In or Out? Putting Write Barriers in Their Place

Stephen M Blackburn*

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

Kathryn S McKinley

Department of Computer Sciences
University of Texas at Austin
Austin, TX, 78712, USA
mckinley@cs.utexas.edu

ABSTRACT

In many garbage collected systems, the mutator performs a *write barrier* for every pointer update. Using generational garbage collectors, we study in depth three code placement options for remembered-set write barriers: inlined, out-of-line, and partially inlined (fast path inlined, slow path out-of-line). The *fast path* determines if the collector needs to remember the pointer update. The *slow path* records the pointer in a list when necessary. Efficient implementations minimize the instructions on the fast path, and record few pointers (from 0.16 to 3% of pointer stores in our benchmarks). We find the mutator performs best with a partially inlined barrier, by a modest 1.5% on average over full inlining.

We also study the *compilation cost* of write-barrier code placement. We find that partial inlining reduces the compilation cost by 20 to 25% compared to full inlining. In the context of just-in-time compilation, the application is exposed to compiler activity. Regardless of the level of compiler activity, partial inlining consistently gives a total running time performance advantage over full inlining on the SPEC JVM98 benchmarks. When the compiler optimizes all application methods on demand and compiler load is highest, partial inlining improves total performance on average by 10.2%, and up to 18.5%.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms

Design, Performance, Algorithms

Keywords

write barriers, copying collection, generational collection, Java

*This author did this work while at the University of Massachusetts. This work is supported by NSF ITR grant CCR-0085792, NSF grant ACI-9982028, DARPA grants F30602-98-1-0101 and F33615-01-C-1892, and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

ISMM'02, June 20-21, 2002, Berlin, Germany.

1. INTRODUCTION

Many garbage collectors remember pointer stores. To avoid collecting the entire heap, garbage collectors divide the heap into regions and track the pointers between them. For example in a generational copying collector, the collector scavenges the nursery independently of higher generations and avoids scanning older generations by conservatively assuming that any remembered pointers into the nursery are live. Collector and mutator performance depend on the frequency of pointer stores, the number of stores remembered, and the benefits from scavanging regions independently. These tradeoffs almost always improve garbage collector and total performance [CWB86, CHL98, LH83, Ung84, SMM99] (see Section 5.4).

The *write-barrier* code sequence determines whether a pointer store needs to be remembered, and if so, remembers it. The *fast path* of a conditional write barrier determines if the pointer update should be remembered (i.e., it crosses independently collected regions and the source will be collected before the target). The fast path is typically short (3 to 5 instructions) and consists of bit operations, comparisons, and perhaps loads. The *slow path* remembers the pointer update. In a *remembered set* scheme, it usually puts the source in a list. The collector then processes the list at the beginning of a collection [Ung84]. Efficient collector organizations minimize the number of remembered pointer stores [App89, BJMM02, HMS92, SMM99]. Other schemes, such as card marking [Sob88, WM89], unconditionally set a bit in a bit vector to mark a region of memory containing the source pointer, and scan for pointers into the increment being collected at collection time. Card marking schemes trade off scanning time for a simpler unconditional barrier.

Previous research has explored implementations of write barriers, remembered sets, card marking, and hybrids [AKPY98, HMS92, HH93]. Fitzgerald and Tarditi [FT00] suggest putting the cold path out-of-line. However, no previous work measures the impact of this choice. This paper investigates the impact of the write barrier on the application code quality, and on the compilation cost in Jikes RVM [AAC+99, AAB+00] with a variety of garbage collectors we developed. We compare no write barrier, a completely inlined write barrier, an out-of-line write barrier, and a partially inlined barrier (fast path inlined, slow path out-of-line). We implement the out-of-line cases with a direct procedure call. We use a variety of copying collectors, and also compare with a semi-space copying collector which has no write barrier.

Our results first confirm that the slow path is rarely taken (between 0.15 and 3%) for the collectors and SPEC JVM98 benchmarks we examine, which is consistent with previous languages and systems [JL96, HH93, Ste99]. Compared with no write-barrier, inlining increases the application code size by on average 81%, partial inlining by 33%, and out-of-line by 21%. Partial inlining however provides the fastest executing application code by a modest 1.5%. Jikes RVM thus integrates and optimizes the fast path instructions well, and the overhead of the direct procedure call on the infrequently taken slow path is minimal. The inlined write barrier suffers because the slow path bloats the code size, and since it is rarely taken, yields no performance benefits.

The compilation cost of the write barrier has two components in the Jikes RVM. (1) The obvious component – the compiler generates and optimizes more code when the write barrier is inlined than when it is out-of-line or partially inlined. (2) The compiler itself executes the write barrier as it runs in the Jikes RVM. Because the Jikes RVM compiler is written in Java and runs along with the application in the JVM, it uses the same write barrier as the application as it compiles the application. Both costs slow down the compiler when it inlines or partially inlines the write barrier compared with an out-of-line write barrier. We show that partial inlining can overcome this degradation with application time improvements. An inlined write barrier slows down the application and a just-in-time compiler twice; by 10.2% on average, and up to 18.5% compared with partial inlining. Even an out-of-line barrier is competitive with inlining (1.4% worse to 15.5% better) when compilation load is a high percent of total time. Although the overall result may now seem predictable to the informed reader, we were startled by the magnitude of the differences. In summary, partial inlining is the best choice for a conditional write barrier, and inlining is especially problematic for a just-in-time compiler.

This paper is organized as follows. We first describe related work on incremental collectors and their write barriers, including policies and mechanisms. We then overview the generational collectors, write barriers, and methodology we use. Our results section demonstrates the static and dynamic application and compilation costs of the different code placement choices. These results show the partially inlined barrier is the best choice, and in some circumstances, dramatically improves performance over inlining or out-of-line barriers.

2. RELATED WORK

In this section, we discuss related work on write barrier function and designs, and general compiler inlining. To our knowledge, no one has studied the impact of write barrier inlining strategies before. Write barriers are required in two distinct contexts: incremental garbage collection and reference counting garbage collection.

Incremental collectors depend on write barriers to record pointers into independently collected regions of memory called *increments*. By tracking all pointers into an increment, the increment can be safely collected by making the conservative assumption that the sources of all incoming pointers are live. If the number of incoming pointers is suitably low, incremental collection can be very efficient. Incremental collection is the basis for a large number of garbage collectors including generational [LH83, Ung84, App89], older-first [SMM99], Beltway [BJMM02], and mature object space collectors [HM92]. Jones and Lins [JL96] describe many more incremental algorithms.

A write barrier for incremental collection can usually be characterized in terms of three implementation choices. 1) A mechanism for determining *whether* to remember a pointer update. 2) A design decision as to *what* should be remembered. 3) A mechanism for *how* it should be remembered. The literature records a large number of alternatives in this space [JL96]. Two broad approaches are widely used: *remembered sets* and *card marking*.

Remembered Sets. Remembered sets typically remember either the source *object* or *slot* (pointer field) [JL96], both of which we study here. Ungar was the first to suggest the object remembering barrier [Ung84], and the Jikes RVM generational garbage collectors [AAB+00] also remember objects. They use a bit in the source object to avoid duplicates, and to avoid remembering source objects that reside in the nursery.

Collectors may remember the exact slot instead of the object containing the pointer [App89, HD90, HMS92, Ste99]. Stefanović et al. developed a very fast *address order write barrier* that exploits an address order organization of collection increments in the older-first collector. The write barrier we use is similar, and depends on generations being organized within major virtual memory alignment boundaries to avoid explicit generational bounds checking. This structure leads to a very fast barrier (see Figure 1a) and c)), that does not require loading explicit generation bounds for comparison with the source and target pointers.

The literature also reports a number of remembered set implementations. Hudson and Diwan use a *sequential store buffer* to remember slots [HD90, HMS92]. They use virtual memory protection to detect buffer overflow. The Jikes RVM collectors use a similar structure for storing remembered objects although it uses an explicit bounds check to detect buffer overflow. Our collectors remember slots in power-of-two aligned buffers, and use the power of two alignment to detect buffer overflow without an explicit buffer bounds pointer.

Card Marking. Card marking uses a table to remember fixed size regions of memory (*cards*) as pointer sources [Sob88, WM89]. The write barrier marks cards when necessary, and the collector treats memory regions corresponding to marked cards as roots, scanning them for pointers. The collector clears the card table at the end of a collection. Several papers consider the efficiency of card marking schemes [WM89, Cha92, H93, HMS92], as well as a hybrid of remembered sets and card marking [HH93]. Hosking et al. compare remembered sets and cards in an interpreted Smalltalk system [HMS92, HH93] and find their performance is similar. Our study is of limited relevance to card marking because most card marking barriers use a very short and unconditional code sequence.

Reference Counting. Reference counting algorithms rely on a write barrier to update reference counts at each pointer store. A classic [Col60] reference counting algorithm might employ a conditional write barrier (reclaim the object if the count is zero). However the more widely used deferred reference counting [DB76] approach depends on an *unconditional* write barrier. It remembers pointer stores unconditionally and processes them from time to time. Our key findings depend on the *conditionality* of the incremental collector's write barrier, and the fact that much of the write barrier is rarely executed. Therefore our results are not likely to apply directly to reference counting garbage collector write barriers.

Inlining. For a long time, compilers have used inlining to improve code performance, and have for the most part used heuristics based on code size and/or profiling to limit the code bloat effects of inlining [AFSS00, ASG97, Sch77]. Cooper et al. find that inlining can degrade highly optimizing compilers, and expose non-linear compiler algorithms [CHT92, CHKT93]. Cooper et al. [CHK92] and Dean and Chambers [DC94] show that not all inlining is equal, and its judicious application improves performance. Both suggest frequency as a criteria for their automatic procedure inlining. We use a Jikes RVM compiler pragma to perform partial inlining and isolate the infrequently executed instructions. Because the write barrier is so prolific, this choice has a large impact. Our results suggest partial inlining for other prolific code sequences with hot and cold paths, such as the allocation sequence, should be profitable. It might also be worth investigating compiler partial inlining using branch-frequency profile feedback.

3. COLLECTORS AND BARRIERS

This section briefly presents the generational garbage collectors that we use in our study. These collectors require a write barrier and this section describes the two styles we implement.

3.1 Garbage Collectors

We use an *Appel-style* generational collector [App89] as the basis of our study because, to our knowledge, it is the best performing generational collector [BJMM02]. We also gather write barrier statistics for a fixed-size two generational collector to explore a wider range of collector behaviors. We compare with a non-generational semi-space collector (which does not require a write barrier) to reveal the overall cost of using a write barrier. We now briefly describe these collectors.

3.1.1 Non-Generational Copying Collection

The *semi-space* copying collector [JL96] simply divides the heap into two equal-sized semi-spaces, *to-space* and *from-space*. It always uses to-space for allocation. When allocation exhausts the to-space, the collector is invoked. It first *flips* the spaces, to-space becomes from-space, and from-space becomes to-space. The collector then identifies all live objects in from-space by tracing from the roots (globals, stack variables, etc.). It copies each live object into to-space. Each time the collector encounters a pointer to an object in from-space, it replaces the pointer with a pointer to the object's new location in to-space. It then reclaims from-space en masse. The mutator resumes allocating into to-space until it is exhausted, at which point the collection cycle resumes.

3.1.2 Generational Copying Collection

Because researchers observed that 'most objects die young', they constructed generational collectors to collect the youngest objects more frequently and thus improve upon the simple semi-space collector. This observation is widely known as the *weak generational hypothesis* [Ung84]. A generational collector extends the semi-space collector by allocating into a *younger generation* (or *nursery*) and collecting it frequently. It copies those objects that *survive* the nursery into the to-space of the *older generation*. Filling the to-space of the older generation triggers a *full heap collection*. This collection considers the entire heap and identifies live objects by tracing from roots just as the semi-space collector does.

In practice most objects do die young, so expensive full heap collections are infrequent, giving generational collection a significant performance advantage over simple semi-space collection. However, this advantage depends on collecting the nursery *independently*, i.e., without tracing the full heap to identify live nursery objects. To achieve this goal, a generational collector remembers all pointers into the nursery from the older generation.[1] When it collects the nursery, it conservatively assumes all pointers into the nursery are live. As we discussed in Section 2, there are a number of approaches to remembering such pointers. All of them depend on a write barrier to trap pointer stores and, when necessary, remember the source of the pointer store.

3.1.3 The Appel-style Generational Collector

Generational collectors are often implemented with a fixed fraction of the heap reserved for use by the nursery [JL96]. Appel [App89] describes an alternative implementation which allows the nursery to consume all the space not used by the higher generation's to-space and from-space. Thus initially, when the higher generation's to-space is empty, the nursery occupies half of the heap.[2] The nursery shrinks as to-space of the older generation grows. When the older generation is full, the collector performs a full heap collection, which will typically shrink the higher generation's to-space and thus expand the nursery. By always deferring collection until all of the available heap space is consumed, the flexible nursery approach makes efficient use of space, and collects the higher generation less frequently. We find this orginization performs quite a bit better than a fixed-size nursery collector [BJMM02].

3.2 Write Barriers

We consider an Appel generational collector with two generations, a nursery and an older generation. We use *remembered sets*, which are lists of remembered pointer sources, to track pointers from the older generation into the nursery. The write barrier produces remembered set entries, when necessary. The collector consumes a remembered set and treats each entry as a root during incremental collection (i.e., nursery collection).

We examine the two common write barriers: a *slot remembering* write barrier, which remembers the slot (object field) containing the source pointer, and an *object remembering* write barrier, which remembers the object containing the source pointer [JL96]. Both are widely used [Ung84, App89, JL96, SMM99, BJMM02, AAB+00]. Each barrier makes slightly different tradeoffs. The slot barrier is likely to remember more pointers. With an object barrier, the collector must scan the source objects during a collection. The point of this paper is not to explore the relative merits of either approach, but by examining both we add a degree of generality to our results.

There are two distinct components to the barrier implementation, a frequently executed *fast path*, which tests whether a pointer should be remembered, and an infrequently executed *slow path*, which stores the pointer into a remembered set when necessary. We now describe our implementation of the slot and object barriers.

3.2.1 Slot Remembering Write Barrier

The slot remembering write barrier remembers the addresses of pointers into the nursery. It tests each pointer store and if the source is outside the nursery and the target is within the nursery, it remembers the address of the source in a remembered set. The collector

[1] Not all generational collectors use a write barrier, some go to the expense of tracing the higher generation as part of each nursery collection, and in some circumstances a read barrier is an effective implementation choice [CFS+00].

[2] Generational collectors must reserve half of the heap for the higher generation's from-space to accommodate the worst case survival rate in a full heap collection. Thus when the higher generation to-space is empty, half of the total heap space is available to the nursery in an Appel-style collector.

```
public static final void writeBarrier(ADDRESS source, ADDRESS target) {
  if (source < ((target>>>HEAP_K)<<HEAP_K)) {
    GCTk_WriteBufferSlot.insert(source);
  }
}
```

a) Java fast path

```
public static final void insert(ADDRESS slot) {
  // (1) establish the bump pointer address
  VM_Processor p = VM_Processor.getCurrentProcessor();
  ADDRESS buf = p.GCTk_WriteBufferBumpPointer;

  // (2) grow buffer if necessary
  if ((buf & (WRITE_BUFFER_BUF_SIZE - 1)) == 0)
    buf = growBuffer(buf, WRITE_BUFFER_BUF_SIZE);

  // (3) add the slot
  buf -= BYTES_IN_WORD;
  VM_Magic.setMemoryWord(buf, slot);

  // (4) store the new buf pointer back
  p.GCTk_WriteBufferBumpPointer = buf;
}
```

b) Java slow path.

```
rlwinm  r3,r3,0,0,3     ; fast 2
cmpw    r4,r3           ; fast
bge     <exit>          ; fast
addis   r3,r2,1         ; slow
lwz     r3,-16508(r3)   ; slow 3
ori     r4,r16,0        ; slow
add     r30,r4,r3       ; slow
lwz     r3,0(r30)       ; slow 4
andi.   r0,r3,32767     ; slow 7
bne     <line 18>       ; slow
addis   r4,r2,1         ; slow
lwz     r4,-30240(r4)   ; slow
lis     r5,1            ; slow
addi    r5,r5,-32768    ; slow
mtctr   r4              ; slow
ori     r4,r5,0         ; slow
bctrl                   ; slow 8
addi    r3,r3,-4        ; slow 11
stw     r31,0(r3)       ; slow 12
stw     r3,0(r30)       ; slow 15
```

c) PowerPC instruction sequence.

Figure 1: The *slot* remembering write barrier used by the GCTk generational collectors.

```
public static final void writeBarrier(Object source) {
  int statusWord = VM_Magic.getIntAtOffset(source, OBJECT_STATUS_OFFSET);
  if ((statusWord & OBJECT_BARRIER_MASK) != 0) {
    GCTk_WriteBufferObject.insert(source, statusWord);
  }
}
```

a) Java fast path

```
public static final void insert(Object source,
                                int statusWord) {
  // (0) mark the source as "remembered"
  statusWord = statusWord ^ OBJECT_BARRIER_MASK;
  VM_Magic.setByteAtOffset(source, OBJECT_STATUS_OFFSET
                           + BARRIER_BIT_BYTE_OFFSET,
                           statusWord);
  ADDRESS src = VM_Magic.objectAsAddress(source);

  // (1) establish the bump pointer address
  VM_Processor p = VM_Processor.getCurrentProcessor();
  ADDRESS buf = p.GCTk_WriteBufferBumpPointer;

  // (2) grow buffer if necessary
  if ((buf & (WRITE_BUFFER_BUF_SIZE - 1)) == 0)
    buf = growBuffer(buf, WRITE_BUFFER_BUF_SIZE);

  // (3) add the source object
  buf -= BYTES_IN_WORD;
  VM_Magic.setMemoryWord(buf, src);

  // (4) store the new buf pointer back
  p.GCTk_WriteBufferBumpPointer = buf;
}
```

b) Java slow path.

```
lwz     r3,-8(r31)      ; fast 2
andi.   r0,r3,2         ; fast 3
beq     <exit>          ; fast
xori    r3,r3,2         ; slow 4
stb     r3,-5(r31)      ; slow 5
addis   r3,r2,1         ; slow
lwz     r3,-16508(r3)   ; slow 10
ori     r4,r16,0        ; slow
add     r30,r4,r3       ; slow
lwz     r3,0(r30)       ; slow 12
andi.   r0,r3,32767     ; slow 15
bne     <line 20>       ; slow
addis   r4,r2,1         ; slow
lwz     r4,-30240(r4)   ; slow
lis     r5,1            ; slow
addi    r5,r5,-32768    ; slow
mtctr   r4              ; slow
ori     r4,r5,0         ; slow
bctrl                   ; slow 16
addi    r3,r3,-4        ; slow 19
stw     r31,0(r3)       ; slow 20
stw     r3,0(r30)       ; slow 23
```

c) PowerPC instruction sequence.

Figure 2: The *object* remembering write barrier used by the GCTk generational collectors.

consumes the remembered set at collection time; it examines each entry to see if the remembered address still contains a pointer into the nursery. (The program may have overwritten the pointer with an uninteresting value after it was recorded in the remembered set.) If so, the collector marks the pointed-to object live. Otherwise, it ignores the pointer.

Figure 1a) illustrates the Java code for our fast path implementation. By locating the nursery and older generation on different sides of a major virtual memory alignment boundary (2^K), we are able to apply Stefanović et al's very cheap address order write barrier [SMM99] to generational collection. We put the nursery in high memory, and older generations in successively lower memory regions. We then simply mask the lower K bits in the target and if the source is less than the shifted target, we remember it (line 2 in Figure 1a)). We use two shifts to perform a mask to avoid a two-step direct mode `and`. In fact the two shifts are folded into a single PowerPC mask operation by the Jikes RVM optimizing compiler. This barrier generalizes to N generations, as long as each generation is contained within a 2^K-aligned virtual memory region, and the generation ordering is preserved in the memory organization. The PowerPC instruction sequence for the fast path appears in Figure 1c), instructions 1 though to 3.

When we remember a pointer, we put it in a *write buffer*, as illustrated in Figure 1b). The write buffer is implemented as a simple *sequential store buffer* [HD90, HMS92]. We implement the buffer using a chain of power-of-two sized chunks (`WRITE_BUFFER_BUF_SIZE` $= 2^n$). We exploit the power of two alignment of each chunk to perform a cheap bounds test (Figure 1b), line 7). The Jikes RVM optimizing compiler produces tight code for the slow path as shown in Figure 1c), instructions 4 through to 20. It further optimizes this code in context after inlining. The slot barrier is the default for the generational collectors in our garbage collection toolkit (GCTk).[3]

3.2.2 Object Remembering Write Barrier

The generational collectors that come with Jikes RVM implement an object remembering barrier [Ung84]. This barrier tests each pointer store, and remembers the source *object* if necessary. At collection time, the collector treats each remembered object as live and scans it for pointers into the nursery. It marks live any pointed-to nursery object. As with the slot remembering barrier, this barrier has a frequently executed fast path, and an occasionally executed slow path. We now describe our implementation of the object remembering barrier in GCTk. (Our implementation closely follows the Jikes RVM implementation. It differs only in the slow path. We use the fast power-of-two bounds check described above for the write buffer, and the Jikes RVM uses a comparison with an explicit end-of-buffer value, which requires an additional load.)

Figure 2a) illustrates the Java code for the fast path implementation. This code remembers a source object if the `OBJECT_BARRIER` bit is *set* in the source object's status word. Once it remembers the object, it clears the bit. Then it will not remember any subsequent stores for an object with a clear `OBJECT_BARRIER` bit. Note that because this bit is clear in any newly allocated object, the barrier automatically ensures that it never remembers nursery objects. When the collector copies an object into the higher generation, it sets the bit to ensure the barrier will remember subsequent stores to it. When it processes the remembered set, it resets each object's `OBJECT_BARRIER` bit (which was cleared during the write barrier). Note that this barrier is somewhat conservative because it remembers all old generation objects into which a pointer is stored regardless of whether the pointer is into the nursery. It need not store older to older generation pointers. The PowerPC instruction sequence for the object remembering barrier fast path appears as lines 1 through to 3 in Figure 2c).

The slow path for the object remembering barrier is identical to that for the slot remembering barrier except that it first clears the `OBJECT_BARRIER` bit in the source object. The Java code for this barrier is illustrated in Figure 2b), and the corresponding instruction sequence appears in Figure 2c).

4. METHODOLOGY

We explore the impact of write barrier inlining strategies by considering two metrics: *code quality* and *compilation workload*. The former measures how the choice of inlining strategies impact the performance of the compiled code, while the latter measures how the choice impacts on the amount of work the compiler must do to compile each barrier. Code quality is of obvious importance, and in a dynamic compilation context, such as Java, compilation workload is also important because the application is exposed to the compiler through just-in-time compilation.

We measure code quality by timing the SPEC JVM98 benchmarks in Jikes RVM, and measure compiler workload by timing the Jikes RVM optimizing compiler building a Jikes RVM boot image while running on the Sun HotSpot JVM. We describe this experimental environment below.

4.1 JIT and GC Environment

We use the Jikes RVM (formerly known as Jalapeño) and GCTk, which is a GC toolkit for Jikes RVM which we have recently developed.[4] We now overview each of these.

4.1.1 Jikes RVM

Jikes RVM is a high performance VM written in Java that includes an aggressive optimizing compiler [AAB$^+$00, AAC$^+$99]. Jikes RVM offers three compiler choices: *baseline*, all methods are compiled by a quick non-optimizing compiler; *optimizing*, all methods are compiled by an aggressive optimizing compiler; and *adaptive*, which initially baseline compiles methods and then adaptively recompiles hot methods using the optimizing compiler. The adaptive compiler uses sampling to select optimization candidates, and thus tends to make slightly different choices for each execution. This non-determinism makes the adaptive compiler a difficult platform for any detailed study of the optimizing compiler. Since our focus is on the behavior of optimizing compilation rather than when to use it, we measure the optimizing compiler for our key results. We use the adaptive compiler to contextualize our results with a realistic indicator of compilation workload.

Jikes RVM can be configured with two levels of ahead-of-time compilation. A minimal configuration only precompiles those classes essential to bootstrapping the VM (which does not include the optimizing compiler). We use the configuration which precompiles as much as possible, including key libraries and the optimizing compiler. We also turn off assertion checking for our experiments.[5]

---

[3] GCTk implements a number of collectors and works in the Jikes RVM. We designed it to be more general and extendible than the existing collectors in the Jikes RVM.

[4] GCTk is publicly available at http://cs.umass.edu/~gctk.

[5] This build-time configuration is known as *Fast*.

4.1.2 GCTk

GCTk is an efficient and flexible platform for GC experimentation that exploits the object-orientation of Java and the VM-in-Java property of Jikes RVM. We have implemented a number of GC algorithms in GCTk and found their performance to be similar to those of existing Jikes RVM GC implementations. The GCTk collectors used here (semi-space, fixed-nursery generational, and Appel-style generational) are well tuned. Each of the collectors shares a common infrastructure. The write barriers share common sequential store-buffer code. The Appel-style and fixed-nursery generational collectors share all the same code except their collection triggering rule.

4.2 Benchmarks

In Section 5.2, we measure compiler code quality by timing the SPEC JVM98 benchmarks. Table 1 shows key compile time and run time characteristics of each of the benchmarks.

	Compile Time			Run Time	
	Bytecodes compiled	Instructions generated	Write Barriers	Minimum heap	Pointer stores
_201_compress	13.5KB	136.8KB	240	19MB	21K
_202_jess	31.1KB	317.1KB	428	11MB	34.0M
_205_raytrace	21.7KB	202.8KB	303	15MB	6.6M
_209_db	15.0KB	154.0KB	236	22MB	30.0M
_213_javac	72.2KB	579.2KB	1929	27MB	25.5M
_222_mpegaudio	28.9KB	197.4KB	344	11MB	<1K
_227_mtrt	21.7KB	202.1KB	303	22MB	8.1M
_228_jack	35.0KB	319.7KB	519	13MB	9.4M

Table 1: Benchmark Characteristics

For these results, we compile each benchmark with the Jikes RVM optimizing compiler using fully inlined write barriers (which is the default behavior). For consistency between executions, we do not use the Jikes RVM adaptive compiler [AFG^{+}00]. The optimizing compiler compiles *all methods* required for the execution of each benchmark. The compilation includes the SPEC JVM98 harness and any additional libraries required by the benchmark that are not precompiled as part of the Jikes RVM boot image. Table 1 indicates the volume of Java bytecodes compiled, the volume of instructions produced, and the (static) number of write barriers compiled.

The minimum heap size column in Table 1 indicates the minimum heap in which the benchmark will run when using the Jikes RVM optimizing compiler and the GCTk Appel-style collector (this heap size is inclusive of the memory requirements of the optimizing compiler compiling the benchmark). The pointer store column shows the number of times the write barrier fast path executes in a run of the benchmark that does not include any compilation overhead. We obtain this statistic by running the benchmarks twice, and measuring the second iteration.

4.3 Experimental Platform

We use Jikes RVM version 2.0.2, turn off run-time assertion checking, and use the optimizing compiler with the build-time configuration which pre-compiles as many classes into the boot image as possible. These experiments use a 733MHz Macintosh PowerMac G4, with 32KB on-chip L1 data and instruction caches, a 256KB unified L2 cache, 1MB L3 off-chip cache, and 384MB of memory, running PPC Linux 2.4.10.

We measure compilation time for the various write barriers by compiling the Jikes RVM boot image using its optimizing compiler running on the Sun HotSpot Client VM version 1.3.1. For this experiment, we use a Dell Precision 340 with a 1.7GHz Intel Pentium 4, with an 8KB L1 data cache, a 12K L1 instruction cache, a 256KB unified L2 on-chip cache, and 512MB of memory running Linux 2.4.7.

5. RESULTS

This results section first shows the dichotomy between the slow and fast path execution frequencies for slot and object write barriers which motivates further exploration of write barrier code placement. We then compare application code quality using the out-of-line, partially inlined, and inlined write barriers. These results exclude both compilation costs and garbage collection time, and show that partial inlining yields a small but consistent advantage in execution time over inlining, which is slightly better than out-of-line, i.e., code quality improves when the slow path is out-of-line.

We then explore compilation costs. We measure the time taken by the Jikes RVM optimizing compiler to compile the methods in the boot image. We attain the expected result that inlining is slowest to compile, and out-of-line is fastest. The partially inlined barriers take between 20% and 25% less time to compile than the fully inlined barrier, with a similar reduction in the number of instructions generated. The magnitude of this difference is unexpected. Many systems inline the entire write barrier.

We also examine the combination of collector and write barrier. We find, not surprisingly, that despite the runtime overhead of the write barrier, the Appel-style generational collector performs substantially better than a semi-space collector. The partially inlined barrier always achieves the best performance for both just-in-time (on average 9.9% better than inlining) and ahead-of-time compilation (on average 1.4% better than inlining).

5.1 Slow Path Execution Frequency

As we discussed in Section 3.1, a two generational copying collector tracks pointers from the older to younger generation to avoid scanning and copying the older generation. Table 2 shows the relative number of dynamic pointer stores for each application, including compilation. We use this configuration instead of just the application take rates, because the latter were even smaller, further exaggerating the results.

We experiment with both a slot and object write barrier, and three different two generational collector configurations: an Appel-style collector, and two fixed-size nursery collectors with nurseries consisting of 5% and 10% of the usable heap. We measure the number of pointers stored over 8 heap sizes from 1 to 3.25 $\times$ the minimum heap size as reported in Table 1, and report the geometric mean of these frequencies. The heap size affects the nursery size, and thus the frequencies. These results are typical [Ste99], and demonstrate that these programs take the slow path less than 3% of the time, and in many configurations, much less than 1%. The next section shows we can exploit this dichotomy.

5.2 Application Code Quality

We investigate three options for write barrier placement: a fully inlined write barrier (*inline*), an out-of-line write barrier implemented with a direct method call (*out*), and a partially inlined write barrier with the fast path inlined and the slow path out-of-line (*partial*). This section presents results for the code quality of the application, as one would see in a traditional ahead-of-time compiler. In addition to excluding the compilation time, we also exclude the garbage

	Appel-style		5% Nursery		10% Nursery	
	Slot	Object	Slot	Object	Slot	Object
_201_compress	0.52%	0.47%	0.77%	0.71%	0.77%	0.71%
_202_jess	0.47%	0.56%	1.12%	1.72%	0.83%	1.03%
_205_raytrace	0.56%	0.56%	1.13%	1.13%	0.77%	1.13%
_209_db	0.16%	0.06%	0.47%	0.41%	0.27%	0.23%
_213_javac	0.58%	1.07%	1.07%	2.02%	1.07%	1.58%
_222_mpegaudio	0.58%	0.45%	1.67%	1.83%	1.19%	1.10%
_227_mtrt	0.38%	0.38%	0.93%	0.93%	0.64%	0.93%
_228_jack	2.47%	1.11%	1.11%	2.81%	1.11%	1.84%
Geometric mean	**0.37%**	**0.29%**	**1.34%**	**1.35%**	**0.93%**	**0.94%**

Table 2: Frequency with which the slow path is taken for three collector configurations and two write barriers.

collection time because of the impact of compiler-generated heap objects on garbage collection.

We measure *application time* by executing each benchmark twice using the optimizing compiler which does all its work on the first iteration. The application time is the total time for the second iteration of the benchmark less any garbage collection time during that iteration. We measure each benchmark five times and record the best time. We run the benchmark using 8 different heap sizes from 1 $\times$ to 3.25 $\times$ the minimum heap size and report the geometric mean of the best times for each of these heap sizes (see Table 1 for minimum heap size data). We also report the geometric mean of the application time results for all benchmarks, and normalize against the inlined barrier.

	Slot Remembering Barrier			Object Remembering Barrier		
	Inline	Partial	Out	Inline	Partial	Out
_201_compress	100.0%	99.0%	99.2%	100.0%	99.8%	100.4%
_202_jess	100.0%	99.0%	113.7%	100.0%	100.1%	105.9%
_205_raytrace	100.0%	92.9%	97.6%	100.0%	99.5%	101.6%
_209_db	100.0%	99.2%	102.2%	100.0%	100.0%	102.4%
_213_javac	100.0%	97.4%	103.4%	100.0%	98.3%	102.0%
_222_mpegaudio	100.0%	99.4%	100.7%	100.0%	99.6%	100.6%
_227_mtrt	100.0%	101.1%	105.0%	100.0%	100.1%	101.2%
_228_jack	100.0%	99.9%	104.6%	100.0%	99.9%	101.7%
Geometric mean	**100.0%**	**98.5%**	**103.2%**	**100.0%**	**99.7%**	**102.0%**

Table 3: Average application running time (excluding GC), normalized against running time with an inlined barrier.

Table 3 shows that partial inlining is the best choice for the slot barrier in all benchmarks, and the best for the object barrier on 5 of the benchmarks. The out-of-line barrier is the worst of the choices, on average 2% (the object barrier) and 3.2% (the slot barrier) worse than inlining, except for _201_compress and _202_jess. We were a little surprised the out-of-line barrier did not do worse, but this machine and compiler apparently implement a direct procedure call very well. On average, partial inlining offers an overall improvement over full inlining of about 1.5% for the slot barrier and 0.3% for the object barrier.

This result is a little surprising. Common practice might suggest that fully inlined code will perform better. However, the slow path frequency numbers in Table 2, the dynamic pointer store statistics in Table 1, and the compilation statistics reported in Table 4 all indicate instruction locality as one explanation. The rarely executed write barrier slow path is proliferated throughout the compiled code, increasing the code volume by around 30% over the out-of-line case (see Table 4). A second potential explanation is that increasing the *register pressure* by inlining the slow path degrades code quality. Most likely a combination of these effects accounts for the performance degradation suffered by the fully inlined code.

A call to a static method in Jikes RVM takes a minimum of three instructions and typically more.[6] Thus the call alone increases the number of instructions inlined by at least 100% on top of the fast path instruction sequence in the case of partial inlining. The compiler could reduce the mixing of hot fast-path and cold slow-path instructions in the partial inlining case by pushing the call sequence to the end of a code block [FG02]. We expect this optimization would further improve the performance of partial inlining, but do not explore it here.

For traditional ahead-of-time compilation, these results suggest that a partially inlined write barrier will attain consistent, but small execution time improvements for the slot barrier on a variety of benchmarks.

5.3 Compile-time Costs

Dynamically compiled languages like Java directly expose the application to the compiler. We now measure the compilation overhead for the three write barrier code placement strategies. In this section, we tease apart the compile time to optimize code with different write barriers, and its effect on compilation time itself (because the Jikes RVM compiler is subject to the choice of write barrier as well).

Because the Jikes RVM optimizing compiler normally executes in the context of Jikes RVM, changing the write barrier changes the performance and allocation of the compiler itself as it executes, i.e., the compiler must execute the same barrier that it is compiling. Direct measurements of the optimizing compiler within the Jikes RVM are thus problematic because the code quality issues raised in Section 5.2 are superimposed onto any variation in compiler workload the different barriers impose. For these reasons, we measure the Jikes RVM optimizing compiler when compiling the Jikes RVM boot image in the context of a host JVM (Sun's HotSpot JVM). This strategy holds constant the execution context for the compiler, and only changes the compiler's results due to the write barrier.

The compilation of the boot image is a substantial test of the optimizing compiler. It compiles 9743 methods and 43004 write barriers. For the fully inlined slot barrier, it generates 11.4MB of instructions, and spends about 15 minutes in compilation. The results in Table 4 show the impact of the three code placement strategies on the compiler workload, as measured by instructions generated, and time taken to compile the Jikes RVM boot image classes. As with our other results, we report the best time from five runs.

	Slot Remembering Barrier			Object Remembering Barrier		
	Inline	Partial	Out	Inline	Partial	Out
Output	100.0%	73.8%	66.9%	100.0%	76.6%	69.4%
Time	100.0%	75.2%	62.2%	100.0%	79.1%	69.0%

Table 4: Compiler workload expressed in terms of instructions generated (Output) and compilation time (Time), for the slot and object barriers all normalized to inlining using the Jikes RVM on the HotSpot JVM.

These results indicate that fully inlining write barriers substantially increases the compiler workload over the other choices. Partial in-

[6] Note that in Figures 1c) and 2c), the static method calls take seven instructions (lines 11–17 and 13–19 respectively).

	Appel-style generational collector						Semi-space
	Slot Remembering Barrier			Object Remembering Barrier			
	Inline	Partial	Out	Inline	Partial	Out	
_201_compress	100.0%	97.0%	97.6%	99.4%	96.7%	99.6%	108.7%
_202_jess	100.0%	86.2%	93.6%	95.0%	88.6%	91.4%	207.8%
_205_raytrace	100.0%	88.5%	92.2%	87.5%	82.1%	86.4%	159.8%
_209_db	100.0%	98.4%	100.2%	99.8%	98.9%	101.4%	114.2%
_213_javac	100.0%	81.5%	84.5%	97.6%	83.0%	84.8%	119.9%
_222_mpegaudio	100.0%	90.4%	91.9%	97.3%	91.6%	93.0%	104.5%
_227_mtrt	100.0%	90.5%	96.2%	97.4%	93.7%	97.1%	138.1%
_228_jack	100.0%	87.0%	89.9%	95.7%	87.5%	89.9%	141.4%
Geometric mean	**100.0%**	**89.8%**	**93.2%**	**96.1%**	**90.1%**	**92.8%**	**133.5%**

a) Total Time *including* compilation (first iteration).

	Appel-style generational collector						Semi-space
	Slot Remembering Barrier			Object Remembering Barrier			
	Inline	Partial	Out	Inline	Partial	Out	
_201_compress	100.0%	98.8%	99.1%	99.2%	98.9%	100.5%	113.0%
_202_jess	100.0%	98.9%	112.6%	97.6%	97.5%	105.4%	372.2%
_205_raytrace	100.0%	93.1%	97.7%	94.4%	93.9%	101.0%	238.4%
_209_db	100.0%	99.1%	102.1%	99.4%	99.3%	102.3%	116.5%
_213_javac	100.0%	96.5%	101.2%	101.1%	98.3%	100.2%	144.6%
_222_mpegaudio	100.0%	99.4%	100.7%	99.8%	99.4%	100.6%	99.1%
_227_mtrt	100.0%	100.8%	104.8%	101.2%	102.0%	101.5%	174.8%
_228_jack	100.0%	99.4%	103.8%	99.5%	98.9%	101.3%	204.6%
Geometric mean	**100.0%**	**98.2%**	**102.6%**	**99.0%**	**98.5%**	**101.6%**	**166.8%**

b) Total Time *excluding* compilation (second iteration).

Table 5: Total running time for one iteration of each of the SPEC JVM98 benchmarks, including garbage collection costs.

lining reduces total compiler costs by around 20% to 25%, and the out-of-line barrier reduces it more, by 30% to 35%. Note also that the improvements in compilation time are approximately linear in the reduction in instructions generated. This result means that the compilation work is proportional to the number of instructions it generates. (Good news for the compiler!) For our study, the fundamental problem is that fully inlining the write barrier increases the compiler load significantly without any corresponding increase in the resulting code performance, and, in fact, slightly degrades code quality.

When compilation time exceeds about 10% of total time and becomes more dominant, these results even suggest an out-of-line barrier as a good choice! As we show in the next section, this suggestion does not hold up in our system. Instead, the cost of executing the out-of-line barrier in the very pointer store intensive compiler degrades the compiler performance by more than the time to compile it.

5.4 Total Running Time

Having shown that a partially inlined write barrier executes faster and substantially reduces compiler workload compared to a fully inlined barrier, and that the out-of-line barrier is more ambiguous with respect to these criteria, we now examine their overall execution time impact. Because the total running time is so heavily impacted by the level of compiler activity, we show best and worst case results for compiler activity. In this context, best is no compilation (everything is already compiled) and worst is all application methods compiled at the time of their execution.

Table 5 illustrates total running time results for slot and object barriers compiled inlined, partial, and out-of-line using the Appel-style generational collector. We include for comparison the running time for the simple semi-space collector, which has no write barrier. We again pick the best of 5 runs and compute the geometric mean for each program and collector over 8 heap sizes. All of these numbers are inclusive of garbage collection time. All but two of the SPEC JVM98 programs spend between 20 and 40% of their time in the Jikes RVM compiler; _201_compress (9-13%) and _209_db (5-7%) are the exceptions. Since the semi-space collector has no barrier, the lowest compilation times are for the semi-space collector (e.g., a compile time of 5% for _209_db with no write barrier). When compiling the Jikes RVM boot image, the full, partial, and out-of-line barriers slowed the compiler down by 71%, 30%, and 10% respectively, relative to the semi-space collector.

Table 5a) shows times for the first iteration of each benchmark and is thus *inclusive* of compilation costs. Since the optimizing compiler compiles all methods prior to their execution, this table represents the *worst* case in terms of compilation load. Comparing between inline, partial, and out-of-line, a partially inlined slot barrier performs best, on average 10.2% better than inlining. With the compiler in the picture, an out-of-line slot barrier is better than an inlined one. The inlined object barrier performs better than an inlined slot barrier, but slot is the best using partial inlining.

Table 5b) shows times for the second iteration of each benchmark and is thus *exclusive* of compilation costs. This table represents no compilation load, but does include the garbage collection time. In these results, inlining always performs better than an out-of-line barrier. Partial inlining offers a small and consistent advantage over full inlining. These results are consistent with the code quality improvements observed in Section 5.2 where we exclude garbage collection performance.

With compilation, even an out-of-line barrier is better than an inlined barrier for the slot barrier by 6.8% on average, and for the object barrier, they differ only by around 3.5% on average. When compilation is eliminated or minimized, the out-of-line barrier takes its place as the worst performing barrier.

Regardless of the compilation cost or the barrier code placement choice, a semi-space collector with no barrier performs worse than Appel with a barrier. As expected, the incrementality of the Appel generational collector always yields significantly better performance than collecting the whole heap every time with the semi-space collector. For example, an inlined slot barrier with Appel is 33% faster than the semi-space collector with the compiler and 66% without the compiler. In fact, these results are somewhat understated because the second iteration will have lower memory requirements than the first because the heap will not contain any compiler objects. In some cases, the semi-space collector is barely exercised in the second iteration (e.g., _222_mpegaudio and _209_db).

These results indicate that the impact of a partially inlined slot write barrier ranges from a 0.8% degradation to 18.5% improvement over a fully inlined write barrier, depending on the level of compilation. The more compilation, the greater the advantage partial inlining has over inlining. We also measure the activity of the adaptive compiler which is typically 12.5% of execution time. This level of activity is between half and one third that of the optimizing compiler, and will thus see the benefit of partial inlining.

We obtain very similar results using the same experiment with a fixed-size nursery collector which, as we show, takes the slow path more frequently. Thus, we believe these results will hold across different collectors and compilers.

6. CONCLUSION

The write barrier is a key to the efficiency of many modern garbage collectors. Garbage collectors pay mutator write-barrier overheads to reduce copying overhead and improve total performance. Many researchers have spent their time trying to minimize the impact of the write barrier on the mutator. We show that the way in which the barrier is compiled can have a considerable impact on overall performance, even if it is highly optimized. Write barriers are prolific and have highly regular bimodal execution patterns. These two characteristics bring into question a common practice of inlining write barriers.

We find that fully inlining write barriers not only produces suboptimal code, but dramatically increases the compiler's workload. By contrast, partial inlining reduces the compiler's workload by between 20% and 25% as compared to full inlining, and consistently leads to better quality code. This result is general in the context of write barrier compilation. Furthermore, it is likely to extend to other contexts, notably compiling allocation sequences which are also prolific and typically have well defined fast and slow paths.

7. ACKNOWLEDGEMENTS

We want to thank IBM Research and in particular Mark Wegman, Vivek Sarkar, Michael Hind, and Mark Mergen for giving us an early version of the Jikes RVM which enabled us to undertake this research. We also thank Brendon Cahoon, Stephen Fink, David Grove, Michael Hind, Richard Jones, and Eliot Moss for their input and discussions.

8. REFERENCES

[AAB+00] Bowen Alpern, Dick Attanasio, John J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, D. Lieber, V. Litvinov, Mark Mergen, Ton Ngo, J. R. Russell, Vivek Sarkar, Manuel J. Serrano, Janice Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1):211–238, February 2000.

[AAC+99] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324, Denver, CO, October 1999. ACM Press.

[AFG+00] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA'00 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 35(10) of *ACM SIGPLAN Notices*, pages 47–65, Minneapolis, MN, October 2000. ACM Press.

[AFSS00] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Boston, MA, 2000.

[AKPY98] Alain Azagury, Elliot K. Kolodner, Erez Petrank, and Zvi Yehudai. Combining card marking with remembered sets: How to save scanning time. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 10–19, Vancouver, October 1998. ACM Press. ISMM is the successor to the IWMM series of workshops.

[App89] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[ASG97] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 134–145, Las Vegas, NV, June 1997.

[BJMM02] Stephen M. Blackburn, Richard E. Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, volume 37(5) of *ACM SIGPLAN Notices*, Berlin, Germany, June 2002. ACM Press.

[CFS+00] A.M. Cheadle, A.J. Field, Marlow S., S. Peyton-Jones, and R.L While. Non-stop Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, volume 35(9) of *ACM SIGPLAN Notices*, pages 257–267, Montreal, September 2000. ACM Press.

[Cha92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Objected-Oriented Programming Language*. PhD thesis, Stanford University, March 1992.

[CHK92] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, pages 96–105, Oakland, CA, April 1992.

[CHKT93] K. Cooper, M. W. Hall, K. Kennedy, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–501, April 1993.

[CHL98] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, volume 33(5) of *ACM SIGPLAN Notices*, pages 162–173, Montreal, June 1998. ACM Press.

[CHT92] Keith D. Cooper, Mary W. Hall, and Linda Torczon. Unexpected side effects of inline substitution. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, March 1992.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[CWB86] Patrick J. Caudill and Allen Wirfs-Brock. A third-generation Smalltalk-80 implementation. In Norman Meyrowitz, editor, *OOPSLA'86 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 21(11) of *ACM SIGPLAN Notices*, pages 119–130. ACM Press, October 1986.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[DC94] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 273–282, Orlando, FL, June 1994.

[FG02] Stephen J. Fink and David Grove. Personal communication. January 2002.

[FT00] Robert Fitzgerald and David Tarditi. The case for profile-directed selection of garbage collectors. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, pages 50–58, Minneapolis, MN, October 2000.

[Ḧ93] Urs Hölzle. A fast write barrier for generational garbage collectors. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[HD90] Richard L. Hudson and Amer Diwan. Adaptive garbage collection for Modula-3 and Smalltalk. In Eric Jul and Niels-Christian Juul, editors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[HH93] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In Andreas Paepcke, editor, *OOPSLA'92 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 27(10) of *ACM SIGPLAN Notices*, pages 92–109, Vancouver, British Columbia, October 1992. ACM Press.

[JL96] Richard E. Jones and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[LH83] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[Sch77] R. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, September 1977.

[SMM99] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 370–381, Denver, CO, October 1999. ACM Press.

[Sob88] Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

[Ste99] Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[WM89] Paul R. Wilson and Thomas G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, 1989.