Appears in the proceedings of the
34th Annual ACM/IEEE International Symposium on Microarchitecture,
Austin, Texas, December 3-5, 2001

Correctly Implementing Value Prediction in Microprocessors that Support Multithreading or Multiprocessing

Milo M. K. Martin[†], Daniel J. Sorin[‡], Harold W. Cain[†], Mark D. Hill[†], Mikko H. Lipasti[‡]
[†]Computer Sciences Department and [‡]Department of Electrical and Computer Engineering
University of Wisconsin—Madison
{milo,sorin,cain,markhill,mikko}@cs.wisc.edu

Abstract

This paper explores the interaction of value prediction with thread-level parallelism techniques, including multithreading and multiprocessing, where correctness is defined by a memory consistency model. Value prediction subtly interacts with the memory consistency model by allowing data dependent instructions to be reordered. We find that predicting a value and later verifying that the value eventually calculated is the same as the value predicted is not always sufficient.

We present an example of a multithreaded pointer manipulation that can generate a surprising and erroneous result when value prediction is implemented without considering memory consistency correctness. We show that this problem can occur with real software, and we discuss how to apply existing techniques to eliminate the problem in both sequentially consistent systems and systems that obey relaxed memory consistency models.

1 Introduction

One prominent trend in micro-architectural research is improving system performance by adding prediction and speculation to a processor's core. Value prediction [26] is a type of prediction that has recently emerged from the research community, and numerous recent papers have demonstrated its performance potential. With value prediction, a mechanism predicts a complete value (e.g., a 64-bit integer), in contrast to a one-bit branch outcome resulting from branch prediction. In principle, value prediction can enable program execution in less time than the lower bound determined by the dataflow limit.

Another trend in micro-architectural research exploits thread-level parallelism (TLP) in the form of simultaneous multithreading (SMT) [13], coarse-grained multithreading (CMT) [2, 6], single-chip multiprocessing (CMP) [5, 11], or traditional multiprocessing (MP) [10]. From the software's perspective, hardware multithreading and multiprocessing are the same, and we treat them similarly in this paper. These techniques have been shown to improve performance substantially for important applications such as database workloads [4, 14, 27], web workloads [7, 28], and desktop applications [15].

This paper explores the correctness issues that arise from the interaction between these two techniques. To date, most value prediction research has assumed a single-threaded uniprocessor system and has ignored multithreading and input/output (I/O) issues. While the correct implementation of value prediction in the context of a single-threaded uniprocessor system without coherent I/O is well-understood, we will show that naïve implementations of value prediction in *TLP systems*—systems with multithreading, multiprocessing, or coherent I/O—can produce incorrect executions.

What do we mean by correctness? In a system with a single-threaded uniprocessor without coherent I/O, correctness is simply defined by program order (i.e., uniprocessor correctness). In this scenario, value prediction is correct if and only if *simple verification* succeeds, i.e., the value predicted equals the actual value eventually obtained. However, when multiple threads, processors, or devices concurrently access a logically shared memory, the definition of correctness becomes more complicated.

In TLP systems, correctness is defined by the *memory consistency model* [1]. The memory consistency model, enforced jointly by the processor and the memory system, is the interface between the hardware and (low-level) software that defines the legal orderings of loads and stores to different memory locations. For example, the memory consistency model of the system answers questions such as: "If a thread writes to two different memory locations, in what order are other threads or devices in the system allowed to observe these writes?" and "Will all of the threads in the system observe these writes in the same order?" Memory consistency models are defined as part of the instruction set architecture (ISA), and the hardware must obey the consistency model, just as the hardware must correctly implement all instructions as specified by the ISA. Thus, TLP system implementations that use value prediction must ensure that value prediction does not cause consistency model violations.

How can value prediction violate the consistency model? The key insight is that *value prediction allows a processor to relax the ordering between data dependent operations*. Normally these dependencies are enforced by dataflow in the processor, but value prediction allows the processor to break the dataflow of a program, allowing dependent operations to speculatively execute out of program order. More generally, any current or future microarchitectural optimization that allows the relaxation of program order between data dependent operations can lead to consistency violations, but we focus on value prediction in this paper.

The key result of this paper is that, in systems with multithreading, multiple processors, or coherent I/O, verifying value prediction by comparing the predicted and actual values is not always sufficient and can cause erroneous behavior. In a TLP system, unlike in a single-threaded uniprocessor, it is possible for a value prediction to be incorrect at the time of the prediction but ‘correct’ by the time the value prediction is to be verified, since another thread, processor, or I/O device could have modified the value in the interval between prediction and verification. With simple verification, value prediction appears correct, but we will show cases in which the execution is incorrect because it violates the memory ordering rules of the consistency model. Any possible violation is sufficient to determine that the implementation is incorrect.

As an informal example of how simple value prediction could cause a problem, consider the following scenario, in which a professor plans to post the results of an exam and an impatient student in the class cannot wait to see her score. The student, assigned student ID #5 for this class, predicts that the results might be posted on bulletin board B. She arrives at bulletin board B and finds that student #5’s score is 60. Unbeknownst to her, the results that she is looking at are old results from another class. Later, the professor posts the correct results (her actual score is 80) on board B, replacing the results the student had seen, and announces that the results are posted on board B. Since that is where the student looked originally, she ‘verifies’ her earlier prediction (of board B) and continues to incorrectly assume that her score was 60. Throughout this paper, we will explore an analogous scenario that arises in multithreaded code used for reading and writing a linked list data structure.

In Section 2, we examine the issues involved in implementing value prediction in the context of sequential consistency, the simplest consistency model. Through an example, we show that adding simple value prediction to a TLP system can be insufficient to implement sequential consistency. We then review techniques used to ensure correctness in aggressive implementations of sequential consistency, and we show that these existing mechanisms are still sufficient when value prediction is added. In Section 3, we investigate relaxed consistency models and demonstrate that simple implementations of some relaxed consistency models are insufficient when value prediction is added. Mechanisms similar to those used in aggressive sequentially consistent implementations can be added to ensure correctness, but these mechanisms are conservative. We then present ideas for less conservative schemes, but detailed evaluation of their relative performances is beyond the scope of this paper. A potential criticism of this work is that the value prediction problems we illuminate might occur in theory but not in practice. In Section 4, we show that the dynamic code sequences necessary to generate value prediction errors in a relaxed memory model can occur in workloads we simulated on a four-processor symmetric multiprocessor (SMP).

2 Value Prediction & Sequential Consistency

The simplest and most intuitive set of rules governing the behavior and ordering of memory operations between threads and devices is *sequential consistency* (SC) [24]. In this section, we describe SC, and we present an example in which a simple system (that implements SC without value prediction) fails to implement SC when value prediction is added naïvely. We describe two techniques for the detection of ordering violations, and these techniques restore the simple system to a valid SC implementation. These two techniques are the same methods used for the correct implementation of SC in aggressive out-of-order processors [17]. Thus, adding value prediction to a system that supports SC with simple processors can result in additional complexity, while adding value prediction to an already dynamically scheduled SC implementation will have minimal additional design impact due to memory consistency model considerations.

2.1 SC and a Simple SC System

The memory consistency model of a system specifies how memory operations appear to the programmer [1]. Many programmers would like to view a TLP system as a multi-tasking uniprocessor. Lamport [24] formalized this notion when he defined a system to be *sequentially consistent* (SC) if (1) the result of any execution is the same as if the operations of all the processors (or threads) were executed in some sequential order, and (2) the operations of each individual processor (or thread) appear in this sequence in the order specified by its program. SC is the most restrictive consistency model that has been implemented in commercial systems, including the MIPS R10000 [34] and the HP PA-8000 [23], and it presents the simplest, most intuitive, and least surprising interface to the programmer.

We first consider a simple in-order processor that implements coarse-grained hardware multithreading in which multiple hardware contexts share a cache. The processor performs all memory operations from each thread in order, and thus it implements SC. A system without multithreaded processors, but with multiple processors and standard invalidation-based cache coherence, exhibits similar interactions between value prediction and memory consistency. We will consider more aggressive processor designs in Section 2.3.

2.2 Simple Value Prediction Can Violate SC

We now show that extending our example system with simple value prediction (i.e., using only simple verification) violates SC.

Simple value prediction. When an instruction is value predicted, the processor predicts the value produced by that instruction and continues executing instructions from the same thread (including dependent instructions) speculatively. For simplicity, we conservatively assume that our implementation waits for any value predictions to be verified before executing any store instructions encountered while speculating. When the predicted instruction completes, possibly many cycles later due to cache misses or other delays, the processor compares the actual value with the predicted value. If the value matches, the prediction is determined to have been successful and execution continues. Otherwise, the processor aborts execution and rolls back the thread using a mechanism similar to that used in recovering from a branch misprediction.

Illustrative example. Figure 1 illustrates a problem with the simple implementation of value prediction, and it will serve as our illustrative example throughout the paper. The example is a pointer-based data structure manipulation that is analogous to our informal example of the professor posting grades and the student checking her grade. One thread (the professor) is inserting at the front of the list, while the other thread (the student) is reading the first element of the list. No further synchronization is necessary if there is only one writer and one reader. The reader or writer may execute its code first, or the instructions may occur interleaved. Under SC, this code segment allows only two possible outcomes: the read happens before the insert, resulting in the reader observing the data value 42, or the insert occurs before the read, and the reader observers the data value 80 (the student's grade). Because, under SC, the store that changes `B.data` from 60 (an old grade from a previous class) to 80 must occur before the store that changes the head pointer to point at `B`, $\texttt{T}_{\texttt{reader}}$ should never observe the stale value 60.

Example execution. The example code sequence works correctly with our SC system implementation without value prediction (returns only 42 or 80), but simple value prediction can generate a surprising result (the value 60). We first examine how value prediction can change the execution of this example. $\texttt{T}_{\texttt{reader}}$ executes first. Assume that $\texttt{T}_{\texttt{reader}}$ value predicts the result of instruction `r1`. Since any value can be predicted, we assume that the hardware predicts the value `B`. (The student predicts that her grade is on board B.) Notice that it *appears* that this value prediction will be incorrect, since the value of `Head` is actually `A` and not `B` at this time. $\texttt{T}_{\texttt{reader}}$ continues executing speculatively and instruction `r2` reads the value 60. (The student sees a score of 60.) Reading an impossible value while speculating is allowed, and it is only discarded when the value prediction is ultimately determined to be incorrect. Before $\texttt{T}_{\texttt{reader}}$ resolves its value prediction, a thread switch occurs, and $\texttt{T}_{\texttt{writer}}$ executes instructions `w1`-`w4`, effectively inserting a new node at the beginning of the list. (The professor posts grades on board B.) The memory system now processes $\texttt{T}_{\texttt{reader}}$'s load for `Head` and returns 'B', the current value. $\texttt{T}_{\texttt{reader}}$ now compares the predicted value (board B) with the actual value (the professor announces that results are on board B) and surprisingly decides the value prediction was correct. Since the value prediction was pronounced correct, $\texttt{T}_{\texttt{reader}}$ continues to execute. This execution violates SC because instruction `r2` reads the value 60 (and the student thinks her score is 60).

Intuition. Why was the value prediction 'correct' when it was clearly initially wrong? By the time the value prediction was resolved, the thread had the new value. In essence, it predicted the future, and this allowed it to read a location that was not ready to be observed. (The student 'verified' that she was looking at the right set of grades, but she was not actually looking at the right grades *when she made the prediction*, so she incorrectly believed that her score was 60 instead of 80.) The key observation is that adding simple value prediction allows us to perform two *dependent* operations (`r1` and `r2`) out of program order. By executing these data dependent memory operations out of order, SC was violated. Moreover, while this particular example was for a load value prediction, value prediction of other types of instructions faces the same issues. For instance, if a load address is dependent upon an instruction whose output has been value predicted, a similar violation is possible.

Formal explanation of why the value 60 is not valid. For the above execution to be correct, the definition of SC requires that we construct a total order of memory operations (i.e., loads and stores). Instruction `w4` precedes `r1` in the global order, because `w4` writes the value read by `r1`. Instruction `r2` precedes `w1`, because `r2` reads the value 60 before `w1` writes the value 80. SC's second requirement,

Code for T_{writer}

w1: store mem[B.data] ← 80
w2: load reg0 ← mem[Head]
w3: store mem[B.next] ← reg0
w4: store mem[Head] ← B

Code for T_{reader}

r1: load reg1 ← mem[Head]
r2: load reg2 ← mem[reg1]

(a) Racing code

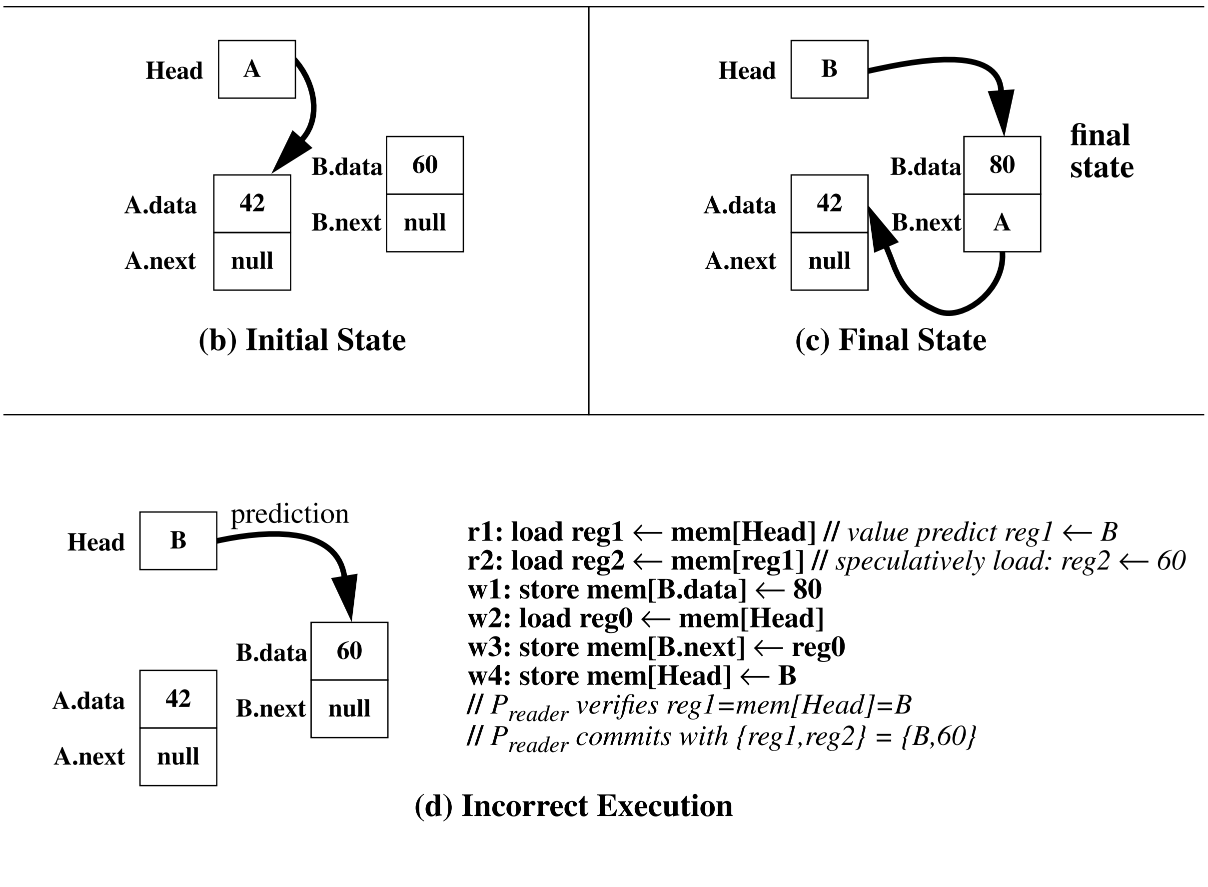

Figure 1. Example Showing Failure of Simple Value Prediction

Part (a) presents code for $\mathtt{T}_{\mathtt{writer}}$ (left) that races $\mathtt{T}_{\mathtt{reader}}$ (right). $\mathtt{T}_{\mathtt{writer}}$ sets element *B*'s value to 80 and links element *B* to the beginning of the list. $\mathtt{T}_{\mathtt{reader}}$ reads the value of the first element.

Part (b) gives the initial state of memory for a singly-linked list with a *Head*, currently-linked element *A*, and unlinked element *B*. Each element has a data value field and a next pointer.

Part (c) shows the final state, after $\mathtt{T}_{\mathtt{writer}}$ atomically inserts element *B*. $\mathtt{T}_{\mathtt{reader}}$ can execute either before or after the atomic insert, obtaining 42 from *A* or 80 from *B*, respectively.

Part (d), however, shows that value prediction with simple verification can allow $\mathtt{T}_{\mathtt{reader}}$ to obtain the incorrect, stale value 60. Technically, this result assumes sequential consistency (SC), but, as we will see, similar problems exist for other memory consistency models.

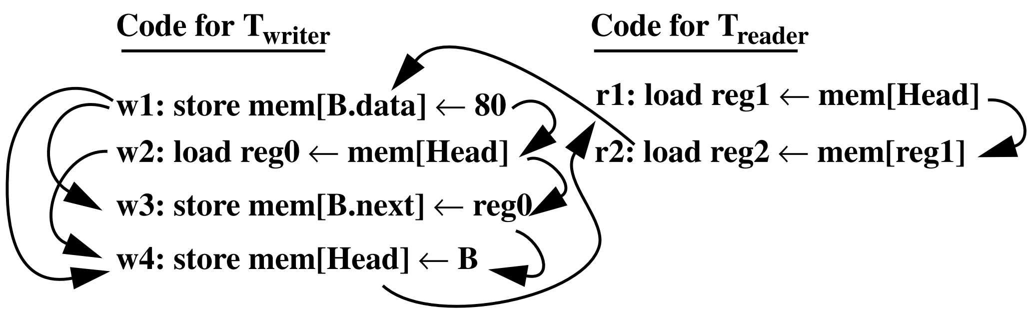

Figure 2. Value Prediction with Simple Verification Violating SC. The example execution of Figure 1(d) forms a cycle, violating SC.

called *program order*, requires that `r1` precedes `r2` and that `w1` precedes `w2`, `w2` precedes `w3`, and `w3` precedes `w4`. The instructions cannot be put in a sequential order (SC's first requirement), because the required order of instructions forms a cycle, as shown in Figure 2. Therefore, this execution does not obey SC, and the example system using simple value prediction does not correctly enforce SC.

2.3 Restoring Correctness

Several alternatives exist for correctly implementing value prediction in a sequentially consistent system. These schemes—which are based on previously developed techniques for implementing dynamically scheduled processors that support SC—can be used to detect when value prediction has violated SC. These techniques suffice for detecting ordering violations caused by value prediction and recovering from them, but they add complexity and increase the cost of the simple processor implementations considered thus far. We first discuss how both techniques enforce SC in systems without value prediction, and then we explain how these mechanisms also detect ordering violations introduced by value prediction.

Address-based detection. Dynamically scheduled (out-of-order) processors that implement SC (e.g., the MIPS R10000 [34]) allow memory operations to occur speculatively out of program order and then rollback if a possible memory ordering violation is detected [17]. To enforce SC, a processor must detect when another thread, processor, or device writes to an address that has been speculatively read from the cache by an unretired instruction. When an ordering violation is detected, the processor rolls back the execution of the speculative thread to a known consistent and non-speculative state. The R10000 implements this approach by augmenting the load/store queue to (1) track the addresses that have been speculatively loaded until the loads retire and (2) compare the addresses to the addresses that are written by other processors. These external writes are revealed through the coherence protocol by the arrival of invalidation messages for these addresses [34]. Future multithreaded processors can implement this similarly, by keeping a per-thread table of speculatively loaded addresses and checking all the stores of the other threads (from any processor) against this table. We refer to the process of comparing other threads' stores and other processors' invalidations against the set of addresses that a thread has loaded speculatively as *read set tracking*. This scheme is overly conservative in that false squashes can be triggered by false sharing (in multiprocessors) or writing the same value [17] (i.e., a silent store [25]).

In Figure 1, the relaxation of program order between `r1` and `r2` that is enabled by value prediction is analogous to the relaxation of program order between load instructions that are not data dependent. Since SC requires that `r1` and `r2` appear to occur in order in both cases, read set tracking is sufficient for identifying all ordering violations. In our example, if T_{reader} detects that T_{writer} writes to `mem[reg1]` (detected by comparing the store address from T_{writer} to the address speculatively loaded by `r2` in T_{reader}) between the prediction and retirement, it knows that simple verification of the prediction might be insufficient. No special check to enforce dependence order is needed, since guaranteeing the appearance of program order also guarantees the appearance of dependence order.

Returning to our running analogy of the student and her test score, the student must detect if the professor posts grades between her prediction and her verification that bulletin board B was the correct prediction. Thus, the student asks a friend to stand by bulletin board B and report if anyone changes the posted grades. In the case where the professor posts to board B after the student's prediction, the student's friend would tell her the information on the board changed, and thus the student would know to check her score again. If the friend reports no violation, the grade is known to be correct and no further validation is required.

Value-based detection. In this approach, all loads that execute with directly or transitively predicted address operands are replayed when their operands become non-speculative (e.g., immediately before retirement) [17]. A speculative load must wait for its own operands to become non-speculative, read its value from the cache a second time, and then compare this non-speculative value with its earlier speculatively loaded value. If the values match, no ordering violation has occurred. If the values do not match, the standard misprediction recovery mechanism is invoked. This approach avoids false squashes due to either false sharing or silent stores. However, this approach has a considerable downside in that some loads must execute twice, thus increasing the contention for cache read ports.

Dynamic verification [3], a recently introduced technique for tolerating transient and design errors in a microprocessor, can be used to implement value-based detection of memory consistency ordering violations. One proposed implementation of dynamic verification uses separate core and checker processors, each with a dedicated data cache [8]. This organization mitigates the cache port pressure problem and removes verification from the critical path, by allowing the main processor to proceed with executing and speculatively retiring instructions. Using this decoupled approach can reduce the cost/performance penalty associated with more traditional implementations of value-based detection.

Value-based detection also fits into our simple analogy. Once the professor has announced where the scores are posted, the student (or a friend) must go to bulletin board B and double-check the score, just in case. In the situation where the speculatively observed score is 60 and the re-observed score is 80, the student would know that her prediction was wrong.

Discussion. Some have noted a connection between memory value prediction and *software-directed binding prefetching*. With binding prefetching, data is brought into registers early under the direction of software. Software must guarantee high-level language semantics, possibly with hardware assistance (e.g., IA-64's ALAT [20]). In contrast, memory value prediction should be implemented in hardware without disturbing the low-level memory consistency model.

2.4 Value Prediction & SC Conclusion

Broadly, SC implementations are either simple (i.e., coarse-grained multithreading, in-order, and non-speculative) or aggressive (i.e., SMT or out-of-order and highly speculative). Correctly applying value prediction in a simple SC system moves the implementation complexity toward that of an aggressive SC implementation, because the mechanisms for verifying value prediction are similar to those that enable aggressive SC implementations. Implementing value prediction in a system that supports dynamic scheduling and SC is straightforward, because the existing mechanisms are already sufficient. In the next section, we will explore the ramifications of adding value prediction to systems that exploit relaxed consistency models.

3 Value Prediction & Relaxed Memory Models

Many common instruction set architectures [18, 19, 20, 30, 33] do not require the strict semantics of sequential consistency. These systems are said to implement relaxed memory consistency models. Relaxed memory models allow the hardware to potentially employ optimizations such as store queues and write buffers, and they can simplify the implementation of out-of-order execution. However, relaxed models require the programmer to add explicit annotations to enforce some memory orderings. Value prediction interacts with relaxed models much like it interacts with SC, and thus similar challenges exist for value prediction with many relaxed consistency models.

Defining relaxed memory consistency models is complex. We refer the reader to Adve and Gharachorloo [1] for a tutorial on the subject and the references to many primary sources. Broadly, there are two classes of relaxed models that we will address in this section. One class—sometimes called *processor consistent (PC) models*—is similar to SC, except that the models allow for FIFO, non-coalescing store buffers by relaxing the order from a thread's write to its subsequent reads. The other class, generally referred to as *weakly ordered models*, allows much more reordering of reads and writes.

3.1 Processor Consistency

PC models, such as SPARC Total Store Order (TSO) [33] and IA-32 [19], allow relaxation of the order from a thread's write to its subsequent reads. Since PC models do not allow relaxation of read-to-read program order, simple implementations must, in our example, execute `r1` and `r2` in program order. If, on the other hand, a more sophisticated implementation allows reads to be reordered by out-of-order execution, it must guarantee the appearance of program order execution by either of the two methods described in Section 2.3.

Although PC relaxes write-to-read order, the result from Figure 1(d) is not valid under PC models. Going back to Figure 2 from the SC example, the only difference for PC is that, since PC relaxes write-to-read ordering, the arc from `w1` to `w2` is not present. Nevertheless, the arc from `w1` to `w3` is still part of a cycle. Therefore, as with SC, correct implementations of value prediction for PC models must not return the surprising result from Figure 1(d).

The issues for correctly implementing value prediction under PC models are substantially similar to those issues for SC discussed in Section 2, because PC models and SC both enforce read-to-read ordering. A mechanism which guarantees the appearance of read-to-read program order suffices to guarantee the appearance of read-to-read dependence order, since two loads serialized by dependence order are also serialized by program order. Thus, as with SC, adding value prediction to a simple processor that supports PC would require an additional mechanism to detect violations introduced by value prediction. Adding value prediction to a more complicated processor—one that already speculatively relaxes read-to-read order and contains a mechanism to detect violations—would not require an additional mechanism.

3.2 Weakly Ordered Models

This section concentrates on the other class of relaxed memory consistency models, including weak ordering and release consistency, that allows a processor to reorder reads and writes, provided that a processor sees its own reads and writes in order. Commercial models in this class include Alpha [30], PowerPC [9, 18], IA-64 [20], and SPARC Relaxed Memory Order (RMO) [33]. These models differ in subtle ways, but they all require that the programmer insert one or more *memory barriers* (a.k.a., *MBs, barriers*, *membars*, *fences*, or *syncs*) or annotations to assert required orderings.

Unlike with SC and PC, dynamically scheduled processors that support most weaker models do not need to implement memory ordering detection mechanisms like those described in Section 2.3. Processors can allow memory operations to speculatively execute out of order without requiring additional inter-thread detection mechanisms. These re-orderings are no longer violations but rather correct semantics allowed by the memory model.

However, these processors must enforce ordering across memory barriers. For simplicity, when a thread encounters a memory barrier in our examples, order is enforced by stalling the execution of all instructions following the memory barrier until it retires.[1] The thread only continues execution when all speculative instructions (including any value predicted instructions) have completed. This implementation is perhaps conservative, but it is similar to that of the Alpha 21264 [12, 22].

One subtle difference among weak memory models, which affects their interaction with value prediction, is whether they establish order between two data dependent reads from the same processor. Some models do not require a memory barrier between data dependent operations (e.g., through a register, as in Figure 2's `r1` and `r2`), while other models always require a memory barrier to enforce the ordering of two reads (even if the reads are data dependent). We say that the former models enforce *data dependence order.* We now discuss both alternatives in turn.

Code for T_{writer}	**Code for T_{reader}**
w1: store mem[B.data] ← 80	**r1: load reg1 ← mem[Head]**
w2: load reg0 ← mem[Head]	**r2: load reg2 ← mem[reg1]**
w3: store mem[B.next] ← reg0	
$\boxed{\textbf{w3b:memory barrier}}$	
w4: store mem[Head] ← B	

Figure 3. Correct Code for Weak Ordering with Data Dependence Enforced

1. To simplify the discussion, we assume one type of memory barrier that enforces all orderings. In reality, most relaxed models have multiple flavors of barriers or annotations with different ordering requirements.

Models that enforce data dependence. We first discuss models, including SPARC RMO [33], PowerPC [9, 18], and IA-64 [20], that require a memory barrier to order independent reads but not dependent reads. In the latter case, hardware is required to preserve the dependence order. Even without considering value prediction, programmers that want the linked list code to allow only the two expected outcomes of Figure 1 must add a memory barrier before instruction `w4`, as shown in Figure 3. The memory barrier before `w4` asserts that instructions `w1`, `w2`, and `w3` (initializing element B) should be ordered before instruction `w4` (which inserts element B at the head of the list). Without the memory barrier, `w4` could be ordered before any of the other three operations, resulting in the addition of a (partially) uninitialized node to the head of the linked list.

To extend the informal analogy of the professor posting grades, consider a different situation in which the professor is too busy to post the results, but instead sends a teaching assistant (TA). Assume that the professor gives the results to the TA to post and then immediately sends an e-mail to the class. If the TA is delayed in posting the results, the student might see the e-mail, go to check her score, and once again believe erroneously (despite not using value prediction) that her score was 60. The professor can prevent this problem by waiting for the TA to report back—acknowledging that the results have been posted—before sending the e-mail. Waiting for the TA to return is analogous to inserting a memory barrier between `w3` and `w4` (which is required of all weak models, regardless of whether they enforce data dependence order).

Weak models allow the relaxation of most read-to-read ordering, so it would appear that our example should require the insertion of an additional memory barrier between `r1` and `r2` to enforce order between these reads.

However, we are now only considering those models that enforce data dependence order. Our example does not require a memory barrier between instructions `r1` and `r2` because the data dependence between `r1` and `r2` orders the instructions in dependence order in RMO [33, page 260], PowerPC [18, page 106], and IA-64 [20, section 13.2.1.7].

While the addition of the memory barrier between `w3` and `w4` ensures correctness *without* value prediction, it does nothing to prevent incorrect execution with naïve value prediction. (The student can still predict that her grade is on board B, see that her score is 60, later verify that board B was the correct location, and still erroneously believe that her grade was 60 and not 80.) Thus, this simple implementation of value prediction violates weak memory models that enforce data dependence order.

One approach to restoring correctness is to change the memory consistency model by removing the enforcement of data dependence order. This change requires programmers to insert a memory barrier between dependent instructions (such as `r1` and `r2`). A memory barrier will explicitly order these instructions, ensuring that only the expected two outcomes of our example can occur. (The student can predict that she should look on board B, but she cannot act on her prediction until she observes the professor's announcement) This approach, however, breaks backward compatibility, since changing the memory model definition of an architecture to require additional memory barriers may break programs written for the old definition. For this reason, we do not consider this to be a practical solution.

A conservative solution is to use the same approach described earlier for aggressive SC and PC implementations: enforce all read-to-read program order by speculatively executing and explicitly detecting possible violations. As in the SC and PC cases, this technique is sufficient to avoid the subtle correctness issues induced by value prediction. However, this solution can reduce performance due to false squashes, and it faces the same implementation issues described in Section 2.3.

Alternatively, a more aggressive strategy could perhaps selectively enforce read-to-read ordering only for dependent operations or only while a value prediction is active in the processor, reducing unnecessary squashes. We leave a detailed description, proof of correctness, and performance evaluation of more aggressive techniques for future work.

Models that do not enforce data dependence. The other class of weak memory models, which includes Alpha [30], requires a memory barrier to order two reads, regardless of whether they are dependent. For example, Alpha programmers that want our linked list example to allow only the two expected outcomes of Figure 1 must insert two memory barriers (shown in Figure 4). First, just as for models that do enforce data dependence, a memory barrier must be inserted before instruction `w4`. Second, Alpha also requires a memory barrier between instructions `r1` and `r2`, because the Alpha model does not enforce data dependence order. Without this second barrier, the surprising result of Figure 1(d) is valid under the Alpha memory consistency model.[2] By adding the second barrier, this result is disallowed.

Code for T_{writer}	**Code for T_{reader}**
w1: store mem[B.data] ← 80	**r1: load reg1 ← mem[Head]**
w2: load reg0 ← mem[Head]	$\boxed{\textbf{r1b:memory barrier}}$
w3: store mem[B.next] ← reg0	**r2: load reg2 ← mem[reg1]**
$\boxed{\textbf{w3b:memory barrier}}$	
w4: store mem[Head] ← B	

Figure 4. Correct Code for Weak Ordering without Data Dependence

One advantage of not enforcing dependence order is that naïve value prediction does not violate the consistency model. A straightforward implementation would not allow value prediction across a memory barrier, preventing subtle reorderings due to value prediction. A side effect of adopting a memory model that does not enforce dependence order is that the programmer must insert a memory barrier to explicitly enforce data dependence order (e.g., between `r1` and `r2`). Requiring these additional memory barriers increases the frequency of memory barriers, possibly reducing performance, and it adds an extra burden on the programmer by requiring barriers in "non-intuitive" locations [20, section 13.2.1.7].

3.3 VP & Relaxed Memory Models Conclusion

Value prediction can complicate some implementations of relaxed memory models that enforce read-to-read ordering of dependent operations. Examples of these relaxed models include SPARC TSO, IA-32, SPARC RMO, PowerPC, and IA-64. However, memory models that do not enforce data dependence order (e.g., Alpha) allow for simple implementations of value prediction, as long as prediction across memory barriers is not allowed.

4 Could Violations Occur in Real Code?

The correctness issues described above are valid regardless of the frequency with which these errors may occur. Nevertheless, the issues might seem less important if

2. At least one Alpha implementation leverages the relaxation of data dependence order and thus could produce undesired results if the memory barrier between `r1` and `r2` is omitted [16].

Table 1. Frequency of Consistency Model Violations Caused by Value Prediction

Application	Instructions Executed (millions)	Possible Violations
TPC-W	3,688	161
Raytrace	979	0
SPECjbb2000	4,559	0
SPECint_rate95	729	0
SPECweb99	1,651	5

they occurred only for contrived code sequences instead of real ones. In this section, we quantify the dynamic frequency of code sequences similar to the relaxed consistency code illustrated in Figure 3, while simulating a set of five multithreaded workloads on a 4-processor PowerPC system. We show that the error potential is real, as two of the five workloads contain code sequences in which simple verification of a value prediction could be incorrect.

We simulate the user and system level instructions of all workloads using the SimOS-PPC full system simulator [27], augmented with a detailed memory hierarchy modeling an IBM RS/6000 S80 server. The workloads chosen are representative of a wide array of applications, each running on the AIX 4.3 operating system. TPC-W [32] is an e-commerce benchmark using IBM's DB2 database and the Zeus Web Server. Raytrace is a parallel image rendering application from the SPLASH benchmark suite [29]. We also use three SPEC benchmarks [31]. SPECjbb2000 is a multithreaded transaction processing application written in Java. SPECint_rate95 uses multiple threads to concurrently execute the SPEC95 integer benchmarks. SPECweb99 is a web serving benchmark using the Zeus web server.

We detect potential consistency model violations using two 100-entry FIFO queues per processor, one for loads and one for stores. The store FIFO associated with each processor tracks dynamic store and memory barrier instances. The load FIFO tracks dynamic load instances, the dependences among them, and whether or not dependent loads were separated by memory barriers. For each execution of a load that is dependent upon another load during this 100 instruction window, without being separated by a memory barrier or other ordering instruction, we scan the store FIFO of all other processors. During this scan, we search for two store instructions which overlap the memory locations read by the dependent loads, where the store instructions are separated by a memory barrier. A match is a potential consistency violation.

As shown in Table 1, potential consistency model violations occur in two of the five applications. Although they occur infrequently, any occurrence indicates a potential system failure, which is not acceptable. We have examined the code surrounding a few of these violations, and we have found that many occur in locking routines called from the operating system task dispatcher. We have found that many of the loads and stores involved are normal load and store operations, and they are not restricted to load and store conditional synchronization operations for which value prediction could be selectively disabled.

5 Conclusions

We have shown that micro-architects must consider system correctness, as defined by the architecture's memory consistency model, if they are implementing value prediction in microprocessors that are to be used in systems with thread level parallelism (TLP). Value prediction can induce violations of read-to-read dependence ordering, and these violations can cause incorrect executions of multithreaded workloads. This issue exists in many commonly applied consistency models—including sequential consistency, processor consistency, and some flavors of weakly ordered models—and thus pertains to many commercial architectures.

For each class of consistency models with which value prediction can induce violations, we have presented solutions that are sufficient to eliminate consistency model violations due to value prediction. For the relaxed models, including processor consistency and weak ordering, one viable solution is to borrow the mechanisms that are used in aggressive implementations of sequential consistency to detect violations of read-to-read program order. This solution, however, has potential drawbacks in terms of performance and complexity. Alternate solutions may alleviate these drawbacks, but an evaluation of their relative performances is beyond the scope of this paper.

Acknowledgments

We thank the following individuals for their contributions to the findings presented in this paper: Gordie Bell, Ras Bodik, Adam Butts, Kourosh Gharachorloo, Kevin Lepak, Kevin Moore, Craig Saldanha, Ed Silha, and David Wood. This work was supported in part by the National Science Foundation with grants CCR-0073440, CCR-0083126, EIA-9971256, and CDA-9623632, the University of Wisconsin Graduate School, and donations from Compaq Computer Corporation, IBM, Intel Corporation, and Sun Microsystems. Milo Martin is supported by an IBM Graduate Fellowship. Daniel Sorin is supported by an Intel Graduate Fellowship.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[4] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[5] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[6] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A Multithreaded PowerPC Processor for Commercial Servers. *IBM Journal of Research and Development*, 44(6):885–898, Nov. 2000.

[7] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An Architectural Evaluation of Java TPC-W. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, pages 229–240, Jan. 2001.

[8] S. Chatterjee, C. Weaver, and T. Austin. Efficient Checker Processor Design. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 87–97, Dec. 2000.

[9] F. Corella, J. M. Stone, and C. M. Barton. A Formal Specification of the PowerPC Shared Memory Architecture. IBM Technical Report RC 18638, Jan. 4, 1993.

[10] D. E. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[11] K. Diefendorff. Power4 Focuses on Memory Bandwidth. *Microprocessor Report*, 13(13):1–8, Oct. 1999.

[12] Digital Equipment Corporation. *Compiler Writer's Guide for the Alpha 21264*, June 1999.

[13] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5):12–18, Sept/Oct 1997.

[14] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of Multithreaded Uniprocessors for Commercial Application Environments. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 203–212, May 1996.

[15] K. Flautner, R. Uhlig, and T. Mudge. Thread Level Parallelism and Interactive Performance of Desktop Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–138, Nov. 2000.

[16] K. Gharachorloo. Personal Communication, July 2001.

[17] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 355–364, Aug. 1991.

[18] IBM Corporation. *Book E: Enhanced PowerPC Architecture, version 0.91*, July 21, 2001.

[19] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual*, Jan. 1996.

[20] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual, Volume 2: IA-64 System Architecture, Revision 1.1*, July 2000.

[21] T. Keller, A. Maynard, R. Simpson, and P. Bohrer. SimOS-PPC Full System Simulator. http://www.cs.utexas.edu/users/cart/simOS.

[22] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, Mar/Apr 1999.

[23] A. Kumar. The HP PA-8000 RISC CPU. *IEEE Micro*, 17(2):27–32, Mar/Apr 1997.

[24] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[25] K. M. Lepak and M. H. Lipasti. On the Value Locality of Store Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 182–191, June 2000.

[26] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct. 1996.

[27] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 39–50, June 1998.

[28] J. A. Redstone, S. J. Eggers, and H. M. Levy. An Analysis of Operating System Behavior on a Simultaneously Multithreaded Architecture. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–256, Nov. 2000.

[29] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, Mar. 1992.

[30] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[31] Systems Performance Evaluation Cooperative. SPEC Benchmarks. http://www.spec.org.

[32] Transaction Processing Performance Council. TPC Benchmark W (Web Commerce), Standard Specification, Revision 1.43, Feb. 2001.

[33] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[34] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.