Journal of Instruction-Level Parallelism 6 (2004) 1-23 Submitted 10/03; published 4/04

Constraint Graph Analysis of Multithreaded Programs

Harold W. Cain CAIN@CS.WISC.EDU
Computer Sciences Dept., University of Wisconsin
1210 W. Dayton St., Madison, WI 53706

Mikko H. Lipasti MIKKO@ENGR.WISC.EDU
Dept. of Electrical and Computer Engineering, University of Wisconsin
1415 Engineering Dr., Madison, WI 53706

Ravi Nair NAIR@US.IBM.COM
IBM T.J. Watson Research Laboratory
P.O. Box 218, Yorktown Heights, NY 10598

Abstract

This paper presents a framework for analyzing the performance of multithreaded programs using a model called a constraint graph. We review previous constraint graph definitions for sequentially consistent systems, and extend these definitions for use in analyzing other memory consistency models. Using this framework, we present two constraint graph analysis case studies using several commercial and scientific workloads running on a full system simulator. The first case study illustrates how a constraint graph can be used to determine the necessary conditions for implementing a memory consistency model, rather than conservative sufficient conditions. Using this method, we classify coherence misses as either required or unnecessary. We determine that on average over 30% of all load instructions that suffer cache misses due to coherence activity are unnecessarily stalled because the original copy of the cache line could have been used without violating the memory consistency model. Based on this observation, we present a novel delayed consistency implementation that uses stale cache lines when possible. The second case study demonstrates the effects of memory consistency constraints on the fundamental limits of instruction level parallelism, compared to previous estimates that did not include multiprocessor constraints. Using this method we determine the differences in exploitable ILP across memory consistency models for processors that do not perform consistency-related speculation.

1. Introduction

The doubling of the number of transistors on a single chip each 18 months has had a profound impact on the performance, complexity, and cost of computer systems [41]. The expanding transistor budget has been used by architects to build increasingly complex and aggressive microprocessor cores, to augment memory systems with larger and deeper cache hierarchies, and to include multiple processors on a single chip. A side-effect of this transistor windfall is an ever-increasing level of design complexity burdening computer architects. Although Moore's law has reduced the per-unit cost of computer systems through increasing levels of integration, the one-time design cost of each computer system has increased due to this complexity. This increased cost is compounded by the relative performance loss caused by delayed designs: upon release, delayed processors may have already been surpassed in performance by the competition, or could have been shipped in a newer process technology had the delays been planned.

Considering this landscape, the need for tools that help designers understand and analyze computer systems should be evident. In this paper we present one such tool, a method of reasoning about multiprocessor systems using a graph-based representation of a multithreaded execution called a *constraint graph*. A constraint graph is a directed graph that models the constraints placed on instructions in a multithreaded execution by the memory consistency model, in addition to data dependences. This representation has been

used in the past to reason about the correctness of multiprocessor systems [13], [34], [39], [46], [50]. In this paper we show how it can be used to analyze the performance of multiprocessor systems. By analyzing the program's constraint graph, we hope to gain insight into the inefficiencies of current multiprocessor consistency protocols and use this knowledge to build future multiprocessors that remove these inefficiencies.

This paper focuses on two case studies of constraint graph analysis using the execution of several commercial and scientific workloads running on a full system simulator. The first study shows how the constraint graph can be used to determine the necessary conditions for implementing a memory consistency model, rather than overly conservative sufficient conditions. We use it to classify coherence misses as either required or unnecessary, and show that many load instructions which suffer cache misses due to coherence activity are unnecessarily stalled, because the stale copy of the cache line could have been used without violating the consistency model. Based on this observation, we motivate a method of applying invalidates to the cache, *selective invalidation*, that extends the lifetime of cache lines in invalidate based coherence protocols beyond that of previous delayed consistency proposals by selectively delaying some invalidations when they are unnecessary and applying other invalidations that are deemed necessary. This study concludes with the description of a preliminary hardware implementation that avoids coherence misses by using stale data when possible.

The second study demonstrates the use of a constraint graph to determine the effects of memory consistency on the fundamental limits of instruction level parallelism. Relative to previous estimates that did not include realistic consistency constraints, we find that the inclusion of consistency constraints significantly reduces available instruction-level parallelism. Using this method, we compare the differences of exploitable ILP for various memory consistency models, and quantify the maximum benefit in parallelism to be gained from more relaxed memory models by processors that do not perform consistency-related speculation. We find that exploitable ILP increases relatively meagerly when moving from sequential consistency to processor consistency (by 43%), compared to the move from sequential consistency to weak ordering (by 180%).

In summary, this paper makes the following contributions:

- We describe extensions to the constraint graph model for reasoning about memory models other than sequential consistency, and show how to use this model to analyze different aspects of the performance of multiprocessor systems
- Using the constraint graph, we demonstrate the potential to avoid coherence misses that are not required by the consistency model, and evaluate this potential across several popular consistency models. We show that on average 30% of the load misses caused by coherence activity are unnecessary. This study motivates future work in unnecessary coherence miss detection and removal.
- Using the constraint graph, we perform the first comparison of the fundamental limits of ILP for non-speculative implementations of several popular consistency models.

2. Related work

This work is indebted to the original authors of the constraint graph model, who used it to reason about the correctness of request and response reordering in sequentially consistent interconnection networks [34]. Prior to Landin et al.'s formalization, a similar model was used by Shasha and Snir to determine the necessary conditions for the insertion of memory barriers by parallelizing compil-

ers to ensure sequentially consistent executions [50]. The constraint graph has since been used to automatically verify sequentially consistent systems [13], [46] and reason about the correctness of value prediction in multiprocessor systems [39]. In this paper, we build upon this work by showing how the constraint graph model can be used to reason about performance in multiprocessor systems. Although previous work has defined the constraint graph for sequentially consistent systems, the vast majority of multiprocessors shipping today employ a relaxed memory model [25], [26], [40], [53], [58]. We show how previous constraint graph definitions can be augmented for reasoning about processor consistent and weakly ordered systems. These extensions are broadly applicable to reasoning about relaxed memory consistency models, whether it be for the purpose of verification or for performance analysis as discussed here.

There has been much prior work on other formalisms for reasoning about shared-memory consistency models. Dubois et al. established the theoretical correctness of delaying data writes until the next synchronization operation [15], and introduced the concept of memory operations being "performed" locally, performed with respect to other processors in the system, and performed globally. Collier presented a formal framework that divided memory operations into several sub-operations to model the non-atomicity of memory operations, and formulated a set of rules regarding the correct observable order of sub-operations allowed by different consistency models [11]. Adve incorporated the best features of Landin et al.'s model (acyclic graph) and Collier's framework (multiple write events) into a unified model that was used to perform a design space exploration of novel consistency models [1] (specifically Chapter 7). Chapter 4 from Gharachorloo's thesis describes a framework for specifying memory models that improves upon previous representations through the addition of a memory operation "initiation" event, which is used to model early store-to-load forwarding from a processor's write buffer [19]. The formal representation described here is most similar to Adve's representation, although it does not include the detailed modeling of non-atomic writes, in favor of simplicity.

In Section 5 we present a study that uses the constraint graph to classify coherence misses into two categories: required or unnecessary. This study is related to prior work in delayed consistency [14], [16], [17], [28] that attacked the false-sharing problem by delaying either the sending of all invalidates (sender-delayed protocols) or the application of all invalidates (receiver-delayed protocols) or both until a processor performs a synchronization operation. These proposals have demonstrated a substantial reduction in false-sharing related coherence misses in multiprocessor systems, but have unfortunately relied on the presence of properly-labeled synchronization operations[1]. Afek et al. and Brown describe delayed consistency algorithms similar to Dubois' in the context of update-based coherence protocols and invalidate-based protocols, respectively. These algorithms limit a processor's use of stale values to those cache blocks that are more recent than the most recent write by that processor [2][8].

The constraint graph characterization presented in this paper corroborates these previous results, measuring the frequency of accesses that may benefit from delayed consistency protocols. However, rather than using an all-or-nothing approach to delaying all invalidates, we illustrate how to use the constraint graph to perform *selective invalidation* to apply only those invalidates that are absolutely necessary to correctly implement the memory consistency model, thus maximizing the cache-resident lifetime of a memory location. Using this mechanism, one can reduce

1. With the exception of IA-64, no commercial instruction set architecture requires using special labeled operations when accessing synchronization variables. In most architectures, synchronization may be implemented using ordinary loads and stores (as opposed to atomic synchronization primitives or labeled acquire/release operations). Consequently, it is not advisable to delay such operations, limiting the practicality of previous delayed consistency mechanisms.

the number of coherence misses to truly shared data in addition to reducing the number of coherence misses to falsely shared data, in systems with or without properly-labeled synchronization operations. Based on the measurements presented in Section 5, we are optimistic that selective invalidation will prove to be a fruitful method of reducing coherence misses in future commercial shared-memory multiprocessors. Rechtschaffen and Ekanadham also describe a cache coherence protocol that allows a processor to modify a cache line while other processors continue to use a stale copy [48]. Their protocol ensures correctness by conservatively constraining certain processors from reading stale data or performing a write while stale copies exist. Using the constraint graph analysis described in Section 5, one can build an aggressive protocol that implements the necessary conditions of the consistency model, rather than the approximate conditions used by Rechtschaffen and Ekanadham.

In Section 6 we present a study of the limits of ILP in multithreaded workloads across several memory consistency models. There have been many studies evaluating uniprocessor performance using graph-based representations. Several of these studies measure the purely theoretical ILP limits of systems that are not constrained by any physical resources [9], [37], [45], while others include a wide range of resource constraints in the model [5], [32], [47], [57]. When evaluating any feature of an ISA, it is important to weigh both what is practically buildable and what is theoretically possible, because today's impracticalities may be practical in the future. We present the first study of the fundamental ILP limits across multiple memory consistency models, which complements recent comparisons of memory models using detailed simulation [20], [43], [60] by quantifying the theoretical performance differences.

3. Constraint graph definitions

A *constraint graph* [13] or an *access graph* [34] is a partially-ordered directed graph that models the execution of a multithreaded program. Nodes in a constraint graph represent dynamic instruction instances. Edges represent the ordering relations between these instructions, and are transitive in nature. The transitive closure of these orders on the constraint graph nodes dictate the sequence in which each operation must *appear* to execute in order to be considered correct. These ordering constraints are divided into two groups: dependence constraints and consistency constraints. The dependence constraints are uniform across architectures, while the consistency edge types vary depending upon the architecture's consistency model.

3.1. Dependence constraints

The dependence constraint category consists of the standard dependence graph edges defined by many other studies [5], [9], [30], [32], [37], [45], [47], [54], [57]. For a thorough explanation of these dependences, we refer the reader to the original sources, although we will briefly summarize them here.

- **Read-after-write true data dependencies (RAW)**: two instruction nodes are connected by RAW dependence edges if one instruction creates a value (in register or memory) that is used by the other. This edge places an order on operations ensuring that values are created before they are read.
- **Write-after-read storage dependencies (WAR)**: an instruction node that writes a memory or register element is preceded through a storage dependency by any instruction that has previously read that element. This edge models a finite number of storage

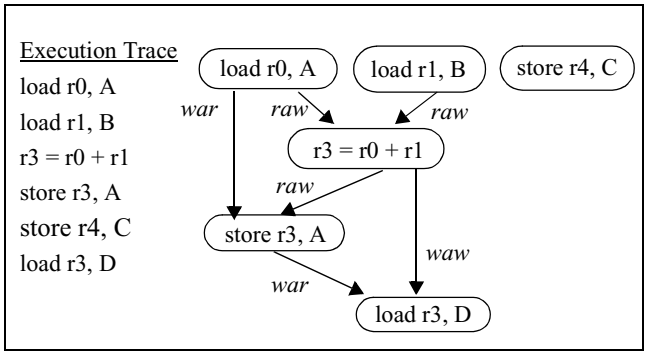

Figure 1: Uniprocessor constraint graph

locations by forcing writes to wait until all previous readers have read the previous value at a location.

- **Write-after-write storage dependencies (WAW)**: an instruction node that writes a memory or register location is preceded through a storage dependency by any instruction that has previously written that location. This edge, like the WAR edge, models a finite number of storage locations, and forces writes to wait until all previous writes to the same storage element are complete.

Figure 1 shows an example of a constraint graph consisting of these true dependence and storage dependence edges. In this example, there are three instructions that may execute immediately because they have no incoming dependence edges. The add instruction will execute when its inputs, the two load instructions, are ready. The store to address A must wait until the load of A completes due to a write-after-read storage dependence on memory location A. The store to A must also be delayed until after the add completes due to a true data dependence. The load from address D is storage-dependent on the add instruction and the store to A.

As shown in Figure 1, certain constraints can limit the instruction-level parallelism in a program's execution. Of course, each of these constraints can be overcome using various microarchitectural features. WAR and WAW edges can be removed using register renaming and memory renaming [42], [56], and RAW edges can be removed using value prediction [38]. We can use constraint graphs to determine the maximum ILP in a program execution, and evaluate the potential performance advantage to be gained by different architectural mechanisms. Using the example in Figure 1, we can see that adding register renaming could maximally reduce execution time from four cycles to three cycles, while memory renaming offers no performance advantage for this particular contrived execution.

Previous studies have also included control dependence and resource constraint edges to more accurately measure ILP in uniprocessor systems, which is orthogonal to the multiprocessor issues that are the focus of this paper. To prevent these uniprocessor issues from obscuring our focus, we omit control dependence and resource constraint edges. In Section 6, we show that despite using such an aggressive machine model, parallelism is still severely limited by the constraints of the consistency model.

3.2. Consistency constraints

Note that the store to address C in Figure 1 can execute immediately because it is not dependent upon any other operations. Previous studies have made this assumption, which is not always correct due to potential interactions with coherent DMA devices in uniprocessor systems and with

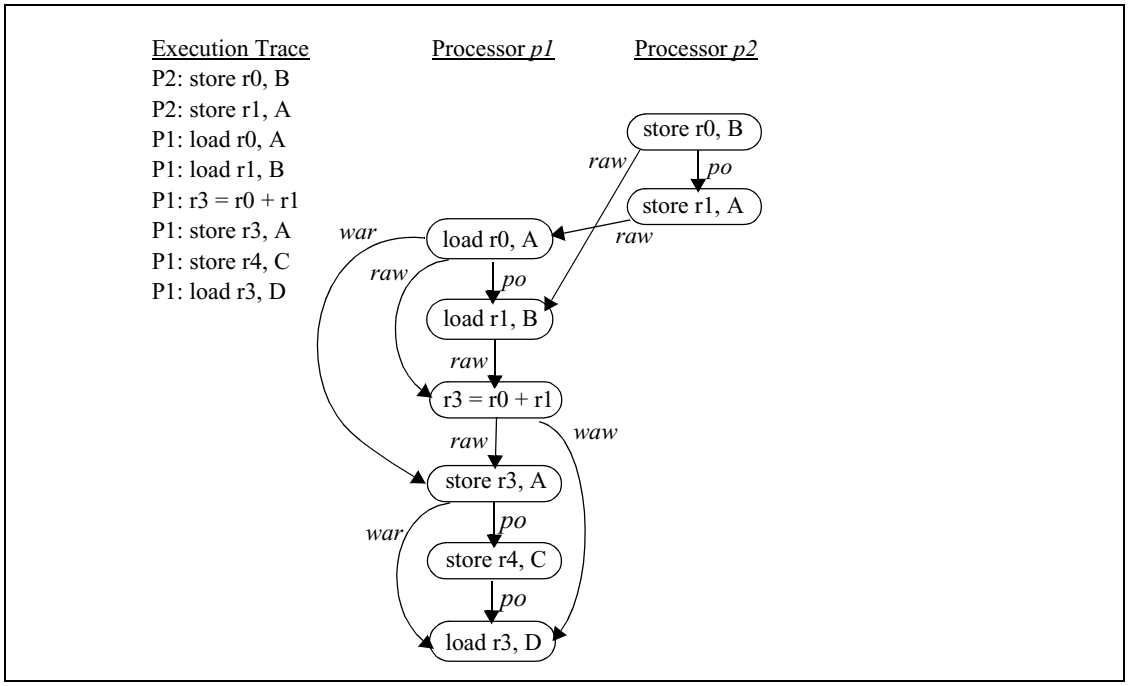

Figure 2: Constraint graph including sequential consistency edges. (Redundant program order edges removed for clarity.)

other processors in multiprocessor systems. In order to maintain correctness, processors must obey the semantics of the memory consistency model in addition to these true dependence and storage dependence edges.

Consistency model edge types vary depending on the supported consistency model. Previous constraint graph definitions have focused solely on sequential consistency, where a single consistency edge type that represents program order (PO) is used. Program order places an order on all of the memory operations executed by a single processor. We use the following criterion to determine the correctness of the execution: if the graph is acyclic, the execution is correct. As shown by Landin et al., a directed cycle implies that the instructions cannot be placed in a total order, which is a violation of sequential consistency [34]. To illustrate this principle, a revised version of the example in Figure 1 that has been augmented with consistency edges for sequential consistency is shown in Figure 2. In this example, each processor must respect program order in addition to the usual dependence edges. For instance, if processor p1 executes the load B before any other operation, there would be a WAR edge from *p1* to processor *p2* instead of a RAW edge to *p1*, and a cycle would be formed among the operations executed by *p1* and *p2*.

In the next two subsections, we specify modifications to the constraint graph model necessary for reasoning about processor consistent systems and weakly ordered systems. These changes simply add or remove certain edge types from the graph to enforce the different ordering requirements found in relaxed memory models, while preserving the property that if the constraint graph is acyclic, then the execution corresponding to the graph is correct.

3.2.1. Processor Consistency Edge Types

Several relaxed memory models have been proposed that remove many of the constraints associated with sequential consistency. Processor consistency (PC), variations of which are used in

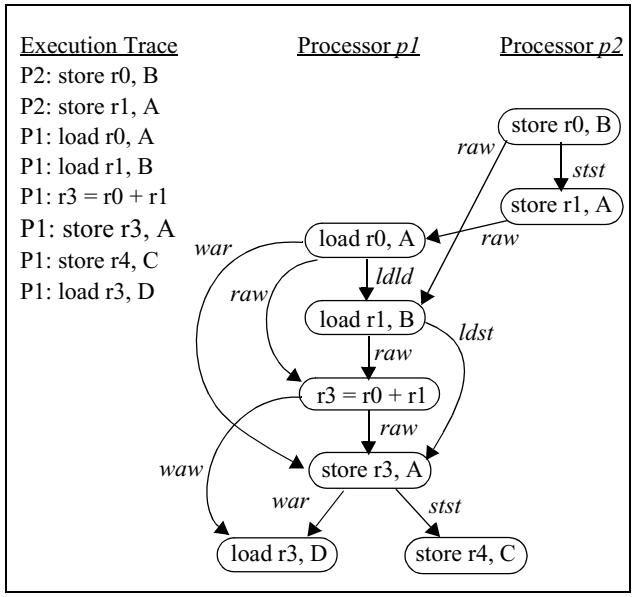

Figure 3: Processor consistent constraint graph

the IA-32, IBM 390, and SPARC total store order memory models[1], relax the program order between stores and subsequent loads while leaving all other ordering requirements the same as sequential consistency [23]. Because of this difference, the constraint graph for processor-consistent systems contains a different set of consistency edge types than those used in sequentially consistent systems. The program order edge type that orders all operations in a sequentially consistent system is removed because load instructions are allowed to be reordered with respect to previous store instructions. The following edge types represent the necessary orders for implementing processor consistency.

- **Load to Load Order (LDLD)**: All load instructions executed by the same processor are ordered with respect to one another by load to load order.
- **Store to Store Order (STST)**: All store instructions executed by the same processor are ordered with respect to one another by store to store order.
- **Load to Store Order (LDST)**: Each store instruction executed by a processor is preceded by all of the load instructions that have previously occurred in that processor's sequential instruction stream.

A revision of the previous example for processor consistent systems is found in Figure 3, where the final load and store instructions executed by *p1* are no longer ordered with respect to one another. As in Figure 2, redundant consistency edges have been removed for clarity (e.g., *p1*'s load r3, D is transitively ordered after its load r0, A).

3.2.2. Weak ordering edge types

Weakly ordered systems (WO) further relax the memory consistency model by removing the three constraints imposed by processor consistency, and enforcing order explicitly using special memory barrier instructions [15]. The semantics of such instructions specify that all memory operations that precede the memory barrier must appear to have executed before any of the instructions

1. For the purposes of this work, we do not distinguish between the variations of processor consistency, although their subtle differences could be explored given further constraint graph refinements.

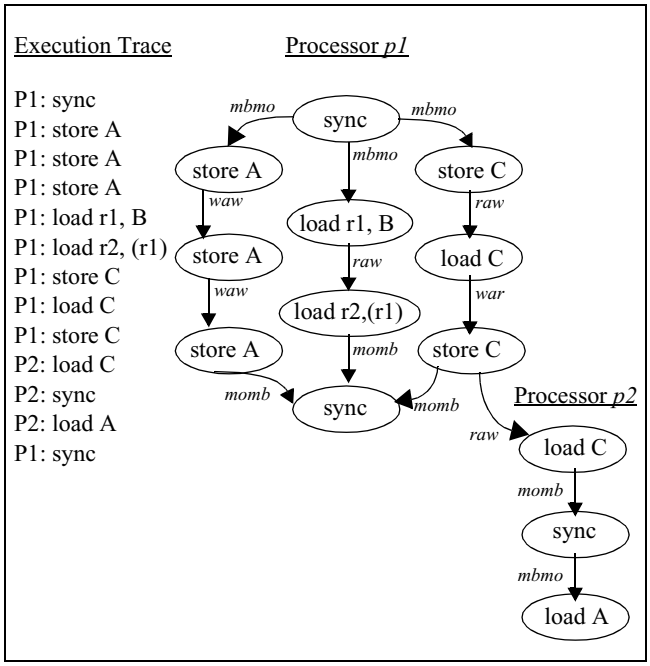

Figure 4: Weakly ordered constraint graph

that follow the memory barrier. Examples of commercially available weakly ordered systems include PowerPC, SPARC RMO, and Alpha multiprocessors. We define the following constraint graph edge types for use in weakly ordered systems:

- **Memory Barrier to Memory Operation Order (MBMO)**: All memory operations that are executed by any single processor are preceded through a MBMO edge by that processor's most recently executed memory barrier.
- **Memory Operation to Memory Barrier Order (MOMB)**: All memory barrier operations that are executed by any single processor are preceded through MOMB edges by all of the memory operations executed by that processor since the previous memory barrier.

Figure 4 shows a constraint graph for a PowerPC execution (different from the previous examples), whose memory barrier operations consist of *sync* instructions. Once again, redundant consistency edges have been removed for clarity. In this execution, processor *p1* executes a series of instructions between two sync operations. The first three instructions all write the memory location A, creating a series of WAW hazards, and are thus connected by edges in the constraint graph. The next two instructions are a load followed by a data dependent load instruction, which are also ordered and connected by a RAW edge in the constraint graph. The next three instructions are a store to location C, followed by a load to C, followed by a store to C. These three instructions are all ordered with respect to each another by RAW or WAR hazards, and are consequently ordered in the constraint graph.

The resulting constraint graph contains three separate independent strands of execution that are not ordered with respect to one another. Within each strand, the operations must appear to execute in a certain order, but across strands, no ordering guarantees are made. Each strand must appear to execute after the first sync operation and before the second sync operation. When processor *p2* executes its load instruction to C, receiving the data written by *p1*, the load instruction is ordered after the entire right-most strand of *p1*'s execution, but is not ordered with respect to the

Table 1: Workload Descriptions

Application	Total non-idle instr count	Coherence Misses Per 10,000 Instr (4 byte block)	Coherence Misses Per 10,000 Instr (128 byte block)	Membars per 10,000 instr	Description
barnes	248,800,151	8.9	2.7	2.2	SPLASH-2 N-body simulation (8K particles)
radiosity	816,280,634	10.2	2.2	2.7	SPLASH-2 Light interaction application (-room -ae 5000.0 -en 0.050 -bf 0.10)
SPECjbb2000	953,399,434	86.4	26.1	37.5	Server-side Java benchmark (IBM jdk 1.1.8 w/ JIT, 400 transactions).
TPC-H	679,717,253	108.4	43.4	32.7	Transaction Processing Council's Decision Support Benchmark (IBM DB2 v. 6.1 running query 12 on 512 MB database)

remainder of *p1*'s operations. Consequently, the load A executed by *p2* does not necessarily need to return a value stored by *p1*. It could correctly return any of the values, or the value that existed prior to any of *p1*'s store operations. Note that if there were a WAR edge from *p2*'s load A to any of *p1*'s stores, the constraint graph would remain acyclic. By visualizing this example using the constraint graph, the amount of additional flexibility allowed by a weaker model should be clear. In Section 5, we will describe mechanisms that take advantage of this flexibility.

Because we use a PowerPC-based weakly-ordered infrastructure for this work, we are able to study any memory model that is as strict or stricter than weak ordering, such as sequential consistency and processor consistency. Constraint graphs for models more relaxed than weak ordering can be described using further refinements to the constraint graph edge types discussed for weakly ordered systems. However, we leave such models out of our discussion because their results will not appear in this paper.

4 . Methodology

We use SimOS-PPC [29], a PowerPC version of the original execution-driven full system simulator [49], to generate an instruction trace containing user and system level instructions of a variety of commercial and scientific parallel applications. Each trace is generated by a 16 processor machine configuration with fixed 1 CPI functional processor model. The traces are fed to a constraint graph analysis tool specifically developed for this work, which dynamically builds a constraint graph and performs analysis as described in subsequent sections.

A summary of the applications used and their setup/tuning parameters is found in Table 1. Each of the SPLASH applications was compiled with the IBM VisualAge Professional C/C++ compiler v5, with optimization flags qfloat=maf -O3 -qipa=level=2. The commercial applications use precompiled binaries distributed by the respective company. Each of these applications was written for a weakly ordered memory model. Consequently, they may safely be executed on weakly ordered, processor consistent, and sequentially consistent hardware. Table 1 also lists the number of coherence misses in an invalidate-based coherence protocol for two cache line sizes: 4 bytes and 128 bytes. Although the number of coherence misses is fairly small for the two scientific applications, these misses have a significant performance impact on commercial applications, as others have observed [6], [31].

The constraint graph constructed by our analysis tool includes all of the edges described in Section 3, excluding WAR and WAW edges for register dependences. Register renaming is a well-

understood and widely used mechanism that removes most WAR and WAW register dependences in current systems, so by removing these edges our constraint graph is more faithful to real executions. However, memory renaming mechanisms have not gained widespread use in processors. Consequently, our tool includes edges for WAW and WAR memory dependences. Unless specified otherwise, memory dependences among instructions are tracked on a four-byte granularity.

5. Identifying Unnecessary Coherence Misses Using a Constraint Graph

There is a close relationship between the performance of shared-memory multiprocessors and the fraction of memory operations that can be satisfied from local cache hierarchies. Due to the growing disparity between system bus frequencies, DRAM latency, and processor core clock frequencies, cache misses are the dominant source of processor stalls for many systems and applications. Inter-processor communication through invalidation-based coherence protocols is a dominant source of cache misses in shared memory multiprocessors. When one processor writes a memory location, all copies of that location must be removed from the caches of other processors. When those processors subsequently access the location, their accesses incur coherence misses. These delays are exacerbated in multiprocessor systems where cache misses must navigate multiple levels of interconnect before they are serviced. It has been projected that as cache hierarchies grow larger, other sources of cache misses (conflict and capacity) will be reduced, resulting in an even greater proportion of coherence misses [31].

In this section, we use the constraint graph representation to evaluate the opportunity for eliminating delays associated with coherence-related cache misses. Using the constraint graph, we can detect the conditions such that RAW coherence misses may be safely avoided by simply using stale data from the invalidated cache line, rather than stalling a load while satisfying the cache miss. We focus on RAW misses as a target for optimization because the latencies associated with WAW and WAR coherence misses can usually be overlapped using write buffers. We compare the potential for avoiding RAW coherence misses across sequential consistency, processor consistency, and weak ordering, and find that there exists a large opportunity in each model. We conclude this section with the outline of a hardware implementation that takes advantage of this opportunity by using stale data whenever possible.

5.1. Unnecessary miss identification

By examining the source and destination processors of edges in the constraint graph, one can identify instances of potential cache-to-cache transfers. Each RAW, WAR, and WAW edge whose two endpoint instructions were executed by different processors equates to a single miss or upgrade between processors in an invalidation-based coherence protocol, neglecting castouts. Inter-processor RAW edges correspond to a read miss that is satisfied by a dirty copy of the memory location residing in another processor's cache. Similarly, inter-processor WAW edges correspond to write misses satisfied by remote processors. Interprocessor WAR edges correspond to writes that result in either a miss or an upgrade message between processors. We filter out the per-processor compulsory misses from this set of references, yielding the set of coherence misses. A count of these misses for each workload is shown in Table 1. Because dependence edge types are uniform across consistency models, the number of interprocessor edges is also uniform, meaning that the number of observed coherence misses is constant across consistency model.

Once a coherence-related load miss has been identified, we determine whether or not that miss

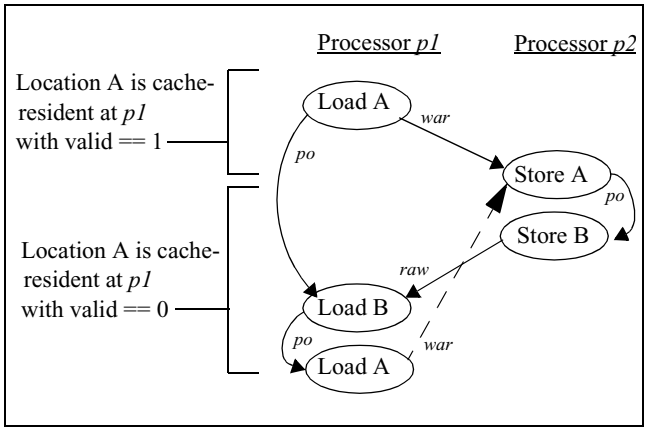

Figure 5: A necessary coherence miss (time progresses from top to bottom)

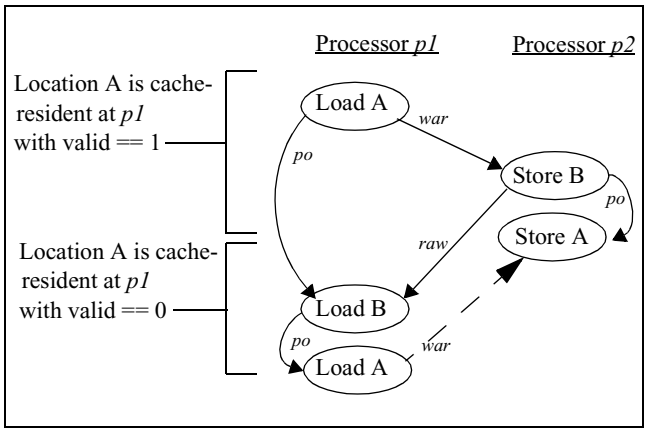

Figure 6: An unnecessary coherence miss

is avoidable based on the following criterion: a RAW edge *e* emanating from writer node *w* and connecting to reader node *r* is necessary, if there exists a directed path in the constraint graph from *w* to *r* that does not include edge *e*. This observation follows from Landin's proof that if a constraint graph is acyclic then the execution corresponding to that constraint graph is correct. If *e* is deemed avoidable, then we are essentially transforming the RAW edge from *w* to *r* into a WAR edge from *r* to *w*. If there is already a directed path from *w* to *r*, this new WAR edge would create a cycle in the graph, and would thus be incorrect. If there is no directed path from *w* to *r*, then the WAR edge cannot create a cycle, and the coherence miss is unnecessary.

An illustration of a necessary coherence miss under sequential consistency is found in Figure 5: In this example, processor *p1* is about to perform its second load to cache block A, but the cache line containing A has been invalidated. In order to determine whether or not *p1* can avoid this cache miss, we look at the constraint graph node that would provide the value to the miss (the "W" in RAW), in this case processor *p2*'s store to A. We would like to use the stale value from the cache, thus creating a WAR edge from *p1*'s load A to *p2*'s store A (indicated by the dotted arrow). However, if there already exists a directed path from this store node to the load miss node, then we know that this miss is necessary, because the load is already transitively dependent upon the store. As we can see in the figure, a directed path already exists from *p2*'s store A to *p1*'s load A through a RAW dependence on memory location B. If this path did not exist, *p1* could safely use the stale

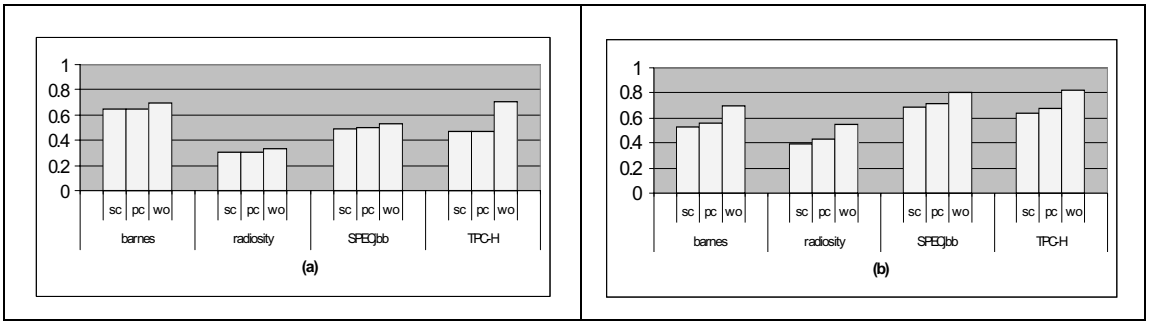

Figure 7: Fraction of coherence load misses avoidable for sequential consistency (sc), processor consistency (pc), and weak ordering (wo). (a) 4 byte coherence unit, and (b) 128 byte coherence unit

value of A. However, because it does exist *p1* must not use the stale value, thus the dotted WAR edge must be transformed to a RAW edge, eliminating the cycle.

Figure 6 illustrates an almost identical code segment, however in this case the load miss to location A by *p1* is avoidable. The miss is avoidable because we can create a legal schedule of operations such that the load miss will return the old value of A. In this example, *p2* performs the stores to memory location A and B in reverse order. Consequently, at the time of *p1*'s load miss, there is not already a directed path from *p2*'s store of A to *p1*'s load of A, therefore the WAR edge caused by using the stale value does not create a cycle, and this coherence miss is avoidable.

Speculative mechanisms might guess that it is safe for *p1* to use the stale data in A because the store to A by *p2* may have been a silent store or may have been to a different word within the same cache line. These mechanisms require a verification step to ensure that *p1* did indeed load the correct value. Given the long latencies associated with fetching data from another processor's cache, this verification operation will most likely stall the processor. However, using the constraint graph, we can detect cases where it is safe to use the stale value, without the need for verification. In the next sub-section, we quantify the frequency of this scenario.

5.2. Results

In this section, we present two sets of results using different address granularities for tracking interprocessor dependences: 4 bytes and 128 bytes. The 4-byte granularity is used to capture the sharing inherent to the application, ignoring false-sharing effects, while the 128-byte granularity captures the interprocessor dependence relationships that would be observed in an invalidate-based cache-coherence protocol using a 128-byte cache block size. Each set of results was collected by building the constraint graph that would have existed for the given trace under each memory model. For instance, we perform six experiments using each of the applications, building three constraint graphs with 4-byte coherence granularities (one for SC, one for PC, one for WO), and three constraint graphs using 128-byte coherence granularities. As the trace is processed and RAW coherence misses are identified, we classify each miss using the algorithm described in Section 5.1 as either avoidable or necessary subject to the given memory model's constraint graph. Figure 7 shows the fraction of these RAW coherence misses that are avoidable for each of the three consistency models. There is a significant fraction of cache misses avoidable for all of the workloads. From Table 1, 41% and 50% of coherence misses are RAW misses for 4-byte and 128-byte coherence granularities, respectively. For 4-byte granularities, the percentage of RAW coherence misses that are avoidable averages 51% across all applications and consistency models, and is

highest for TPC-H under weak ordering at 71%. The scientific application *barnes* exhibits the most potential for coherence miss avoidance, in which 66% of RAW coherence misses are avoidable averaged across consistency models. For all applications, there is a very small benefit moving from sequential consistency to processor consistency, and significant advantage moving to a weakly ordered model.

For 128-byte inter-processor dependence granularities, there are many fewer coherence misses, as reported in Table 1, so the two graphs are not comparable to one another in terms of absolute numbers. (Coherence misses, like other types of misses, exhibit spatial and temporal locality.) At this granularity, we find that there is even greater opportunity for using stale data, 62% on average across all consistency models and applications, ranging from 38% (*radiosity*, *sc*) to 82% (*TPC-H, wo*). Once again, there is a significant gain in opportunity to avoid coherence misses when moving from the stricter models to weak ordering.

This data is very encouraging, because it indicates that most coherence misses caused by loads are avoidable. However, using stale values may not always be the correct decision. There are certain scenarios where using stale values might cause performance and correctness problems, for example a lock variable used in an attempted lock acquisition. If the lock is initially held by another processor, the processor performing the lock acquire will continue to test the variable until it is released. If this release is never observed by the acquiring processor, it could test the lock forever, resulting in deadlock. Consequently, one should not use stale values when referencing synchronization variables.

Figure 8 shows the data from Figure 7(b), with each bar broken down into three components: misses that are definitely synchronization, misses that are possibly synchronization, and misses that are not synchronization. We categorize an avoidable miss as definite synchronization if the load miss is caused by the PowerPC *lwarx* and *ldarx* synchronization instructions (which have load-linked semantics). Unfortunately, this method of categorization does not encompass all possible synchronization, because ordinary loads are often used to implement synchronization (e.g., the test+test&set construct). Consequently, we further classify avoidable load misses as possible synchronization if they read a memory location that has *ever* been touched by a synchronization instruction (*lwarx, ldarx, stwcx, stdcx*), throughout the lifetime of the program. This "possible" synchronization is conservative because it may classify a load as synchronization when it is actually to data (e.g. if a particular memory location were used once as a lock, but subsequently reallocated as data due to dynamic memory allocation). This classification can therefore be thought of as an upper bound on synchronization in the program, with the actual amount of synchronization more likely lying somewhere in the middle. If we ignore those avoidable misses that may be synchronization, we can see from Figure 8 that between 18% and 51% of all RAW coherence misses are still avoidable, on average 30%. Even when using a conservative estimate of synchronization, there remains significant opportunity to exploit this phenomenon.

In the next subsection, we outline one such hardware mechanism that can detect and eliminate this class of avoidable coherence misses.

5.3. Implementation

In this section, we present a delayed consistency implementation that distinguishes between avoidable and necessary misses by tracking the causal dependencies among processors using a system of vector clocks. There have been prior efforts that exploit the difference between physical time and logical time to extend the useful lifetime of cached data. Each of these proposals has inferred the correctness of using stale data based on partial knowledge of the dependences among

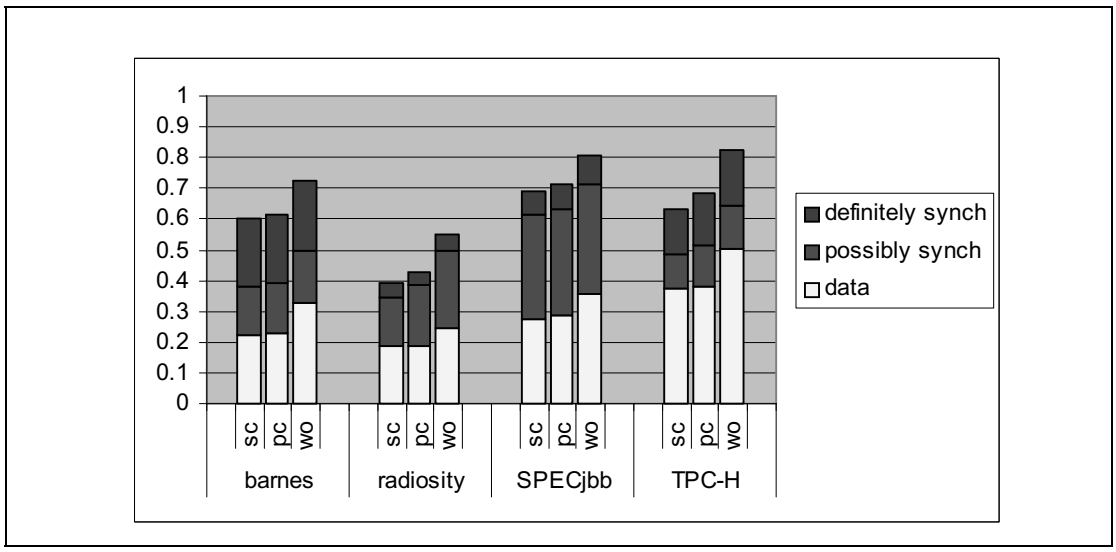

Figure 8: Fraction of RAW coherence misses avoidable at 128-byte coherence granularity, classified by type

processors. Dubois et al. observed that the invalidation of a cache block could be delayed until the subsequent cache miss and still maintain sequential consistency [15]. This observation led to an implementation that buffered invalidations until specially marked synchronization operations were performed, in order to reduce the program's miss rate [17]. Lebeck and Wood also took advantage of this observation by marking certain blocks in a directory-based coherence protocol as "tear-off blocks", which would be automatically invalidated by the cache controller upon the next miss (under sequential consistency) or the next synchronizing operation (under weak ordering) [36]. Consequently, the tear-off blocks would not need to be tracked by the directory controller, avoiding the transmission of invalidation messages when the block was subsequently written by another processor.

There exists a spectrum of implementations that are able to take advantage of stale cache lines, ranging from implementations with low hardware cost that can use stale lines during a short window (Dubois et al., Lebeck and Wood) to those with a higher cost and prolonged window. Rechtschaffen and Ekanadham's proposal falls somewhere in the middle. Rather than inferring the correctness of reading a stale line based solely on a lack of recent communication with other processors, their protocol tracks the lines shared among processors within a limited window of time, during which a cycle in the constraint graph can be identified and prevented. Once this window has expired, all stale blocks in the system are invalidated.

Our implementation is one step further in this spectrum, extending the window that stale cache lines may be used, but also requiring more hardware in order to precisely track inter-processor dependences. Our goal is to reduce the average latency of shared memory operations. We assume a weakly-ordered directory-based shared-memory multiprocessor, using unordered address and data interconnects and a SGI Origin-like coherence protocol in which write misses/upgrades are explicitly acknowledged by current sharers [35]. Our implementation allows a processor to non-speculatively read from a stale cache line in certain circumstances. Writes to a stale cache line are not allowed. This is a simple example of how one can implement a delayed consistency protocol without relying on properly labeled synchronization operations. As such, it is not optimal in terms of necessary implementation overheads, however we discuss a few potential improvements after

Processor p1	Processor p2
load A	store B
membar	membar
load B	store A

Figure 9: If p1's load A returns the value written by p2, then p1's load B must return the value written by p2.

the initial presentation.

In order to ensure correct operation, any delayed consistency mechanism must satisfy the following three requirements: 1) Ensuring that writes become visible to other processors in the correct order 2) forcing the eventual visibility of writes 3) Ensuring the expedient visibility of time-critical writes. In the next three subsections, we describe a solution to each of the three problems.

5.3.1. Ensuring the causal visibility of writes

Figure 9 shows a code segment where relaxing the causal visibility of writes would result in an incorrect execution. If processor p1's first load returns the value from processor p2's write, then p1's second load must observe processor p2's write to B. To ensure the correct execution of the code segment, our delayed consistency implementation employs a system of vector clocks to track the causal relationships between processors, as in the lazy release consistency protocol [28] and causal memory [4]. Vector clocks are a logical time base that preserve the partial ordering of events in a distributed system. In a system composed of *n* processors, a vector clock is an *n*-entry vector of integers, where each entry represents the logical time at processor *n*. The lazy release consistency protocol uses vector clocks to minimize the number of messages exchanged between processors by communicating only those writes that *must* be observed by a processor performing an acquire operation. Our implementation similarly uses vector clocks to track the causal dependences among processors, however we use them to infer when a processor may safely temporarily ignore the writes of other processors, using stale data until the new data can be obtained, thus overlapping the miss latency with useful computation.

In a weakly ordered system, a processor's execution can be divided into a series of intervals, where each interval is demarcated by the execution of a memory barrier operation at that processor. A vector clock V_p is maintained at each processor p, which tracks the current state of p's execution by incrementing $V_p[p]$ when p executes a memory barrier instruction. The other entries in V_p represent the logical time at which other processors' executions have been observed by p through the reading or writing of shared memory. When p performs a load or store operation to shared memory, a vector clock associated with that cache block-sized chunk of shared memory V_m is updated using V_p, such that $V_m[i] = max(V_m[i], V_p[i]), \forall i \in P$. The entries of V_p are also updated similarly $V_p[i] = max(V_m[i], V_p[i]), \forall i \in P$. If a processor's write is a cache miss, the processor augments its invalidate request with its own vector timestamp. When other processors that have a cache-resident copy of the cache line receive the invalidate message, their cache controller marks the cache line as stale, and stores the incoming vector timestamp with the cache line.

Should a processor attempt to read from a stale cache line, its cache controller must evaluate the correctness of using the stale version. Given a processor p performing a load operation from cache line m, if $V_p[i] < V_m[i], \forall i \in P$, then p has not observed any operation more recent than the

	Processor *p1*	**Processor *p2***	
V_{p1}			V_{p2}
[1, 0]			[0, 1]
[1, 2]	**ld r1 ⟵ [A]**	**st r1 ⟶ [B]**	[0, 1]
[2, 2]	**membar**	**membar**	[0, 2]
[2, 2]	**ld r2 ⟵ [B]**	**st r2 ⟶ [A]**	[0, 2]

Figure 10: Causal dependency recognized using vector clocks. Because *p1* already depends on *p2*'s timestamp 2 when executing the ld B, *p1* can recognize that it is not safe to use a stale copy of B with timestamp [0,0], since B's current timestamp is[0, 1], as updated by *p2*.

most recent operation to *m*, therefore it is safe to use the stale copy of *m*. As shown in Figure 10, if processor *p1* reads the value for A written by *p2*, it inherits the vector timestamp associated with A after *p2*'s write [0, 2]. In this case, assume that *p2*'s store to B was a cache miss, and *p1*'s cache contained a copy of the line. When *p1* executes its load B, if it tries to read from the stale line, it compares its timestamp vector [2,2] to the timestamp vector associated with the stale line [0, 1]. Because 2 > 1, *p1* will know that the stale data is no longer valid, and it must refetch the data from *p2*.

Vector clocks have been used extensively in the distributed systems research community, for example to ensure liveness and fairness in distributed algorithms [52], to characterize concurrency in parallel applications [10], to implement distributed debuggers [18], and to implement software-based distributed shared memory systems [4][28]. There has been no previous work using vector clocks to efficiently implement common consistency models (i.e. sequential consistency and other commercially used models) or in hardware-based implementations. The vector clock logical time system is closely related to Lamport's scalar logical time base [33], which has been used by others to reason about and verify consistency models [12],[44],[55]. We prefer the vector clock system because it preserves the partial ordering of operations among processors, rather than enforcing an unnecessary total order, allowing more freedom to delay/reorder operations.

5.3.2. Enforcing the eventual visibility of writes

To ensure an acyclic constraint graph and hence correct execution, we must take special care to track inter-processor WAR edges. Because our protocol preserves the single-writer property of the coherence protocol, it is fairly simple to update the vector clocks properly to reflect inter-processor RAW and inter-processor WAW edges, because there is a single source for the data block: the writer, and that writer loses its write privilege at the time that the RAW or WAW edge appears. In the case of WAR edges, however, load instructions may continue to read a stale value in a delayed consistency protocol, and it is therefore unknown at the time of the write which load operations precede the write in the constraint graph via WAR edges. If a processor continues to use a stale value for an arbitrarily long time, the delayed write must be ordered after an arbitrary number of instructions.

To show a concrete example of the problem, we refer to Dekker's algorithm in Figure 11(a). At the time that *p1* and *p2* perform their stores, the processors are at logical time [1, 0] and [0,1] respectively. Assuming both stores are cache misses, their invalidates will carry these vector

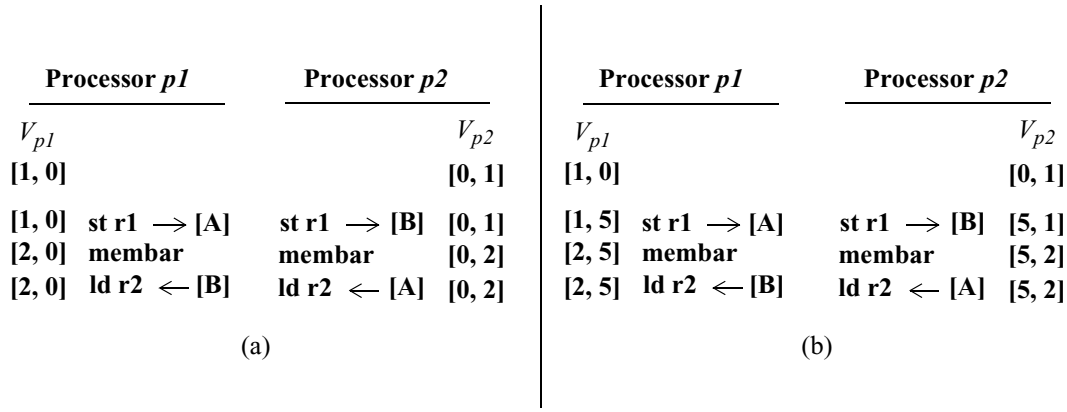

Figure 11: Dekker's algorithm: Each operation is labeled with the value of the processor's vector clock after the execution of that instruction. In example (a), *p1* and *p2* can both incorrectly read the stale value. In example (b), the store instructions inherit a timestamp indicating the stale data's time of expiration.

timestamps, and will deposit them with the stale line in the other processor's cache. Because the load instructions have not yet occurred, it is not possible for each store instruction to inherit a vector timestamp from the load instruction through the WAR edges that will form when the loads read the stale value. Consequently, when each processor performs its load, it incorrectly thinks that it is safe to read the stale value.

To solve this problem, any processor that expects to use a stale cache line returns as part of its invalidate acknowledgment a vector timestamp indicating the maximum timestamp at which it will use the stale data. This timestamp can be thought of as an "expiration date" for the stale data. In the example shown above in Figure 11(b), each processor responds to the other processor with its current timestamp (1) plus 4 (= 5), indicating that it will continue to use the stale value until it has executed four more sync instructions. Consequently, when a processor attempts to perform a load from the stale value, it has already inherited a newer vector timestamp, so the stale value may no longer be used, thus ensuring a correct execution.

In this example, we arbitrarily chose the value 4 when constructing the expiration date. This constant must be chosen carefully, because it determines the trade-off between maximizing the period of time that a stale cache line can be used, and minimizing the period of time that the writing processor must artificially causally depend on the processor using the stale value. If the chosen value is too low, then the stale cache line "expires" before it is ever used, yielding no performance benefit. If the chosen value is too high, then it will be a long time before the processor that performed the write can use data that is stale with respect to the other processor, because it is forced to causally follow the other processor for an extended period of time.

5.3.3. Ensuring the immediate visibility of time-critical writes

The above mechanism does not guarantee that a particular write will ever be observed by another processor if that processor does not execute a memory barrier operation. Obviously, this is problematic in the presence of synchronization operations, which should not be delayed to ensure forward-progress. We choose the simple solution of issuing a hardware initiated prefetch operation at the time that a stale value is first read. Reads to the stale cache line are able to complete while the prefetch is in flight, however the new value will eventually return. Figure 12 illustrates the relative timing of the dynamic sequence of instructions that are part of a spin lock test loop, in both a

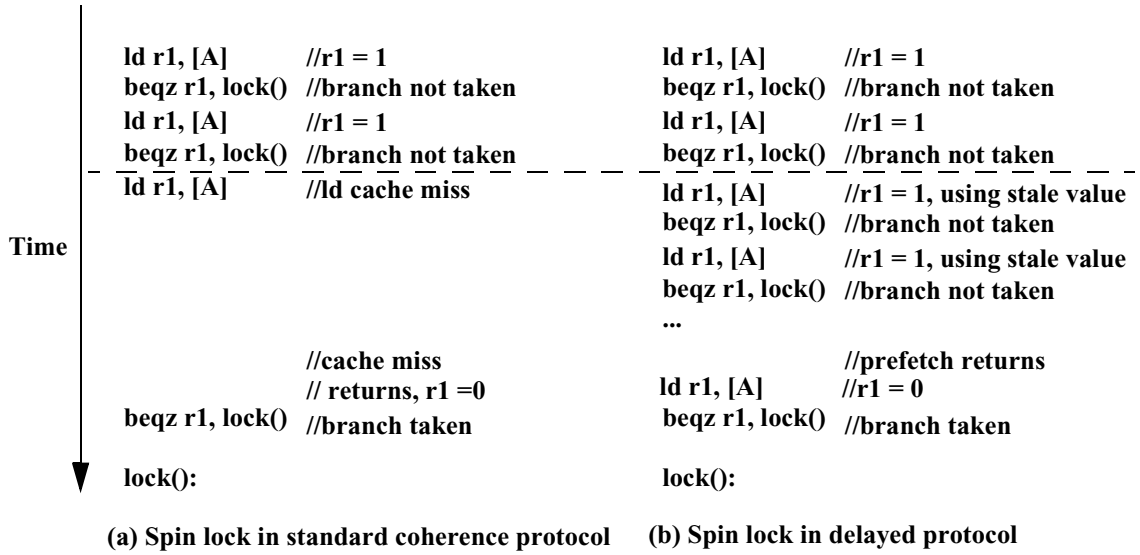

Figure 12: Dynamic sequence of instructions for a spin lock test loop in (a) standard coherence protocol and (b) delayed consistency implementation. The dotted line signifies the occurrence of the lock release at a different processor. The lock release becomes visible at each processor at the same time. In the delayed case, the processor spins on the stale value for a few more iterations.

standard coherence protocol and this delayed consistency implementation. Because the processor in Figure 12(a) suffers a cache miss to address A when it is invalidated, its load will stall the processor. In the delayed consistency mechanism, instead of stalling on the cache miss the processor continues to spin using the stale value, as shown in Figure 12(b). Nothing is gained by the extra spinning, but by issuing the prefetch at the first touch of the stale value, we guarantee that the synchronization performance of the delayed consistency implementation will match that of a traditional implementation. This solution sacrifices the potential bandwidth advantages of the delayed consistency implementation to guarantee the observance of time-critical writes.

5.3.4. Potential Improvements

Obviously, this initial implementation leaves much room for improvement in terms of the overheads associated with the maintenance of vector clocks, because it requires a vector clock to be associated with each cache line in a processor's cache. These storage requirements can be largely reduced by only maintaining vector clocks for cache lines that are a) shared among processors, and b) have been recently written. Because the window during which operations are delayed is fixed and known, it is only necessary to maintain the vector clock for writes during that window. All cache lines without an attached vector clock are assumed to be older than those in the window. Through careful selection of those blocks to treat as stale, we believe that vector clock storage and bandwidth overheads can be minimized. It may also be possible to reduce the required width of each vector clock using mechanisms similar to those that have been previously proposed to minimize the size of directory entries [3], [24].

Prior work on delayed consistency protocols used an all-or-nothing approach to the delaying of writes; however, using the constraint graph it is clear when it is safe to delay some writes while

applying others. Through this additional level of flexibility we identify an opportunity for both increasing average cache block lifetime as well as reducing buffer space requirements for storing delayed invalidation messages.

6. Using the constraint graph to measure the effect of memory consistency on ILP

As discussed in Section 3, memory consistency models can negatively affect performance by disallowing certain instruction reorderings, thus constraining both dynamic scheduling in hardware and static scheduling in compilers. Many recent implementations of processor consistency and sequential consistency overcome these scheduling constraints by performing speculative loads and non-binding prefetching [21], where a processor may speculatively hoist a load instruction above previous unresolved loads and stores, and detect instances where this is incorrect using the system's coherence protocol [25], [27], [59]. It has also been shown that given sufficient support for buffering speculative state, the performance of sequential consistency can approach the performance of relaxed memory models [22].

In this section, we quantify the effects of the consistency model on the levels of parallelism inherent to several executions of commercial and scientific applications. Although it has been shown that the parallelism-limiting effects of consistency models can be removed using speculation [21], [22], multiprocessors are increasingly being used in power constrained environments, such as high-density servers and embedded systems [7], [51], where issues like power dissipation and distribution suggest avoiding the use of many power-hungry speculative mechanisms. Although we believe that studying the fundamental levels of parallelism offered by different memory consistency models is intrinsically interesting, these real-world power constraints further motivate this work. We find that even when assuming an extremely (absurdly) aggressive machine configuration (perfect branch prediction, perfect memory disambiguation), the constraints imposed by the consistency model severely limit the performance of the sequentially consistent and processor consistent models, resulting in an average ILP of 1.5 and 2.2, respectively.

We measure the level of parallelism in these applications using the same calculation that has been used for measuring ILP in single-threaded programs [5],[30],[45], only we apply it to constraint graphs for multithreaded executions instead of single threaded executions. Given a graph-based representation of an execution, we measure parallelism as the number of instructions in the graph divided by the length of the longest path through the graph. In previous studies, this calculation produced the instruction-level parallelism metric. Our study measures a combination of instruction-level parallelism and thread-level parallelism, which we term *aggregate parallelism*. The aggregate parallelism metric is equivalent to the sum of the instruction-level parallelism achieved by each of the 16 processors in the system.

Figure 13 shows the aggregate parallelism for each application across each consistency model. The right most bar for each application shows the aggregate parallelism when ignoring all consistency constraints. As one would expect, as the memory model becomes weaker, the inherent parallelism in each application increases because the scheduling of each instruction is subject to fewer constraints, enabling some instructions to be executed prior to preceding independent instructions. On average, aggregate parallelism increases by 43% moving from sequential consistency to processor consistency and by 92% moving from processor consistency to weak ordering. Despite the increased flexibility offered by weak ordering, this model still incurs a significant impact on ILP for these applications. When removing the remaining consistency constraints (the memory barrier

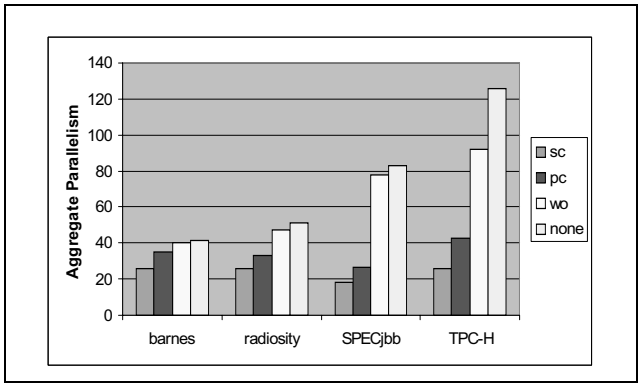

Figure 13: Aggregate parallelism across consistency models

constraints from weak ordering), ILP is increased by 14% on average over weak ordering. The impact of weak ordering is especially pronounced for TPC-H, whose instruction stream contains the most frequent occurrence of sync (memory barrier) instructions. Although SPECjbb also contains a relatively high percentage of sync instructions (compared to the scientific applications), the constraints imposed by these instructions affects ILP much less than TPC-H, indicating that instructions ordered by memory barriers are likely to already be ordered by dependence edges.

Although the aggregate parallelism may seem high in absolute terms, ranging from 18 to 126, this equates to modest amounts of ILP per processor (1.1 to 7.9). Although we use an aggressive machine configuration, enforcing consistency constraints on execution order severely limits instruction level parallelism, especially for sequential consistency and processor consistency. The results in Figure 13 illustrate the performance differences among non-speculative implementations of memory consistency models. Although the performance disadvantage of strict models can be compensated using hardware for speculative buffering, detection, and recovery, should a designer be unable to pay the cost of such hardware (in terms of power, area, or complexity), there is much potential for performance improvement from weaker models.

7. Conclusion

These case studies, an analysis of parallelism in multithreaded applications and an evaluation of RAW coherence miss avoidance that exploits the necessary conditions of different consistency models, have illustrated two types of studies that can be performed using the constraint graph model. Although most implementations of consistency models have been based on satisfying conservative sufficient conditions of the model, we have shown how the constraint graph can be used to reason about the necessary conditions, thus enabling more aggressive implementations. We outline one such implementation that enables the use of previously invalidated cache lines when the communication of new data is unnecessary, thus tolerating the latency of some coherence misses. In future work, we plan to evaluate a refined version of this protocol in the context of a hardware-based shared-memory multiprocessor.

We have also contributed extensions to the constraint graph representation for use in analyzing weak memory models. These extensions are also broadly applicable to the task of verifying the correctness of relaxed memory consistency implementations, beyond the scope of the performance analysis discussed here. In general, the constraint graph offers an intuitive framework for reasoning about memory consistency models, because it transforms a complex and subtle topic into

something more concrete and therefore more manageable. We advocate its use for future education and research in shared-memory multiprocessors.

Acknowledgments

This work was made possible through an IBM Fellowship, an internship at IBM T.J. Watson Research Lab, generous equipment donations and financial support from IBM and Intel, and NSF grants CCR-0073440, CCR-0083126, EIA-0103670 and CCR-0133437. The authors would like to thank Ilhyun Kim and Kevin Lepak for their comments on earlier drafts.

References

[1] S. V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, November 1993.

[2] Y. Afek, G. Brown, and M. Merritt. "Lazy caching." *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[3] A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz. "An evaluation of directory schemes for cache coherence." In *Proc. of the 15th Int'l Symp. on Computer Architecture*, pages 280–289, June 1998.

[4] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. "Causal memory: Definitions, implementation, and programming." *Distributed Computing*, 9(1):37–49, 1995.

[5] T. M. Austin and G. S. Sohi. "Dynamic dependency analysis of ordinary programs." In *Proc. of the 19th Intl. Symp. on Computer Architecture*, pages 342–351, 1992.

[6] L. Barroso, K. Gharachorloo, and E. Bugnion. "Memory system characterization of commercial workloads." In *Proc. of the 26th Intl. Symp. on Computer Architecture*, May 1999.

[7] Broadcom Corp. *SB-1250 Mercurian Processor Datasheet*, 2000.

[8] G. Brown. "Asynchronous multicaches." *Distributed Computing*, 4:31–36, 1990.

[9] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. "Single instruction stream parallelism is greater than two." In *Proc. of the 18th Intl. Symp. on Computer architecture*, pages 276–286, 1991.

[10] B. Charron-Bost. "Combinatorics and geometry of consistent cuts: Application to concurrency theory." In J.-C. Bermond, editor, *Distributed algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 45–56, Sept 1989.

[11] W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[12] A. Condon, M. Hill, M. Plakal, and D. Sorin. "Using Lamport clocks to reason about relaxed memory models." In *Proc. of the 5th IEEE Symp. on High-Performance Computer Architecture*, January 1999.

[13] A. Condon and A. J. Hu. "Automatable verification of sequential consistency." In *Proc. of the 13th Symp. on Parallel Algorithms and Architectures*, January 2001.

[14] F. Dahlgren and P. Stenstrom. "Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors." *Journal of Parallel and Distributed Computing*, 26(2):193–210, April 1995.

[15] M. Dubois, C. Scheurich, and F. Briggs. "Memory access buffering in multiprocessors." In *Proc. of the 13th Intl. Symp. on Computer Architecture*, pages 434–442, June 1986.

[16] M. Dubois, J. Skeppstedt, and P. Strenstrom. "Essential misses and data traffic in coherence protocols." *Journal of Parallel and Distributed Computing*, 29(2):108–125, 1995.

[17] M. Dubois, J.-C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. "Delayed consistency and its effects on the miss rate of parallel programs." In *Supercomputing*, pages 197–206, 1991.

[18] C. J. Fidge. "Partial orders for parallel debugging." In *Proc. of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, 1989.

[19] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[20] K. Gharachorloo, A. Gupta, and J. Hennessy. "Performance evaluation of memory consistency models for shared memory multiprocessors." In *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating System*, pages 245–259, 1991.

[21] K. Gharachorloo, A. Gupta, and J. Hennessy. "Two techniques to enhance the performance of memory consistency models." In *Proc. of the 1991 Intl. Conf. on Parallel Processing*, pages 355–364, August 1991.

[22] C. Gniady, B. Falsafi, and T. N. Vijaykumar. "Is SC + ILP = RC?" In *Proc. of the 26th Intl. Symp. on Computer Architecture*, pages 162–171, May 1999.

[23] J. Goodman. “Cache consistency and sequential consistency.” Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[24] A. Gupta, W.-D. Weber, and T. Mowry. “Reducing memory and traffic requirements for scalable directory-based cache-coherence schemes.” In *Proc. of the Intl. Conf. on Parallel Processing I*, pages 312–321, August 1990.

[25] Intel Corporation. *Pentium Pro Family Developers Manual, Volume 3: Operating System Writers Manual*, Jan. 1996.

[26] Intel Corporation. *Intel IA-64 Architecture Software Developers Manual, Volume 2: IA-64 System Architecture, Revision 1.1*, July 2000.

[27] G. Kane. *PA-RISC 2.0 Architecture*. PTR Prentice-Hall, 1995.

[28] P. Keleher, A. L. Cox, and W. Zwaenepoel. “Lazy release consistency for software distributed shared memory.” In *Proc. of the 19th Int’l Symp. on Computer Architecture*, pages 13–21, 1992.

[29] T. Keller, A. Maynard, R. Simpson, and P. Bohrer. “Simos-ppc full system simulator.” http://www.cs.utexas.edu/users/cart/simOS.

[30] M. Kumar. “Measuring parallelism in computation-intensive scientific/engineering applications.” *IEEE Transactions on Computers*, 37(9):1088–1098, September 1988.

[31] S. Kunkel, B. Armstrong, and P. Vitale. “System optimization for OLTP workloads.” *IEEE Micro*, pages 56–64, May/June 1999.

[32] M. S. Lam and R. P. Wilson. “Limits of control flow on parallelism.” In *Proc. of the 19th Intl. Symp. on Computer Architecture*, pages 46–57, 1992.

[33] L. Lamport. “Time, clocks, and the ordering of events in a distributed system.” *Communication of the ACM*, 21(7):558–565, July 1978.

[34] A. Landin, E. Hagersten, and S. Haridi. “Race-free interconnection networks and multiprocessor consistency.” In *Proc. of the 18th Intl. Symp. on Comp. Architecture*, 1991.

[35] J. Laudon and D. Lenoski. “The SGI origin: a ccNUMA highly scalable server.” In *Proc. of the 24th Intl. Symp. on Computer architecture*, pages 241–251. ACM Press, 1997.

[36] A. R. Lebeck and D. A. Wood. “Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors.” In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 48–59. ACM Press, 1995.

[37] H. Lee, Y. Wu, and G. Tyson. “Quantifying instruction-level parallelism limits on an epic architecture.” In *Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software*, pages 21–27, 2000.

[38] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. “Value locality and load value prediction.” In *Proc. of Seventh Intl. Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[39] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. “Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing.” In *Proc. of the 34th Intl. Symp. on Microarchitecture*, pages 328–337, December 2001.

[40] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors, 2nd edition*. Morgan Kauffman, San Francisco, California, 1994.

[41] G. Moore. “Cramming more components onto digital circuits.” *Electronics*, 38(8):114–117, 1965.

[42] A. Moshovos and G. Sohi. “Speculative memory cloaking and bypassing.” *Intl. Journal of Parallel Programming*, 27(6):427–456, 1999.

[43] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. “An evaluation of memory consistency models for shared-memory systems with ILP processors.” In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1996.

[44] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. “Lamport clocks: verifying a directory cache-coherence protocol.” In *Proc. of the 10th Symp. on Parallel Algorithms and Architectures*, June 1998.

[45] M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge. “The limits of instruction level parallelism in SPEC95 applications.” *Computer Architecture News*, 217(1):31–34, 1999.

[46] S. Qadeer. “On the verification of memory models of shared-memory multiprocessors.” In *Proc. of the 12th Intl. Conf. on Computer Aided Verification*, 2000.

[47] L. Rauchwerger, P. K. Dubey, and R. Nair. “Measuring limits of parallelism and characterizing its vulnerability to resource constraints.” In *Proc. of the 26th Intl. Symp. on Microarchitecture*, pages 105–117, 1993.

[48] R. N. Rechtschaffen and K. Ekanadham. “Multi-processor cache coherency protocol allowing asynchronous modification of cache data.” United States Patent 5,787,477, July 1998.

[49] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. “Complete computer simulation: the simos approach.” *IEEE Parallel and Distributed Technology*, 3(4):34–43, 1995.

[50] D. Shasha and M. Snir. "Efficient and correct execution of parallel programs that share memory." *Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[51] Silicon Graphics Inc. *SGI Origin 3900 Product Overview*, 2002. http://www.sgi.com/origin/3000/overview.html.

[52] M. Singhal. "A heuristically-aided mutual exclusion algorithm for distributed systems." *IEEE Transactions on Computers*, 38(5):651–662, May 1989.

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

[54] M. D. Smith, M. Johnson, and M. A. Horowitz. "Limits on multiple instruction issue." In *Proc. of the 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating System*, pages 290–302, 1989.

[55] D. J. Sorin, M. Plakal, M. D. Hill, and A. E. Condon. "Lamport clocks: reasoning about shared-memory correctness." Technical Report CS-TR-1367, University of Wisconsin-Madison, March 1998.

[56] G. Tyson and T. Austin. "Memory renaming: Fast, early and accurate processing of memory communication." *Intl. Journal of Parallel programming*, 27(5):357–380, 1999.

[57] D. W. Wall. "Limits of instruction-level parallelism." In *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating System*, pages 176–189, 1991.

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

[59] K. C. Yeager. "The MIPS R10000 superscalar microprocessor." *IEEE Micro*, 16(2):28–40, April 1996.

[60] R. Zucker and J.-L. Baer. "A performance study of memory consistency models." In *Proc. of the 19th Intl. Symp. on Computer Architecture*, 1992.