Barriers: Friend or Foe?

Stephen M Blackburn
Department of Computer Science
Australia National University
Canberra, ACT, 0200, Australia
Steve.Blackburn@anu.edu.au

Antony L Hosking
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
hosking@cs.purdue.edu

ABSTRACT

Modern garbage collectors rely on read and write barriers imposed on heap accesses by the mutator, to keep track of references between different regions of the garbage collected heap, and to synchronize actions of the mutator with those of the collector. It has been a long-standing untested assumption that barriers impose significant overhead to garbage-collected applications. As a result, researchers have devoted effort to development of optimization approaches for elimination of unnecessary barriers, or proposed new algorithms for garbage collection that avoid the need for barriers while retaining the capability for independent collection of heap partitions. On the basis of the results presented here, we dispel the assumption that barrier overhead should be a primary motivator for such efforts.

We present a methodology for precise measurement of mutator overheads for barriers associated with mutator heap accesses. We provide a taxonomy of different styles of barrier and measure the cost of a range of popular barriers used for different garbage collectors within Jikes RVM. Our results demonstrate that barriers impose surprisingly low cost on the mutator, though results vary by architecture. We found that the average overhead for a reasonable generational write barrier was less than 2% on average, and less than 6% in the worst case. Furthermore, we found that the average overhead of a read barrier consisting of just an unconditional mask of the low order bits read on the PowerPC was only 0.85%, while on the AMD it was 8.05%. With both read and write barriers, we found that second order locality effects were sometimes more important than the overhead of the barriers themselves, leading to counter-intuitive speedups in a number of situations.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection), run-time environments*

General Terms

Languages, design, performance, algorithms

Keywords

Write barriers, memory management, garbage collection, Java

ISMM'04, October 24–25, 2004, Vancouver, British Columbia, Canada.

1. Introduction

Modern garbage collectors rely heavily on mechanisms to monitor accesses by the mutator to objects in the garbage-collected heap. Commonly referred to as *read* or *write barriers*, they encapsulate actions to be performed whenever the mutator reads/writes a *reference* from/to some field of a heap object. Typical actions include recording the accessed object or object field, perhaps conditionally with respect to the object/field and the reference itself. The information recorded by a barrier can be used to partition the heap into separately-collected regions, and to synchronize actions of the mutator and the garbage collector. A general overview of such barriers and their use in garbage collection can be found in Jones and Lins [20].

The impact of different barriers on *garbage collector* performance and on overall application performance has been widely studied [30, 12, 26, 31, 2, 16, 15, 14, 4, 29, 9]. Questions regarding the *form* of the barrier (e.g., conditional, inlined) and the impact of such decisions on mutator code quality and compile times have also been studied [6]. There have been scattered measurements of direct barrier overheads: Zorn reported write barrier overheads of 2-6% and read barrier costs of up to 20%; Bacon et al. [5] report the cost of their Brooks-style [11] indirection-based read barrier as 4% on average and 10% maximum for SPECjvm98. Thus, it has been widely assumed that barriers impose significant overhead on the mutator. This assumption has led to efforts to reduce barrier overheads using approaches based on static analysis and compiler optimization [10, 17, 33, 5]. Others have cited barrier cost to justify the development of new garbage collection algorithms that eliminate the need for barriers while preserving garbage collection properties such as partitioned collection of the heap [13].

What has been lacking is a comprehensive and comparative evaluation of precise barrier costs on modern processors. Understanding the costs of the barriers is important for implementors, since barriers impact aspects of the compiler, optimizer, and run-time system. Barrier costs are also important to algorithm designers, since the style of barrier can directly influence algorithmic design choices (e.g., copying, generational, concurrent, parallel). The results we report here reveal that, depending on the hardware platform, carefully engineered read and write barriers typically consume a minimal fraction of total execution time. Our results have two-fold impact: language implementors can choose among barriers depending on their target platform and the GC algorithm they must support, while algorithm designers can judiciously incorporate barrier requirements into their algorithms without fear of unnecessarily expensive mutator overheads. Moreover, the evaluation methodology we use is one that other implementors should be encouraged to replicate for evaluation of barriers within their own systems.

1.1 Our contributions

Our specific contributions include:

- An evaluation methodology allowing meaningful comparison of the precise overheads of different barrier implementations. We supplement the reachability information contained in remembered sets with exact reachability information, thus allowing us to *ignore* remembered sets while holding all other elements of program behavior constant. This allows us to include or exclude barriers without impacting the correctness of the collector.
- Implementation of this methodology for MMTk and Jikes RVM.
- Evaluation of a range of barriers for standard Java benchmarks on a number of platforms, revealing that certain barriers have overheads that are a minimal fraction of total execution time. We implement standard write barriers and a number of read barriers that are representative of broad classes of read barrier.

2. Related work

There are three broad areas of prior work related to the study we present here. The most direct precursors are studies of mutator barrier overheads. Zorn [34] gathered heap access profiles for several large Lisp programs, in the form of counts for the access events to which barriers need to be applied. Based on these counts, and by timing the cost of postulated barriers within a tight loop on the SPARC, MIPS, and MC68020 processors, Zorn was able to come up with an estimate of the total cost of the barriers per application and platform. By measuring total execution time for the benchmarks on each platform (without the overhead of profiling) he was able to calculate the barrier overhead as a fraction of total execution time. We note that this measure is not necessarily accurate, since it does not consider the in-place effects of the barriers. On modern processors, these are magnified by secondary compiler/hardware effects such as register pressure, branch prediction, cache locality, etc. We do measure these effects. Zorn's measured overheads range from 2-6% for inlined fast-path write barriers and $< 20\%$ for read barriers. Since Lisp is untyped, his barriers must also filter for non-pointer stores. Our results for Java reveal the impact of static disambiguation of non-pointer stores from pointer stores, showing that a reasonable generational barrier can have average mutator overhead under 2%. Interestingly, our worst case overhead is just under 6%, so the maximum bound is consistent with Zorn's. However, we are operating in a *fully compiled* setting (Jikes RVM), whereas Zorn's work was based on Franz Lisp which features interpretation. Moreover, our *average* overhead is 2%, while some benchmarks experience almost no overhead.

Bacon et al. [5] report the overhead of their Brooks-style [11] indirection-based read barrier as 4% on average and 10% maximum for SPECjvm98. Note that to achieve this result Bacon et al. apply a number of optimizations including those naturally performed by the Jikes RVM compiler (e.g., common subexpression elimination), as well as special-purpose optimizations like barrier-sinking, in which the barrier is moved to its point of use, which allows the null-check required by the read access to be combined with the null-check required by the barrier. Our results do not include any special-purpose optimizations other than those the Jikes RVM compiler already applies.

A broader area of related work includes studies of the primary effects of different barriers: that is, on the collector, or on overall performance itself [30, 12, 26, 31, 2, 16, 15, 14, 4, 29, 9]. Several of these consider use of hardware-supported barriers (e.g., using virtual memory protection primitives supported by the operating system), but find such approaches generally too expensive, both because of the coarse granularity of virtual memory pages and the high cost of fielding the protection traps using user-level signal handlers. Here, we focus solely on software barriers, which none of these prior studies have accurately measured as a fraction of mutator time. Other work looks at the impact of barriers on compile times and code quality [6].

Finally, we also mention that the spectre of high barrier costs has driven diverse work on elimination of barriers through compiler optimization [10, 17, 33, 5] or synthesis of new collector algorithms that forgo reliance on barriers [13]. Our results question the motivation for such efforts (if their point is only to avoid barrier costs), since we show such costs to be surprisingly low.

3. Methodology

Perhaps the reason why write barrier performance has been the subject of so much speculation rather than evaluation is the lack of a suitable methodology. The only prior attempt to measure barrier performance we are aware of [34] used a combination of simulation and measurement of barriers in tight loops. By contrast, our methodology allows *in vivo* measurement of the barrier. This is significant, because as we shall show, the interplay between barrier code, its surrounding context, and the compiler can be subtle and unexpected.

Note that we do not measure the impact of different barriers on the compiler (see Blackburn and McKinley [6]), nor do we measure the indirect impact a choice of barrier may have on collection time (some barriers trade barrier simplicity for extra work at collection time). Our focus is solely on the mutator performance of the barriers themselves. We now describe garbage collector and compiler configurations that allow us to measure this.

3.1 Ignoring remembered sets

To achieve *in vivo* measurement, we want to be able to remove a barrier from its natural environment and then compare the mutator performance with and without the barrier in place. Our focus is on generational collectors, where write barriers are used to identify and remember pointers into the nursery so that the nursery collection can be performed in isolation. Pointer sources can be remembered with a variety of different mechanisms, with varying degrees of precision, including remembering *cards* (regions of memory that may contain pointers into the nursery), *objects* (objects that may contain pointers into the nursery), and *slots* (addresses that may contain a pointer into the nursery).

Our approach is simple. We augment nursery collection with a trace of the entire heap so that we can identify those objects in the nursery that are live (via reachability rather than via remembered sets). This allows us to dispense with the remembered data or not use a barrier at all, and still correctly collect the nursery.

We only *trace* for liveness at nursery collections, we do not actually *collect* the entire heap. We perform full heap collections according to the usual regime (e.g., when the heap is full), so the space utilization and movement of objects is as faithful as possible to the original generational collector. The only point of difference is that in the original generational collector mature space objects are conservatively assumed to be live, so excess retention occurs when a nursery object is pointed to by a dead (but uncollected) mature space object. There is no excess retention in our collector.[1]

[1] Incidentally, our collector could therefore be used to quantify the extent of excess retention.

Of course this collector has a substantial overhead at collection time, undermining the very purpose of a generational collector. However, our goal here is not to produce an efficient collector but to develop a methodology for measuring mutator performance, so this overhead of no consequence. A potential source of concern is the effect major collections might impart on the mutator through their impact on the memory hierarchy through purging of caches. However, consistent with previous results [8] we did not see any such effect.[2]

Because we are only concerned with *mutator* performance, we perform all of our experiments at a single large heap size (700MB). We use a bounded nursery of 4MB. The survival rates of our benchmarks are not sufficient to trigger a full heap collection under these circumstance. Since there is never pressure on the heap size, the nursery always remains at its upper bound of 4MB.

This mechanism is publicly available as part of MMTk, allowing the research community to apply our methodology easily in evaluating other barriers.

3.2 Pseudo-adaptive compilation

The compiler is a key factor in barrier performance, although measuring the impact of different compilers on barrier performance is beyond the scope of this work. Instead our goal is to measure the impact of barriers in a realistic setting within one high performance virtual machine.[3] We use Jikes RVM and a deterministic variant of its adaptive optimizing compiler [1]. As with most modern virtual machines, the Jikes RVM compiler focuses its effort, applying the heaviest optimizations to the most frequently executed code [3]. While it is possible to fully optimize all code in Jikes RVM, we focus on an adaptive compilation mix on the grounds that it is more realistic. We measured both and found that while full optimization leads to measurable overall improvements in code performance, the relative impact of the various barriers was not significantly different from that in the adaptive compilation setting.

Unfortunately Jikes RVM's adaptive compilation is not deterministic, as it uses timer-based sampling to identify hot methods. We circumvent this with the *pseudo-adaptive* driver for the Jikes RVM compiler, which applies the optimizing compiler to code according to an advice file generated ahead of time.[4] The pseudo-adaptive compiler thus mimics the adaptive compiler in a deterministic manner. We generate advice by running each benchmark five times while logging compiler decisions. We then use the log from the fastest of the five runs as advice during timing runs of the benchmarks.

Because we are only interested in the impact of the barriers on the mutator performance, not their impact on the compiler, we want to exclude compilation costs from our measurements. We therefore perform two iterations of each benchmark, measuring only the second iteration (both timers and performance counters are started only at the beginning of the second iteration).

4. Barrier implementations

We have implemented a range of popular write barriers that vary in what information they record, and in what in-line filtering they apply to each store to decide whether it generates new information of interest to the collector. We also consider representative (if not comprehensive) read barrier variants: one that unconditionally masks out low-order tag bits from references as they are accessed, and the other that conditionally tests whether any tag bit is set. All of the write barriers except for the card marking approach, record their pertinent information into a sequential store buffer (SSB) [2]. The SSB is updated out-of-line in a sub-routine when a given store passes its in-line filter. We use the term *fast-path* to refer to the the in-line portion of the barrier, including the filter but excluding the code to dispatch the call. The term *slow-path* refers to both the in-line dispatch code, and the called method. The fast-path represents the primary mutator overhead of any barrier. The frequency of slow-path execution directly impacts secondary overhead.

MMTk has recently implemented an optimized array copy barrier that significantly improves performance on benchmarks such as `_202_jess` which perform a substantial number of array copies. All of our write barriers exploit that optimization.

We now detail the various barriers and their relative advantages and disadvantages. Figure 1 shows the code skeleton for the MMTk write barrier, while Figure 4 shows the code skeleton for the read barrier. Note that this code is the barrier fast-path, generated in-line by the Jikes RVM optimizing compiler (which the Jikes RVM idiom `VM_PragmaInline` enforces). The parameters to the write barrier include the source object being modified (`src`), the location within that object to which the store occurs (`slot`), the target reference being stored (`tgt`), and an integer parameter (`mode`) indicating what kind of store is being performed. Note also that the write barrier is what we call a *substituting* barrier: the barrier itself is responsible for effecting the store to the appropriate location. The read barrier is also a substituting barrier with a similar interface.

4.1 Write barriers

The various write barrier implementations we evaluate here are given in Figure 2, showing the Java code that implements each particular barrier (i.e., to be inserted at line 3 in the skeleton), along with the assembly code generated by the optimizing compiler for the PowerPC and Intel x86 platforms.

Boundary. The **boundary** barrier is the current default barrier for MMTk generational collectors. It tests whether the source and target lie on different sides of a static boundary address, and records the source location (or object *slot*) to which the target reference is stored. Such a barrier is useful for copying generational collectors to record references from older objects to younger objects, using a bounded nursery at a fixed virtual memory location. Recording the slot holding the reference is most precise for GC, since only pointers of definite interest need to be processed at GC time.

Object. The **object** barrier was used in the original Jikes RVM collectors prior to adoption of MMTk. When a target reference is stored into any field of a source object, the source object's reference is recorded in the SSB. To avoid multiple entries of the same source object, the object barrier filters duplicates by setting a flag in the object's header when it is first entered in the SSB, and checking this flag in the fast-path. The object barrier has the advantage of concisely recording pointer updates to multiple fields of hot objects, at the expense of interpreting object pointer maps at GC time to find the pointers of interest to GC.

Hybrid. The **hybrid** barrier uses the **boundary** barrier for arrays and the **object** barrier otherwise; the distinction is made statically at each store site based on the type of the source object. This avoids **object** barrier GC overhead incurred to scan large arrays looking for interesting pointers. The Java code fragment for the hybrid barrier is given in Figure 3. The choice of barrier is statically chosen at compile time by passing a literal constant as the `mode` parameter and relying on constant folding to eliminate the unreached branch

[2] We determined this by comparing mutator performance with and without full heap traces at nursery collection time.

[3] Jikes RVM has, at various times, been shown to be competitive with the IBM product JVM for the x86 architecture.

[4] Xianglong Huang and Narendran Sachindran jointly implemented the pseudo-adaptive compilation mechanism.

```
1  public final void writeBarrier(VM_Address src, VM_Address slot,
2                                 VM_Address tgt, int mode) throws VM_PragmaInline {
3    // insert write barrier code here
4    VM_Magic.setMemoryAddress(slot, tgt);
5  }
```

Figure 1: Write barrier skeleton code

boundary

```
3    if (slot.LT(NURSERY_START)
4        && tgt.GE(NURSERY_START))
5      remSlots.insert(slot);
```

```
1    liu  R3,0x6e10
2  cmplW  cr1,R30,R3
3    bge  1 54
4    liu  R3,0x6e10
5  cmplW  cr1,R31,R3
6    bge  1 7c
```

```
1   cmp  edi 0xa0200000
2  jlge  0
3   cmp  ebx 0xa0200000
4  jlge  0
```

object

```
3    if (getHeader(src)
4        .and(LOGGING_MASK)
5        .EQ(UNLOGGED))
6      rememberObject(src);
```

```
1    lwz  R4,-8(R5)
2  rlinm  R4,R4,0x0,0x1d,0x1d
3  cmpiW  cr1,R4,0x4
4    beq  1 78
```

```
1   mov  ecx -8[edx]
2   and  ecx 4
3   cmp  ecx 4
4   jeq  0
```

zone

```
3    if (slot.xor(tgt)
4        .GE(ZONE_SIZE))
5      remSlots.insert(slot);
```

```
1    xor  R3,R30,R31
2    liu  R5,0x40
3  cmplW  cr1,R3,R5
4    bge  1 74
```

```
1   mov  edi eax
2   mov  eax edi
3   xor  eax ebx
4   cmp  eax 0x400000
5  jlge  0
```

card

```
3    int card = src.rshl(LOG_CARD_SIZE);
4    VM_Magic.setByteAtOffset
5      (cardTable, card, (byte) 1);
```

```
1    lwz  R5,0x1664(JT)
2  rlinm  R6,R3,0x16,0xa,0x1f
3    lil  R7,0x1
4   stbx  R7,R5,R6
```

```
1   mov  ebx [0x290279a]
2   shr  eax 10
3   mov  [0+ebx+eax<<0] 1
```

Figure 2: Write barrier code

```
3    if (mode == AASTORE_WRITE_BARRIER) {
4      if (slot.LT(NURSERY_START) && tgt.GE(NURSERY_START))
5        remSlots.insert(slot);
6    } else {
7      if (getHeader(src).and(LOGGING_MASK).EQ(UNLOGGED))
8        rememberObject(src);
9    }
```

Figure 3: Hybrid write barrier code

of the conditional. The fast-path result is compiled code that looks exactly like that of the **object** or **boundary** barrier.

Zone. The **zone** barrier assumes the heap is divided into fixed-size 2^k-byte logical *zones*, aligned on 2^k-byte address boundaries, and captures pointers that cross from one region to another, recording the source slot into which such target reference is stored. Such a barrier is useful for collectors that partition the heap into more than two regions. Examples of such collectors include MOS [18, 25], generalized age-based collectors [29], and Beltway [9], among others. The zone barrier has the same precision as the boundary barrier, remembering precisely which locations hold interesting pointers. We use a zone size of 4MB, matching the nursery size, and thus avoiding the slow path for any intra-nursery pointers.

Card. The **card** barrier has long been promoted for the fact that it is entirely in-line and unconditional. Like the zone barrier, the card barrier assumes a heap divided into fixed-size 2^k-byte logical *cards* [26, 32], where typically $7 \leq k \leq 10$. Each card maps to an entry in an array or *card table* that records whether the card has been *dirtied* by stores of target references into any of the source locations in the card. The index in the card table for a given location is obtained by a simple shift operation. At GC time the dirty cards must be scanned to discover pointers of interest. The precision of the card barrier varies with the size of the cards: smaller cards give more precision at the cost of maintaining and processing the correspondingly larger card table [16].

4.2 Read barriers

Figure 5 shows the code for the read barriers.

Read unconditional. The unconditional form of the read barrier simply masks out low-order address bits of references as they are loaded from a source object field. Such a barrier is useful in situations that need to tag object references transparently to the mutator, such as memory managers that rely on marking unique references [24].

Read conditional. The conditional read barrier includes a test whether the tag bits are set. Such a barrier is needed any time a mutator must perform some action conditionally on reading a target reference from a source object field. Note that this is an 'eager'

```
1  public final VM_Address readBarrier(VM_Address obj, VM_Address slot,
2                                      int mode) throws VM_PragmaInline {
3    VM_Address value = VM_Magic.getMemoryAddress(slot);
4    return value;   // replace with barrier code
5  }
```

Figure 4: Read barrier skeleton code

Java	PowerPC	x86
unconditional		
4 `return value.and(~3);`	1 `rlinm R3,R3,0x0,0x0,0x1d`	1 `and eax -4`
conditional		
4 `if (value.and(1).NE(1))` 5 `  return value;` 6 `else` 7 `  return 0;`	1 `rlinm R4,R3,0x0,0x1f,0x1f` 2 `cmpiW cr1,R4,0x1` 3 `bne 1 3c`	1 `mov edx eax` 2 `and edx 1` 3 `cmp edx 1` 4 `mov edx 0` 5 `cmovne edx eax` 6 `mov eax edx`

Figure 5: Read barrier code

read barrier, using the terminology of Bacon et al. [5]. While they used a Brooks-style unconditional read barrier, our goal is to expose the cost of any conditional associated with the read barrier, and so implement a more general barrier here.

4.3 Discussion

There are aspects of our barrier implementations that could be improved, depending on hardware platform. Given sufficient registers, the card barrier could devote a register to hold the base address of the card table, so avoiding the need to load it from a global variable. Moreover, on certain RISC platforms the literal constant 0 is available in a 'zero' register hard-wired to the value 0. If clean cards are represented by a non-zero card table entry then dirtying a card can be achieved by clearing it: a store of 0 to the entry. This eliminates the need to load a non-zero immediate operand to the store. Together, these changes can eliminate 2 instructions from the sequence. Unfortunately, on the PowerPC the 'zero' register is only available to arithmetic instructions, not to stores, so without a devoted card table base register our instruction sequence is optimal.

We also plan to compare other specialized barriers to the more general ones we consider here. In particular, there is a directional alternative to the zone barrier that records references that point from one zone to another zone at a higher (or lower) address in the heap [29]. Other barriers of interest include the various flavors used by reference counting collectors [7].

5. Experimental methodology

We now describe the experimental context in MMTk and Jikes RVM, the machines we use, the benchmarks measured, and the significance and accuracy of our measurements.

5.1 MMTk and Jikes RVM

We use MMTk in Jikes RVM version 2.3.2+CVS [1], patched to support performance counters, *pseudo-adaptive* compilation, our *ignore remsets* GC configuration, and *read barriers*.[5] MMTk is a flexible high performance memory management toolkit used by Jikes RVM [8]. Jikes RVM is a high-performance VM written in Java with an aggressive optimizing compiler. We use configurations that precompile as much as possible, including key libraries and the optimizing compiler and turn off assertion checking (the *Fast* build-time configuration).

[5]The *ignore remsets* functionality is now integrated into MMTk. All of our other patches are publicly available from the first author's web pages.

5.2 Experimental platform

We perform our experiments on three architectures: Athlon, Pentium 4, and Power PC.

We use a 1.9GHz AMD Athlon XP 2600+. It has a 64 byte L1 and L2 cache line size. The data and instruction L1 caches are 64KB 2-way set associative. It has a unified, *exclusive* 512KB 16-way set associative L2 cache. The Athlon has 1GB of dual channel 333MHz DDR RAM configured as 2×512MB DIMMs with an nForce2 K7N2G motherboard and 333MHz front-side bus.

The 2.6GHz Pentium 4 has hyperthreading disabled. It has a 64 byte L1 and L2 cache line size, an 8KB 4-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, and a 512KB unified 8-way set associative L2 on-chip cache. The machine has 1GB of dual channel 400MHz DDR RAM configured as 2×512MB DIMMs with an Intel i865 motherboard and 800MHz front-side bus.

We also use an Apple Power Mac G5 with a 1.6HGz IBM PowerPC 970. It has a 128 byte L1 and L2 cache line size, a 64KB direct mapped L1 instruction cache and 32KB 2-way set associative L1 data cache, and a 512KB unified 8-way set associative L2 on-chip cache. The machine has 768MB of 333MHz DDR RAM with an Apple motherboard and 800MHz front-side bus.

All three platforms run the same configuration of Debian Linux with a 2.6.0 kernel. We run all experiments in a standalone mode with all non essential daemons and services (including the network interface) shut down. We instrument MMTk and Jikes RVM to use the AMD and Intel performance counters to measure cycles, retired instructions, L1 cache misses, L2 cache misses, and TLB misses of both the mutator and collector, separately. Because of hardware limitations, each performance counter requires a separate execution. We use version 2.6.5 of the *perfctr* Intel/x86 hardware performance counters for Linux with the associated kernel patch and libraries [23]. At the time of writing, perfctr was unavailable for the PowerPC 970.

5.3 Benchmarks

Table 1 shows key characteristics of each of our benchmarks. We use the eight SPECjvm98 benchmarks, and pseudojbb, a variant of SPECjbb2000 [27, 28] that executes a fixed number of transactions

benchmark	Allocation			Write barrier	
	alloc	MS min	alloc:MS	total	rem set
_202_jess	403MB	16MB	25:1	28.63	0.16%
_213_javac	593MB	26MB	23:1	20.78	2.41%
_228_jack	307MB	14MB	22:1	10.44	7.23%
_205_raytrace	215MB	18MB	12:1	7.35	0.98%
_227_mtrt	224MB	21MB	11:1	8.49	1.00%
_201_compress	138MB	17MB	8:1	1.53	0.71%
pseudojbb	339MB	46MB	7:1	23.31	3.66%
_209_db	119MB	20MB	6:1	35.03	0.52%
_222_mpegaudio	51MB	12MB	4:1	9.79	0.23%
mean	**265MB**	**21MB**	**13:1**	**16.15**	**1.74%**
geometric mean	**216MB**	**20MB**	**11:1**	**11.89**	**0.98%**

Table 1: Benchmark allocation statistics and write barrier events (in millions)

to perform comparisons under a fixed garbage collection load. The *alloc* column in Table 1 indicates the total number of megabytes allocated. The *min* column shows the minimum heap size in which the benchmark can run to completion with a generational copying mark-sweep hybrid (GenMS). The *alloc:min* column quantifies garbage collection load as the ratio of total allocation to the minimum heap size in which GenMS executes. The *total* column indicates the dynamic count of write barrier invocations (fast path). The *rem set* column indicates the frequency with which the write barrier leads to a remembered set entry for the boundary barrier.

5.4 Significance and accuracy

To assess the significance of our results we: 1) measured the variation in timing data, and 2) compared instruction counts on the P4 and AMD. Since only one performance metric could be gathered at a time, it was necessary to run each experiment four times on the AMD and P4 (recall that each 'experiment' takes the fastest of five invocations of the second iteration of the benchmark). We then compared the variation across each set of four results, measured as the standard deviation divided by the mean for that set. We found that the variation ranged from 0.04% to 0.8%, with a geometric mean across all benchmarks of 0.1%. When we compared instruction counts between P4 and AMD, we found that they were almost identical, despite wide variations in running time and miss rate. This gives us a high degree of confidence in the measurements.

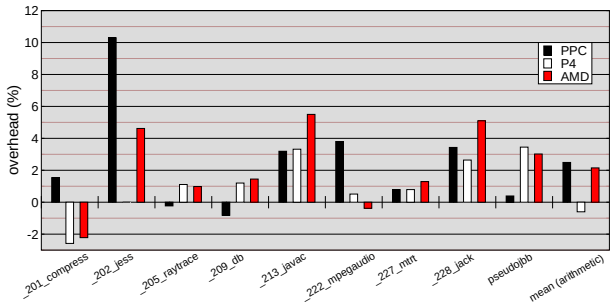
6. Results

In this section we examine the mutator performance of the read and write barriers. It is important to remember that our methodology measures mutator overhead only. The choice of barrier can have a significant impact on garbage collection overheads, but we do not measure that here. Furthermore, the contribution of mutator-time overheads reported here to total time will be diluted by garbage collection.

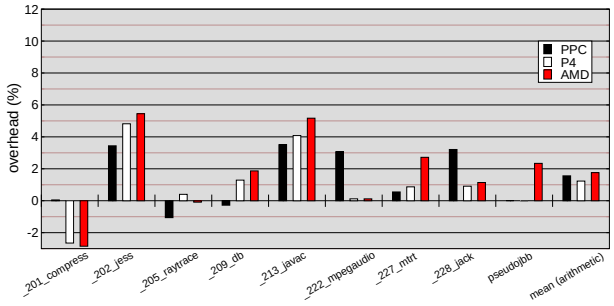
6.1 Write barrier performance

Figure 6 shows the performance overhead of each of the write barriers on the three platforms. The overhead is generally quite low, on average 2% or less for most of the barriers. Surprisingly, in some cases the addition of barriers *improves* performance. Hardware performance counters reveal that these improvements are due to better locality. (As we would hope, the performance counters also show that the addition of the barrier does *not* reduce the instruction count!). At this stage we can only speculate on why the addition of the barrier would improve locality.

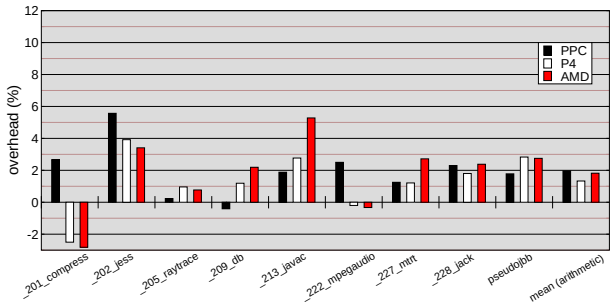
We now examine the results for each barrier in more detail.



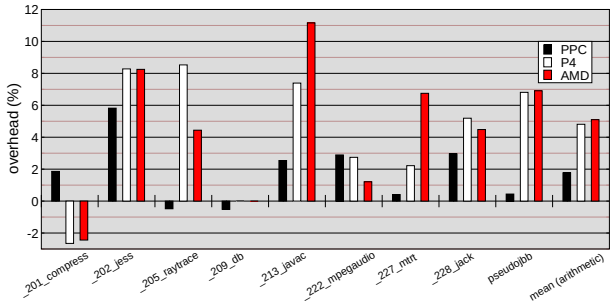
(a) Boundary



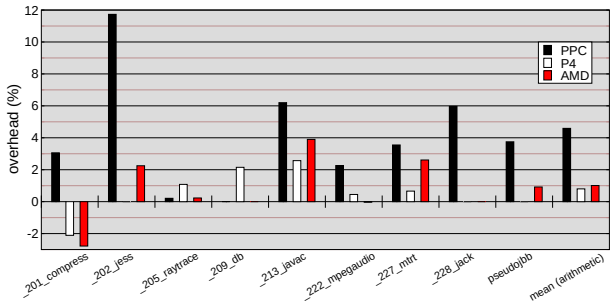
(b) Object



(c) Hybrid

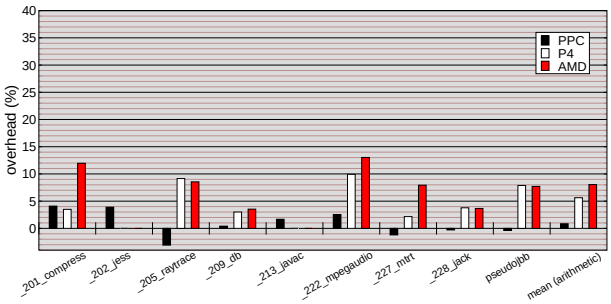


(d) Zone

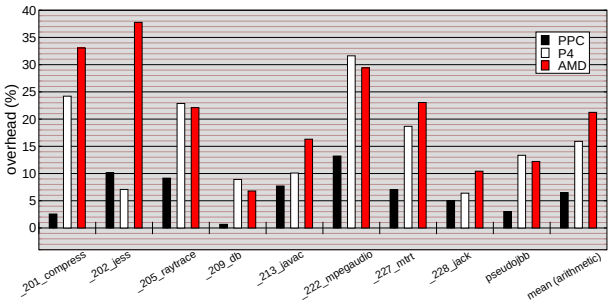


(e) Card

Figure 6: Write barrier running time overheads



(a) Read unconditional



(b) Read conditional

Figure 7: Read barrier running time overheads

Boundary. The boundary barrier costs on average 2.15%, 1.31%, and 2.49% for the AMD, P4 and PPC architectures, respectively. This barrier shows the biggest variation in results, with a best case *improvement* of 2.58% on _201_compress on the P4 and a worst case degradation of 10.30% on the PPC. The actual increase in retired instructions is 2.33% on average, with increases of 4.68% and 4.43% on _202_jess and _213_javac, and an increase of only 0.0007% on _201_compress. The miss rates explain the performance win on _201_compress, decreasing consistently with the running time. Somehow the boundary barrier leads to improved locality on that benchmark, with very little increase in retired instructions (Table 1 shows that _201_compress executes very few write barriers). Improvements in _222_mpegaudio (AMD) and _209_db (P4) are also due to lower miss rates. Without access to performance counters, we can only speculate on the reasons behind the substantial slowdown on _202_jess (10.30%) and speedups on _205_raytrace (0.23%) and _209_db (0.82%) on the PPC.

Object. The object barrier costs on average 1.76%, 1.23%, and 1.56% for the AMD, P4 and PPC architectures, respectively. These numbers are all slightly better than those for the boundary barrier. The worst case is substantially better (5.45% on _202_jess for the AMD), and minor improvements are seen on _201_compress (2.85 % AMD, 2.65% P4), _205_raytrace (0.09% AMD, 1.06% PPC), and _209_db (0.28% PPC). This barrier shows lower mutator overheads than the boundary barrier, but it is less precise, and must therefore perform more work at collection time. In the case of large, sparsely updated objects (e.g., arrays), this overhead could be substantial.

Hybrid. The hybrid barrier uses an object barrier for scalar objects (which are typically small and dense), and the boundary barrier for arrays (which may be large and sparse). The average mutator costs for this barrier are roughly between those for the two barriers from which it is composed: 1.82%, 1.33%, and 1.97% for AMD, P4 and PPC architectures, respectively. The worst case is relatively low (5.57% on _202_jess for PPC), and similar small improvements are seen on _201_compress, _222_mpegaudio, and _209_db. This barrier is designed to balance the trade-off between mutator and collection costs in the boundary and object barriers.

Zone. The zone barrier is the first where we see a marked architectural dependency. The average costs are 5.10%, 4.81%, and 1.77% for the AMD, P4 and PPC, respectively. The x86 architectures thus find this barrier significantly more expensive than the other barriers, while on the PPC this barrier is among the cheapest. The dynamic instruction count on the x86 architectures is increased by 4.65%. Unfortunately, we don't know how the PPC instruction count is impacted. However, Figure 2 shows that, statically, the PPC uses about the same number of instructions for the zone and object boundaries and sees similar run-time performance. By contrast, the x86 needs quite a long instruction sequence for the zone barrier. We are unsure to what extent this is an artefact of the x86 backend of the Jikes RVM optimizing compiler, however we note that the zone barrier uses an arithmetic instruction (`xor`), which must use the `eax` register, further increasing pressure on the few available x86 registers.

Card. The card barrier also shows a strong architectural difference, this time the PPC is substantially slower. The average costs are 1.01%, 0.80%, and 4.59% for the AMD, P4 and PPC, respectively. The x86 results are somewhat consistent with conventional wisdom; however, it is debatable as to whether they are sufficiently lower than the other barriers to warrant the extra GC time effort of scanning the cards, and the space overhead of the card tables. Also, we do not include here the allocation time work that must be performed to ensure that the card offset table is kept consistent. In short, it is not a decidedly better choice. We are unsure why the PPC results are so poor, however we are investigating the possibility that the PPC does not perform stores of bytes very efficiently. It may be necessary to use a word store on the PPC, trading off time for space a little further.

These results indicate a number of conclusions. First, barrier costs have often been overstated. We show that a reasonable generational barrier (the hybrid) has an average mutator overhead under 2%, with a worst case of less than 6%. Sometimes the barrier overhead is less important than noise due to second order effects on locality. This is encouraging for those designing GC algorithms, and should give pause to those who dismiss write barriers as unreasonably expensive. Second, barrier costs are very architecturally dependent. We have seen that both the ISA (PPC v. x86) and the implementation (AMD v. P4) can have a significant impact on barrier performance. Implementers should consider this carefully when choosing their collector and associated barriers. Third, the often stated view that card marking is the cheapest form of barrier is questionable. Our card marking implementation is appreciably slower on the PPC, and the performance difference on x86 may be overwhelmed by the secondary costs associated with card marking, such as maintaining the card offset table and scanning the cards at GC time.

6.2 Read barrier performance

Figure 7 shows the performance overhead of each of the read barriers on the three platforms. The overhead varies substantially and is extremely architecturally sensitive. On the PPC the unconditional barrier actually showed performance improvements on some benchmarks, which we ascribe to locality affects, though without hardware performance counters we cannot be sure. Unfortunately our read barriers are not as robust as the write barriers, and in a number of cases we were unable to produce complete results.

We now examine the results for each barrier in more detail.

Unconditional. The average costs for the unconditional barrier are 8.05%, 5.04%, and 0.85% for the AMD, P4 and PPC, respectively. The results tend to be somewhat more consistent than the write barrier results with the worst case of 13.03% on the AMD (less than 2× the average on the AMD). Most strikingly, the barrier is extremely cheap on the PPC, about half the price of the cheapest write barrier on that architecture, and nearly an order of magnitude cheaper than on the AMD. There are a number of speedups on the PPC, which we guess are due to locality effects (in the absence of hardware performance counters this is hard to show).

Conditional. Unsurprisingly, the conditional barrier is substantially more expensive than the unconditional barrier, with average overheads of 21.24%, 15.91%, and 6.49% for the AMD, P4 and PPC, respectively. Once again, we see a striking architectural dependency, with the PPC much better able to absorb the overhead. Interestingly, the read-intensive _209_db benchmark is barely affected by this barrier on the PPC (0.67%). It is possible that branch prediction is playing a significant role in these results, as the conditional we have used will always evaluate to true (although this is not something the compiler can determine statically).

We draw two strong conclusions from these results. First, read barrier costs are very architecturally dependent. If read barriers are known to be cheap, then this opens a number of algorithmic possibilities that may otherwise have been ruled out as too expensive. This invites consideration by architects. It also suggests that great care should be taken in a) interpreting published results which are architecture-specific, and b) in collector design for portable runtimes. Second, the nature of the read barrier substantially impacts on its overhead. An unconditional read barrier can be extremely cheap and should be considered as a viable approach in cases where it might be useful. A conditional barrier will be moderately expensive, although the predictability of the branch may greatly influence just how expensive it is.

7. Conclusions

Read and write barriers are algorithmically powerful mechanisms with significant application to garbage collection and other applications. In this paper we evaluate the untested assumption that barriers impose a significant overhead. We present a methodology for measuring barrier costs *in vivo*, and evaluate a range of common write barriers and two read barriers on three platforms, using 9 standard Java benchmarks.

We show that the overhead of a reasonable write barrier for a generational collector is low on average (less than 2%), and less than 6% in the worst case. We also show that this is architecturally sensitive. We also show that read barriers can be very low cost (0.85% on average on the PPC), and that this is extremely sensitive to the style of barrier and the underlying architecture.

Our methodology is publicly available as part of MMTk. We hope that this will encourage systematic empirical study of barrier performance within the community, rather than allowing key algorithmic decisions to be determined by untested assumptions about the cost of the underlying mechanisms. We also hope that this will invite further creativity and more aggressive exploitation of the algorithmic power of barriers by the memory management community. This work should encourage implementors to carefully consider the choice of barrier in light of the workload, their compiler, and the underlying architecture. Finally, architectural sensitivity to this key mechanism for garbage collected languages may invite some interest from the architecture research community.

Acknowledgments

This work is supported by the National Science Foundation under grant No. CCR-0085792, the Australian Research Council under grant No. DP0452011, by the Defense Advanced Research Program Agency, and by IBM. Any opinions, findings and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

We are grateful to Kathryn S. McKinley for her encouragement and numerous helpful suggestions that improved this paper.

8. REFERENCES

[1] ALPERN, B., ATTANASIO, C. R., BARTON, J. J., COCCHI, A., HUMMEL, S. F., LIEBER, D., NGO, T., MERGEN, M., SHEPHERD, J. C., AND SMITH, S. Implementing Jalapeño in Java. In OOPSLA'99 [22], pp. 314–324.

[2] APPEL, A. W. Simple generational garbage collection and fast allocation. *Software—Practice and Experience 19*, 2 (Feb. 1989), 171–183.

[3] ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., AND SWEENEY, P. F. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Minneapolis, Minnesota, Oct.). *ACM SIGPLAN Notices 35*, 10 (Oct. 2000), pp. 47–65.

[4] AZAGURY, A., KOLODNER, E. K., PETRANK, E., AND YEHUDAI, Z. Combining card marking with remembered sets: How to save scanning time. In ISMM'98 [19], pp. 10–19.

[5] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (New Orleans, Lousiana, Jan.). *ACM SIGPLAN Notices 38*, 1 (Jan. 2003), pp. 285–298.

[6] BLACKBURN, S., AND MCKINLEY, K. S. In or out?: Putting write barriers in their place. In *Proceedings of the ACM International Symposium on Memory Management* (Berlin, Germany, Jun., 2002). *ACM SIGPLAN Notices 38*, 2 (Feb. 2003), pp. 281–290.

[7] BLACKBURN, S., AND MCKINLEY, K. S. Ulterior reference counting: fast garbage collection without a long wait. In OOPSLA'03 [21], pp. 344–358.

[8] BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. Myths and reality: The performance impact of garbage collection. In *ACM International Conference on Measurement and Modeling of Computer Systems* (New York, New York, June). 2004, p. To appear.

[9] BLACKBURN, S. M., JONES, R. E., MCKINLEY, K. S., AND MOSS, J. E. B. Beltway: Getting around garbage collection gridlock. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Berlin, Germany, June). *ACM SIGPLAN Notices 37*, 5 (May 2002), pp. 153–164.

[10] BRAHNMATH, K., NYSTROM, N., HOSKING, A. L., AND CUTTS, Q. Swizzle barrier optimizations for orthogonal persistence in Java. In *Proceedings of the Third International Workshop on Persistence and Java* (Tiburon, California, August 1998), R. Morrison, M. Jordan, and M. Atkinson, Eds. Advances in Persistent Object Systems. Morgan Kaufmann, 1999, pp. 268–278.

[11] BROOKS, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (Austin, Texas, Aug.). 1984, pp. 256–262.

[12] CAUDILL, P. J., AND WIRFS-BROCK, A. A third generation Smalltalk-80 implementation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, Sept.). *ACM SIGPLAN Notices 21*, 11 (Nov. 1986), pp. 119–130.

[13] HIRZEL, M., DIWAN, A., AND HERTZ, M. Connectivity-based garbage collection. In OOPSLA'03 [21], pp. 359–373.

[14] HOSKING, A. L., AND HUDSON, R. L. Remembered sets can also play cards. In *Proceedings of the OOPSLA Workshop on Memory Management and Garbage Collection* (Washington, DC, Sept.). 1993.

[15] HOSKING, A. L., AND MOSS, J. E. B. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, Dec.). *ACM Operating Systems Review 27*, 5 (Dec. 1993), pp. 106–119.

[16] HOSKING, A. L., MOSS, J. E. B., AND STEFANOVIĆ, D. A comparative performance evaluation of write barrier implementations. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, Canada, Oct.). *ACM SIGPLAN Notices 27*, 10 (Oct. 1992), pp. 92–109.

[17] HOSKING, A. L., NYSTROM, N., CUTTS, Q., AND BRAHNMATH, K. Optimizing the read and write barriers for orthogonal persistence. In *Proceedings of the Eighth International Workshop on Persistent Object Systems* (Tiburon, California, August 1998), R. Morrison, M. Jordan, and M. Atkinson, Eds. Advances in Persistent Object Systems. Morgan Kaufmann, 1999, pp. 149–159.

[18] HUDSON, R. L., AND MOSS, J. E. B. Incremental collection of mature objects. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France, Sept.), Y. Bekkers and J. Cohen, Eds. vol. 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992, pp. 388–403.

[19] *Proceedings of the ACM International Symposium on Memory Management* (Vancouver, Canada, Oct., 1998). *ACM SIGPLAN Notices 34*, 3 (Mar. 1999).

[20] JONES, R., AND LINS, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, May 1996. Chapter on distributed collection by Lins.

[21] *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Anaheim, California, Nov.). *ACM SIGPLAN Notices 38*, 11 (Nov. 2003).

[22] *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Nov.). *ACM SIGPLAN Notices 34*, 10 (Oct. 1999).

[23] PETTERSSON, M. Linux Intel/x86 performance counters, 2003. http://user.it.uu.se/ mikpe/linux/perfctr/.

[24] ROTH, D. J., AND WISE, D. S. One-bit counts between unique and sticky. In ISMM'98 [19], pp. 49–56.

[25] SELIGMANN, J., AND 235-252, S. G. . E. . Incremental mature garbage collection using the Train algorithm. In *Proceedings of the European Conference on Object-Oriented Programming* (Åarhus, Denmark, Aug.). vol. 952 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995, pp. 235–252.

[26] SOBALVARRO, P. G. A lifetime-based garbage collector for LISP systems on general-purpose computers, 1988. B.S. Thesis, Dept. of EECS, Massachusetts Institute of Technology, Cambridge.

[27] SPEC. SPECjvm98 benchmarks, 1998. http://www.spec.org/osg/jvm98.

[28] SPEC. SPECjbb2000 benchmarks, 2000. http://www.spec.org/jbb2000.

[29] STEFANOVIĆ, D., MCKINLEY, K. S., AND MOSS, J. E. B. Age-based garbage collection. In OOPSLA'99 [22], pp. 370–381.

[30] UNGAR, D. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, Apr.). 1984, pp. 157–167.

[31] WILSON, P. R., AND MOHER, T. G. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices 24*, 5 (May 1989), 87–92.

[32] WILSON, P. R., AND MOHER, T. G. Design of the opportunistic garbage collector. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, Louisiana, Oct.). *ACM SIGPLAN Notices 24*, 10 (Oct. 1989), pp. 23–35.

[33] ZEE, K., AND RINARD, M. C. Write barrier removal by static analysis. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Seattle, Washington, Nov.). *ACM SIGPLAN Notices 37*, 11 (Nov. 2002), pp. 191–210.

[34] ZORN, B. Barrier methods for garbage collection. Tech. Rep. CU-CS-494-90, University of Colorado at Boulder, Nov. 1990.