An Analysis of a Combined Hardware-software Mechanism for Speculative Loads

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

TR-455-94

April 5, 1994

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 three speculative load mechanisms based on a MIPS processor and a detailed memory system. The results of scientific kernel loops indicate that speculative load techniques can hide memory latency effectively.

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 about 50% each year whereas DRAM speeds have increased at a rate of 10% or less each year [16]. 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.

We propose to hide memory latency using a non-blocking load instruction that handles faults gracefully. The instruction is non-blocking to allow the processor to hide latency by continuing to execute instructions while waiting for the load to complete. To take advantage of this, we must move a load far enough away from the first instruction that uses its value to prevent the processor from having to stall to waiting for it to complete. In many cases, this means moving an instruction above a conditional branch, which requires a mechanism that handles, possibly unnecessary, faults gracefully.

The architectural support for such our proposed 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 examine three different approaches towards incorporating speculative loads into the memory system. All three methods store the value directly into a register. The mechanisms differ in how much data they retrieve when servicing a speculative load and how they interact with the data cache. The first method, called *Bypassing*, retrieves only the required four bytes and bypasses the cache completely.

The second method, called *Unified*, retrieves a whole cache line and stores it into the data cache. The final model, called *Spec-cache*, also retrieves a whole cache line, but stores it in a separate cache.

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[22], 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. And our results show that the Unified model is the best of the three proposed memory systems, because it outperforms the Bypassing model and is less complicated than the Spec-cache model.

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 [21, 32]. Baer and Chen propose a preloading mechanism based on a reference prediction table and a lookahead program counter[4]. Eickemeyer and Vassiliadis present a load address prediction technique using a *load delta table* (LDT) [9]. The address of a load instruction is used to index the LDT, which records the previous address referenced and stride for the load instruction. These values are then used to generate an address to prefetch. Various methods for updating the LDT are presented.

Software-controlled prefetching techniques do not have such limitations [7, 22] and work quite well with scientific loops that do not have simple access patterns. Mowry, Lam, and Gupta [27] present a compiler algorithm for inserting prefetch instructions in dense matrix code. Their compiler uses locality analysis to prefetch only those references that are likely to miss in the cache. Software pipelining is used to schedule the prefetch instructions. A disadvantage of software-controlled schemes is that prefetch 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 [34], Tomasulo's algorithm [36], 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 [18] uses a scoreboard to implement load instructions in a non-blocking fashion. The IBM RS/6000 [17] also implements non-blocking loads. The Intel i860 [19] 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 the memory hierarchy effectively [1, 7, 12, 24]. Blocking can be applied to any level of memory hierarchy, including virtual memory, caches, and registers. In particular, blocking can be very effective for scientific programs, but automatic blocking transformations are quite limited and manual transformation is difficult.

Much work has been done on code scheduling algorithms and heuristics for pipelined architectures [6,

13, 15, 37]. 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 [29] addresses the scheduling of instructions for delayed-load architectures with a fixed number of delay slots. Their algorithm does not handle cases beyond basic blocks 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 [11] and its implementations [8, 10] 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, 5, 23] 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 [30]. Many architectures, the SPARC [20] for example, have hardware support to annul the instruction in the delay slot following a branch. 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 [33]. Sentinel scheduling is used to support speculative execution, this technique leaves a sentinel instruction in the home block of every potential exception causing instruction that is speculatively executed [26]. The sentinel instruction reports any exceptions that were caused by the speculative instruction. These methods are more general than our mechanism, but they are also more complex.

This paper is an extension to our previous work [31]. Our earlier paper described a speculative load mechanism that bypasses the first level cache. In this paper, we propose two design alternatives that cache speculatively loaded values, either in the data cache (Unified) or in a separate cache (Spec-cache). We simulate and compare the three design alternatives.

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 our Bypassing mechanism for speculative load instructions. For simplicity, we explain our speculative load mechanisms in terms of the Bypassing model. The other two models, Unified and Spec-cache, differ from Bypassing only in how much data is accessed when servicing a speculative load and where that data is stored. We address these issues at the end of this section. In the next section, we describe several algorithms for scheduling speculative loads aggressively.

Speculative loads load memory values into registers directly without blocking the pipeline. This allows other instructions to execute 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 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 set when the speculative load completes.

The non-faulting character of speculative load instructions allows compilers and programmers to issue 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 make an illegal memory access that would not have happened, if the branch had not been taken. Thereby falsely signaling an exception. If speculative loads are to be lifted above conditional branches, unnecessary page faults must be avoided. In prefetching

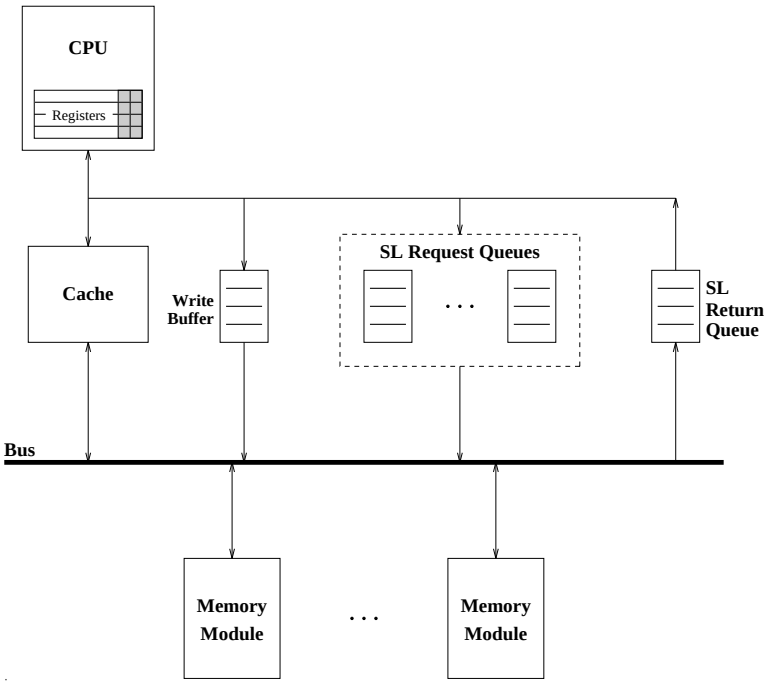


Figure 1: Speculative Load Mechanism

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: *pending, present, 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 on the state of the register. Table 1 summarizes the required actions.

Table 1: Read/Write Actions

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		

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, it could force a load of the desired address into the pipeline. If the force load caused a page fault, it would be serviced by a software fault handler, which would clear the poison bit and set the presence bit. The load might not cause a fault, because the offending page may have been retrieved in the interim. In this case, the load will clear the poison bit and set the presence 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 (see Figure 1). Bus requests from the cache to memory and from memory to the processor have priority over speculative load requests. But, 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 table to determine if such a reference would cause a page fault. In parallel, the processor determines 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.

This last action raises two questions. First, how much data should be retrieved when servicing a speculative load and where should it go? And second, what mechanism should be used to write the returned value into the register file?

As mentioned earlier, we examine three different approaches of how much data to retrieve and where to put it:

Bypassing Retrieve four bytes and bypass the cache completely.

Unified Retrieve a whole cache line and store it in the data cache.

Spec-cache Retrieve a whole cache line and store it into a dedicated cache, called the *spec-cache*.

Figure 2 illustrates the three cache models. The rationale behind the Bypassing approach is that data that does not have much locality will not benefit from being cached and would unnecessarily pollute the data cache. The Unified approach takes a slightly less extreme view: speculatively loaded data may not have enough locality to amortize the cost of servicing an initial cache miss effectively, but it may have enough locality to benefit from caching once the latency of the initial miss is hidden. The Spec-cache approach tries to achieve the best of both worlds by using a dedicated cache for speculatively loaded values. This avoids the problem of polluting the data cache, while maintaining the benefits of caching. Note that in all three models, the return value plus the destination address are added to the return queue.

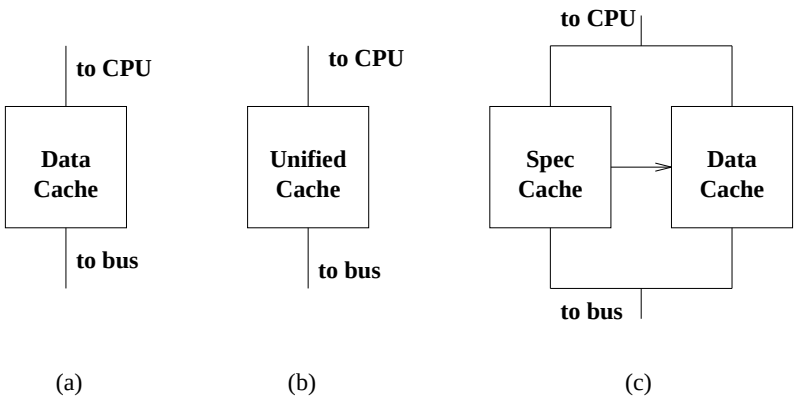


Figure 2: Cache Designs

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 our simulations.

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 bit 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 three compiler techniques for using speculative loads effectively. All three techniques need information concerning dependences between instructions. Two instructions have a data dependence between them, if one writes to a register or memory location, that the other reads or writes. Dependences that arises because of register conflicts are easy to identify at the assembly language level. Dependences that occur because of stores can be handled in two ways. The easiest is to assume that there is a dependence between every load and store. A better idea is to use dependence analysis[3, 28] in an earlier phase in the compiler to determine when there is a dependence between an array reference and an array update. Recently proposed intermediate representations [14] make this information available for use in later stages of the compiler.

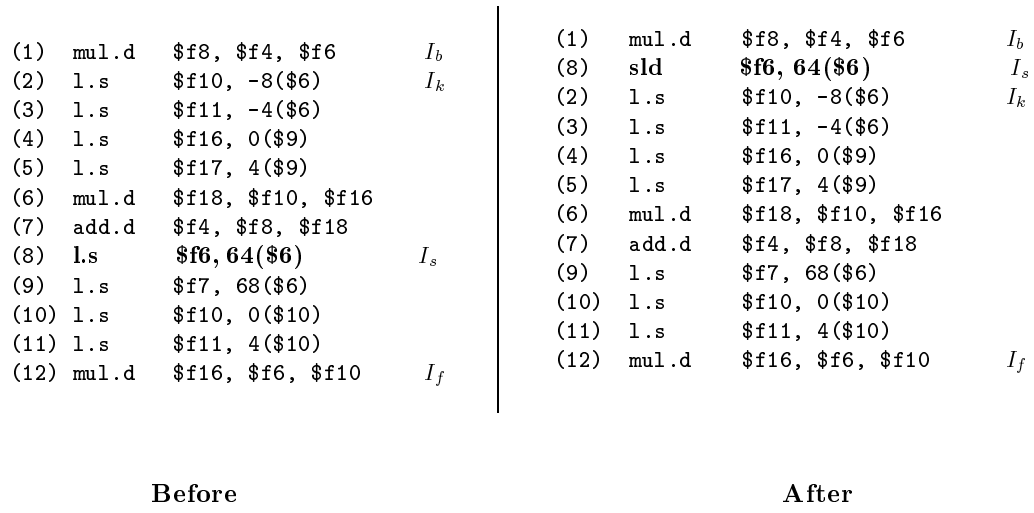


Figure 3: Basic Block Lifting Example

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.

Figure 3 contains a fragment of a basic block from the MIPS R2000/R3000 assembly code for K23, one of the benchmarks we discuss in Section 7. (On the R2000/R3000, floating point doubles are loaded into even-odd register pairs using two load single instructions (`l.s`).) Consider converting the load labelled I_s, in line 8 into a speculative load. The multiply instruction (labelled I_f) in line 12 is the first use of register `f6`, the destination register of the load, that follows I_s. The multiply instruction in line 1 (I_b) is the blocking instruction for I_s, because it uses register `f6` as an operand.

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 and assumes that loads and stores take one cycle, speculative loads take zero cycles, and all integer operations can be overlapped with floating point operations. In our example, the distance between I_b and I_f is 11 (assuming that `mul.d` takes 5 cycles and `add.d` takes 2). Suppose that the basic block has n instructions $I_1 I_2 \cdots I_n$.

The algorithm works as follows. For each speculative load, I_s:

1. Identify I_f and I_b. I_f can be identified in a simple bottom up pass through the basic block. I_b can be identified in simple top down pass.

2. Calculate 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. 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 between two speculative loads that are lifted above of the same instruction.

Returning to our example, if we assume a latency of 14 cycles for a speculative load, then $D' = 11$ and I_k is the load instruction in line 2. See the box on the right in Figure 3 for the rearranged code. Note that the load has been converted into a speculative load (`sld`).

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, as we noted earlier, double floating point loads are converted by the MIPS R2000/R3000 assembler into two single floating point load instructions. The second of these two instructions will be delayed until the first one has completed because the loads will reference the same bank. This type 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 an addition $l - Dist(s1, s2)$ cycles 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 chose this method over lifting all the way to I_b to reduce register pressure and to reduce the likelyhood that one 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[35]. Lifting speculative loads above conditional branches can hide much longer memory latencies than just 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 left half of Figure 4 is true most of the time, then we lift the load from the corresponding basic block above the conditional branch, as in the right half of the figure.

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` will have to stall, if the speculative load of `A` into `Rx` has not completed.

Lifting Across Loop Iterations

This method is for small inner loops with very few instructions, which may not be sufficient to hide the memory latency of speculative loads.

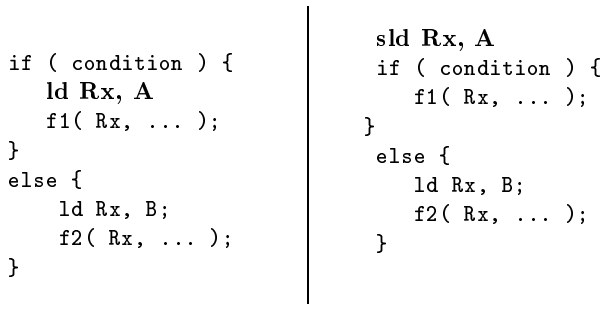


Figure 4: Conditional Branch Example

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 a set of 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 may be possible to convert it into a move instruction, assuming enough registers are available.

For example, consider the small loop in the left half of Figure 5, which uses m registers. 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 (see the right half of Figure 5).

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.

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 the code in the left half of Figure 6. Suppose it is not possible to hide the memory latency of loading value of `A` into `R1`. The loop would be transformed into the loop in the right half of the figure. 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, called *CL-SPIM*, for our experiments. CL-SPIM consists of a CPU simulator, a data cache, an instruction cache, and a memory system.[1] CL-SPIM simulates the MIPS R2000/R2010 instruction set augmented with a speculative load instruction. It implements both the R2000 and the R2010 pipelines. Bypassing is simulated for the R2000 pipeline but not for the R2010

[1] CL-SPIM is an extension of Jim Larus's instruction level R2000 simulator, SPIM[25].

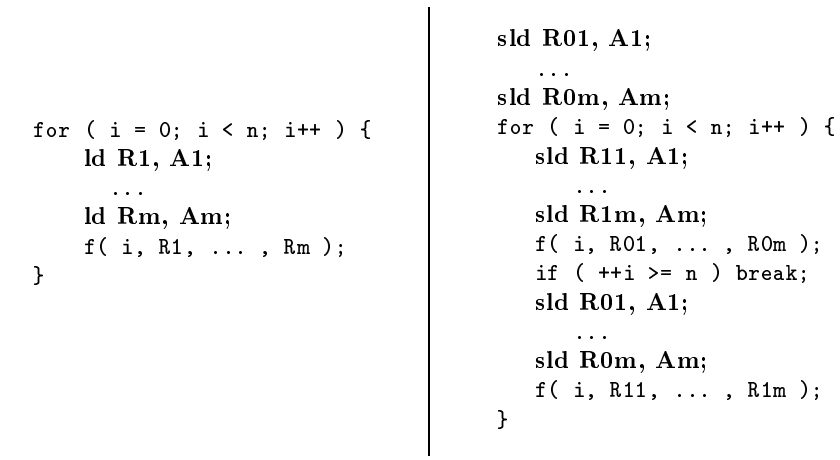


Figure 5: Lifting Across Loop Iterations

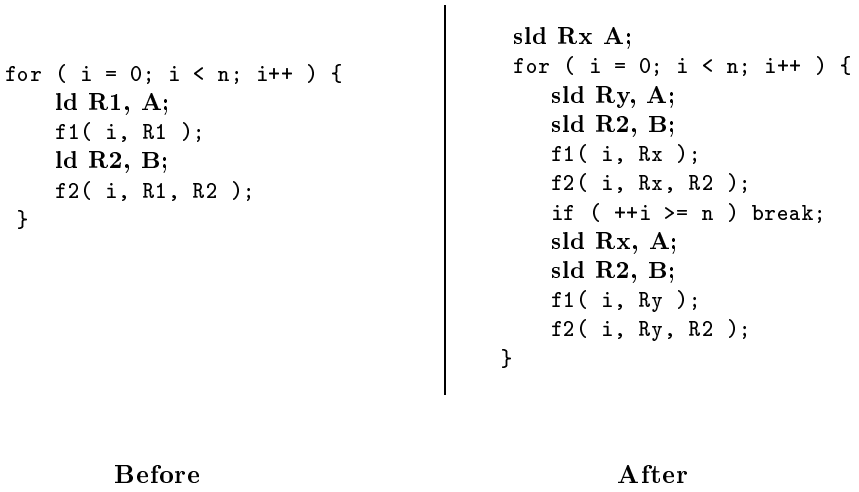


Figure 6: Lifting Across and Within an Iteration

pipeline. In addition, CL-SPIM implements both the interlock and non-faulting mechanisms needed for the speculative load instruction.

The simulator implements four different memory systems: the three mentioned earlier (Bypassing, Unified, and Spec-cache), plus a Baseline model that serves as a point of comparison. All four memory systems use the same cache model: physically-addressed, direct-mapped, with a write-through/no-write-allocate policy, and a six-deep write buffer. Only the line sizes and number of lines in the cache differ between models. We assume that read hits and most writes take 1 cycle. Read misses are delayed due to a non-empty write buffer only when the contents of the write-buffer conflict with the requested line and writes are delayed only when the write buffer is full. The differences among the four models follow:

Baseline The Baseline model provides 16K data cache that is organized into 512 32-byte lines. This model uses a single memory controller and memory that is interleaved to provide one word per cycle, once the cost to access the first word in a line has been paid.

Bypassing The Bypassing model is similar to the Baseline model in many ways. The data cache is smaller (8K organized as 512 16-byte lines[2]) to account for the extra real estate needed by the speculative load queues. The Bypassing memory model is 16-way interleaved 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.

Unified The Unified model differs from the Bypassing model in how much data is retrieved when servicing a speculative load and where that data is stored. The Unified model retrieves a whole cache line (16 bytes), instead of a single word when servicing a speculative load. In addition to filling the speculative load's destination register with the necessary value, the retrieved line is stored in the data cache.

Like the Bypassing model, the data cache is an 8K physically addressed, direct-mapped cache, organized as 512, 16-byte lines and the memory system is 16-way interleaved at the line level.

Spec-Cache The difference between the Spec-cache and Unified models is where to store the cache line retrieved with with speculative load. Unlike the Unified model, the Spec-cache model stores the cache line in a separate dedicated cache. The data cache is the same size as in the Unified and Bypassing models: 8K bytes organized as 512, 16-byte lines. We use a 4K spec-cache organized as 256, 16-byte lines.

Note that none of the models includes a second level cache. And that the results reported in Section 7 are based on a warm-started cache.

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

All models use a split transaction bus that multiplexes addresses and data. A simple arbitration scheme manages 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 transaction 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 other models, the difference being the length of the cache lines (4 words vs. 8 words). In all models, servicing a 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 and on how much data is retrieved when servicing the

[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.

Table 2: Transaction Costs in Cycles

Transaction	Write Request	Read Request		Speculative Load Request		
		32 bytes	16 bytes	Bypassing	Unified	Spec-Cache
Arbitrate	1	1	1	1	1	1
Addr/data	2	1	1	1	1	1
Latency	2/9	2/9	2/9	2/9	2/9	2/9
Arbitrate		1	1	1	1	1
Data		8	4	1	4	4
Fix Up		1	1	1	1	1
Total	5/12	14/21	10/17	7/14	10/17	10/17

request. 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) for the Bypassing model and 10 cycles (17 cycles) for the Unified and Spec-cache models.

6 Experiments

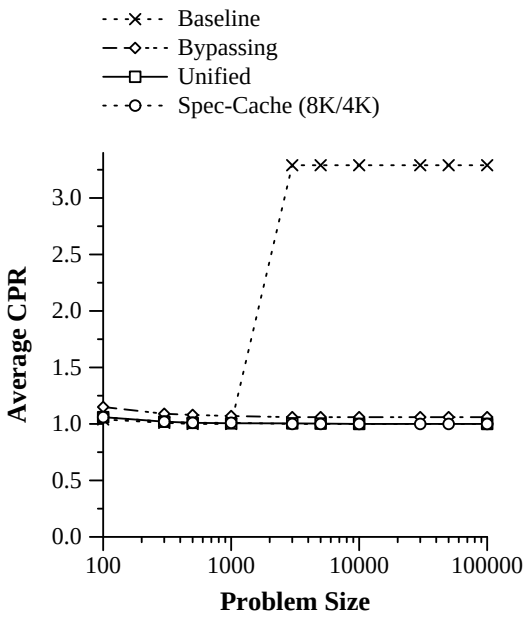
We used a selection of the Livermore Loop kernels written in C as a basis for our experiments. For the Baseline model, we compiled the C code using the MIPS C compiler (v 2.10) with the -O switch.

Generating code for the various speculative models took more work, because we did not have access to a a compiler capable of unrolling loops and inserting speculative loads. To generate code we unrolled the inner loop in the C code by hand, used the MIPS C compiler (v 2.10, -O, -S) to generate MIPS assembly code, inserted speculative loads in place of some regular load instructions, and assembled the resulting code using the MIPS reorganizing assembler. Note that 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 code within a loop body.

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

7 Results

Following Klaiber and Levy[22], we report the average number of cycles per memory reference (CPR). 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 add extra instructions; speculative loads replace load instructions. Therefore, in the best case, the average CPR will be 1.0. Our Baseline results differ from those reported by Klaiber and Levy, because our Baseline model is different. Our simulations model a write-through cache with a no-allocate-on-write policy instead of write-back cache, and we do not model a second level cache.




```
#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 7: Kernel K1

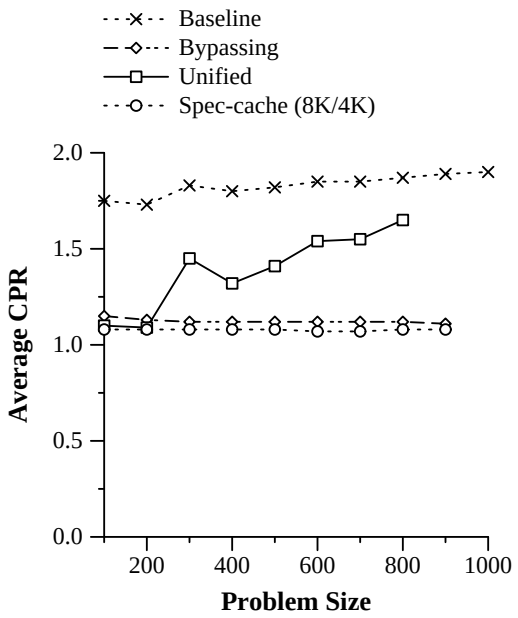
7.1 Loop Unrolling

Unrolling loops to lift speculative loads is the simplest of the techniques suggested in Section 4. In this section, 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.

7.1.1 Kernel K1

The first benchmark, Kernel K1, is a simple loop that walks through two arrays to compute the values for a third array (see Figure 7). The loop contains three double loads and a double store. The compiler can determine that one load (`ZX[k+11]`) is unnecessary. As the graph in Figure 7 shows, for $N < 1000$ the average CPR 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 memory traffic arises from write-buffer requests, and are stalled for several extra cycles waiting for write-buffer requests to complete.

Speculative loads work very well for this example. Both remaining loads can be replaced with speculative loads (speculatively loaded values are highlighted in boldface), because neither is involved in a troublesome data dependence. Unrolling the loop body once provides approximately 37 cycles for hiding latency. The Bypassing model performs reasonably well (1.07 CPR for `N`=1000). Roughly 86% of the speculative loads complete within 36 cycles and 98% complete within 39 cycles. The speculative loads do not complete faster, because they must compete for the bus with write-buffer requests, which have a higher priority. This causes a delay of approximately one half cycle per iteration. The Unified model performs better than the Bypassing model for this benchmark (1.006 CPR vs. 1.07 CPR for `N`=1000), because interference with the write-buffer is less of a problem. Only one quarter of the speculative loads use the bus (to complete at the memory system); half complete at the cache immediately upon issue and the remaining one quarter complete at the cache upon reaching the head of a speculative load queue.




```
#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 8: Kernel K6

Because the cache does not hold any data that is not speculatively loaded, the Spec-cache model does not provide any additional benefit (1.007 CPR for `N`=1000).

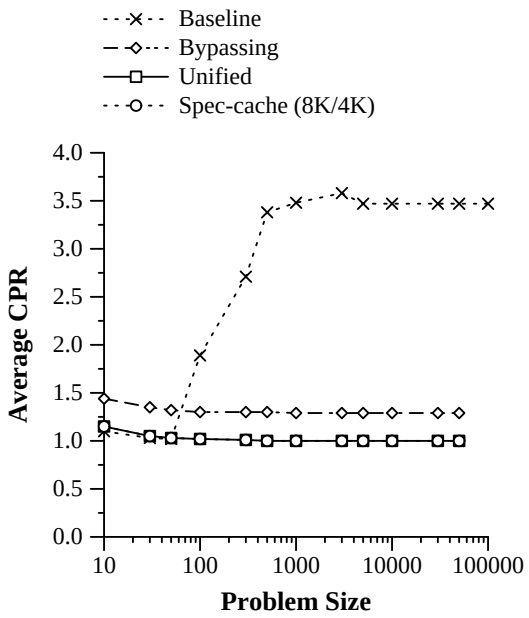
7.1.2 Kernel K6

The second benchmark, Kernel K6, is a simple general linear recurrence (see Figure 8). The code contains three double loads and a double store. For all simulated values of `N`, the data sets are too large for the cache, because each `B[i][k]` reference accesses a different element of the array. Even though the working set is large, the average CPR is low (under 2.0), in part, because each reference to `W[i]` inside an iteration of the outer loop references the same array element. These repeated references help inflate the reference count 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]` cannot be converted into a speculative load, because it has a loop carried dependence with the store. Since the reference to `W[i]` must be loaded through the cache, we did not convert the references `W[i-k]` into speculative loads. The loop body consumes at least 14 cycles between when the speculative loads are issued and when the first reference can occur.

Using the Bypassing model, the speculative loads of `B[i][k]` have an average latency of 15.12 cycles and more than 78% complete within 15 cycles. For $N = 100$, the average CPR is 1.15. The impact of the write-buffer on the latency of a speculative load is substantial. To determine its effect, we replaced each store with load that was guaranteed to hit in the cache. This trick maintains the appropriate number of memory references, while removing all write-buffer related bus traffic. Doing so reduces the average latency of a speculative load to 11.96. Even so, some of the speculative loads (14%) still take more than 14 cycles and cause the processor to stall (1.04 CPR).

The Unified model performs better than the Baseline model, but not as well as the other two speculative load models on this benchmark. Speculatively loaded cache lines replace cached data that is still live. One partial solution to this problem would be to use a 2-way set associative cache, which would decrease, but not completely eliminate the misses. The other solution is to use a separate cache for speculatively loaded values. The Spec-cache performed well, because `W` fits in the cache for all values of `N` tested and the latency of the `B[i][k]` loads can be hidden.




```
#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 9: Kernel K9

7.1.3 Discussion

In summary, both kernels K1 and K6 can use the simple unrolling technique to insert speculative loads and thereby improve the average CPR. Caching speculative loads can improve performance in comparison with the Bypassing model as long as the cached data does not displace non-speculatively load data.

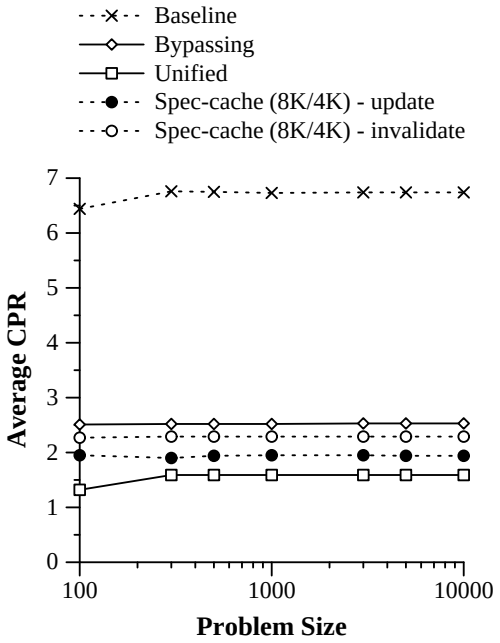
7.2 Basic Block Lifting

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.

7.2.1 Kernel K9

Kernel K9 integrates predictors (see Figure 9). The loop uses all available registers, which will be an issue when we try to introduce speculative loads. The body of the loop contains ten double loads and one double store. The constants (`DM28, DM27, ...`) are loaded outside the loop. From the graph in Figure 9, we can see that the data fits in the cache for `N` < 100. The data should fit in the cache for `N`=100, yielding an average CPR of 1.0, but conflict misses increase the cost to 1.89 CPR. For `N` > 300, the data no longer fits in the cache. Each line is reference 8.8 times on average, which yields an average CPR 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 20 loads, which were converted into speculative loads and lifted the appropriate distance using the basic block algorithm. Six speculative loads (those corresponding to `PX[13][i], PX[12][i],` and `PX[11][i]`) were lifted above the 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 replicated and lifted across all edges).




```
#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

The resulting code performs well for all three models. Both cached models perform better than the Bypassing model, because there is good spatial locality in the access pattern for the array `PX`. Only one quarter of the speculative loads complete at the memory system in the cached models. As a result, the average cost of servicing a speculative load is less than nine cycles, as compared to greater than 18 cycles for the Bypassing model.

7.2.2 Kernel K23

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 ten 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 10 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 conflict misses and the poor locality shown by the references to the other arrays. 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. But, the loop has enough computation (approximately 60 cycles) to hide most of the latency of the loads using the technique for lifting within a basic block. We converted all ten loads into speculative loads.

The transformed code performs much better under all three models. For example, using the Bypassing model the average CPR for `N`=1000 is 2.52. The Unified and Spec-cache models perform better than the Bypassing model, because the references to `ZA` benefit from caching. The remaining references (to the arrays `ZR, ZB, ZU, ZV`, and `ZZ`) do not benefit from caching the speculative loads, because the reference pattern conspires with the row major layout imposed by C to show little spatial or temporal locality. A column major layout would bring the CPR down from 1.6 for the Unified model to close to 1.0.

This is the first benchmark that writes to speculatively loaded data. To avoid possible incoherence between the data cache and the spec-cache, we prohibit writes to lines in the spec-cache. There are two alternatives for handling writes that hit in the spec-cache: invalidate and update. The invalidate-on-

write alternative invalidates the spec-cache line and treats the write as a normal write to the data cache. The update-on-write alternative transfers the line to the data cache and then proceeds with the write. In both cases, the value is written through to the memory system via the write-buffer, as usual. The results in Figure 12 show that Spec-cache with invalidate-on-write performs better than Bypassing, but not as well as Spec-cache with update-on-write. And that the update-on-write model does not perform as well as the Unified model. The invalidate-on-write Spec-cache model performs better than Bypassing, because it can can exploit some locality in the `ZA` array. It does not perform as well as the update-on-write model, because the writes in K23 update the array `ZA` and the resulting invalidates shorten the window of opportunity for exploiting locality. The Spec-cache model using update-on-write does not perform quite as well as the Unified model, because the Spec-cache is smaller than the data cache and as a a result can hold less of `ZA`.

7.2.3 Discussion

In summary, both kernels K9 and K23 can use the basic block lifting technique to insert speculative loads. In both cases, we saw dramatic improvements in the average CPR in comparison to the Baseline model. The two cached models performed as well or better than the Bypassing model.

7.3 Unrolling Revisited

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

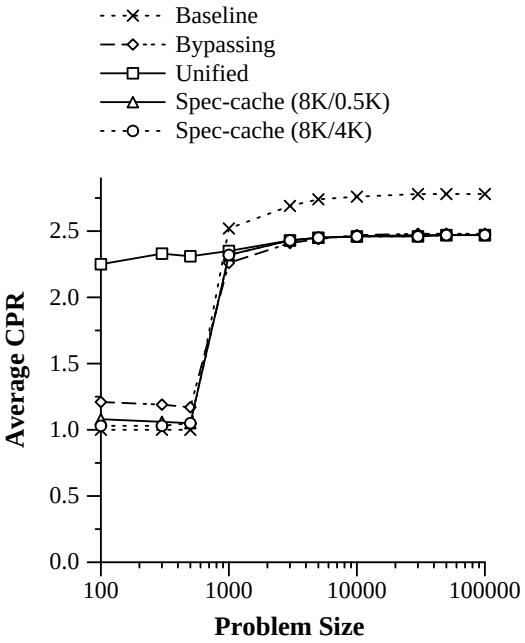
7.3.1 Kernel K2

The first benchmark, Kernel K2, is an excerpt from an incomplete Cholesky-conjugate gradient method. The code (see Figure 11) 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]` with a move instruction, because `k` is incremented by two for each iteration. Examining the graph in Figure 11 shows that the data fits in the Baseline cache for $\texttt{N} < 1000$, yielding an average CPR of 1.0. This benchmark fares 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.

Introducing speculative loads into this example provides 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.

The Unified model's curve is essentially flat at CPR=2.4 (even for small problem sizes), because speculatively loaded cache lines displace useful data in the cache. As we discussed earlier, there are two solutions to this problem. The first is to use a set associative cache. The second is to use a separate cache for speculatively loaded data. Even a small 0.5K spec-cache eliminates the cache line conflicts that arise in the Unified model. A small spec-cache is sufficient, because the accesses to array `V` have locality only over a very small window in time.

Once the problem no longer fits in the cache ($\texttt{N} > 1000$) all the speculative load models perform similarly. The addition of cached speculative loads does not improve the performance of this benchmark, because much of the data is still being loaded using regular loads, which contribute 98.6% of the memory stall cycles for `N`=1000.




```
#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 11: Kernel K2

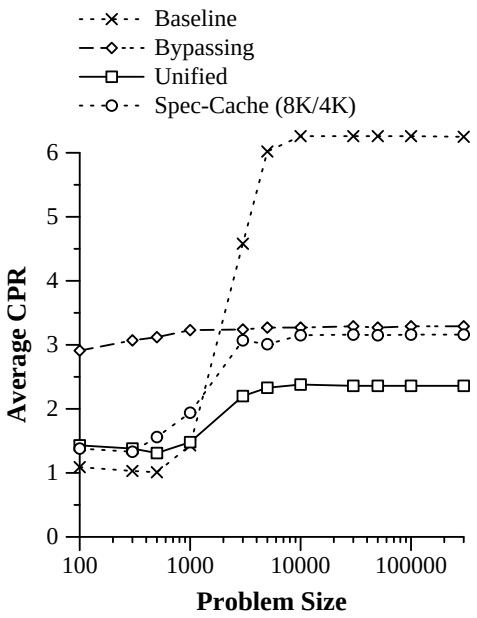
7.3.2 Kernel K4

The performance of K2 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 12). The inner loop contains a pair of double loads 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 the graph in Figure 12, the data fits in the Baseline cache for `N` < 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 CPR. For `N` > 3000, the number of references to a line before it is kicked out of the cache drops precipitously, resulting in an average CPR over 6.

Unrolling the loop once and replacing both loads with equivalent speculative loads has a dramatic effect on the performance of this benchmark. 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 models perform 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 using the Bypassing model (causing an 8 cycle delay) and 15.5 cycles using the Unified model (causing a 2-3 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




```
#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 12: Kernel K4

associated memory requests will sit in a queue until the earlier requests have been satisfied, which has the effect of increasing their latency significantly.

The performance of the Unified and Spec-cache models is similar to the Baseline model for $N < 1000$. The Spec-cache performs like the Bypassing model once the data is too large to fit in the cache, because there is not enough computation to hide the latency of the speculative loads and the spec-cache is not large enough to reduce the latency significantly. The Unified model performs the best. The average CPR increases to only 2.3, once the data is too large to fit into the cache. Because the Unified cache is larger than the Spec-cache, caching reduces the latency of a speculative load (from 21 cycles to 15.5 cycles).

7.3.3 Discussion

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 CPR due to memory/bus bandwidth limitations. Furthermore, caching the speculative loads perform better than Bypassing and there is no benefit to using a separate spec-cache.

7.4 TLB Issues

There are two possible approaches to handling a TLB miss caused by a speculative load: treat it as exceptions or trap to software.

TLB misses can be treated as exceptions, in which case, the speculative load mechanism poisons 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 to dealing a TLB miss caused by a speculative load is to always trap when it occurs and let the trap handler generate the speculative load. This approach is also simple and allows

Table 3: Average TLB Misses

Read Misses	0.00872%
Write Misses	0.04135%
Read/Write Average	0.02504%

the speculative load instruction to complete. The disadvantage is that the TLB trap handler now needs to distinguish between speculative loads and regular loads.

We instrumented the simulator to measure the number of TLB misses that occur. The results shown in Table 3 indicate that TLB misses happen infrequently. These averages are for all the Livermore Loops given in this paper.

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. 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.

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 an 8K direct-mapped data cache, speculative loads can hide memory latency effectively.

We illustrated that caching speculative loads results in a lower CPR than simply bypassing the cache. We presented two cached speculative load models (Unified and Spec-Cache). Only one benchmark (Kernel K6) performed better using the Spec-cache model than using Unified model. The additional implementation complexity of the Spec-cache model is not warranted given these results.

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 that 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.

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. Stefanos Damianakis was supported, in part, by a 1967 Scholarship from the Natural Sciences and Engineering Research Council of Canada. Kai Li was supported, in part, by NSF grant CCR-9020893, by ARPA 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.R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] R. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37(4):547–564, July 1993.

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

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

[12] 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.

[13] 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.

[14] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured representations. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Machines: 5th International Workshop (LNCS 757)*, pages 406–420. Springer-Verlag, 1993.

[15] 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.

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

[17] IBM. Special issue on RISC system/6000. *IBM Journal Research and Development*, 34(1), January 1990.

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

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

[20] SPARC International. *The SPARC Architecture Manual*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] S. Mahlke, W. Chen, W. Hwu, B. Rau, and M. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *The 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247, May 1992.

[27] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *The 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, May 1992.

[28] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12), December 1986.

[29] 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.

[30] 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.

[31] A. Rogers and K. Li. Software support for speculative loads. In *The 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, May 1992.

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

[33] 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.

[34] J.E. Thorton. Parallel operation in the Control Data 6600. In *Proceedings of the Fall Joint Computer Conference*, 1964.

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

[36] R.M. Tomasulo. An efficient algorithm for exploring multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

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