On the Effectiveness of Speculative and Selective Memory Fences

Oliver Trachsel, Christoph von Praun, and Thomas R. Gross

Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland

Abstract

Memory fences inhibit the reordering of memory accesses in modern microprocessors; fences are useful to implement synchronization and strong shared memory semantics in multi-threaded programs. A naive implementation of memory fences can result in a significant performance penalty for processors with deep pipelines supporting multiple concurrent memory accesses.

The paper compares three techniques to reduce the impact of memory fences: (1) Read-speculation allows reads that follow a fence to be issued while the fence is being processed; (2) Write-ahead additionally allows writes following a fence to proceed early; (3) Selective fences distinguish between memory accesses to thread-local and shared memory and enforce ordering only among accesses to shared memory.

We evaluate and compare the effectiveness of these techniques with a simulator derived from the Pentium 4 architecture. We report data for a storage model that uses memory fences to enforce the memory semantics at monitor boundaries.

1. Introduction

Modern microprocessors overlap and reorder memory operations to smooth out the increasing delay of transferring data between memory and CPU. Constraints on reordering are defined in a *(shared) memory model* that addresses the visibility of memory actions in a multiprocessor environment with respect to the program order of statements. These rules, together with data dependences among memory references on the same processor, determine legal execution orders. For current multiprocessor architectures, the memory model is usually *weaker* than the intuitive model of sequential consistency [18], i.e., the architecture imposes fewer constraints on ordering and hence provides more leeway for reordering and overlap.

Common programming models for multi-threaded applications require guarantees of memory consistency that are stronger than those provided by modern architectures. Most inter-thread synchronization models, e.g., specify that the memory contents be consistent across threads at each point where control flow is synchronized. This combination of control flow and memory view synchronization has consequences for the compiler and/or runtime system; a point of flow synchronization implies a barrier to the reordering of memory accesses [11]. Memory models that are defined at the programming language level [3, 7, 21] are another example. Such models must control the access reordering explicitly and often require a high number of memory fences to achieve this goal.

Current processor architectures provide *memory fence* instructions (aka *serializing instructions* or *barriers*) to inhibit the internal reordering of memory accesses. According to [15, Volume 3, Chapter 7], "fence instructions provide a performance-efficient way of ensuring load and store memory ordering between routines that produce weakly-ordered results and routines that consume data". Memory fences are currently used in the context of inter-thread synchronization and they can significantly degrade execution performance [6].

Static compiler analysis and dynamic techniques have been proposed to remove unnecessary synchronization locks [1, 2, 4, 22, 26]. However, these techniques are not successful in all contexts.

This paper studies three ideas to reduce the impact of memory fences: (1) *Read-speculation*, as proposed by Gharachorloo et al. [9], which allows reads that follow a fence to be issued while the fence is being processed; (2) *Write-ahead*, used in combination with

This research was supported, in part, by a gift from the Microprocessor Research Lab (MRL) of Intel Corporation. Christoph von Praun's current address is: IBM T.J. Watson Research Center, Yorktown Heights, NY

read-speculation, which additionally allows writes following a fence to proceed early; similar techniques have been addressed by Gharachorloo et al. [8], and Hammond et al. [12]; and (3) *Selective fences*, which distinguish between memory accesses to thread-local and shared memory and enforce ordering only between accesses to shared memory[1]. We compare these techniques against a simple baseline model that does not apply any of these optimizations.

To investigate these issues, we have developed a model and simulator that are derived from the Intel Pentium 4 processor architecture. Specific emphasis is put on the aspects of the *read/write queue* (RWQ), *write buffer* (WB), and the memory subsystem. The simulation is based on execution traces of multi-threaded Java programs. We report data for a storage model that uses memory fences to enforce the memory semantics at monitor boundaries. The described optimizations reduce the cost per fence by up to 99%, with an average reduction of 83.2% for selective fences.

2. Fence characterization

The execution of a memory operation is not an instantaneous action; rather, it extends over a time interval during which the processor and memory system can carry out concurrent work (e.g., initiate further memory operations). We say that the memory operation is *issued* at the beginning of that interval and that the memory operation *has performed* at the end of that interval [5]. During the interval, the result of a read operation is determined, and the value of a write operation is made available to other processors. There are several features of microprocessors that cause reordering of memory accesses:

- *Write buffering* allows read operations that follow a write to be hoisted above the write;
- *Write combining* allows writes to bypass other writes;
- *Non-blocking reads* allow reads and writes to be hoisted above a pending read operation.

A fence sets a bound on the start and/or end of the execution intervals of memory operations that surround the fence. Fences control the behavior of the processor on which they execute and do not influence the ordering of memory operations that are executed by other processors. There are several variants of fences, and they can be classified according to the types of memory operations they affect [19].

- *read-read fence (lfence)*: Reads following the lfence are not issued before preceding reads have completed; an lfence disables reordering due to non-blocking reads.
- *write-write fence (sfence)*: An sfence ensures that writes are performed in order; it disables reordering due to write combining.
- *read-write fence*: This fence disables an optimization that allows a write operation to complete earlier than a pending read. For performance reasons, processors typically try to execute reads early and delay the completion of writes; hence most common processor architectures (including SPARC and IA-32) obey this ordering without mention of a fence.
- *write-read fence*: A write-read fence ensures that reads following writes obtain values that are observed after the writes become visible to other processors. In particular, such a fence ensures that reads that follow a write to the same location can observe a value that is loaded from memory, not from the WB. This fence disables write buffering.
- *memory fence (mfence)*: All memory operations that precede an mfence must have performed. Memory operations that follow an mfence must not be issued before the mfence; the mfence combines the reordering constraints of all other fence types.

3. Processor and fence model

We use a simulator to evaluate the performance of fences and the effect of the optimizations that we study. This section explains the underlying processor model with a detailed discussion of the memory hierarchy. We also describe the four implementation variants of fences (one baseline model, three optimizations).

The model abstracts from the details of the execution unit as follows. The execution cost of operations other than memory accesses is estimated using approximate throughput information. We use the initiation interval as provided by [14, Appendix C] as an estimate of the throughput capacity.

3.1. Execution path of memory operations

The execution of a memory operation involves several architectural components that are illustrated in

[1] There is anecdotal evidence that this technique has been considered before, but we are not aware of any publication on this topic.

Figure 1. The execution proceeds along the following steps. We use the terms *issued* and *performed* as defined by Dubois et al. [5].

1. Memory operations are fetched from the instruction stream and entered into the RWQ, which keeps track of all currently performing memory accesses; there can be several memory operations per instruction. If the RWQ is full, the execution stalls. Operations enter the RWQ in the order of their occurrence in the instruction stream at a maximal rate of one operation per cycle.

2. Write operations are given to the memory system, i.e., *issued*, in order. A read operation in the RWQ is issued as soon as all register- or address dependences with earlier memory accesses are met.

3. When issued, reads and writes are handed over to the L1 subsystem; writes are, in addition, entered into the WB. If a read does not hit in L1, the data request is handed over to the L2 subunit and subsequently (in case of an L2 miss) to the main memory system.

4. A read operation *has performed* if its value is available. A write operation *has performed* as soon as it appears in the write buffer. Accesses that are issued and have not performed are said to be *pending*.

5. Operations that have performed are removed from the RWQ in the order of their occurrence in the instruction stream. An operation that is removed from the RWQ is said to *retire*.

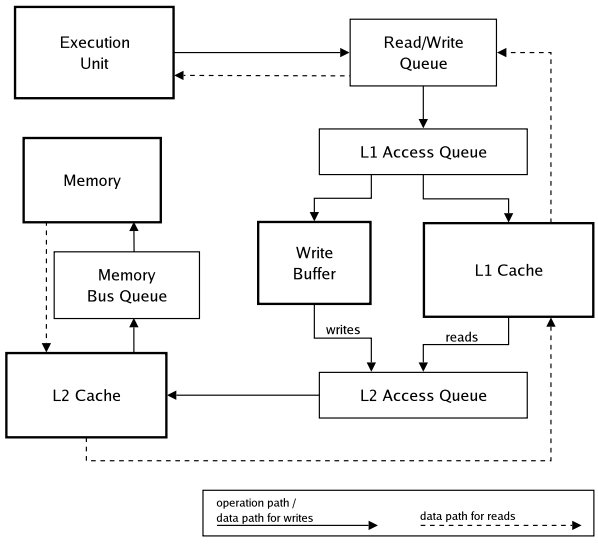

Figure 1. Architectural components of the processor and memory system.

Unit	*Parameters*
L1 cache	2/6 cycles integer/float latency, 1 new access per cycle, hits with probability $p1$
L2 cache	7 cycles latency, 1 new access every other cycle, hits with probability $p2$
Memory	50 cycles latency, new access every 12 cycles
Write buffer(s)	one 24-entry buffer or two separate 12-entry buffers for thread-local and shared data, each entry 64 bytes wide, retire-at-20 (one buffer) or retire-at-8 (two buffers), 50000 cycles maximal reside time, full entries retire in bursts, partials in 8 bytes chunks

Table 1. Summary of the cache, memory, and write buffer configuration. The cache and memory access times and the buffer geometry are similar to those of Pentium 4 processors.

3.2. Caches and write buffer

We use a two-level cache hierarchy, as shown in Figure 1. The parameters of caches and WB are summarized in Table 1. L1 and L2 are statistically modeled; the benchmark-specific hit rates are measured on a real machine. The access latency to L1 is 2 cycles for integer data and 6 cycles for floating point data, following [13]. A new access can be issued every cycle. L2 has an access latency of 7 cycles, and a new access can be issued every other cycle. The memory latency is 50 CPU cycles and a new memory access can be issued every 12 cycles (bus transaction time). These numbers are taken as an average from [14, Chapter 1]. If a write misses, no data is fetched from the next cache level or from memory.

L1 has write-through semantics, i.e., every write is propagated to L2. The WB is responsible for passing on the writes to L2: The WB avoids the blocking of the execution unit when writes are issued faster than L2 is able to handle them; moreover, the WB reduces the internal bus traffic by combining writes to the same cache block. Many writes are word-sized (4 bytes), so writes to the same block address are combined into the same WB entry, which has the same width as an L1

cache block (64 bytes).

Blocks are transferred from the WB to L2 according to a retirement strategy that we formulate according to Skadron and Clark [23]: In our model, the entry with the oldest data is retired if there are 20 or more valid entries in the buffer (*retire-at-20*), or if the entry contains data older than 50,000 cycles.

To retire an entry, the WB arbitrates for access to L2 and writes the valid data. If every byte of the entry is dirty, then all 64 bytes are written out in one burst access. If some bytes have not been written to, then the data is retired in 8 byte chunks. Both transfers, a burst- or a chunk-write, to L2 take 7 cycles. When the WB is full, it can cause processor stalls because no more writes can be issued until the buffer has written an old entry to L2. New writes can be combined with all entries except those currently being retired or marked for retirement.

An important characteristic of a combining WB is that it may reorder writes when combining a write with an existing entry in the buffer [23]. Our model allows for this reordering and is therefore somewhat more general than common IA-32 implementations, which do not reorder writes. In addition, our model allows reads to bypass writes that reside in the WB. Due to this reordering, the WB plays a crucial role for memory fences. The buffer needs to be flushed when strict ordering of reads and writes is mandated.

Our processor model supports two separate WBs (both having the same size) to separate writes to thread-local memory from those to shared memory. On a fence, it is only necessary to flush the buffer with shared data. The duration of a flush may thereby be reduced.

3.3. Implementation of fences

This section presents four implementation alternatives of fences, a baseline model and three variants (read-speculation, write-ahead, and selective fences). Each model is discussed in detail in the following sections.

Fences affect the operation of two architectural components that implement the reordering of memory accesses: (1) the RWQ that allows for processing multiple concurrent memory operations, and (2) the WB. The presentation focuses on a memory fence (mfence) because all issues arise here; other fence variants (Section 2) can be optimized accordingly.

(a) Baseline. On a memory fence, the processor unwinds the processor pipeline to make sure that all preceding memory operations have performed at the memory system; at the memory interface, all pending

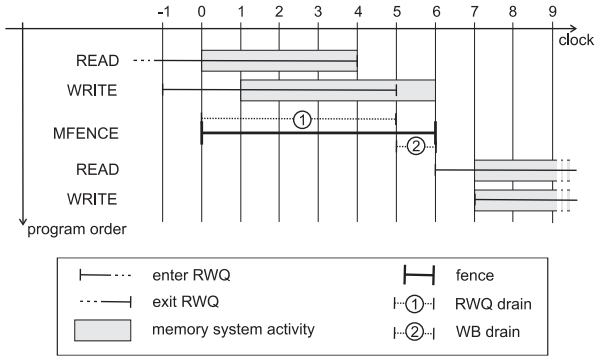

Figure 2. Execution scenario under the baseline model.

memory operations are drained from the RWQ; then the WB is flushed to memory. Operations that follow the fence are not issued until the fence has completed.

Figure 2 shows the execution of the instruction sequence (`read, write, mfence, read, write`) in the baseline model.

Read and write operations enter the RWQ in program-order; at most one operation is entered per clock tick; several operations may be removed at a time. The gray bars specify the period during which an operation is pending, i.e., active in the memory system. The fence execution happens in two phases: During the first phase, all operations are drained from the RWQ. During the second phase, writes are flushed from the WB to L2. The phases may overlap; the second phase starts as soon as there are no more writes in the RWQ. The read and write operations succeeding the fence are issued only once the two phases have completed.

(b) Read-speculation. The read-speculation model aggressively issues reads that follow a fence while the fence is active. Such reads are marked speculative and must be redone if L2 coherence traffic for the concerned address is observed before the fence finishes. As the speculation period is typically small, i.e., at most the duration of a fence, the complexity of auxiliary hardware structures to track the speculative execution is moderate. Speculative execution was first introduced by Gharachorloo et al. [9].

In the scenario of Figure 3(a), the read instruction following the fence is issued speculatively while the write is delayed.

(c) Write-ahead. At the occurrence of a fence, subsequent writes can proceed and may even be entered into the WB as long as they are not combined with writes issued before the fence. Similar techniques have been addressed by Gharachorloo et al. [8], and Hammond et al. [12].

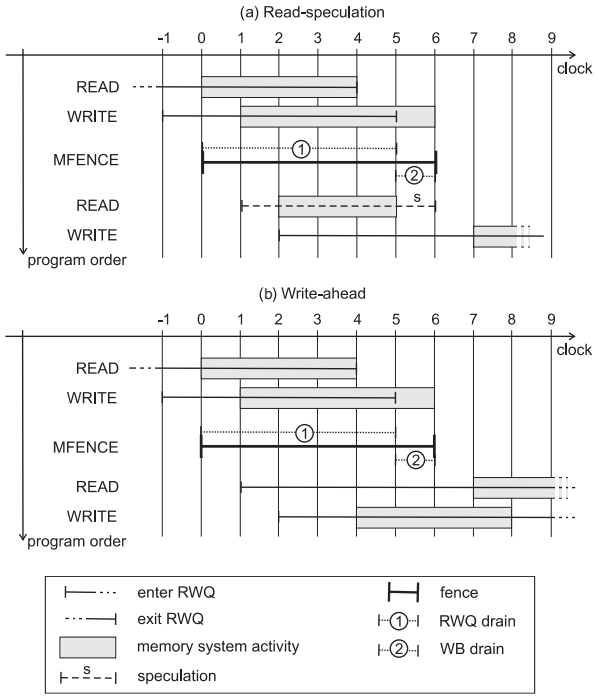

Figure 3. Execution scenario under the read-speculation and write-ahead models.

In Figure 3(b), both memory operations that follow the fence enter the RWQ; provided that there are no data dependences between the read and the write, the write can proceed early, while the read is not issued to the memory system until the fence is performed.

Write-ahead and read-speculation can be combined. In this case, writes that depend on speculative reads may need to be undone if a mis-speculation on that read is detected. This is, however, not a problem, since writes that occur during the activity of a fence remain in the WB and can be simply undone during rollback. For the evaluation, we consider only this combined model, because write-ahead reveals no significant improvement of fence performance when applied on its own.

(d) Selective fences. Selective fences affect only a subset of memory operations: operations that target shared memory follow the rules of the fence; operations that operate on thread-local memory remain unaffected.

In Figure 4, local memory operations that follow the fence proceed while the fence is active. We assume that there is no data dependence between the read and the subsequent write operations. Note that operations that target shared data enter the RWQ but are delayed, i.e., not issued to the memory system, until the fence

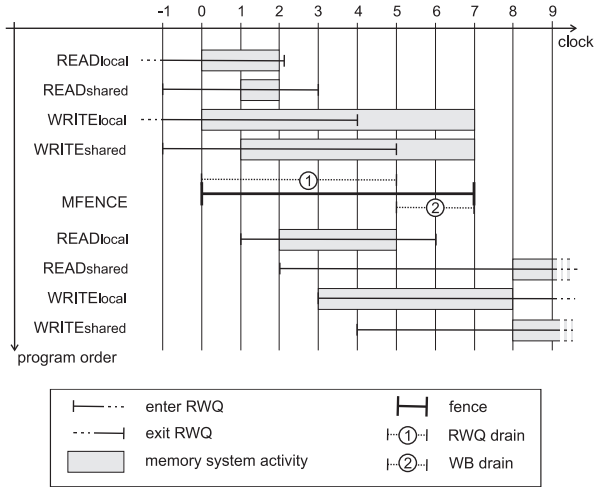

Figure 4. Execution scenario with selective fences.

completes.

There are two factors that make the selective fence model more efficient than the baseline model:

1. When flushing the WB, only entries to *shared memory* are evicted. Entries that involve shared memory can either be kept in a separate buffer (as in our model) or can be marked in a single buffer. The benefit is a reduction of the flush time.

2. When executing a fence, memory accesses to *thread-local* memory can proceed during the execution of the fence. This refinement re-enables ILP to a certain degree, as operations that depend only on thread-local data need no longer be blocked during a fence.

4. Methodology

Our experimental setup consists of a compiler, a trace collector, and the simulator.

We extended an ahead-of-time Java compiler [17] by a code instrumenter to enable trace collection during execution of the compiled programs. The generated traces contain all necessary information about memory accesses, fences, and non-memory instructions. We simulate these traces with an event-based processor simulator that implements the processor model from Section 3.

5. Evaluation

The first part of this section discusses one application scenario of fences. We then present the benchmarks used for evaluation along with statistics about memory access and fence usage. The succeeding subsection discusses the improvements that can be obtained by the optimized fence models. Finally, we make some general comments on memory access disambiguation.

5.1. Application scenario

An important application of fences is to implement monitors; monitors are the predominant synchronization mechanism in Java and C#, and hence an efficient handling of fences is important to obtain good performance.

A common model for inter-thread synchronization is to synchronize both the execution and memory view simultaneously. In Java, for example, the basic monitor semantics are, besides the effect on the progress of the calling process, to provide an *acquire* semantics at `monitorenter` and *release* semantics at `monitorexit` [20]. This combination means that accesses to shared variables inside a monitor must be performed *after* the acquire and before the *release*.

We compare four models for bidirectional fences. The four models differ in the kind of operations that are constrained by the fence. In the simulation, acquire-release semantics are established though the use of mfence (Section 2) at monitor boundaries.

5.2. Benchmark characterization

The following benchmarks are used for the evaluation:

- sor and sparse from the Java Grande Forum multi-threaded benchmark suite [16]. The data sizes of the benchmarks have been reduced to shorten their execution time.
- mtrt: A multi-threaded raytracer from the SPEC JVM98 suite [24].
- tsp: A TSP solver ported from C.
- jigsaw: The web-server of the W3 consortium in version 1.0 [27].
- specjbb: The SPEC JBB2000 Java business benchmark [25].

	L1 hit rate	L2 hit rate
sor	97.7%	95.3%
sparse	90.1%	98.2%
mtrt	96.8%	88.3%
tsp	99.5%	60.8%
jigsaw	96.0%	82.9%
specjbb	96.9%	77.0%

Table 2. Cache hit rates measured on the real CPU (Intel Xeon with a 256KB L2).

	total [10^6]	*stack*	*R/W-ratio*	*accesses per fence*
sor	537	79.4%	4.5	1,491
sparse	165	72.1%	3.2	1,865
mtrt	282	80.3%	1.5	140
tsp	166	62.6%	3.5	152,782
jigsaw	234	85.4%	1.5	656
specjbb	2604	82.8%	1.3	84

Table 3. Memory access statistics of the benchmarks. Accesses to the stack are thread-local. All heap accesses are assumed to go to shared memory.

The L1 and L2 hit rates are shown in Table 2. They are measured on the real CPU and used as input parameters for the simulations.

Table 3 shows the memory access and fence statistics of the benchmarks. tsp has almost no thread synchronization. This fact is indicated by the high number of memory accesses per fence. Benchmarks sor and sparse use moderate synchronization, and the remaining three benchmarks have high synchronization density. All accesses to the stack are thread-local (over 70% for all benchmarks); all heap accesses are assumed to go to shared memory.

5.3. Fence performance

We evaluate the fence performance and compare the different fence models. Table 4 shows for the baseline model how cycles during which a fence is active are partitioned among the main phases (draining of the RWQ and draining of the WB).

For most benchmarks, the draining of the RWQ corresponds to about 30% of the fence time. An exception to this is tsp, for which the WB draining takes longer (but for which the total fence time corresponds to less than 0.1% of total execution time).

Table 5 shows the runtime overhead due to the

	fence time	*RWQ-drain*	*WB-drain*
sor	1.5%	32.4%	67.6%
sparse	1.4%	31.1%	68.9%
mtrt	15.0%	29.7%	70.3%
tsp	<0.1%	15.8%	84.2%
jigsaw	4.1%	27.7%	72.3%
specjbb	31.1%	30.7%	69.3%

Table 4. The time relative to the total execution time during which a fence was active and the partition between RWQ-drain and WB-drain of that time.

	base-line	*selective fences*	*read-spec.*	*read-spec.+ write-ahead*
sor	0.8%	<0.1%	0.2%	<0.1%
sparse	0.8%	<0.1%	0.1%	0.1%
mtrt	13.5%	3.8%	7.9%	6.1%
tsp	<0.1%	<0.1%	<0.1%	<0.1%
jigsaw	3.1%	0.7%	1.9%	1.4%
specjbb	21.5%	2.3%	8.7%	4.3%

Table 5. Runtime overhead due to the presence of fences under the different models.

	baseline	*selective fences*		*read-spec.*		*read-spec.+ write-ahead*	
sor	21.1	3.2	84.9%	5.2	75.1%	1.0	95.3%
sparse	26.3	0.1	99.8%	8.0	69.4%	4.7	82.3%
mtrt	28.9	8.1	71.9%	16.9	41.7%	13.0	55.2%
tsp	38.5	9.1	76.4%	26.1	32.0%	17.4	54.9%
jigsaw	29.3	6.8	76.8%	17.5	40.1%	12.9	55.9%
specjbb	29.5	3.1	89.3%	11.9	59.7%	5.9	80.1%

Table 6. Average cost of a fence in clock cycles and the fence cost reduction relative to the baseline model.

presence of fences under four different fence models. The overhead is calculated as the increase in execution time relative to a hypothetical model where fences would cost nothing. Table 6 shows the average cost per fence in number of clock cycles. We assume that read-speculation never fails and hence do not account for rollback cost. Gniady et al. [10] showed that misspeculations are indeed very infrequent in well-synchronized applications.

We can group the results into three classes: A first class is formed by tsp, sor and sparse, for which the impact of fences on the execution is below 1%. We report these benchmarks, even though the absolute performance gain through optimized fences is small, because we are interested in the reduction of the cost per fence. If the cost of a fence can be significantly reduced by the improved fence models for different applications and memory access patterns around fences, it is of course most beneficial for applications which make frequent use of fences.

A second class is formed by jigsaw, for which 3% of the total execution time is attributed to fences.

mtrt and specjbb form the third class with the most significant overhead due to fence instructions: more than 10% of the total execution time is due to memory fences under the baseline model.

The results in Table 6 show that, compared to the other models, selective fences achieve the highest reduction of fence cost for all benchmarks except sor. The relative improvement lies between 71.9% for mtrt and more than 99% for sparse with an average of 83.2%.

The speculative-read model also results in a remarkable improvement of fence performance, although less than with selective fences. The improvement varies from 32.0% for tsp to 75.1% for sor with an average of 53.0%. There are two limiting bottlenecks for this model: (1) The maximal refill rate of the RWQ which is one access (read or write) per clock cycle in our model, and (2) reads that miss in L1 must wait for the WB drain to finish because eviction accesses are prioritized at L2.

The first bottleneck can partly be avoided by combining write-ahead and read-speculation. The corresponding numbers are given in the last column of Table 6. The reduction of the mean cost per fence increases to 54.9%–95.3% (tsp and sor, respectively), with an average of 70.6%;

An important metric for selective fences is the number of thread-local accesses that surround a fence. If there are no, or only very few, accesses to non-shared

	Local acc. per fence	*Cost red.*
tsp	103186	76.4%
sparse	1408	99.8%
sor	1217	84.9%
jigsaw	648	76.8%
mtrt	139	71.9%
specjbb	80	89.3%

Table 7. Thread-local accesses per fence vs. fence cost reduction for the selective fence model.

	R/W-ratio	*Cost red.*
sor	4.5	75.1%
tsp	3.5	32.0%
sparse	3.2	69.4%
jigsaw	1.5	40.1%
mtrt	1.5	41.7%
specjbb	1.3	59.7%

Table 8. Ratio between reads and writes vs. fence cost reduction for the model with read-speculation.

memory before and after a fence, then the performance gain with the selective fence model is small. If, on the other hand, most accesses go to thread-local memory and accesses to shared memory are sporadic, then the full potential of selective fences can be exploited. Table 7 provides another view of this issue; it shows the average number of local accesses per fence as well as the relative fence cost reduction.

The ratio between reads and writes is a similar metric to indicate the effectiveness of the model with speculative-reads. Generally, the more reads per write, the higher the potential for speculative-reads. Table 8 contains the corresponding data.

5.4. Discussion

Although the compiler we use has already implemented a more advanced shared/local disambiguation scheme, we do not take advantage of it in our tests; instead, the complete heap of an application is assumed to be thread-shared. The performance improvement could be even higher if the differentiation between thread-local and thread-shared for heap accesses that is determined by the compiler were taken into account.

Today's microprocessors already contain mechanisms to declare different properties for distinct memory regions (e.g., memory type range registers [15, Volume 3]) which could provide the foundation for address-based memory disambiguation.

The implementation of sequential consistency through a compiler at the programming language level [6, 7] requires the selective insertion of memory fences at ordinary memory access operations (in addition to fences accompanying synchronization operations). These additional fences can result in an even higher performance degradation than the application scenario we consider in this paper.

6. Concluding remarks

The efficient implementation of fences will become more important in future generations of microprocessors as the gap between processor cycle time and memory access time widens. In current application scenarios, e.g., thread synchronization with fences at monitor boundaries, fences can occur frequently and hence their optimization is important to the overall execution performance of a parallel (multi-threaded) program. On today's processors, fences can be expensive, with the corresponding negative impact on the execution time of parallel programs.

However, memory fences need not be expensive: this paper studies three practical optimizations that do not require expensive hardware extensions and that reduce the execution time of fences by up to 99% compared to a non-optimized implementation. The optimizations pursue two principal strategies: First, write-ahead and read-speculation aim at advancing the execution window past a fence while the fence is active; they are very effective and reduce fence cycle time by an average of 70.6%. Second, selective fences differentiate thread-local and shared data and constrain only the execution order among operations on shared data; even using the most rudimentary heuristic for disambiguation yields an average reduction of 83.2%, making this the most effective technique we studied.

Multi-threaded programs are likely to increase in importance as new languages include appropriate abstractions; in addition, the spread of multiprocessor and multicore machines will further encourage the development of multi-threaded programs. Since fences are a key component of many synchronization constructs and language-level memory consistency models, improving their efficiency is crucial to providing a suitable platform for the execution of parallel programs.

Acknowledgments We thank Niko Matsakis and the anonymous reviewers for their detailed and insightful comments.

References

[1] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proc. of Static Analysis Symposium (SAS'99)*, pages 19–38, Sept. 1999.

[2] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 35–46, Nov. 1999.

[3] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. ICC++ — A C++ dialect for high performance parallel computing. In *2nd Intl. Symp. on Object Technologies for Advanced Software (ISOTAS)*, pages 190–205, Mar. 1996.

[4] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 1–19, Nov. 1999.

[5] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of the 13th Intl. Symp. on Computer Architecture (ISCA'86)*, pages 434–442, 1986.

[6] X. Fang. Inserting fences to guarantee sequential consistency. Technical Report MSU-CSE-02-27, Department of Computer Science, Michigan State University, East Lansing, Michigan, December 2002.

[7] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proc. of the 17th Intl. Conf. on Supercomputing (ICS'03)*, pages 285–294, 2003.

[8] 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 Systems (ASPLOS'91)*, pages 245–257, 1991.

[9] 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 (ICPP'91)*, pages 355–364, 1991.

[10] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Intl. Symp. on Computer Architecture (ISCA'99)*, pages 162–171, 1999.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 2nd Edition.* Addison-Wesley, 2000.

[12] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. of the 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 58–69, 1998.

[13] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, (1), Feb. 2001.

[14] Intel Corporation. IA-32 Intel Architecture Optimization Reference Manual. http://www.intel.com/products/processor/pentium4, June 2005.

[15] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual, Volumes 1-3. http://www.intel.com/products/processor/pentium4, Jan. 2006.

[16] Java Grande Forum. Multi-threaded benchmark suite. http://www.epcc.ed.ac.uk/javagrande/, 1999.

[17] Laboratory for Software Technology (LST), ETH Zurich. ERCO – ETH Research Compiler Framework, 2004.

[18] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. on Computers*, 46(7):779–782, July 1997.

[19] D. Lea. The JSR-133 cookbook for compiler writers. http://gee.cs.oswego.edu/dl/jmm/cookbook.html, 2005.

[20] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd Symp. on Principles of Programming Languages*, pages 378–391, 2005.

[21] S. Midkiff, J. Lee, and D. Padua. A compiler for multiple memory models. In *Rec. Workshop Compilers for Parallel Computers (CPC'01)*, June 2001.

[22] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. of the 34th Intl. Symp. on Microarchitecture (MICRO'01)*, pages 294–305, 2001.

[23] K. Skadron and D. Clark. Design issues and tradeoffs for write buffers. In *Proc. 3rd Intl. Symp. on High-Performance Computer Architecture (HPCA'97)*, pages 144–155, Feb. 1997.

[24] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

[25] The Standard Performance Evaluation Corporation. SPEC JBB2000 Benchmark. http://www.spec.org/osg/jbb2000, 2000.

[26] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 187–206, Nov. 1999.

[27] World Wide Web Consortium (W3C). Jigsaw Webserver. http://www.w3c.org/Jigsaw/.