Comparison of Memory System Behavior in Java and Non-Java Commercial Workloads

Morris Marden[1], Shih-Lien Lu[1], Konrad Lai[1], Mikko Lipasti[2]

[1]Microprocessor Research, Intel Labs
Intel Corporation
Hillsboro, OR 97124
{morris.marden, shih-lien.lu, konrad.lai}@intel.com

[2]Dept. of Electrical and Computer Engineering
University of Wisconsin
Madison, WI 53706
mikko@ece.wisc.edu

Abstract

In this paper, we compare the memory system behavior of Perl and Java versions of SPECweb99. We find that the memory behaviors of these two versions of SPECweb99 are very different. The Java version incurs 213% more cache misses than the Perl version for a 1 MB second level cache and 179% more misses for a 16 MB cache. Much of this increase is due to true sharing, which is 222% to 415% higher in the Java workload than in Perl. We find that for 1 MB caches, the exclusive state is effective, since it is able to remove 24% and 61% of the write upgrades for Perl and Java respectively. However, for 8 MB caches, the exclusive state only removes 1% to 5% of the upgrades. Also, it is important to have an efficient mechanism for cache to cache transfers, since 71% to 87% of the second level cache misses hit a modified line in another processor's cache when using 8 MB second level caches. Our early results show that processor consistency can improve the performance of Perl and Java SPECweb99 by 14% and 22% over sequential consistency. Surprisingly, we find that removing all of the serializations in processor consistency degrades the performance of the Java workload by 11%.

1. Introduction

With the growth of the Internet and the World-Wide Web, there has been a dramatic increase in the number of types and styles of programs that computers are being required to run. Traditionally, software engineers wrote programs in high-level languages, like C and C++, which then were compiled into assembly language long before users ran the programs. Later, scripting languages were developed, such as Perl, which interpret each line of code of a program at runtime. Recently, dynamic runtime compiled languages have gained in popularity. These languages, including Java and .Net, are compiled into an intermediate machine language, which is independent of all machines that it is run on. When the user runs these programs, the Virtual Machine compiles these programs on demand for the machine that they run on. Each of these program styles are common today and place different demands on the hardware of the computer. Thus, today's computers must be versatile and perform well with very different program behaviors.

While Java is now a few years old, its memory system behavior is still somewhat a mystery. Recently, there have been a number of papers [2, 4, 9-12, 16, 18-19] that have studied this behavior. However, few studies have compared the behavior of Java workloads to other workloads. Luo and John [12] compared VolanoMark, a Java chat room server, and SPECjbb2000, a java server-side business benchmark that models a warehouse system, to SPECint2000, a general processor benchmark. Cain et al. [2] compared a Java implementation of TPC-W, an online store benchmark, and SPECjbb2000, to SPECweb99, an ISP web server benchmark, and SPECint95. Lepak et al. [10] study the amount of silent stores in a Java implementation of TPC-W, SPECjbb2000, TPC-B (an online transactions database workload), and the SPLASH-2 benchmark suite (a suite of scientific workloads). In these three studies, the Java and non-Java workloads are very different, so one cannot compare the behaviors of the two sets of workloads. In this paper, we compare the memory system behavior of SPECweb99, which uses Perl, to a version that we ported to Java. In addition, of the earlier works of Java workloads, only Cain et al. [2] and Lepak et al. [10] studied the impact of multiprocessor behavior on memory systems. We study the memory system behavior of a four processor machine in this paper.

Even though there have been a large number of studies on memory consistency models, few studies measure their performance differences. The studies that measure performance include [1, 6, 13-14, 17]. Even fewer studies measure performance of consistency models for commercial workloads. Of the above papers, only Martin et al. [13] studied this. In this paper, we include early results of the performance of simple implementations of sequential and processor consistency.

2. Methodology

To compare the memory behavior of Java and non-Java workloads, we used the SPECweb99 benchmark [20], which models a server for the home-pages of users of an ISP. SPECweb99 includes requests for static and dynamic web pages, keep alive and persistent connections, and user targeted rotating advertisements that use cookies. We ported this benchmark from a Perl CGI to a Java Servlet. The Java Servlet performs the same work as the Perl CGI, except that we use shared memory based locks instead of file-based locks in the Servlet. Shared memory locks are more efficient than file locks, since they do not require intervention by the operating system, file system, and disks. However, Perl scripts do not have the ability to use shared memory between processes, so the original Perl implementation is forced to use file-based locks. We believe that this distinction is important, since shared memory locks can have a great impact on memory behavior. Table 1 lists the software that we used for the workload.

To measure the behavior of the memory system, we use the Simics full system simulator [21]. Simics is a functional simulator that simulates multiprocessor systems, using unmodified binary programs. To measure memory behavior, we wrote a memory system model that simulates a two level cache hierarchy and a cycle-accurate multiprocessor split-transaction bus. The bus protocols in our memory model are based on the Pentium II MESI protocol [7] and are tuned for characteristics of processors a few years in the future. Simics sends each memory request to our memory model, which analyzes the effects of the requests and sends the timing information back to Simics. To prevent our results from being skewed, the memory model detects and removes instruction and data read accesses in idle and spin loops. Our memory model also classifies the causes of cache misses and uses Dubois' definitions of sharing [5] to compute the number of true and false sharing misses.

Table 1 shows the configurations that we used for the simulated system. In our experiments, we varied the second level cache line size and second level cache size. The first level cache used the same sized lines as the second level cache. The memory model maintains inclusion between the first level caches and the second level cache, and maintains exclusion between the first level caches (so that self modified code will be handled correctly). We also measured the differences between simple implementations of sequential and processor consistency. Under sequential consistency, the processor can only execute new instructions when the node has completed all outstanding memory requests. However, under processor consistency, the processor does not need to stall for cache misses on writes unless there is a cache miss to the line for a read. Our implementations of these consistency models are discussed in more detail in Section 4.

Operating System	Red Hat Linux 6.2
Web Server:	Apache 1.3.20
Perl:	Perl 5.005_03
Java Server:	Apache JServ 1.1.2
Java:	Sun Java SDK 1.4.0 β3
Processor:	Intel Pentium II
# of processors:	4
Ratio processor to bus frequency:	10:1
Main memory:	1 GB
Minimum memory access latency:	120 processor cycles
Consistency model:	Sequential, Processor
L2 cache read/write ports (total):	1
L2 cache associativity:	4
L2 cache line size:	64 B, 128 B, 512 B
L2 cache size:	1 MB, 8 MB, 16 MB
L2 cache latency:	6 processor cycles
L1 D-cache read ports:	4
L1 D-cache write ports:	2
L1 cache associativity:	1
L1 cache line size:	*same as L2 cache*
L1 cache size (I & D, each):	128 KB
L1 cache latency:	0 cycles (i.e., scheduled in pipeline)

Table 1: Simulated server configuration

SPECweb99 is composed of two parts: a client and a server. The client emulates user requests for static and dynamic web pages from the web server. The server handles these requests and sends responses back to the client. Since we are only interested in the behavior of the server (the system under test or SUT), we simulated the client and server on separate machines and used Simics' network capability to simulate a network between the two simulated machines. We then collected measurements for the behavior of the server machine alone. Figure 1 shows a diagram of the simulated machines.

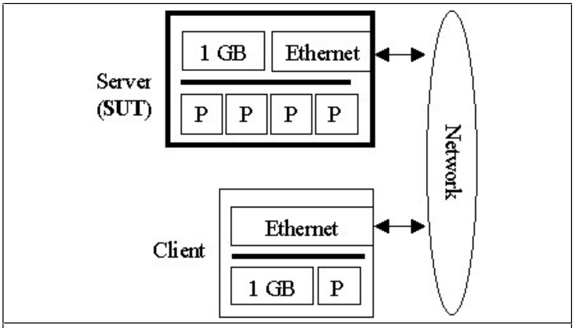


Figure 1: Diagram of simulated machines setup

To obtain reliable steady state results, we first warmed up each of the workloads for a few minutes of simulated time without our memory model (it would be too slow to do this with the memory model). This ensures that our simulator will measure the steady state behavior of the workload. We tuned the load of the workloads to maximize the utilization of the processors. If the load is too high, then the system will spend too much time handling disk accesses and the processors will then be idle while waiting for the accesses to complete. On the other hand, if the load is too low, then there will not be enough work for the processors. In our simulations, we set load of Java SPECweb99 to be the maximum that the client could produce, which was 33% higher than the optimal load of the Perl version.

After warming up the workloads without the memory module, we then ran the workloads with the memory module to collect results. After the first simulated processor (of the server) has executed 250 million instructions, the memory model cleared all of the measurements collected so far. This prevents our results from being skewed due to initial behavior of the simulated system. For example, at the beginning of the simulation, the simulated caches are completely empty, so most of the misses will be compulsory misses. However, when the workload reaches steady state, there will be few compulsory misses. After resetting the measurements, the memory model then measures the memory behavior of the server until the first processor has executed another one billion instructions.

3. Sequential Consistency Results

3.1. Miss Rate

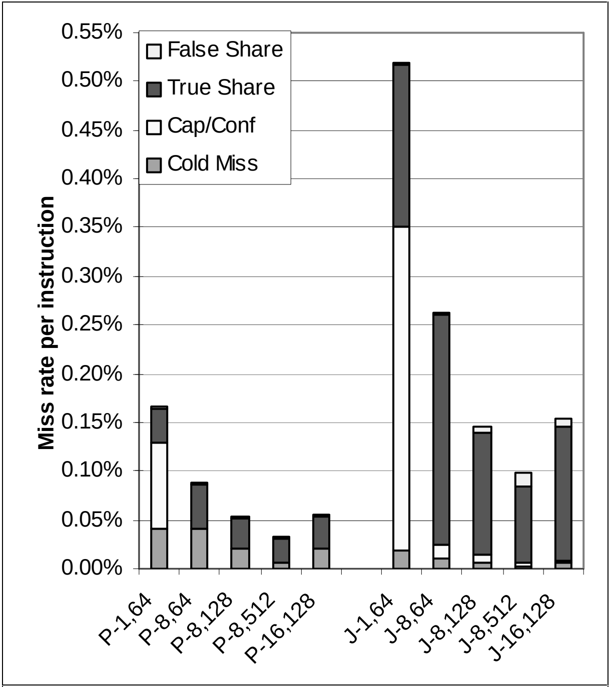


Figure 2: Cache miss rates
Note that X-Y,Z stands for X workload (P for Perl, J for Java), Y MB sized L2 cache, with Z byte lines.

Figure 2 shows the miss rates of the second level caches for the workloads and why these misses occur. Our results show that our Java version of SPECweb99 has a higher miss rate than the Perl version. The Java implementation has a larger working set than the Perl, since the Java version suffers more capacity and conflict misses, especially with relatively small second level caches. The Java Servlet also has much more true sharing misses than the Perl CGI, much of which is due to the shared memory locks. In addition, Java SPECweb99 also incurs much more false sharing misses than Perl, especially at large line sizes.

The sharing behavior of the workloads varies greatly with cache configuration. In relatively small second level caches, the large number of capacity and conflict misses reduce the number of sharing misses, since it is less likely that a cache line will be in a remote cache when a processor wishes to write to the line. The number of true sharing misses greatly reduces with line size, especially for the Java implementation. We believe that this occurs since reads and writes are clustered in adjacent cache lines (due to spatial locality). Therefore, when two or more processors share data, a processor will invalidate a large number of consecutive lines in other processors' caches when it writes the data. Later, when the other processors access this data, the large cache lines prefetch more of this data on each access, thereby reducing the number of cache misses. In addition, this reduction in true sharing misses is larger than the increase in false sharing misses, indicating that 512 byte line sizes are useful. Our results also show that increasing the line size is more effective in reducing the miss rate than increasing the size of the cache. In the case of the Java workload, the miss rate actually becomes worse when one increases the size of the cache, since it is more likely that shared data will be in a remote cache.

3.2. Remote Cache Hits

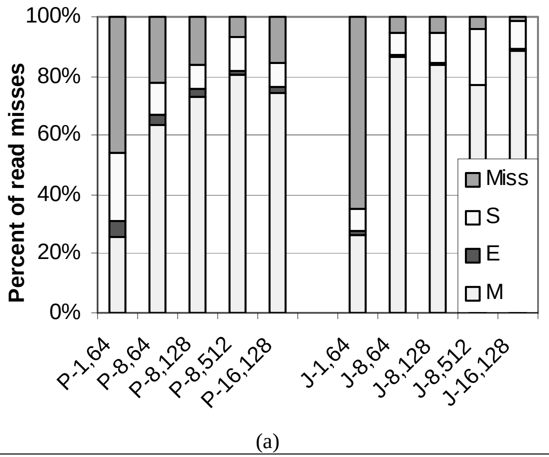
To better understand the sharing behavior of the caches, Figures 3a shows the number of cache misses for data reads that hit in one or more of the other processors' caches. In the Perl version, we see that more than half of the reads hit in a remote cache. For relatively large caches, most of the remote read hits are to lines in the modified state, indicating that it is important to implement an efficient cache to cache transfer mechanism for modified lines. The Java port has an even larger number of remote read hits to the modified state, in part due to the shared memory lock.

The behavior of write misses is similar to that of read misses, as shown in Figure 3b. In the Perl version, there are more write misses that hit in remote modified lines than in the case of read misses. However, there are much fewer write misses that hit in the shared state in a remote cache than for read misses. In contrast, in the Java version, the differences in the number of write misses and read misses that hit in remote modified and shared lines varies based on the line size. One reason that the Java version has more write misses that hit in remote caches in the shared state is that under a highly contented shared memory lock, each processor frequently reads the value of the lock to see if it can acquire the lock. Therefore, the lock will often be in the shared state in a processor's cache when another processor writes to the lock to acquire or release the lock.

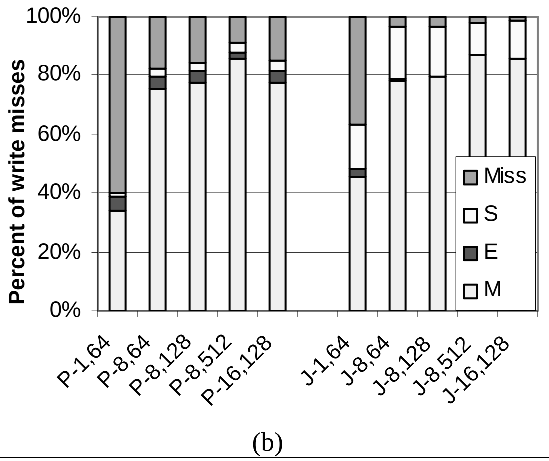
3.3. Exclusive State in Shared Memory Multiprocessors

The exclusive state of the MESI protocol is designed to improve cache performance reducing the number of invalidations that processors use for write upgrades. Recent studies have debated the effectiveness of the exclusive state. Some studies have argued that the exclusive state is not useful, since they found that there are few writes that hit in the exclusive state [3, 8]. However, we believe that a better measure for deciding the usefulness of the exclusive state is the number of lines that enter the cache in the exclusive state and the number of times that the processors can perform silent upgrades instead of invalidations to go from shared to modified. Most writes hit in cache lines in the modified state due to spatial and temporal locality. The exclusive state is not aimed at improving the performance of writes that hit in the cache in the modified state. Using this method, Cain et al. [2] found that the exclusive state is useful for reducing the number of invalidations for write upgrades.

Figure 4 shows that most lines enter the cache in shared state on read misses. The number of shared lines increase with cache size and line size, since it is



(a)



(b)

Figure 3: Misses that hit in remote caches
(a) Read misses
(b) Write misses

Note that X-Y,Z stands for X workload (P for Perl, J for Java), Y MB sized L2 cache, with Z byte lines.

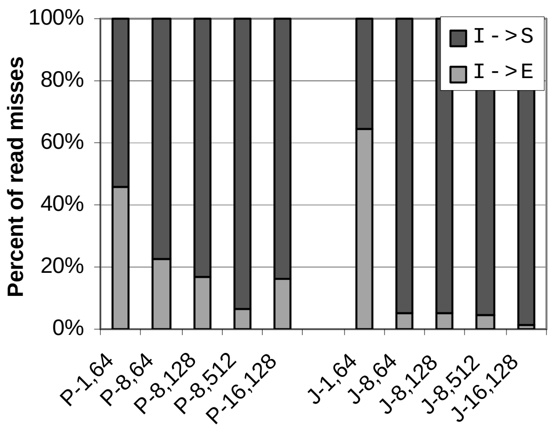


Figure 4: State that lines for read misses enter

Note that X-Y,Z stands for X workload (P for Perl, J for Java), Y MB sized L2 cache, with Z byte lines.

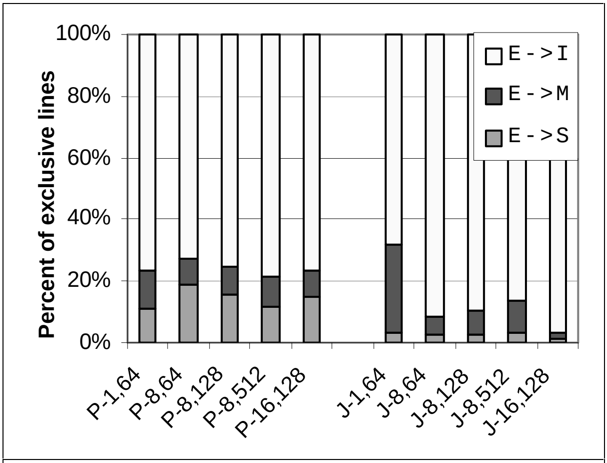


Figure 5: Transitions leaving the exclusive state

Note that X-Y,Z stands for X workload (P for Perl, J for Java), Y MB sized L2 cache, with Z byte lines.

more likely that the line will be present in another processor's cache. Here, we see that for relatively large caches, there are more exclusive lines in the Perl version than Java. We also see that the exclusive state is more sensitive to cache line size in the Perl implementation than Java. However, for small caches, the exclusive state is used more in the Java Servlet than the Perl CGI.

In Figure 5, we see that most lines that are in the exclusive state never change to the modified or shared states. In the Perl implementation of SPECweb99, we see that more exclusive lines are downgraded to shared than those that are upgraded to modified state. However, in the Java port, more exclusive lines change to modified than shared, indicating that the exclusive state may be more useful in the Java version than in the Perl version.

We can see the overall impact of the exclusive state by looking at the origins of modified lines in the caches, shown in Figure 6. For each pair of bars in the above graphs, the first bar includes all sources of modified lines and the second bar includes only the write upgrades. First, we see that more than half of the lines become modified on write misses rather than upgrades. Interestingly enough, the number of modified lines that enter the cache on misses is highly dependent on the cache size and line size for Perl, but is mostly independent of the cache configuration for Java. A reason for this is that most of the write misses are due to the sharing in the Java version, which is largely from the shared memory lock. For relatively small caches, there are a large number of upgrades from exclusive to modified. In addition, for small caches, the Java implementation performs more upgrades from exclusive to modified than the Perl version. In particular, 62% of the upgrades are from the exclusive state in the Java Servlet, but only 24% of the upgrades are silent for the Perl CGI. However, for relatively large caches, there are few upgrades from the exclusive state, so it is less useful.

3.4. Dirty Lines

Figure 7 shows the MESI transitions for dirty cache lines. Note that the `M->M` is for modified lines that never leave the modified state. First, we see that a large number of modified lines downgrade to shared state due to another processor's read. As in the case of exclusive lines, we see that the behavior of the Perl version is highly dependent on the cache configuration, whereas the Java version is mostly independent of the configuration.

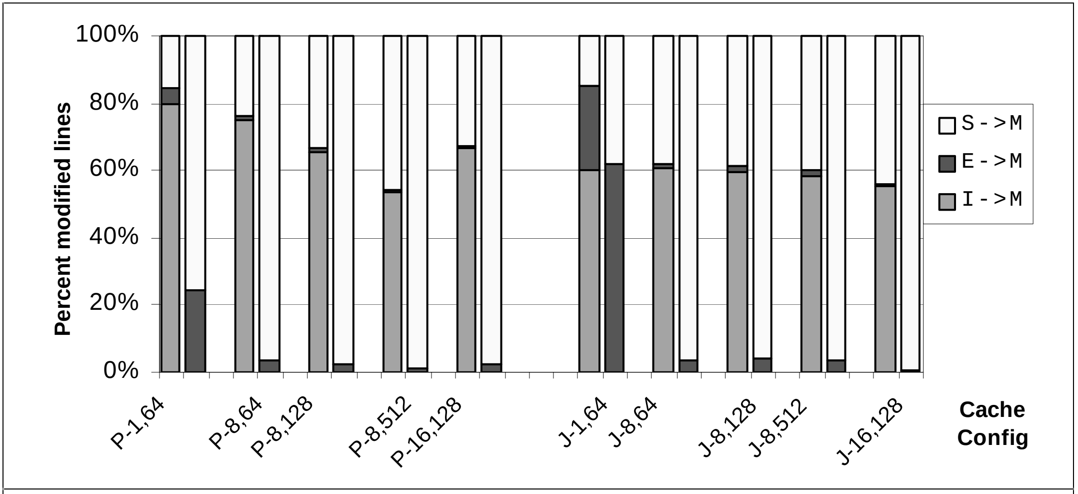


Figure 6: Sources of modified lines

Note that in each pair of bars, the first bar includes all sources of modified lines and the second bar only includes lines that upgraded from shared or exclusive state.

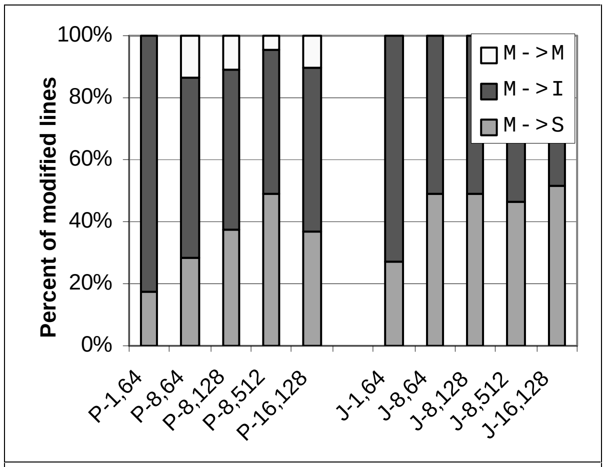


Figure 7: Transitions from modified lines

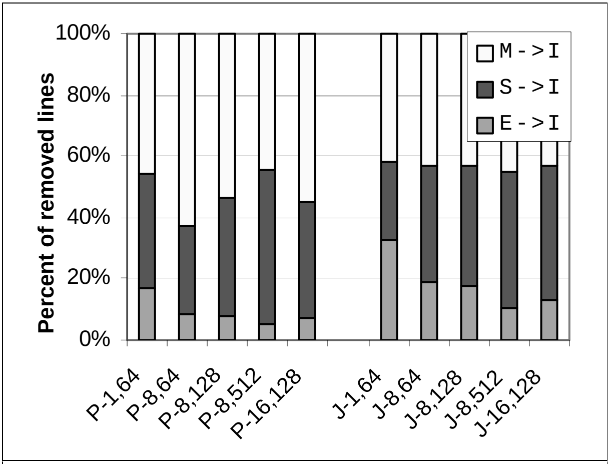


Figure 8: Lines removed from the cache

Similarly, we see that this is also true for the number of modified lines that are removed from the cache, as seen in Figure 8. Here, we see that about 40% to 65% of the lines that are removed from the cache are dirty, indicating that it is important to have an efficient mechanism for handling dirty removals. We also see that the number of shared lines increase with the cache size and line size, since it is more likely that a line will be shared as the caches become larger. Surprisingly, we see that the exclusive lines have the opposite behavior of the modified lines. In particular, the number of exclusive lines removed from the cache is relatively constant in the Perl version, but is dependent on the cache configuration in the Java implementation. This trend is also the opposite of the number of lines that enter the cache in exclusive state on read misses.

3.5. Stall Cycles

We show the number of stall cycles that each instruction has in Figure 9. Note that write back stalls occur when a processor has a second level cache miss after it has started performing a write back, so the processor cannot start handling the miss until the write back completes. In addition, on second level cache misses, the latency of the caches are included in the number of cycles of stall for the miss and not counted under second level cache hits. The restart processor stalls refer to a one processor cycle stall that occurs after the processor finishes handling a series of one or more first level cache misses, to model the overhead of restarting a processor after a cache miss. Lastly, the miscellaneous stall cycles contain the remaining stalls that the processors suffer, including first level data cache hazards and negative acknowledgements.

First, we see that the number of stall cycles is much larger for Java SPECweb99 than Perl, due to a larger cache miss rate. Note that the load of the Java

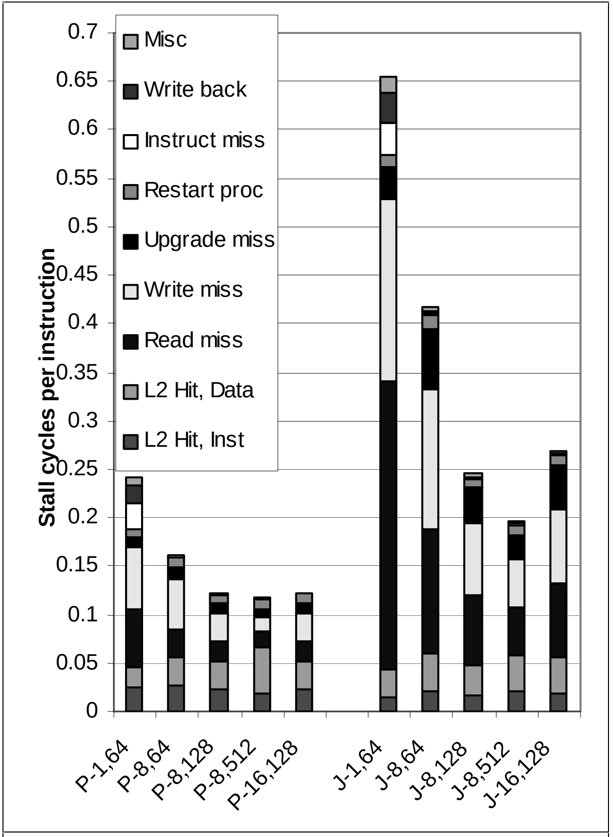


Figure 9: Stall cycles per instruction
Note that X-Y,Z stands for X workload (P for Perl, J for Java), Y MB sized L2 cache, with Z byte lines.

version is higher than the Perl version, so it is handling more transactions for the same amount of time. We also see that see that for both the Perl and Java versions, the number of stall cycles dramatically drops when one increases the cache from 1 MB to 8 MB and when one increases the line size from 64 bytes to 128 bytes. Our results show that the stall cycles due to instruction misses and write backs quickly disappear when one increases the cache size to 8 MB. However, in the Perl version, the number of stall cycles does not change significantly when one increases the cache size to 16 MB or the line size to 512 bytes. In the Java implementation, though, the number of stalls decrease dramatically again when one increases the line size to 512 bytes. Surprisingly, in the Java version the stalls increase when one increases the cache size to 16 MB, due to an increase in the number of true sharing misses.

4. Processor Consistency

4.1. Implementation

To better understand the impact of consistency models on commercial workloads and whether consistency models warrant further study, we added simple implementations of sequential and processor consistency to our memory model. Under sequential consistency, all memory operations must appear to occur in the same order. For example, it is illegal for one processor to observe that a store occurs before a load and another processor to observe that the write occurs after the load[1]. To this end, our implementation of sequential consistency stalls each processor until all memory accesses complete for an instruction. By stalling the processor until all accesses complete, the processor cannot observe its own accesses before any other processor, since the accesses will have been broadcast over the bus.

Processor consistency relaxes memory ordering by allowing a processor to observe write to read ordering in a different order than another processor. In the previous example, processor consistency allows the first processor to observe the store before the load, even though the second processor observes the store after the load. Note that to ensure that programs work correctly, programmers may need to explicitly add serializations to their programs to ensure that a particular ordering of operations occur. In our simple implementation, under processor consistency, the processor stalls when a read miss occurs, but does not normally stall when a write miss occurs. Note that our simple implementation does not assign higher priority to reads than writes. Therefore, to avoid starvation, our implementation only allows there to be a maximum of 31 outstanding write misses, without stalling the processor. When the processor has more than 31 outstanding writes misses, it must stall until some of the misses complete and there are only 31 pending write misses.

[1]Note that it is legal for the load and store be performed in a different order with respect to different processors, as long as the two processors appear to observe that the operations occur in the same order.

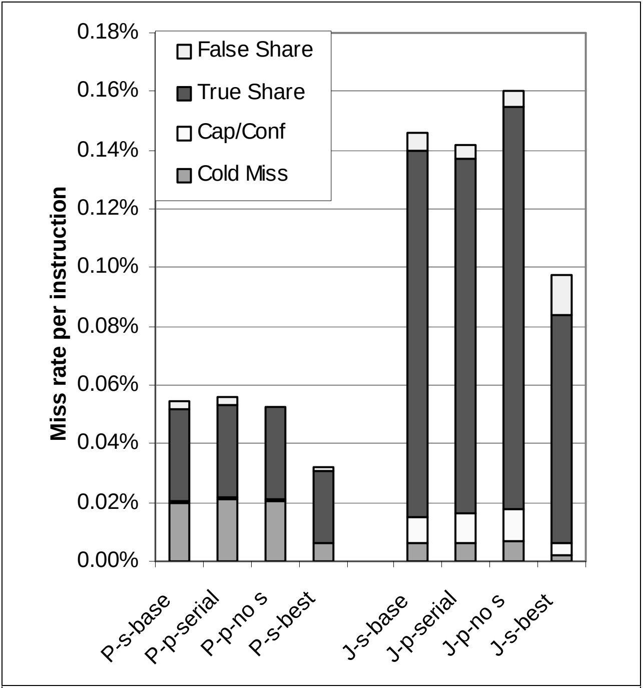


Figure 10: Cache miss rates for consistency models

X-s-base: *baseline sequential consistency (8MB, 128 B)*
X-p-serial: *processor consistency with serializations*
X-p-no s: *processor consistency without serializations*
X-s-best: *sequential consistency with the highest performance cache configuration (8 MB, 512 B)*

4.2. Results

In Figure 10, we show the second level cache miss rates for four different test cases for each workload. In the first test case, the simulated system uses sequential consistency with 8 MB second level caches and 128 byte cache lines (from Section 3). For the second case, the simulated system uses processor consistency with 8 MB second level cache with 128 bytes. The third test case shows the miss rate of a system that uses processor consistency, but is somehow able to ignore serializations and does not need to enforce the ordering that the programmer specified (this is similar to [15]). In the last test case, we show the miss rate of a system with sequential consistency, 8 MB second level caches, and 512 byte lines, which had the highest performance of all the sequential consistency cases (as seen in Section 3).

In the case of the Perl version of SPECweb99, we see that the miss rate of the system with processor consistency is slightly higher the system with sequential consistency. When a system with processor consistency does not need to perform serializations, the false sharing misses disappear.

Our results show that the interaction between the consistency model and miss rates is different for our Java implementation than Perl. In particular, we see that the number of sharing misses is smaller for processor consistency than sequential consistency. Surprisingly, when the system with processor consistency does not need to perform serializations, the number of sharing misses increase, such that the overall miss rate is 10% higher than in the case of sequential consistency. We believe that this occurs because there are some cases in which the serialization enforces that only one processor can use a cache line at a time. When the serialization is removed, then two or more processors use the cache lines at once, so they frequently steal the lines from each other before any of them can finish working with the line, increasing the amount of sharing.

Table 2 shows the number of serializations that occur in each workload. The processors do not need to add a memory barrier each time that they reach a serialization point. In particular, if there are no outstanding cache misses when a processor reaches a serialization point, then it does not need to insert a memory barrier. In addition, since our implementation of processor consistency does not prioritize reads over writes, if the last cache miss was for a read, then the processor does not need to add a memory barrier, since the processor will be stalled until all of the outstanding cache misses complete anyway. Note that due to current limitations in our simulator, we were not able to detect all serialization points for the Perl workload. Surprisingly, the processor does not need to add a memory barrier at most serialization points.

Figure 11 shows the number of stall cycles occur for each instruction for the four cases of memory consistency. The decrease in the number of stall cycles of the test cycles over the baseline (sequential consistency with 8 MB cache and 128 byte lines) is shown in Table 3. We see that adding processor consistency reduces the number of stall cycles for second level cache data hits, write misses, and upgrade misses by about 20% each, for the Perl version. Overall, the reductions in stalls create a respectable 13.68% overall improvement, which is much larger improvement than if one increases the cache line size to 512 bytes for sequential consistency. While removing the serialization points from processor consistency only improves the performance by another 2%, this shows that even a small number of serializations can have a significant impact on performance.

Our results that show that the Java version of SPECweb99 benefits from processor consistency more than the Perl version. In the Java version, the number of stalls for write misses and upgrade misses decrease by 47% and 43% respectively. However, the number of stall cycles for second level cache data hits decrease by only 2%. The reduction in stall cycles yields an overall 22.2% improvement, which is a little better than the improvement from increasing the cache line size to 512 bytes for the sequential consistency. Surprisingly, removing the serializations actually worsens the performance of the performance of the workload by 11%. The number of stall cycles for second level data misses, data read misses, write misses, and upgrade misses each increase by 10% to 16%. As

Workload	Memory Barriers Inserted	Serialization Points
Perl	22	?
Java	237	28,109

Table 2: Number of serializations

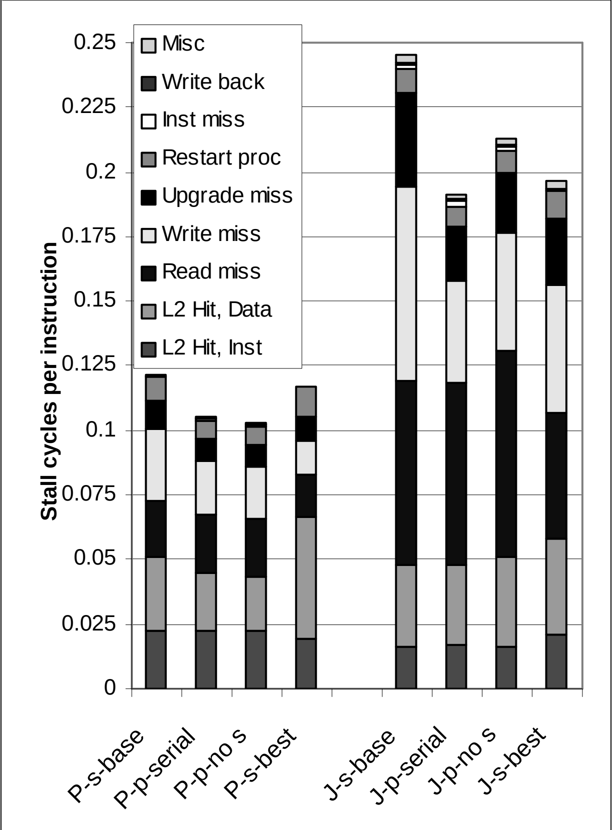


Figure 11: Stall cycles for consistency models

X-s-base:	*baseline sequential consistency (8MB, 128 B)*
X-p-serial:	*processor consistency with serializations*
X-p-no s:	*processor consistency without serializations*
X-s-best:	*sequential consistency with the highest performance cache configuration (8 MB, 512 B)*

	Perl Improve	Java Improve
Processor, serialization	13.68%	22.20%
Processor, no serialization	15.55%	13.43%
Sequential, best caches	3.38%	20.24%

Table 3: Decrease in stall cycles over baseline

mentioned before, the second level cache miss rate increases by 10% when the serializations are removed. Note however that even though the cache miss rate of processor consistency without serialization is higher than that of the baseline sequential consistency, it still manages to perform 13.43% better than sequential consistency.

5. Conclusion

In this work, using a functional execution driven full system simulator and a detailed memory model, we show that the memory system behavior of Java workloads is very different from that of Perl workloads. We find that our Java implementation of SPECweb99 suffers 169% to 213% more cache misses than the original Perl version.

Our results reveal several means for improving performance for both Perl and Java SPECweb99. First, for relatively large caches, increasing the line size is more effective for reducing the cache miss rates than increasing the size of the cache, even with very long line sizes. In addition, we find that when using relatively small caches (i.e., 1 MB second level caches), the exclusive state is effective at reducing stalls for write upgrades, by removing 24% of the upgrades for Perl and 61% of the upgrades for Java. However, the exclusive state is less useful for multiprocessor workloads with relatively large caches and is only able to remove about 1% to 5% of the upgrades. Note, however, that while the exclusive state is less useful for large caches in multiprocessor workloads, it is still important in the case of single processor workloads. Third, our simulations show that it is important to have an efficient cache to cache transfer mechanism for modified lines. For relatively large lines, 71% to 87% of the second level cache misses hit a modified line in another processor's cache for both the Java and Perl implementations of SPECweb99. Lastly, we find that relaxing the consistency model is effective at reducing the number of cycles that instructions are stalled and warrants further research. Our early results of consistency models show that the Perl CGI and the Java Servlet can improve performance by 13.68% and 22.2% by using processor consistency instead of sequential consistency. These results also show that if one attempts to optimize processor consistency by removing serialization points, one must be careful about which serialization points are removed. If one removes the wrong serializations, then the caches may suffer more cache misses, degrading overall performance.

Acknowledgments

We thank Shih-Chang Lai, Trung Diep, and the Virtutech support team for their valuable help with Simics. We wish to thank Yongjoon Lee for helping with the initial memory model. We also thank Harold Cain for the initial motivation for this work. This work was supported in part by NSF Grants CCR-0073440, CCR-0083126, and EIA-0103670.

References

[1] S. Adve, V. Pai, P. Ranganathan. "Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems." In *Proceedings of the IEEE: Special Issue on Distributed Shared Memory Systems*, pages 445-455. March 1999.

[2] H. Cain, R. Rajwar, M. Marden and M. Lipasti. "An Architectural Evaluation of Java TPC-W." In *Proceedings of The Seventh International Symposium on High-Performance Computer Architecture (HPCA-VII)*, January 2001.

[3] Q. Cao, P. Trancoso, J. Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won. "Detailed Characterization of a Quad Pentium Pro Server Running TPC-D." In *Proceedings of the Third International Symposium on High-Performance Computer Architecture (HPCA-III)*, February 1997.

[4] S. Dieckmann and U. Hölzle. "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks." In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99)*, June 1999.

[5] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramaurthy, and P. Stenström. "The Detection and Elimination of Useless Misses in Multiprocessors." In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA 20)*, May 1993.

[6] K. Gharachorloo, A. Gupta, and J. Hennessy. "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, 1991.

[7] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 1: Specification*, 1996.

[8] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker. "Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads." In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA 25)*, June 1998.

[9] J. Kim and Y. Hsu. "Memory System Behavior of Java Programs: Methodology and Analysis." In *Proceedings of the 2000 International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS 2000)*, June 2000.

[10] K. Lepak, G. Bell, and M. Lipasti. "Silent Stores and Store Value Locality." In *IEEE Transactions on Computers*, Vol. 50, No. 11, November 2001.

[11] T. Li, L. John, N. Vijaykrishnan, A. Sivasubramaniam, J. Sabarinathan and A. Murthy. "Using Complete System Simulation to Characterize SPECjvm98 Benchmarks." In *Proceedings of ACM International Conference on Supercomputing (ICS 2000)*, May 2000.

[12] Y. Luo and L. John. "Workload Characterization of Multithreaded Java Servers." In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2001)*, April 2001.

[13] M. Martin, D. Sorin, H. Cain, M. Hill, and M. Lipasti. "Correctly Implementing Value Prediction in Microprocessors that Support Multithreading or Multiprocessing." In *34th Annual International Symposium on Microarchitecture (MICRO-34)*, December 2001.

[14] V. Pai, O. Ranganathan, S. Adve and, T. Harton. "An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors." In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.

[15] R. Rajwar and J. Goodman. "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution." In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34)*, December 2001.

[16] R. Radhakrishnan, N. Vijaykrishnan, L. John, and A. Sivasubramaniam. "Architectural Issues in Java Runtime Systems." In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture (HPCA-VI)*, January 2000.

[17] P. Ranganathan, V. Pai, and S. Adve. "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models." In *Proceedings of the Ninth Symposium on Parallel Algorithms and Architecture (SPAA-9)*, June 1997.

[18] B. Rychlik and J. Shen. "Characterization of Value Locality in Java Programs." In *Proceedings of the Workshop on Workload Characterization, ICCD*, September 2000.

[19] Y. Shuf, M. Serrano, M. Gupta, and J. Singh. "Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations." In *Proceedings of the 2001 International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS 2001)*, June 2001.

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

[21] Virtutech Corporation. Simics Full System Simulator. Information available at http://www.simics.com.