stalls on both cache read misses and cache write misses. All other models only stall on a reference to a register whose value is not yet available. In the case of the blocking processor model, the excess CPI due to memory latency is shown with a line that divides the delay due to read misses from the delay due to write misses.

The second model (one miss) allows the processor to continue execution after a single cache miss, but a second cache miss is held off until the first one is processed. This models a system where the bus bandwidth is limited to a single outstanding transaction at a time (also called "hit-under-miss"). Loads and stores following the first cache miss which hit in the cache are allowed to proceed. Reordering is enabled in this case, which means that if there are several cache accesses to the same cache line which all miss, they are placed in a cache line busy queue, and subsequent loads and stores to other cache lines are allowed to proceed. However, if the second cache miss occurs on a separate cache line this blocks all subsequent loads and stores.

The third model (no reorder) allows an unlimited number of concurrent cache misses, but doesn't allow loads and stores to be reordered. When reordering is turned off, there are no cache line busy queues. In this case, if two cache misses occur on distinct lines, they will both be processed and placed on the bus in order. Subsequent loads and stores can then be processed. However, if two cache misses occur on the same cache line, the second cache miss blocks all subsequent loads and stores.

The fourth model (unlimited) allows an unlimited number of concurrent cache misses, and allows reordering of loads and stores.

In all cases it can be seen that the latency due to write misses in the blocking processor model is easily recovered. So it appears that the nonblocking cache functions as an excellent write buffer. However, to go beyond this depends heavily on the benchmark. In some cases (tomcatv and wave5), more than half of the total memory latency is recovered, but in others (compress and espresso) there is very little additional improvement beyond reducing write miss latency.

An interesting feature of these graphs is the way that the importance of reordering varies with the application. In the case of tomcatv and doduc, when the processor is not allowed to reorder loads and stores, its performance drops compared to the one miss case. The only case where the one miss model will outperform the no reorder model is if there are multiple cache misses on the same cache line. So the cache misses in doduc and tomcatv must be primarily of this kind. This would happen, for example, if these programs stride through large arrays using unit strides.

On the other hand, wave5 gets nearly its full performance without benefit of reordering. This indicates that cache misses in wave5 mostly occur on distinct cache lines, which would be the case if it was randomly accessing a large array, or if it typically strides through the array using strides larger than the cache line size.

The lack of improvement in compress and espresso indicates that for these applications, in most cases, values are referenced soon after the load is issued. Then the benefit of letting the processor proceed until the register is referenced is minimal.

Although this study demonstrates that a nonblocking cache offers a performance advantage, there is still in all of these graphs some amount of memory latency left. This raises the question of whether these results can be improved. One idea is to reschedule the code, taking into account the potential memory latency. This approach is investigated in the next section.

5 Study 2: Scheduling for Expected Latency

To incorporate memory latency concerns into the compiler, the scheduling algorithm (see section 2.1) needs to be modified. The simplest change would be to replace the load latency with the potential latency (i.e. the cache miss penalty) in the annotated dependency graph. However, since most cache references hit, this could produce a bad schedule.

For example, in a basic block with a series of loads and floating point operations, the original scheduler would attempt to schedule the operations in the load delay slots and the loads in the latency slots of the floating point operations. The resulting schedule would then tend to intersperse the loads and operations.

On the other hand, if the cache miss penalty were used as the load latency, then the loads would tend to be scheduled at the top of the basic block, and the floating point operations would be pushed to the bottom. If, when the code is executed, all of the loads actually do miss in the cache, then this schedule will perform better than the originally produced schedule. However, if the loads all hit in the cache, the second schedule will suffer because it doesn't overlap loads and operations the way the first schedule does.

An example of this effect is shown in figure 8, where two schedules are shown for the same computation ($z = w + v * (y + 2x)$). The NOP's were added to indicate where stalls would occur, assuming a two cycle latency for both loads and floating point operations. The original scheduler produces a schedule (schedule 1) that is two cycles shorter than the schedule based on a longer load latency (schedule 2). However, if the load into R4 misses in the cache, then schedule 2 executes in fewer cycles than schedule 1, because the miss processing starts two cycles earlier.

The overall effect of the new schedule then depends on the miss rate of the loads. In this case, if the loads miss in the cache more than half the time, then schedule 2 would give better overall performance than schedule 1. To decide which schedule is better, the probabilities of the misses have to be considered. In general, the schedule which has the lowest *expected value* for its length should be chosen [Hoe71].

```
Schedule 1                              Schedule 2

LOAD     R1, <addr>                     LOAD     R1, <addr>
LOAD     R2, <addr>                     LOAD     R2, <addr>
ADD      R5, R1, R1                     LOAD     R3, <addr>
LOAD     R3, <addr>                     LOAD     R4, <addr>
ADD      R6, R5, R2                     ADD      R5, R1, R1
LOAD     R4, <addr>                     NOP
MULT     R7, R6, R3                     ADD      R6, R5, R2
NOP                                     NOP
ADD      R8, R7, R4                     MULT     R7, R6, R3
NOP                                     NOP
STORE    R8, <addr>                     ADD      R8, R7, R4
                                        NOP
                                        STORE    R8, <addr>
```

Figure 8: Rescheduling for Latency

The MXS compiler was modified to compute this expected value and to minimize it during scheduling (see [KE93] for an alternative approach to scheduling for memory latency). The details of this operation are as follows:

- Associated with each operation is a fixed latency (L) and a probabilistic latency (PL), both expressed in cycles (The probabilistic latency is equal to the cache miss penalty).
- The dependency graph is annotated with both these latencies, and when an instruction is scheduled, its destination register is marked with the latencies of that instruction.
- The cost of scheduling an instruction which references a particular register is the expected value of the latency associated with that register, $L + MR * PL$, where MR is the miss rate.
- Among all instructions that are ready at each step, the instruction with the lowest cost is scheduled. When there is a choice between instructions with an equal cost, the instruction with the longest path to the final node is scheduled.
- The length of the path is also computed as $L + MR * PL$, where in this case L and PL refer to the sum of the latencies and probabilistic latencies, respectively, along the path.
- After an instruction is scheduled, the latencies and probabilistic latencies associated with each register are updated. They are set to zero for registers that the instruction references, and otherwise are decremented by the number of cycles consumed by the instruction.

To test out the effect of this modified scheduling algorithm, some benchmarks from the SPEC92 benchmark suite were run with both scheduling algorithms, and their performance was compared. The processor model in all cases was the same as the unlimited model from section 4. A selection of the results are shown in figures 9 through 12 for the different compiler options (which are defined in table 2). The benchmarks were chosen to show the variety of responses to this optimization.

The somewhat surprising result was that the performance generally got worse when probabilistic scheduling was enabled. The only exception to this among the benchmarks was tomcatv (see

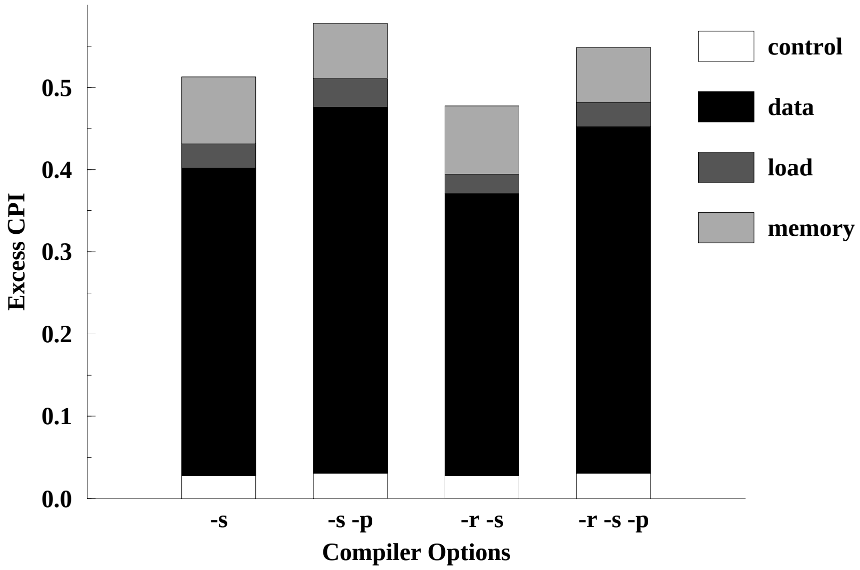

Figure 9: Doduc

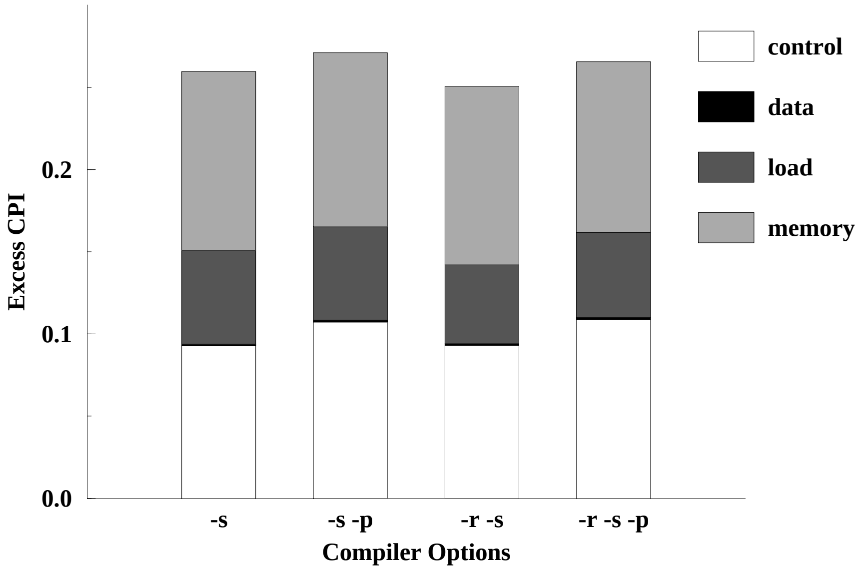

Figure 10: Espresso

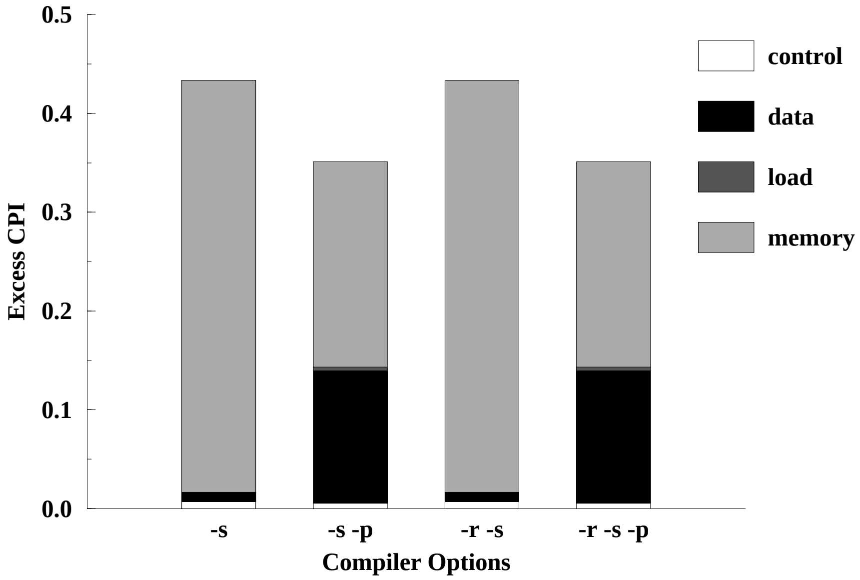

Figure 11: Tomcatv

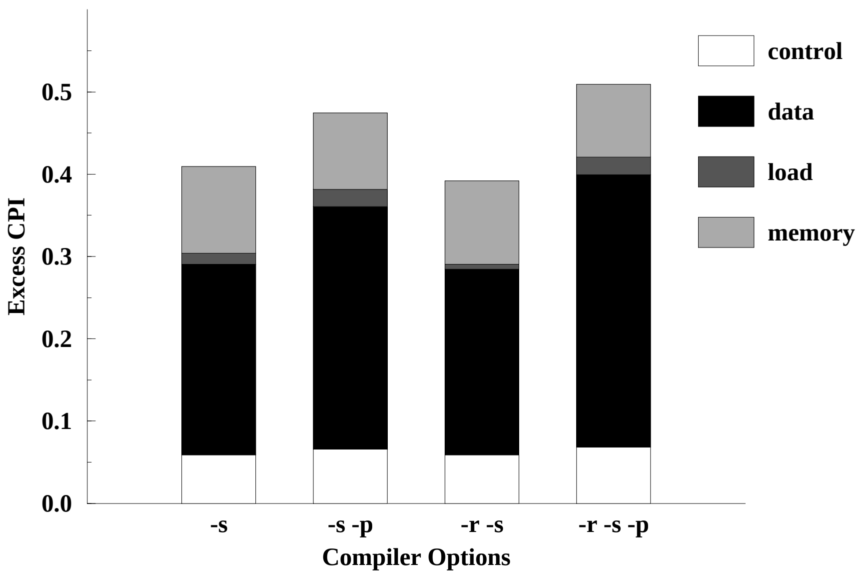

Figure 12: Mdljdp2

Option	*Use*
-s	Reschedule basic blocks
-p	Use probabilistic scheduling for loads
-r	Rename registers

Table 2: Compiler Options

figure 11). To test if perhaps the scheduler was doing a poor job because it was overconstrained, the same tests were run again with register renaming enabled. Registers were renamed assuming an unlimited number of registers, to eliminate the dependencies caused by register reuse. However this did not change the relative impact of enabling probabilistic scheduling. In fact, in one case (see figure 12) the scheduler used this extra freedom to produce a schedule that performed worse than when no register renaming was performed.

In all cases, the scheduler was able to reduce the excess CPI due to memory latency, however the cost of doing this was an increase in the other components of the excess CPI. Usually the tradeoff was with delay due to data dependencies, although for espresso (see figure 10) the tradeoff was with delay due to control dependencies (i.e. unfilled branch delay slots). This tradeoff occurred even when the overall performance improved; the end result of the rescheduling depended on whether the benefit of the memory latency reduction was outweighed by the increase in the other components.

The scheduler was supposed to take this tradeoff into account, using the miss probabilities, as described above. To give the scheduler the best available information, it was passed the actual read miss rate of the benchmark, as determined by a previous run. How could the performance actually decline in this case?

The big assumption that is made when the scheduler is computing the expected value of the latency is that all loads have an equal probability of missing. In fact, different load instructions have different miss rates [ASW+93]. For example, an instruction that is loading a scalar variable within a loop will almost always hit in the cache, while an instruction that is striding through a large array in the same loop will frequently miss. From the results of this study it is clear that these difference are significant, and should be taken into account by any scheduler which attempts to minimize expected latency.

6 Conclusion

In study 1, nonblocking caches were shown to be effective in hiding write miss latency, and for some benchmarks (but not all) in hiding a significant portion of the read miss latency. Performance also generally increased as the bus bandwidth was increased, and the amount of the increase was again application dependent. These results agree with other studies of nonblocking caches, for example [FJ94], which has a detailed study of the performance benefits obtained with varying numbers of miss information/status holding registers.

The most interesting result of this study was the extent to which the ability to reorder loads and stores impacted the effectiveness of the nonblocking cache. Even a hit under miss cache benefited from reordering. In this case the reordering could be implemented using a single cache line busy queue, which seems to be a worthwhile addition when this type of cache is used.

In study 2, the scheduler was shown to be generally ineffective in hiding memory latency. Although the memory latency was reduced by the scheduler, the excess CPI due to other sources was increased. In most cases, the net effect was to reduce the performance of the benchmark, although there was one case where the performance was significantly improved. This result emphasizes the point that miss rates are not uniform when considered on an instruction by instruction basis. The scheduler needs to know which loads are likely to miss and which are likely to hit in the cache in order to do a good job of scheduling for memory latency.

A logical follow on to this study would be to use either profiling information or static analysis to estimate the miss rate of individual loads. Then the compiler could use this estimate to derive the probabilistic latencies for the scheduling algorithm described in section 5. For example, static analysis could be based on stride estimation in loops, or on whether loads are or aren't stack relative. This approach is currently under investigation.

Our current plans for the simulation environment are to add support for superscalar and VLIW architectures, dynamic scheduling, and speculative executions. This will allow us to study a wider range of latency tolerant architectures. In addition, the compiler component needs to be enhanced to perform loop analysis and code motion. All of the studies presented here were limited to rescheduling code within basic blocks, whereas other studies indicate that moving code between basic blocks or loop iterations is necessary in order to hide large amounts of memory latency [MLG92].

References

[ASU88] A. Aho, R. Sethi, and J. Ullman. *Compilers.* Addison-Wesley Publishing, Reading, MA, 1988.

[ASW+93] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proc. of the 26th International Symposium on Microarchitecture*, pages 139–152, December 1993.

[CB92] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *SIGPLAN Notices*, pages 51–61, September 1992.

[Fis81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–90, July 1981.

[FJ94] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proc. of the 21st International Symposium on Computer Architecture*, pages 211–22, April 1994.

[GGH92] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *19th International Symposium on Computer Architecture*, pages 22–33, May 1992.

[GHG+91] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *18th International Symposium on Computer Architecture*, pages 254–63, May 1991.

[GM86] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proc. of SIGPLAN Symposium on Compiler Construction*, July 1986.

[Hoe71] Paul G. Hoel. *Introduction to Mathematical Statistics.* John Wiley & Sons, Inc., New York, NY, 1971.

[Kan88] Gerry Kane. *MIPS RISC Architecture.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

[KE93] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 278–89, June 1993.

[Kro81] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. Eighth Symposium on Computer Architecture*, pages 81–87, May 1981.

[MLG92] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *SIGPLAN Notices*, pages 62–73, September 1992.