Constraint Graph Analysis of Multithreaded Programs

Harold W. Cain
Computer Sciences Dept.
Univ. of Wisconsin-Madison
cain@cs.wisc.edu

Mikko H. Lipasti
Dept. of Elec. and Comp. Engr.
Univ. of Wisconsin-Madison
mikko@engr.wisc.edu

Ravi Nair
IBM T.J. Watson Research Center
Yorktown Heights, NY
nair@us.ibm.com

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 one half of all load instructions which 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. The second case study demonstrates the effects of memory consistency constraints on the fundamental limits of instruction level parallelism, compared to previous estimates which did not include multiprocessor constraints. Using this method we determine the fundamental performance differences of various memory consistency models for processors which 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 [31]. 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 which 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 [8][26][29][35][39]. 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 most 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*, which 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 which are deemed necessary.

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 which did not include realistic

consistency constraints, we find that the inclusion of consistency constraints significantly reduces available instruction-level parallelism. Using this method, we also compare the fundamental performance differences of various memory consistency models, and quantify the maximum benefit in parallelism to be gained from more relaxed memory models by processors which do not perform consistency-related speculation.

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 which are not required by the consistency model, and evaluate this potential across several popular consistency models. We show that on average more than 60% 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 [26]. 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 compilers to ensure sequentially consistent executions [39]. The constraint graph has since been used to automatically verify sequentially consistent systems [8][35] and reason about the correctness of value prediction in multiprocessor systems [29]. 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 [18][19][30][41][45]. We show how previous constraint graph definitions can be augmented for use in 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 a plethora of 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 [10], and introduced the concept of the memory operations being "performed" locally, performed with respect to other processors in the system, and performed globally. Collier presented a formal framework which divided memory operations into several sub-operations in order 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 [7]. Adve incorporated the best features of Landin et al.'s model (acyclic graph) and Collier's framework (multiple write events) into a unified model which 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 which 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 [13]. The formal representation described here is most similar to Adve's representation, although in favor of simplicity it lacks the detailed modeling of non-atomic writes.

In Section 5 we present a study which uses the constraint graph to classify coherence misses as required or unnecessary. This study is related to prior work in delayed consistency [9][11][12][21], which 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]. The constraint graph characterization presented in this paper corroborates these previous results, measuring the frequency of accesses which 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 which 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 the number of coherence misses to truly shared data in

1. With the exception of IA-64, no commercial instruction set architecture requires using special labeled operations for synchronization. In most architectures, synchronization may be implemented using ordinary loads and stores. Consequently, it is not advisable to delay such operations, limiting the practicality of previous delayed consistency mechanisms.

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 describe an example of a cache coherence protocol which allows a processor to modify a cache line while other processors continue to use a stale copy [37]. 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 which 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 which are not constrained by any physical resources [6][27][34], while others include a wide range of resource constraints in the model [3][25][36][44]. 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 [14][33][47] by quantifying the theoretical performance differences.

3. Constraint graph definitions

A *constraint graph* [8] or an *access graph* [26] 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 [3][6][23][25][27][34][36][42][44]. For a thorough

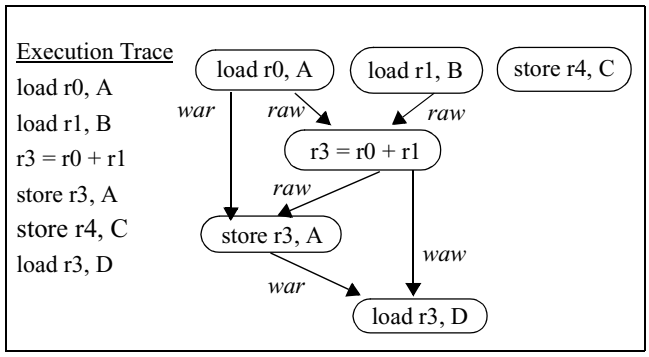

Figure 1. Uniprocessor constraint graph

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 which writes a memory or register element is preceded through a storage dependency by any instruction which has previously read that element. This edge models a finite number of storage 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 which writes a memory or register location is preceded through a storage dependency by any instruction which 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 [32][43], and RAW edges can be removed using value prediction [28]. 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 are orthogonal to the multiprocessor issues which 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 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, in which 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. [26]. To illustrate this principle, a revised version of the example in Figure 1 which has been augmented with consistency edges for sequential consistency can be found 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 to 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 first two operations executed by *p1* and the operations executed by *p2*.

In the next two subsections, we specify modifications

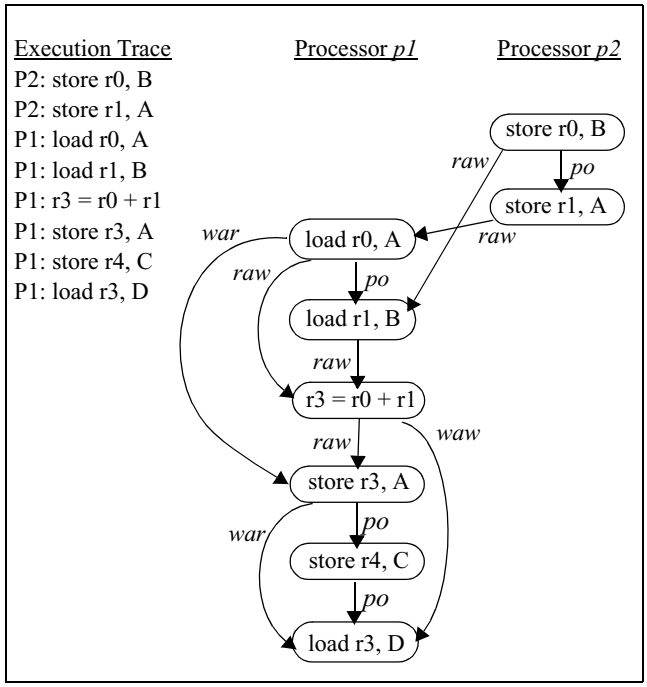

Figure 2. Constraint graph including sequential consistency edges. (Redundant program order edges removed for clarity.)

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 which remove many of the constraints associated with sequential consistency. Processor consistency (PC), variations of which are used in the x86, 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 [17]. Because of this difference, the constraint graph for processor-consistent systems contains a different set of consistency edge types than used for 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.

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.

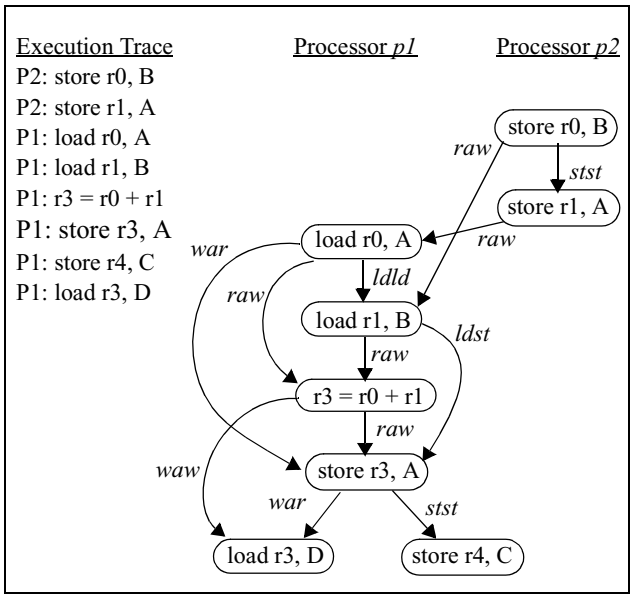

Figure 3. Processor consistent constraint graph

- **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, in which 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 [10]. The semantics of such instructions specify that all memory operations which precede the memory barrier must appear to have executed before any of the instructions 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.

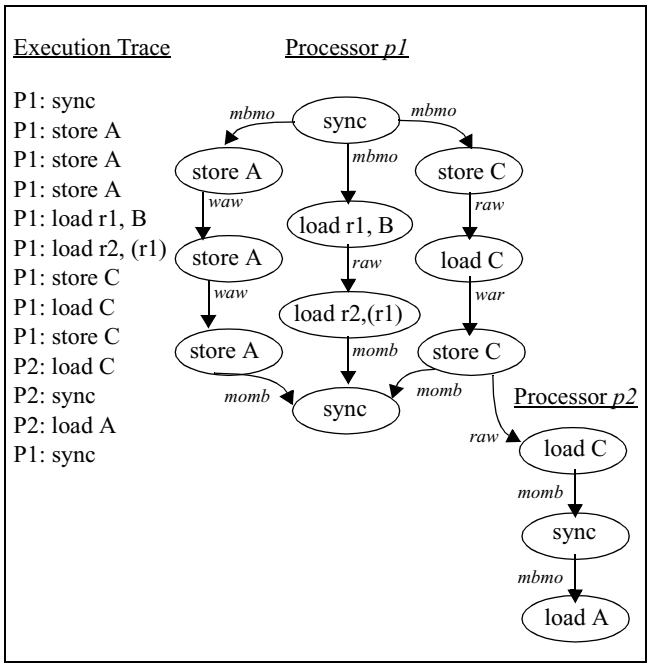

Figure 4. Weakly ordered constraint graph

- **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, yielding three separate independent strands of execution which are not ordered with respect to one another. Each strand must appear to execute after the first sync operation and before the second sync operation, however there is no required ordering among the strands. The load instruction executed by *p2* is ordered after the entire right-most strand of *p1*'s execution, but is not ordered with respect to the remainder of *p1*'s operations. Consequently, the load A executed by *p2* does not necessarily need to return one or any of the values 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.

Because we use a PowerPC-based weakly-ordered infrastructure for this work, we are able to study any memory model as strict or stricter than weak ordering, such as sequential consistency and processor consistency. Con-

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)

straint graphs for models more relaxed than weak ordering can be expressed 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 [22], a PowerPC version of the original execution-driven full system simulator [38], to generate an instruction trace containing user and system level instructions of a variety of commercial and scientific parallel workloads. 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 each trace's constraint graph and performs analysis as described in subsequent sections.

A summary of workloads 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 the commercial workloads, as others have observed [4][24].

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 which 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 an unequivocal 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 reference the location, their references incur coherence misses.These delays are exacerbated in multiprocessor systems in which cache misses must navigate multiple levels of interconnect before they may be 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 [24].

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 in which 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.

5.3 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 which 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 which result in either a miss or an upgrade message between processors. We filter out the per-processor compulsory misses from this set of references, which yields the set of coherence misses. A count of these misses for each workload is found in Table 1. Because the dependence edge types are uniform across consistency models, the number of interprocessor edges is also uniform.

Once a coherence-related load miss has been identified, we determine whether or not that miss is avoidable based on the following criterion: given a potentially avoidable RAW edge *e* emanating from writer node *w* and connecting to reader node *r*, if there exists a directed path in the constraint graph from *w* to *r* that does not include edge *e*, then RAW edge *e* is necessary. 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 avoidance of the cache miss must be correct.

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 which 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

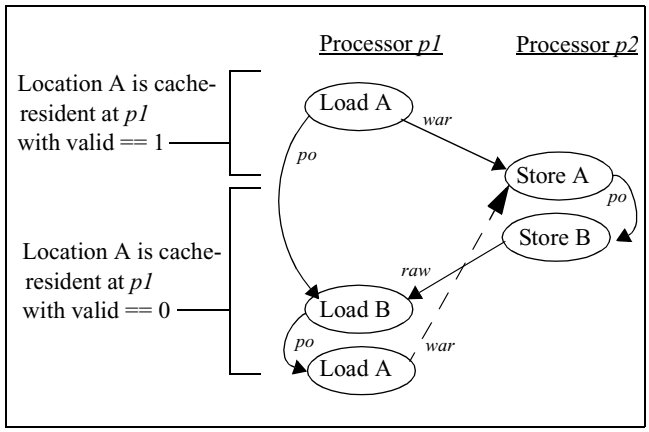

Figure 5. A necessary coherence miss (time progresses from top to bottom)

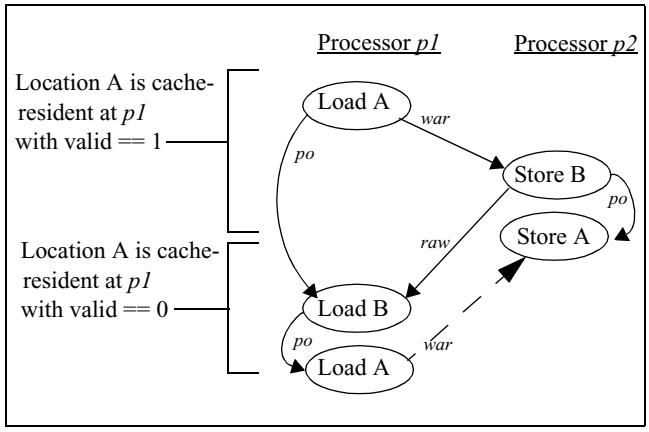

Figure 6. An unnecessary coherence miss

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 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. 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. However, using the constraint graph, we can detect cases where it is safe to use the stale value. In

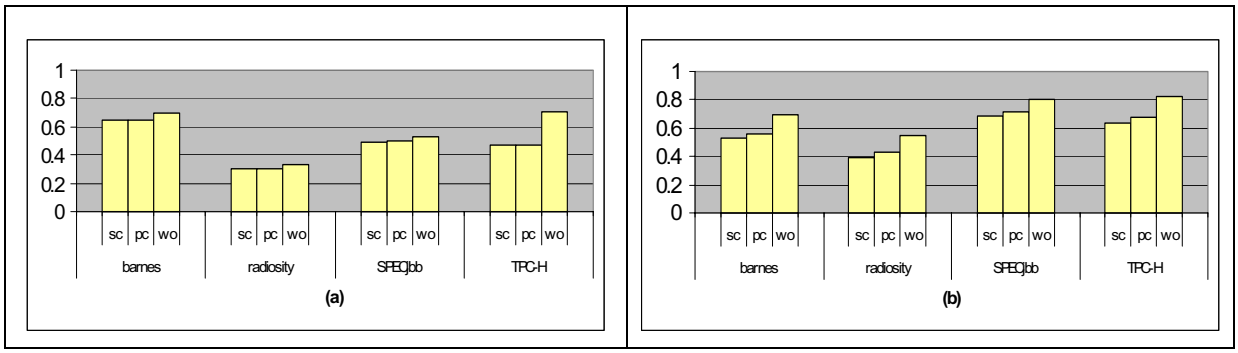

Figure 7. Fraction of coherence load misses avoidable for sequential consistency (sc), processor consistency (pc), and weak ordering (wo). (a) 4 byte coherence unit, and (b) 128 byte coherence unit

the next sub-section, we quantify the frequency of this scenario.

5.4 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 which 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 which 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 5.3 as either avoidable or necessary subject to the given memory model's constraint graph. Figure 7 shows the fraction of these RAW coherence misses which are avoidable for each of the three consistency models. There is a significant fraction of cache misses avoidable for all of the workloads. The absolute number of coherence misses is shown in Table 1, of which 41% and 50% 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 on average 66% of RAW coherence misses are avoidable 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).

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 right decision. There are certain scenarios in which 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 fraction of avoidable RAW coherence misses which are not caused by the PowerPC *lwarx* and *ldarx* synchronization instructions (which have load-linked semantics). By filtering out those misses which are caused by *lwarx* or *ldarx* instructions, we can estimate the fraction of avoidable load misses attributable to synchronization. We can see from the chart that between 9% and 41% of all of the avoidable load misses are caused by synchronization accesses. On average across the models and applications, 79% of the RAW coherence misses which were labeled avoidable are not caused by synchronization.

We leave the exploration of a hardware mechanism that takes advantage of this property to future work. However, it should be clear that there is much potential for cache controllers that can perform selective invalidation based on the knowledge that some invalidates must be applied immediately while many invalidates may be delayed (31% to 71%, 50% on average across models and applications). In order to solve the synchronization prob-

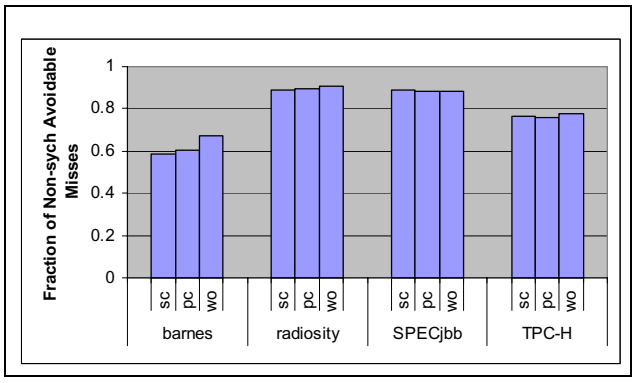

Figure 8. Fraction of avoidable RAW coherence misses that are not attributable to synchronization at 128-byte coherence granularity

lem, one can simply continue using the stale value but issue a prefetch request for the data, thus ensuring that a lock release or any other time-critical data will be observed. 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. Using this framework, we can determine that because using stale data does not create a cycle in the constraint graph, it is necessarily correct. We believe that reasoning about consistency model implementations using the constraint graph framework will open the door for many other optimizations as well.

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 [15], 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 [18][20][46]. 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 [16].

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 [15][16], multiprocessors are increasingly being used in power constrained environments, such as high-density servers and embedded systems [5][40], where issues like power dissipation and distribution suggest rethinking 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 [3][23][34], 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 9 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 an additional 87% 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 constraints from weak ordering), ILP is increased by 17% 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

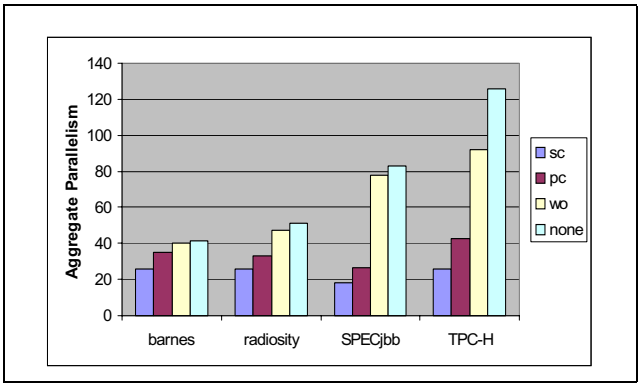

Figure 9. Aggregate parallelism across consistency models

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 9 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 which 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 have 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.

Despite the availability of much published literature and a widely read tutorial on the subject [2], we have experienced much confusion when thinking about what we believe is the most subtle and complex topic in computer architecture. We believe that the constraint graph offers an intuitive framework for reasoning about memory consistency models, because it transforms an otherwise confusing 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 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] S. V. Adve and K. Gharachorloo. “Shared memory consistency models: A tutorial.” *IEEE Computer*, 29(12):66–76, 1996.

[3] 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.

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

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

[42] 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.

[43] 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.

[44] 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.

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

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

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