Software Support for Speculative Loads

Anne Rogers
Kai Li
Department of Computer Science
Princeton University
Princeton, New Jersey

Abstract

This paper describes a simple hardware mechanism and related compiler support for software-controlled *speculative loads.* The compiler issues speculative load instructions based on anticipated data references and the ability of the memory system to hide memory latency in high-performance processors. The architectural support for such a mechanism is simple and minimal, yet handles faults gracefully. We have simulated the speculative load mechanism based on a MIPS processor and a detailed memory system. The results of scientific kernel loops indicate that our speculative load technique is an effective approaches to hiding memory latency.

1 Introduction

The performance gap between processors and memory has widened in the last few years. In the last decade, microprocessor speeds have increased at a rate of 50% to 100% each year whereas DRAM speeds have increased at a rate of 10% or less each year [13]. As the performance gap becomes wider, high-performance processors become more sensitive to stalls for cache misses. This is particularly true for scientific programs that do not exhibit a high degree of locality in data memory references. Hiding memory latency is an attractive way to reduce the effect of the performance gap.

This paper describes a non-blocking memory load mechanism that handles faults gracefully. The architectural support for such a mechanism is simple. It uses two extra bits per data register to implement interlocking and fault handling. A *presence* bit is used to implement interlocking and a *poison* bit allows a fault to be marked without signaling an exception. This architectural support greatly simplifies the use of non-blocking load instructions. We call such instructions *speculative loads.*

We also present several simple compiler techniques for exploiting speculative loads. These techniques include lifting speculative loads within a basic block, above conditional branches, and across loop iterations.

To evaluate the effectiveness of the speculative load mechanism and its accompanying compiler support, we have integrated it into a cycle-level MIPS processor simulator with a detailed memory system similar to that of the DEC 5000/200 memory system. Inspired by Klaiber and Levy's approach[18], we report the average memory reference cost in processor cycles for several scientific kernel loops (Livermore loops). Our results show that the speculative loads can reduce the average memory latency significantly. For loops that are not memory bandwidth bound, speculative loads can often hide memory latency completely.

2 Related Work

There is a large body of related work including prefetching cache lines, non-blocking loads, blocking, scheduling techniques, and speculative executions.

Prefetching reduces cache misses by fetching data into the data cache or a prefetch buffer before load instructions are executed. Prefetching can be controlled by hardware or software. Hardware approaches are often effective only for programs with sequential accesses in ascending/descending order [17, 24]. Baer and Chen propose a preloading mechanism based on a reference prediction table and a lookahead program counter[3].

Software-controlled prefetching techniques do not have such limitations [6, 18] and work quite well with scientific loops that do not have simple access patterns. A disadvantage of software-controlled schemes is that prefetching instructions consume cycles in the loop. Another disadvantage is that supporting safe prefetching and regular cache misses complicates the hardware design and implementation.

The idea of non-blocking loads has existed for many years. Such loads can be implemented with dynamic scheduling techniques, such as, scoreboarding, Tomasulo's algorithm, or by using a simple presence-bit method. An early example of scoreboarding is the CDC-6600 which has 8 address/data register pairs. When a new address is stored into an address register, a load is initiated and the data is loaded into the paired data register several cycles later. The Motorola 88100 [15] uses a scoreboard to implement load instructions in a non-blocking fashion. The IBM RS6000 [14] also implements non-blocking loads. The Intel i860 [16] supports the pipelined floating-point load instruction `fpld` that bypasses the cache and pipelines loads into the destination register. There are three pipeline stages for such loads and the pipeline must be advanced explicitly by issuing a `fpld` instruction. The main disadvantage of these approaches is that compilers cannot issue these non-blocking load instructions speculatively above conditional branches because they may cause unnecessary faults.

Blocking is a software technique that reduces the working set size of a program to use memory hierarchy effectively [1, 6, 10, 20]. Blocking can be applied to any levels of memory hierarchy, including virtual memory, caches, and registers. In particular, blocking can be very effective for scientific programs, but automatic blocking transformation is quite limited and manual transformation is difficult.

Much work has been done on code scheduling algorithms and heuristics for pipelined architectures [5, 11, 12, 27]. These methods attack the problem of scheduling expressions represented by directed acyclic graphs (DAGs) for architectures with pipeline constraints, rather than memory latency constraints. The algorithm proposed by Proebsting and Fischer [22] addresses the scheduling of instructions for delayed-load architectures with a fixed number of delay slots. Their algorithm does not handle cases beyond a basic block and does not apply to speculative loads.

Most static, global scheduling techniques have been limited by the unsafeness of executing instructions speculatively. The trace-scheduling technique proposed by Fisher [9] and its implementations [7, 8] allow compilers to pick a trace across basic blocks and schedule instructions for Very Long Instruction Word machines. The VLIW Trace included a dismissable load instruction which turned off exceptions. Percolation scheduling and software pipelining techniques [2, 4, 19] schedule instructions statically. They either use simple rules to percolate instructions across conditional branches and iteration boundaries or use heuristics to shorten the iteration time interval with resource constraints.

Speculative execution has not been studied extensively. Most work was based on quite complicated hardware support for superscalar or VLIW machines. Cydra has hardware support to annul the instructions on the not-taken branch path [23]. Torch uses a hardware shadow structure to hold the side effects of boosted instructions in order to commit or squash the side effects of executed speculative instructions [25]. These methods only apply to VLIW or superscaler machines whereas our method works for simple RISC processors.

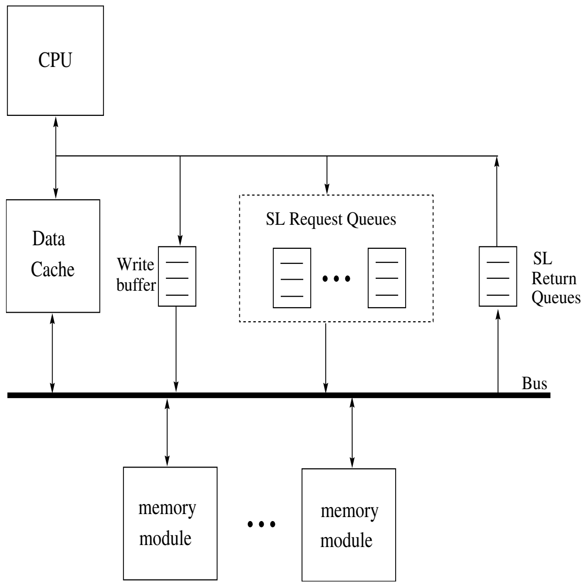

Figure 1: Speculative Load Mechanism

3 Speculative Load Mechanism

A load instruction is called a *speculative load* if it is non-blocking and can be issued speculatively without introducing unnecessary faults into the program. Figure 1 shows a block diagram of a simple mechanism for speculative load instructions, which we describe in this section. In the next section, we describe several algorithms for scheduling speculative loads aggressively.

Speculative loads load memory values into registers directly, bypassing the cache, without blocking the pipeline. This allows other instructions to be executed while the load is being serviced. To prevent an instruction from reading a register before a pending speculative load has completed requires an interlock mechanism. This interlock mechanism can be implemented with a simple presence bit for each register, like the presence bits used to prevent read-after-write and write-after-write hazards in pipelined architectures. Initially, all presence bits are set and every non-speculative load instruction that writes to a register sets the register's presence bit. A speculative load instruction clears the presence bit of its destination register when the instruction is executed. The bit is reset when the speculative load completes.

The non-faulting character of speculative load instructions allows compilers and programmers to issue

State	*Bits*	*Read Action*	*Write Action*
pending	0 0	stall until present	stall until present
present	1 0	proceed normally	proceed normally
poison	0 1	force a load into the pipeline or trap	proceed normally
unused	1 1		

Table 1: Read/Write Actions

instructions aggressively without introducing unnecessary faults. Traditional non-blocking load instructions do not handle faults gracefully making them difficult to use. A non-blocking load cannot be lifted above a conditional branch because it may access an illegal memory page and if the branch is not taken the exception should not be signaled. If speculative loads are to be lifted above conditional branches, unnecessary page faults must be avoided. In prefetching schemes, a prefetch that would cause a page fault can be dropped on the floor because a load instruction is guaranteed to appear if the value is actually required. This technique does not apply to speculative loads. Our solution is to use a bit for each register, called the *poison* bit, to handle faults gracefully. Initially, all poison bits are cleared. If a speculative load of address `A` would cause a fault, the poison bit is set and the address `A` is stored in the register.

The presence bit and poison bit together encode up to four possible states, but we only need three of them: *present, pending, and poison.*

The actions required to satisfy references to speculatively loaded registers depend on the whether the reference is a read or a write and the state of the register. A summary of the required actions can be found in Table 1. A read or write to a register in the *present* state can proceed normally because the necessary value is available and no other update to the register can be pending. When the register state is *pending*, both reads and writes must stall until the state changes to *present*. A cleared presence bit indicates that a speculative load that is guaranteed to complete is pending. A reader must wait for the desired value to become available. A writer must wait because the newly written value would be overwritten by the completing speculative load. The actions required when the state is *poison* depend on the type of the reference. A write may proceed normally because, as in the *present* case, no other updates may be pending; the speculative load that caused the poison bit to be set will have been dropped. A read of a poisoned register can be handled in two ways. First, a read of a poisoned register could force a load of the desired address into the pipeline. The page fault caused by such a load will be handled by a software fault handler which will clear the poison bit. Note that this load will not necessarily cause a page fault since the offending page may have been retrieved in the interim. In this case, the load will clear the poison bit. A second alternative is to trap on reading from a poisoned register. The trap handler will complete any necessary loads and restart the instruction.

Since speculative loads are non-blocking, they should interfere as little as possible with regular, blocking loads and stores. Therefore, load misses and write-through traffic should have priority over speculative loads. To accomplish this, we add a set of queues to the processor, one for each bank of an interleaved memory. Bus requests from the cache to memory and from memory to the processor have priority over speculative load requests. However, once a speculative load request has been sent to a memory module it is allowed to complete even if a regular load is stalled a few extra cycles as a result. Such a design has simple hardware and does not affect the cost of servicing a cache miss unduly.

From the processor's point of view, the mechanics of speculative loads and subsequent references are straightforward. To issue a speculative load, the processor does a virtual to physical address translation and checks the page tables to determine if such a reference would cause a page fault. In parallel, the processor checks to determine if the required data resides in the cache. The action taken depends on the results of these checks.

- If the speculative load would cause a fault or if the appropriate queue is full, the processor simply sets the poison bit and stores the requested address in the destination register.
- If no fault would occur and the value is in the cache, the speculative load can be serviced like a regular load.
- If no fault would occur and the value is not in the cache, the physical address plus the name of the destination register are inserted into the appropriate queue. Eventually, this request will reach the head of the queue and will be granted access to the bus. The request will be marked as pending and will be issued. Once the memory side is completed, the requested value is returned. The returned value and the name of the destination register are inserted into a FIFO return queue.

There are several design choices for writing the returned value into the register file: a dual write-port register file, a pipeline driven by extra speculative loads, or making use of unused write-back stage cycles. A dual write-port register file allows the processor to retire a request every cycle, but requires more chip real estate. Using extra speculative loads to drive the pipeline retires a request on every speculative load instruction.

The idea is similar to the pipelined floating point load instruction in the i860 instruction set. Such a mechanism yields a simple pipeline design. Using unused write-back stage cycles is also simple. If the write-back stage is not in use by an instruction, the processor can retire a request from the return queue, writing the value into the appropriate register. In all cases, the processor should probably provide a *pipeline flush* instruction to flush the pending requests in order to make the speculative load instruction easy to use. We use the third alternative in the simulations discussed in the Results section.

One question remains: how are context switches handled in the presence of our speculative load mechanism? Three alternatives are possible. First, all pending speculative loads could be dropped by clearing the queues and setting the poison bits on any register waiting for a pending speculative load. Second, the processor could stall until all pending speculative loads are completed. And third, the processor could stall until all speculative loads issued to memory have completed while dropping any unissued requests by clearing the request queues and setting the corresponding poison bits. We feel that the third alternative, which represents a middle ground between the other two, is the most reasonable approach; it is simple and yet easy for an operating system to handle. Note that with all three schemes, only the poison bits need to be saved. Since no speculative loads are left pending, the value of the presence bits can be inferred.

4 Lifting Speculative Loads

This section describes several compiler techniques for using speculative loads effectively.

Lifting within a basic block

The most intuitive way to think about the problem of issuing speculative loads is to start with a basic block of code using regular load instructions and to replace the regular loads with speculative loads. The interlocking mechanism of speculative loads guarantees the correctness of such substitutions. These speculative loads are then lifted upwards to hide their memory latency. The correctness condition for lifting a speculative load is that data dependences must be preserved.

Our goal is to lift a speculative load far enough from the first instruction that uses its value to hide the latency of obtaining the value while maintaining the constraint that data dependences are preserved and not extending the lifetime of the value more than necessary. To do this we must identify two instructions, I_f and I_b, for each speculative load, I_s. Called the *first use* instruction, I_f, is the first instruction following I_s that uses its value. I_b, called the *blocking* instruction, is the first instruction before I_s that has either a write-after-read, a read-after-write, or a write-after-write dependence with I_s, that is, the data dependences would not be preserved if we lift I_s above I_b and furthermore, I_b is the nearest such instruction to I_s.

We need a notion of the distance between two instructions in the following algorithm. Let $Dist(I_1, I_2)$ be the distance in cycles from instruction I_1 to instruction I_2. We use a simple counting method that does not model overlap between floating point operations, assumes that loads and stores take one cycle, assumes that intervening speculative loads take zero cycles, and assumes that all integer operations can be overlapped with floating point operations. Suppose that the basic block has n instructions $I_1 I_2 \cdots I_n$. The algorithm works as follows:

1. For each speculative load, I_s, compute I_f and I_b. I_f can be computed in a simple bottom up pass through the basic block. I_b can be computed in simple top down pass.

2. Calculate for each speculative load, I_s, the distance, $D_{bf} = Dist(I_b, I_f)$, from I_b to I_f. Let $D' = min(D_{bf}, l)$, where l is the predicted latency of a speculative load.

3. For each speculative load, determine k such that $b < k \leq s$, $Dist(I_k, I_f) \geq D'$, and $Dist(I_{k+1}, I_f) < D'$. Lift instruction I_s to immediately before I_k. We prefer a top down pass for this because it maintains the original order for speculative loads that are lifted in front of the same instruction.

Intuitively, this algorithm preserves data dependences because it never lifts a speculative load above a blocking instruction. Write-after-read register conflicts may limit the distance a speculative load can be moved unnecessarily. To solve this problem, registers can be renamed in step 2. In practice, the combination of this method of lifting references within a basic block and register renaming works quite well.

This algorithm does not take into account memory bank conflicts. Some bank conflicts are easy to handle. For example, on the MIPS R2000/R3000, double floating point loads are converted by the assembler into two single floating point load instructions. The second of these two instructions will be delayed until the first one has completed because loads will reference the same bank. This kind of conflict is easy to handle with our algorithm. The pair of loads can be treated as a single load with a longer predicted latency. Other cases are more complicated, if the distance between two speculative loads, $s1$ and $s2$, is less than the latency of the first one ($s1$), then the effective latency of the second speculative load ($s2$) will increase if $s1$ and $s2$ reference the same memory bank. One solution to this problem is to lift either the first speculative load by $l - Dist(s1, s2)$ or to lift the second by $l + Dist(s1, s2)$ if possible. This may in turn introduce conflicts with other speculative loads.

Notice that we use the minimum of the latency and the distance between the blocking and first use instructions. We choose this method over lifting all the way to I_b to reduce register pressure and to reduce the likelyhood that a speculative load will leap-frog another speculative load that uses the same memory bank.

Lifting above a conditional branch

Previous research shows that the average size of a basic block is less than 10 instructions in most programs[26]. Lifting speculative loads above conditional branches can hide much longer memory latencies than using the loads within basic blocks.

For a conditional branch connecting two basic blocks, we lift speculative loads from the basic block that is predicted to be executed most often. For example, if the `condition` in the following code is true most of the time, then we lift the load from the corresponding basic block above the conditional branch.

```
if ( condition ) {
   ld Rx, A;
   f1( Rx, ... );
}
else {
    ld Rx, B;
    f2( Rx, ... );
}
```

The resulting code is:

```
sld Rx, A
if ( condition ) {
   f1( Rx, ... );
}
else {
   ld Rx, B;
   f2( Rx, ... );
}
```

The non-faulting feature of our simple speculative load mechanism allows compilers to lift such loads without causing unnecessary faults. Using regular non-blocking loads, a load must be shown to be *safe*, that is, that it will not cause a fault, before it can be lifted above the branch.

Note that if the condition turns out to be false, the load of `B` into register `Rx` may have to stall if the speculative load of `A` into `Rx` is not completed.

Lifting Across Loop Iterations

This method is for small inner loops with very few instructions. As a result, there may not be enough cycles within an iteration to hide the memory latency of speculative loads.

Our idea is to let each iteration speculatively load data for an iteration far enough from the current one to hide the latency. For K consecutive iterations, each iteration has its own private registers for its loads and related computation. These registers are speculatively loaded by the previous (K-1)-st iteration. As with scheduling speculative loads within a basic block, we must take data dependences into account. One condition prevents a load from being converted into a speculative load in a loop that has been unrolled K times: the reference has a loop-carried dependence of distance K or less. Such a reference can be converted into a speculative load using the basic block technique described above or it can be converted into a move instruction, assuming enough registers are available.

For example, consider a small loop, which uses m registers:

```
for ( i = 0; i < n; i++ ) {
    ld R1, A1;
       ⋮
    ld Rm, Am;
    f( i, R1, ... , Rm );
}
```

If two iterations are adequate for hiding the memory latency for speculative loads, then we can use two private register sets for two consecutive iterations:

```
sld R01, A1;
   ⋮
sld R0m, Am;
for ( i = 0; i < n; i++ ) {
   sld R11, A1;
     ⋮
   sld R1m, Am;
   f( i, R01, ... , R0m );
   if ( ++i >= n ) break;
   sld R01, A1;
     ⋮
   sld R0m, Am;
   f( i, R11, ... , R1m );
}
```

Such a transformation is different from the traditional loop unrolling where the purpose is to enlarge the basic block size of a loop body. The above transformation allows the compiler to issue speculative loads one iteration early and does not change the basic block size at all.

It is always a good idea to perform speculative loads for the next iteration at the beginning of the current iteration in order to maximize memory latency hiding. Like lifting speculative loads above conditional branches, loading registers for the n+1st iteration does not introduce unnecessary exceptions.

This method can be generalized to deal with medium size loops, which are small in the sense that the loop body cannot hide the memory latency of all speculative loads, but are also large in the sense that there are not enough registers to use the previous method.

The idea is to lift some speculative loads within the same iteration and some across iterations. Suppose the original loop is:

```
for ( i = 0; i < n; i++ ) {
   ld R1, A;
   f1( i, R1 );
   ld R2, B;
   f2( i, R1, R2 );
}
```

Suppose it is not possible to hide the memory latency of loading value of `A` into `R1`. The loop would be transformed the loop into the following form:

```
sld Rx A;
for ( i = 0; i < n; i++ ) {
   sld Ry, A;
   sld R2, B;
   f1( i, Rx );
   f2( i, Rx, R2 );
   if ( ++i >= n ) break;
   sld Rx, A;
   sld R2, B;
   f1( i, Ry );
   f2( i, Ry, R2 );
}
```

In this case, `Rx` is loaded by the even iterations and used by the odd iterations whereas `Ry` is loaded by the odd iterations and used by the even iterations. Loads related to other registers are placed within their own iterations.

5 Simulation Framework

We have implemented a cycle-level simulator for our experiments. The simulator consists of a CPU simulator, a 16K data cache, and a memory system. The simulator can execute standard Unix executables or programs in MIPS assembly, and report memory related statistics.

The cycle-level CPU simulator is for the MIPS R2000/R2010 instruction set[1] augmented with a speculative load instruction. The interlock mechanism and non-faulting mechanism of speculative load instructions have been implemented in the simulator. It also implements both the R2000 and the R2010 pipelines. Bypassing is simulated for the R2000 pipeline but not for the R2010 pipeline. The speculative instruction mechanism is as described in Section 3 using speculative load request queues of depth 12 and a return queue of depth 16.

The simulator implements two memory systems: a baseline model and a speculative load model. The baseline model assumes a 16K direct-mapped physical data cache organized into 512 32-byte lines. It uses a write-through/no write allocate policy with a six-stage write buffer. Read misses are delayed due to a non-empty write-buffer only if the contents of the write-buffer conflict with the requested line. We assume single cycle access for read hits and for all writes unless the write buffer is full. This model assumes that there is a single memory controller and that memory is interleaved to provide one word per cycle once the cost to access the first word in a line has been paid.

The speculative load model (SL) is similar to the baseline model in many ways. The data cache is also a physical, direct mapped cache with a write through/no write allocate policy and a six-stage write buffer. To account for the extra real estate needed by the speculative load queues, the SL model reduces the size of the data cache to 8K organized as 512 16-byte lines.[2] This model also assumes single cycle access for read hits and for most writes. The SL memory model is n-way interleaved (4, 8, or 16) with banks that are interleaved at the cache line level. In addition, each bank is 2-way interleaved to provide one word per cycle once the latency required to access the first word in a line has passed.

Note that neither model includes a second level cache. And that the results reported in the Results section assume a warm-started cache.

The memory module uses 4 MBit DRAMS with page mode (8K byte sized pages), where the first access to a page has a 9 cycle latency and subsequent accesses to the page have a 2 cycle latency. Each memory module has two banks of interleaved memory.

For both models we assume a split transaction bus that multiplexes addresses and data. A simple arbitration scheme is used to manage access to the bus. Cache read requests have the highest priority, followed by write-buffer requests, memory to processor requests, and finally, speculative load requests.

Table 2 contains a detailed cost breakdown for various memory operations as provided by the simulator. We assume the following cycle costs: 1 cycle to arbitrate, 1 cycle to transmit an address, and 1 cycle to transmit a single word of data. Arbitration is not overlapped with other bus activities. Therefore, in the absence of bus contention, servicing a read miss takes 14 cycles (or 21 cycles depending on the DRAM page) in the baseline model and 10 cycles (17 cycles) in the speculative load model, the difference being the length of the cache lines (4 words). In both models, servicing a

[1] This simulator is an extension of Jim Larus's instruction level R2000 simulator, SPIM[21].

[2] Using 256 32-byte lines changes the results slightly. In some cases, the performance is better than the model with 512 16-byte lines and in others it is slightly worse.

	Baseline		SL	
	Read Miss	Write	Read Miss	Speculative Load Miss
Arbitrate	1	1	1	1
Addr/data	1	2	1	1
Latency	2/9	2/9	2/9	2/9
Arbitrate	1		1	1
Data	8		4	1
Fix Up	1		1	1
Total	14/21	5/12	10/17	7/14

Table 2: Transaction Costs in Cycles

write buffer request takes either 5 or 12 cycles depending on the DRAM page. The cost of servicing a speculative load depends on whether the data is in the cache or not. Speculative load hits take one cycle. Servicing a speculative load miss, once the request has reached the head of the queue, takes at least 7 cycles (14 cycles). Recall that a speculative load retrieves only one word of data from the memory module, and the requested value is stored directly in a register without updating the cache.

6 Experiments

We used a selection of the Livermore Loop kernels written in C as a basis for our experiments. No intervention was necessary for the baseline model, the C code was compiled using the MIPS C compiler (v 2 10) with the -O switch.

The SL model code took more work. Without a compiler capable of unrolling loops and inserting speculative loads, we were forced to intervene. The process was basically: unroll the inner loop in the C code by hand, use the MIPS C compiler (v 2 10, -O, -S) to generate MIPS assembly code, insert speculative loads in place of some regular load instructions, and assemble using the MIPS reorganizing assembler. Two comments. First, all the load instructions of interest load double floating point values. The MIPS assembler converts these into two single floating point loads. Second, since the experiments were being run on a simulator, an existing R2000 instruction (lwc2) could be used as the speculative load instruction. This allowed us to use the MIPS assembler to generate machine code. Judicious use of the `reorder` and `noreorder` directives prevented the assembler from lifting instructions above the speculative loads without preventing it from reorganizing the code within an unrolling of the loop body.

Inserting speculative loads increases the lifetime of the speculatively loaded values, thereby increasing register pressure. As a result, when inserting speculative loads, some reorganization of register usage was necessary. The reorganization was limited to reusing registers earlier than the -O option does naturally. No values were spilled to make way for speculatively loaded values.

7 Results

Following Klaiber and Levy[18], we report the average number of cycles per memory reference. This measure includes the cost of loads, stores, speculative loads, and any time spent waiting for a speculative load to complete. Unlike software prefetching schemes, we do not have prefetch instructions; speculative loads replace load instructions. Therefore, in the best case, the average cost per memory reference will be 1.0. Note that our baseline results differ from the results reported by Klaiber and Levy because our baseline model is different. Our model is write-through with no allocate on writes instead of write-back and we do not use a second level cache.

Loop Unrolling

Unrolling loops to lift speculative loads is the simplest of the techniques suggested in Section 4. In the sequel, we discuss two kernels, K1 and K6, for which unrolling the loop and lifting the speculative loads works very well. Later, we shall see examples where this technique is less effective than might be expected.

The first benchmark, Kernel K1, is a simple loop that walks through two arrays to compute the values for a third array (see Figure 2). The loop contains three loads and a store. The compiler can determine that one load (ZX[k+11]) is unnecessary. As you can see in Figure 3, for N up to 1000, the average cost per memory reference is 1.0 for the baseline model. The data fits in the cache and writes are never stalled. After that, the first reference to every line causes a miss. The amortized cost of a miss is fairly high because each word in a line is referenced only once. The average cost of servicing a miss depends on the number of DRAM page hits/misses and on how often a write-buffer request is in progress when a load miss occurs. Most load misses pay the full DRAM latency, which is not surprising since much of the processor-memory traffic arises from write-buffer requests. In addition, most load misses are stalled for several extra cycles waiting for the completion of a write-buffer request.

Speculative loads work very well for this example. Both remaining loads can be replaced with speculative loads because neither is involved in a troublesome data dependence. Unrolling the loop body once provides sufficient computation (approx. 37 cycles) to hide the latency of most of the speculative loads. Figure 4 contains

```
#define MAX 300000
double X[MAX+20];
double Y[MAX+20];
double ZX[MAX+20]
double END[100];
double Q,R,T;
for (k=1; k<n; k++) {
    X[k] = Q + Y[k] * (R*ZX[k+10] + T*ZX[k+11]);
}
```

Figure 2: Kernel K1

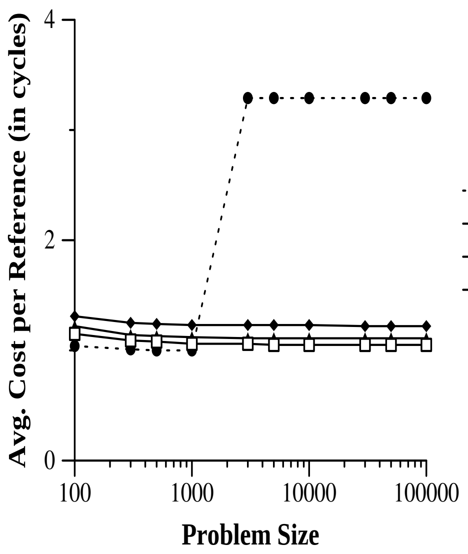

Figure 3: Kernel K1

a graph that plots latency in cycles against the percentage of speculative loads with that latency. Roughly 86% of the speculative loads take fewer than 38 cycles and roughly 98% take fewer than 39 cycles. The added cost of the 14% of speculative loads that cause the processor to stall can be amortized over all references. In addition, some of these are coincident, which further reduces their effect on the cost of a reference.

The average cost per reference (1.06 for N = 1000, SL-16) is close to ideal but it is not exactly 1.0. Speculative loads have a lower priority than write-buffer requests, which causes approximately a half-cycle delay per iteration. We can determine the effect of pending write-buffer requests by replacing the stores with loads that are guaranteed to take one cycle. This provides the right number of cycles and memory references while guaranteeing that there will be no traffic generated by the write-buffer. Roughly 99% of the speculative loads take fewer than 38 cycles when the writes are replaced by one cycle loads. As a result, the average number of

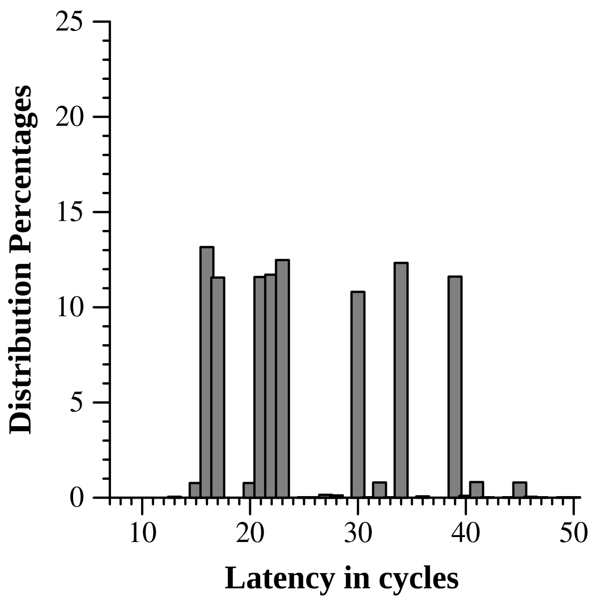

Figure 4: Kernel K1 Distribution Percentages, N = 1000, SL-16

```
#define MAX 1024
double W[MAX+4];
double B[MAX+4][MAX+4];
for (i=2; i<=n; ++i)
   for (k=1; k<=i-1; ++k)
      W[i] = W[i] + B[i][k] * W[i-k];
```

Figure 5: Kernel K6

cycles per reference drops very close to 1.0.

The effect of reducing the number of memory banks from 16 to 4 or 8 is small. For this benchmark the difference can be attributed to the increased likelyhood that a memory module will be busy servicing a write-buffer request. The average depth of the speculative load request queues, which increases as the number of bank conflicts increase, does not change substantially.

The second benchmark, Kernel K6, is a simple general linear recurrence (see Figure 5). The code contains three loads (W[i], W[i-k], B[i][k]), and a store. For all simulated values of N, the data sets are too large for the cache because each B[i][k] reference accesses a distinct element. Even though the working set is large the average number of cycles per reference is low (under 2.0). Within an iteration of the outer loop each reference to W[i] refers to the same array element. These repeated references help inflate the reference counts to more than 20 accesses per line.

For this benchmark, we unrolled the loop once and introduced speculative loads for the references to B[i][k]. The reference W[i] has a loop carried dependence with

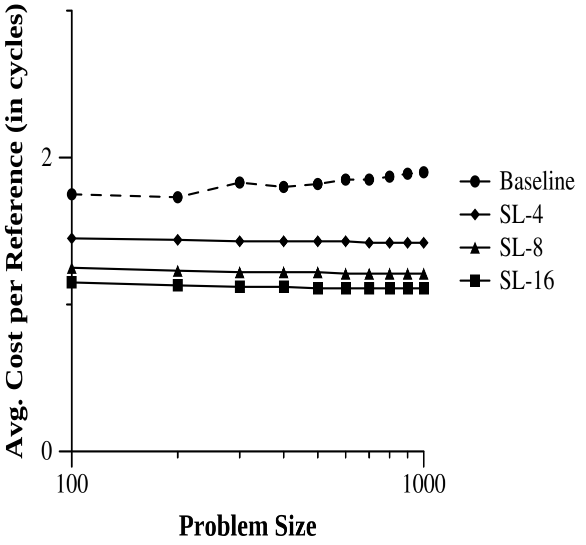

Figure 6: Kernel K6

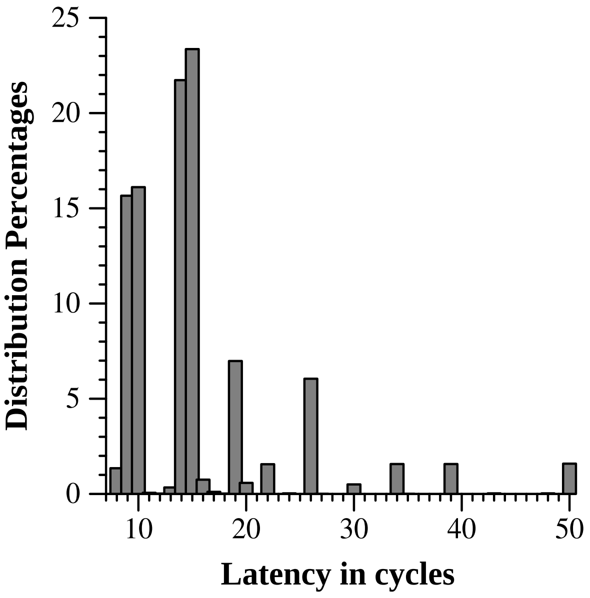

Figure 7: Kernel K6, Distribution Graph, N = 100, SL-16

the store. Since the reference to W[i] must be loaded through the cache, it seems pointless to speculatively load the references to W[i-k]. The loop body consumes at least 14 cycles between when the speculative loads are issued and when the first reference can occur. For this example, the speculative loads have an average latency of 15.12 cycles and more than 78% are completed within 15 cycles. For N = 100, the average cost per reference is 1.15. The impact of the write-buffer on the

```
#define MAX 100000
double PX[15][MAX+1];
double DM28, DM27, DM26, DM25, DM24, DM23, DM22,
    C0;
for (i=1; i<=n; ++i)
    PX[1][i]  =  DM28*PX[13][i] + DM27*PX[12][i]
               + DM26*PX[11][i] + DM25*PX[10][i]
               + DM24*PX[ 9][i] + DM23*PX[ 8][i]
               + DM22*PX[ 7][i]
               + C0*(PX[ 5][i] + PX[6][i])
               + PX[3][i];
```

Figure 8: Kernel K9

latency of a speculative load is substantial. Replacing the stores with one-cycle loads reduces the latency of a speculative load to 11.96 and the average cost per memory reference to 1.04. The effect of reducing the bank-level interleaving to 4 or 8 has a more pronounced effect for this benchmark. This effect is attributable to write-buffer interaction.

In summary, both kernels K1 and K6 can use the simple unrolling technique to insert speculative loads and thereby improve the average cost of a memory reference.

Basic Block Lifting

The two kernels presented thus far have fairly modest register needs. The next two benchmarks, K9 and K23, use enough registers that loop unrolling is impractical. But, both perform well when the basic block lifting technique is used to introduce and schedule speculative loads.

Kernel K9 integrates predictors (see Figure 8). The loop uses all available registers, which will be an issue when we try to introduce speculative loads. The body of the loop contains 10 loads and a store. The constants (DM28, DM27, ...) are loaded outside the loop. From the graph in Figure 9 we can see that for N less than 100 the data fits in the cache. For N = 100, the data should fit in the cache, yielding an average number of cycles per reference of 1.0, but there are conflict misses which increase the cost. Above N = 300, the data no longer fits in the cache. Each line is reference 8.8 times on average which yields an average number of cycles per reference of 3.38.

As mentioned earlier, Kernel K9 uses many registers; the constants alone use half the available registers. Another two registers are required to hold temporary values, which leaves six registers for speculatively loaded values. These registers were used for the remaining ten loads, which were converted into speculative loads and lifted the appropriate distance using the basic block algorithm. Three speculative loads (PX[13][i],

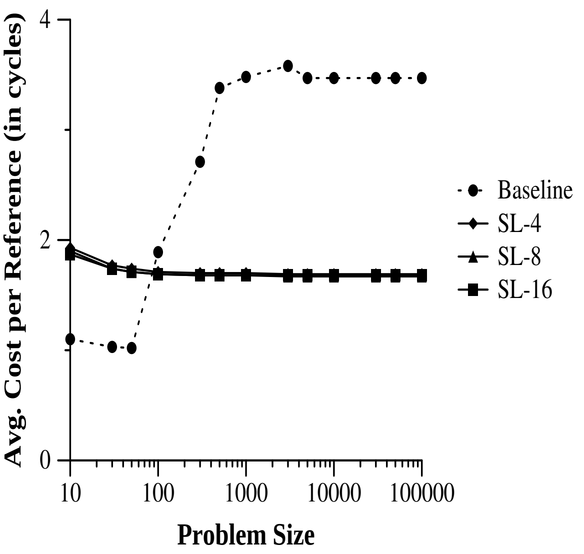

Figure 9: Kernel K9

```
#define MAX 50002
double ZA[MAX][8];
double ZB[MAX][7];
double ZR[MAX][7];
double ZU[MAX][7];
double ZV[MAX][7];
double ZZ[MAX][7];
fw= 0.17500E0;
for (j=2; j<=6; ++j) {
   for (k=2; k<=n; ++k) {
        QA = ZA[k][j+1]*ZR[k][j]
           + ZA[k][j-1]*ZB[k][j]
             + ZA[k+1][j]*ZU[k][j]
           + ZA[k-1][j]*ZV[k][j]
           + ZZ[k][j];
        ZA[k][j]= ZA[k][j] + fw*(QA -ZA[k][j]);
   }
}
```

Figure 10: Kernel K23

PX[12][i], and PX[11][i]) were lifted above loop branch into both the preheader and the bottom of the basic block (as with any method that lifts instructions above branches, if a basic block has more than one incoming edge, the instruction must be duplicated and lifted across all edges). The resulting code performs very well, with an average cost per memory reference of 1.87 for N = 10. The effect of reducing the bank interleaving factor is negligible.

The second basic block lifting example is Kernel K23, a fragment of a 2-D implicit hydrodynamics code (see Figure 10). The loop body contains 10 loads and a store. Unlike K9, none of these values are loop invariant and as a result, the original loop does not use as many registers as K9. The graph in Figure 11 shows that the data set is too large for the cache for all benchmarked values of `n`. Even though there are repeated references to the array, KA, the average number of references per cache line retrieved is low (3.48 for N=100) because of the poor locality shown by the rest of the references and conflict misses. As a result, the average number of cycles per reference is a very high, 6.74 for N = 1000.

The number of values referenced in the K23 loop body makes it difficult to use the simple unrolling technique. However, the loop has enough computation (approx. 60 cycles) to hide the latency of the loads using the lifting within a basic block technique. We converted all but one load (ZA[k][j+1]) into speculative loads and lifted them the appropriate number of cycles. The one remaining load was not converted into a speculative load because of the repeated references to ZA. Loading lines that show an appreciable degree of locality of

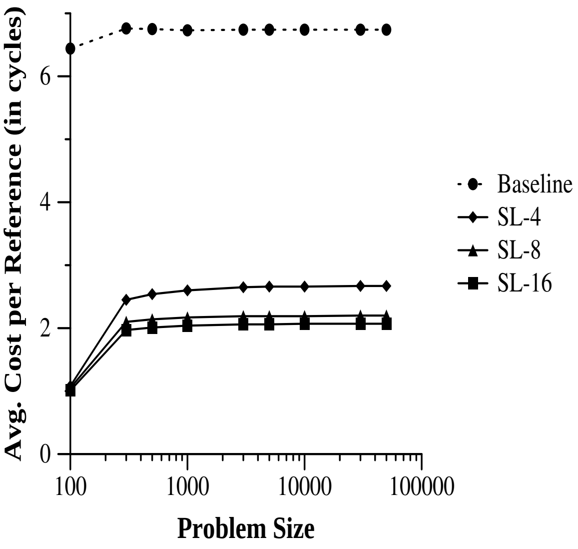

Figure 11: Kernel K23

reference through the cache should produce better performance. The other references to ZA could be left as ordinary loads or they could be converted into speculative loads to hide the latency of retrieving the data in case it is not the cache as expected. We choose this second option. The transformed code performs much better than the original code; the average number of cycles per reference for N = 1000 is 2.04 for SL-16. Reducing the the interleaving factor has a noticeable effect on this benchmark. Unlike previous examples, this effect is due to increased bank conflicts. The average speculative load queue depth jumps from 1.56 for

```
#define MAX 300000
double X[MAX+20];
double V[MAX+20];
II= n;
IPNTP= 0;
do {
   IPNT = IPNTP;
   IPNTP = IPNTP+II;
   II = II/2;
   i = IPNTP;
   for (k=IPNT+2; k<=IPNTP; k+=2) {
      i = i+1;
      X[i] = X[k] - V[k]*X[k-1] - V[k+1]*X[k+1];
   }
} while (II > 1);
```

Figure 12: Kernel K2

SL-16 to 2.91 for SL-4.

Unrolling Revisited

In this section we discuss two benchmarks for which the unrolling method does not work as well as in the earlier examples.

The first benchmark, Kernel K2, is an excerpt from an incomplete Cholesky-conjugate gradient method. The code (see Figure 12) has more loads than either K1 or K6, but not nearly as many as K9. Notice that the compiler cannot replace the load of V[k+1] by a move because k is incremented by 2 for each iteration. Examining the curve in Figure 13 shows that the data fits in the baseline cache for N less than 1000, yielding an average cost per reference of 1.0. This benchmark fairs better than Kernel K1 once the data set becomes too large for the cache. A line is referenced 12 times on average, which lowers the amortized cost of obtaining it. In this example, load misses account for more of the bus/memory traffic and therefore servicing write-buffer requests has a diminished effect.

Introducing speculative loads into this example yields a small performance improvement (10-12%). As in the previous unrolling examples, we unrolled the loop only once. The three references to array X were not replaced by speculative loads because of the apparent loop-carried dependence of X[k+1] on the store. Both references to array V were replaced, which accounts for the slight improvement in the speculative load curve. The load misses plus the computation provide enough time for the speculative loads to complete before the values are required. To be more precise, the loop body without any load misses consumes between 32 and 38 cycles and for N = 5000, 71% of the speculative loads finish within 32 cycles and 96.8% finish within 38 cycles (Figure 14 contains the distribution graph for N = 5000

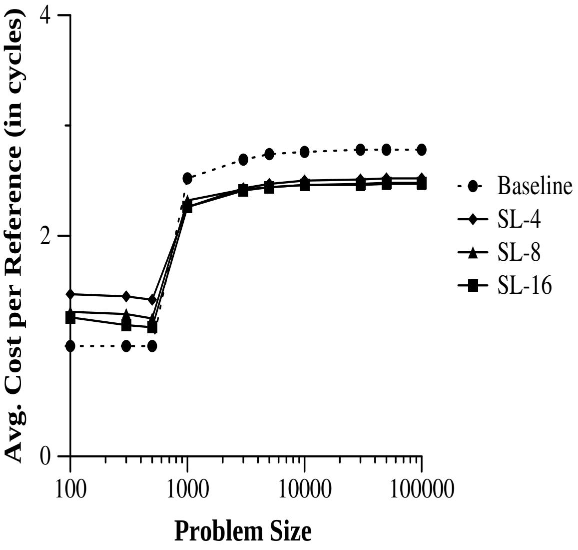

Figure 13: Kernel K2

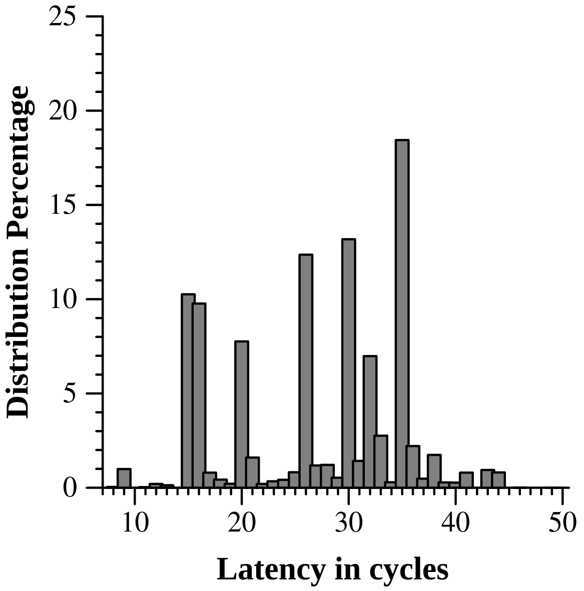

Figure 14: Kernel K2 Distribution Percentages, N = 5000, SL-16

for SL-16). The difference between the various SL models is noticeable only when the problem size is small.

The performance of this first benchmark does not improve dramatically because much of the data is still being loaded through the cache. Our second benchmark, Kernel K4, displays the effects of a lack of memory bandwidth.

Kernel K4 is a banded linear equation (see Figure 15). The inner loop contains a pair of loads (X[lw] and

```
#define MAX 300000
double X[MAX+20];
double Y[MAX+20];
double ZX[MAX+20];
double Q,R,T;
m = (n-7)/2;
fw = 1.000E-25;
for (k=7; k<=n; k+=m) {
   lw = k-6;
   temp = X[k-1];
   for (j=5; j<=n; j+=5) {
      temp = temp - X[lw]*Y[j];
      lw = lw+1;
   }
   X[k-1] = Y[5]*temp;
}
```

Figure 15: Kernel K4

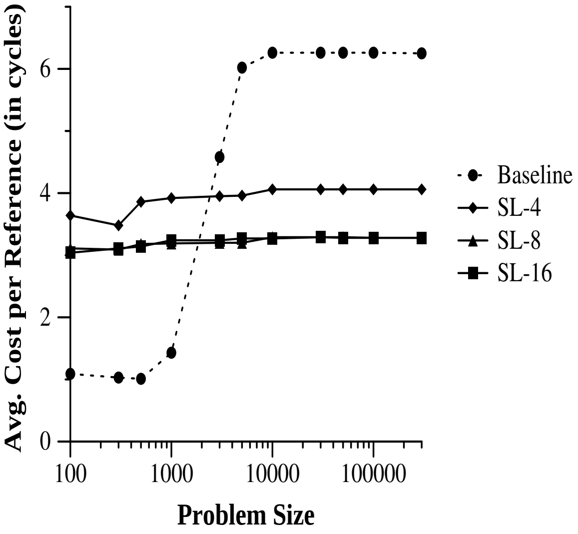

Figure 16: Kernel K4

Y[j]) and very little computation. The compiler stores the variable temp in a register, which removes the need for loading and storing it within the inner loop. As indicated by Figure 16, the data fits in the baseline cache for N less than 1000. In most of the kernels we have seen so far, the cache lines have had short lifetimes, once a line is brought into the cache the necessary references occur quickly. In contrast, Kernel K4, has both short and long lived cache lines. The lifetimes of lines associated with array Y are short; each line is referenced only twice with both references occurring in the same iteration. The lines associated with array X have long lifetimes; roughly the same set of data is referenced in each iteration of the outer loop. For N=1000 and N=3000, some of the long lived data remains in the cache, which helps reduce the average cost per reference. For N greater than 3000, the number of references to a line before it is kicked out of the cache drops precipitously, resulting in an average number of cycles per reference over 6.

Unrolling the loop once and replacing both loads with equivalent speculative loads has a dramatic effect on the performance of this benchmark. As should be expected, the baseline model performs better when the data fits in the cache. However, once the data is too big for the cache, the speculative load model performs significantly better. The best case performance of one cycle per reference is unattainable for this example because the loop body does not have enough computation to hide the latency of the four speculative loads. A loop iteration takes approximately 13 cycles. On average, a speculative load takes roughly 21 cycles to complete, which causes an 8 cycle delay.

It seems reasonable to believe that unrolling the loop a second time and inserting the speculative loads two iterations ahead rather than just one, should further improve the performance of this example. Unfortunately, unrolling a second or third time does not help. The problem is a lack of memory/bus bandwidth. The values referenced by the speculative loads for iteration three, for example, are likely to in the same memory bank as the values that are speculatively loaded for iteration two. As a result, the associated memory requests will sit in a queue until the earlier requests have been satisfied which has the effect of increasing their latency significantly.

In summary, unrolling yields performance gains, but the performance improvement is not always dramatic (K2, for example) and it is not always possible to achieve the ideal value for average number of cycles per reference due to memory/bus bandwidth limitations.

8 Limitations and Conclusions

A software-controlled speculative load requires very simple architectural support, yet it is a practical technique for hiding memory latency. Our cycle-level simulation shows that with a 16K direct-mapped data cache, speculative loads can hide memory latency effectively.

Our simulations demonstrate the performance advantage of speculative loads for a particular memory system, similar to the DEC 5000/200 memory system. It is not clear to what extent the speculative load mechanism could improve the performance of other memory systems such as a cache with a write-back policy or a memory system with a large second level cache. Also, the simulation framework only includes speculative loads for single words. In an implementation, one would need to consider speculative loads for byte, short words, and double words.

In this paper, we described several compiler techniques for lifting speculative load instructions within a basic block, across loop boundaries and across conditional branches. We have applied these techniques to several Livermore kernel loops, which shows that they are effective. A more extensive analysis based on larger benchmarks is needed.

The disadvantage of using speculative loads is more registers may be required than if ordinary loads were used. Informally speaking, speculative loads increase the lifetime of registers by the memory latency time.

There are two cases where speculative loads cannot hide memory latency completely. First, data dependences and register pressure may constrain the motion of speculative loads. And second, the memory bandwidth required by a program may exceed the actual memory system bandwidth. Combining the speculative load lifting techniques with global instruction scheduling and register allocation may help memory latency hiding for the first case. For the second case, one can implement a speculative load mechanism that puts data into the data cache. Such a mechanism is a combination of speculative load and cache prefetching. We are in the progress of investigating the impact of such a mechanism.

In the paper, TLB misses are treated as exceptions; the speculative load mechanism will poison the resulting register. The TLB misses will be initiated when read of a poisoned register is performed. This approach is very simple because it requires nothing from either the hardware or the software. The disadvantage is that if TLB misses are frequent, speculative loads may not hide memory latency effectively. Another approach is to always trap on a TLB miss and let the trap handler generate the speculative load. This approach is also simple and allows the speculative load instruction to complete. The disadvantage is that the TLB trap handler now needs to distinguish between speculative loads and regular loads. More experimental is necessary to study the tradeoffs between these two approaches.

Although the speculative load techniques described in this paper are used to hide memory latency for cache misses, they can also be used to hide latency for TLB misses on architectures with software implementation of TLB updates.[3] Latency hiding for TLB misses requires the compiler to have knowledge of which memory locations have been loaded and which memory locations will be loaded in order to issue TLB updates at compile time.

[3] This was suggested by a referee

Acknowledgement

Michael Nielsen and Neil Wilhelm helped us understand the DEC-5000/200 memory system. We had several helpful discussions with Mark Greenstreet. Karin Petersen and John Reppy read early drafts of this paper and made very helpful suggestions. Josh Fisher explained the Trace's dismissable load instruction. An anonymous referee suggested that trapping upon reading from a poisoned register would perform better than forcing a load into the pipeline. Kai Li was supported in part by NSF grant CCR-9020893, by DARPA and ONR under contract N00014-91-J-4039, and by Intel Supercomputer Systems Division. Anne Rogers was supported in part by NSF grant ASC-9110766.

References

[1] W. Abu-Sufah, D.J. Kuck, and D.H. Lawrie. Automatic program transformations for virtual memory computers. In *Proceedings of the 1979 National Computer Conference*, pages 969–974, June 1979.

[2] A. Aiken and A. Nicolau. Optimal loop parallelization. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 308–317, June 1988.

[3] J.L. Baer and T.F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91*, pages 176–186, November 1991.

[4] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[5] D.G. Bradlee, S.J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for riscs. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.

[6] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[7] P.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, and P.K. Rodman. A VLIW architecture for a trace scheduling compiler. In *The 2nd International conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, October 1987.

[8] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1985.

[9] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7), 1981.

[10] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, October 1988.

[11] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, June 1986.

[12] J. L. Hennessy and T. R. Gross. Postpass code optimization of pipeline constriants. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[13] J. L. Hennessy and N. P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, 24(9):18–29, September 1991.

[14] IBM. *IBM J. Research and Development, Special Issue on RISC System/6000*. IBM, January 1990.

[15] Motorola Inc. *MCS88100: RISC Microprocessor User's Manual*. Motorola Inc., 1988.

[16] $i860^{TM}$ microprocessor programmer's reference manual. Intel Corp., 1989.

[17] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 364–375, May 1990.

[18] A.C. Klaiber and H.M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual Symposium on Computer Architecture*, pages 43–53, May 1991.

[19] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[20] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[21] J. R. Larus. SPIM S20: A MIPS R2000 simulator. Technical Report 966, University of Wisconsin-Madison, September 1990.

[22] T. A. Proebsting and C. N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 256–267, June 1991.

[23] B.R. Rau, D.W.L. Yen, W. Yen, and R.A. Towle. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Computer*, 22(1), January 1989.

[24] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[25] M.D. Smith, M. Lam, and M.A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 344–354, May 1990.

[26] G.S. Tjaden and M. J. Flynn. Detection and parallel execution of parallel instructions. *IEEE Transactions on Computers*, 19(10):889–895, October 1970.

[27] H. S. Warren. Instruction scheduling for the IBM RISC system/6000 processor. *IBM Journal of Research and Development*, 34(1):85–92, 1990.