Barriers Reconsidered, Friendlier Still! *

Xi Yang†, Stephen M. Blackburn†, Daniel Frampton†, Antony L. Hosking‡

†Australian National University ‡Purdue University

†{Xi.Yang, Steve.Blackburn, Daniel.Frampton}@anu.edu.au ‡hosking@cs.purdue.edu

Abstract

Read and write barriers mediate access to the heap allowing the collector to control and monitor mutator actions. For this reason, barriers are a powerful tool in the design of any heap management algorithm, but the prevailing wisdom is that they impose significant costs. However, changes in hardware and workloads make these costs a moving target. Here, we measure the cost of a range of useful barriers on a range of modern hardware and workloads. We confirm some old results and overturn others. We evaluate the microarchitectural sensitivity of barrier performance and the differences among benchmark suites. We also consider barriers in context, focusing on their behavior when used in combination, and investigate a known pathology and evaluate solutions. Our results show that read and write barriers have average overheads as low as 5.4% and 0.9% respectively. We find that barrier overheads are more exposed on the workload provided by the modern DaCapo benchmarks than on old SPECjvm98 benchmarks. Moreover, there are differences in barrier behavior between in-order and out-of-order machines, and their respective memory subsystems, which indicate different barrier choices for different platforms. These changing costs mean that algorithm designers need to reconsider their design choices and the nature of their resulting algorithms in order to exploit the opportunities presented by modern hardware.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection), Run-time environments

General Terms Experimentation, Languages, Performance, Measurement

Keywords Write barriers, Memory management, Garbage collection, Java

1. Introduction

Software read and write barriers are small code fragments transparently inserted into a program by the compiler or interpreter to mediate run-time accesses to memory. By observing and/or intercepting a program's accesses, they allow the run-time system to: a) present richer memory abstractions, and b) transparently implement aggressive memory management strategies. For example, array bounds checks enforce memory safety and read and write barriers are used to ensure correctness of concurrent garbage collection. The creative possibilities opened up by the mediating role of barriers are large. However, because barriers add instructions to the program, any opportunities they present are held in tight check by performance concerns. Because their overhead is often hard to measure, in practice it is typically *perceptions of overhead* that curtail the creative use of barriers.

Software barriers are particularly interesting today because of the growing use of garbage collected languages, and simultaneously, significant disruption to hardware trends. These developments invite deeper investigation into barrier costs because they indicate both a growing dependence on barriers and a growing need for creative memory management strategies that might minimize the impact of disruptive and complex hardware changes. Furthermore, as hardware evolves the folklore surrounding software barrier costs must also be re-examined.

We measure the mutator cost of a range of barriers on a range of hardware and Java workloads. We find that: a) modern benchmarks expose the overheads of barriers more than older benchmarks did, b) write barrier costs are lower on modern machines, and c) barrier overheads can be sensitive to microarchitecture. We consider commonly used barriers as well as more fundamental barriers from which other barriers can be composed. This strategy allows us to tease apart the sources of overhead and will guide future implementers in understanding important influences on barrier overhead.

We compare our findings with the prior study by Blackburn and Hosking [5]. We took some care to optimize each barrier. Using similar hardware, we measured a much lower overhead for a read barrier of 8.5%, down from 15.9%, which we largely attribute to our optimization of the barrier code. The prior work was published before the DaCapo benchmarks became available [11]. We found that the newer benchmarks expose the barrier overheads more than the older ones, so the results in the previous study are understated. We found that on a modern i7-2600 processor, read and write barriers have average overheads as low as 5.4% and 0.9% respectively. We were surprised to see that the overhead of the read barrier on the in-order Atom is almost the same as on the aggressive out-of-order i7 processor. On the other hand the write barrier overhead on the Atom is twice that of the i7. We also examine a barrier pathology [14] that is known to exist in a popular commercial Java virtual machine (JVM). Our study of this pathology for card marking barriers shows it to be very real. We evaluate a number of possible solutions to this pathology.

These changing costs mean that algorithm designers need to reconsider their design choices and the nature of their resulting algorithms in order to exploit the opportunities presented by modern hardware.

2. Related Work

In 2004 Blackburn and Hosking [5] measured the cost of barriers on hardware of that era, including both x86 and PowerPC. Our methodology is essentially the same. Here, we focus on modern x86 platforms, and consider a broader range of benchmarks and additional useful barriers. We explore barrier costs more precisely us-

* This work supported by grants ARC DP0666059 and NSF CCF-0811691.

ISMM'12 June 15–16, 2012, Beijing, China.

Reprinted from ISMM'12, [Unknown Proceedings], June 15–16, 2012, Beijing, China., pp. 1–9.

ing hardware performance counters. Our results using similar hardware broadly confirm theirs, but we observe a number of interesting differences in behavior on current platforms, such as a greatly reduced read barrier overhead. Details appear in Section 5.

Previous direct studies of barrier overheads used less direct approaches to studying barrier cost. Zorn [26] timed the cost of barrier implementations in a tight loop and then used heap access profiles for several large Lisp programs to estimate the total cost of the barriers executing in those programs. These ranged from 2% to 6% for inlined fast path write barriers, and up to 20% for read barriers. Because Lisp is dynamically-typed the overhead includes the cost of dynamically filtering out non-pointer accesses.

Real-time copying collectors commonly use an indirection barrier (for both reads and writes) so that all accesses forward to the most recent copy of the object in constant time [12]. Bacon et al. [4] measured the cost of this style of barrier as 4% on average and 10% maximum for the SPECjvm98 benchmarks. To obtain these results they applied standard optimizations (such as common-subexpression elimination) and special-purpose optimizations (such as barrier-sinking, which moves the barrier to the point of use, so allowing the null-check for the access to combine with that for the barrier).

There is much other related work looking at the effects of different barriers on both mutator and collector execution [2, 3, 9, 13, 18–21, 24, 25]. Some use hardware and/or operating system support. As in Blackburn and Hosking [5], we focus here on pure software techniques. Blackburn and McKinley [7] detail the impact of barriers on compile times and code quality. Hirzel et al. [17] designed a region-based collector to avoid the need for barriers entirely so as to eliminate their overhead. Hellyer et al. [16] study the locality effects of barriers in concurrent collectors.

3. Barriers

We now describe each of the barriers we evaluate. We first describe the most simple barriers, which we term *primitive* barriers, before describing *compound* barriers, which combine primitive barriers to build more sophisticated barriers. We consider both because primitive barriers have the attraction of being easier to analyze, while compound barriers may be more interesting because of their broader application.

In practice, barriers also occur when array elements are written, arrays are copied, and in a number of other more obscure circumstances. Table 2 shows the relative frequency of the important cases. Depending on the barrier semantics, an array copy (`System.arraycopy`) can be greatly optimized, and need not consist of naïve element-by-element application of the simple barrier. In our performance analysis, we apply array copy optimizations aggressively. For simplicity, in this section we present only the code for barriers corresponding to `putfield` and `getfield` bytecode operations.

3.1 Primitive Barriers

Figure 2 presents Java and x86 assembly code for each of the primitive barriers. Each of the Java code segments belongs in the context of the skeleton code shown in Figure 1. We assume that a read barrier must load at least the reference `value` held in the source (`src`) object's `slot` and return the (equivalent, possibly modified) reference (i.e., the read barrier substitutes for the load). In contrast, we assume that the actual store of the target (`tgt`) reference is performed separately from the write barrier (i.e., the write barrier is additional to the store). Figure 2, and subsequent figures showing barrier code, present only the barrier operations (without showing the actual heap load/store).

```
1  @Inline
2  public ObjectReference objectReferenceRead(
3    ObjectReference src,
4    Address slot)
5  {
6    ObjectReference value;
7    value = slot.loadObjectReference;
8    /* barrier-specific code here */
9    return value;
10 }
```

(a) Read

```
1  @Inline
2  public final void objectReferenceWrite(
3    ObjectReference src,
4    Address slot,
5    ObjectReference tgt)
6  {
7    /* barrier-specific code here */
8  }
```

(b) Write

Figure 1: Skeleton code for generic read and write barriers.

Card The card marking barrier is widely used to identify inter-generational pointers in a generational garbage collector. It consists of an unconditional store of a byte to a computed offset in a card table. We improved over the previous version of the card marking code [5] by making the card base constant, reducing the barrier from three instructions to two. At collection time, the card table is scanned to identify the location of regions (cards) that contain mutated reference fields. The size of each card (2^9 bytes in our case) dictates how precise the card table is. A more precise card table has a large footprint but reduces the scanning load at collection time. The card marking barrier is widely used in commercial Java virtual machines (JVMs) and is popular primarily because of its simplicity and the fact that it is unconditional. The barrier has a number of potential pathologies which we explore in detail in Section 5.3. Moreover, frequent unconditional stores generate significant write traffic, so if write bandwidth is scarce, the card marking barrier may perform poorly. The collection-time cost of the barrier is a function of the size of the table, which is typically linear in the size of the heap. If the heap is very large the scanning overhead may be considerable. This overhead is imposed on every nursery collection, so discouraging small nurseries, which may otherwise be a desirable choice.

Object The object barrier is also used in generational collection and also conservatively records areas containing mutated reference fields. However, it works by remembering objects whose reference fields were mutated rather than remembering regions of memory (cards). The barrier is conditional. It checks the header of the object being mutated and only remembers the object if a bit is set in the header to indicate that the object has not yet been remembered. The barrier slow path then clears the bit so that the object is not remembered again. The collector scans each remembered object and re-sets its not-remembered bit. A simple optimization allocates new objects with the bit clear, which means they will not be needlessly remembered. We optimized Jikes RVM's implementation of the check for the non-remembered bit (`HeaderByte.isUnlogged()`) to use the x86 `TEST` instruction, reducing the four lines of assembly code reported in Blackburn and Hosking [5] down to two.

Because it remembers objects rather than cards, the object barrier is more precise than the card marking barrier. Because it is conditional, the object barrier generates much less write traffic. The first method call within the slow path (line 2) is very small and is explicitly inlined with an `@Inline` pragma. By contrast, the

```
1 LOG_CARD = 9;
2 offset = src.rshl(LOG_CARD);
3 Gen.cardBuffer.store((byte)1, offset);
4
```

```
1 SHR     EAX     9
2 MOVB    1048576[EAX] 1
3
4
```

(a) Card

```
1 if (HeaderByte.isUnlogged(src)) {
2   HeaderByte.markAsLogged(src);
3   modbuf.insertOutOfLine(src)
4 }
```

```
1 TEST    -8[EBX] -128
2 JEQ     45
3
4
```

(b) Object

```
1 if (!Gen.inNursery(slot) &&
2       Gen.inNursery(tgt))
3   remset.insert(slot);
4
```

```
1 CMP     EDX     -1530920960
2 JGE     0
3 CMP     EAX     -1530920960
4 JGE     38
```

(c) Boundary

```
1 REGION_SIZE = 32*1024*1024;
2 if (tgt.toAddress().NE(Address.zero()))
3   if (src.xor(tgt).GE(REGION_SIZE))
4     remset.insert(slot)
```

```
1 TEST    EAX     EAX
2 JEQ     0
3 XOR     EBX     EAX
4 CMP     EBX     33554432
5 JGE     36
```

(d) Zone

```
1 return value.and(^3);
2
3
4
```

```
1 AND     EDX     -4
2
3
4
5
```

(e) Read

```
1 if (value.and(1).NE(0))
2   return insertNOP(src);
3 return value;
4
```

```
1 TEST    EAX     1
2 JNE     35
3
4
5
```

(f) Conditional Read

Figure 2: Primitive barriers, showing Java source for the barrier and x86 assembler for the fast path.

second method is larger and is explicitly forced out of line with the `@NoInline` pragma. The barrier slow path therefore includes a couple of instructions and a call. We show later (in Table 4) that the object barrier only remembers an object once every thousand times it is invoked. In the presence of a good branch predictor, the object barrier's very low take-rate may help mitigate the fact that the barrier is conditional. This very low take-rate may also argue for forcing the entire slow path out of line, reducing i-cache pressure, however we have not explored this.

Boundary The boundary barrier remembers the address of any reference that crosses a fixed address boundary in a particular direction. This can be used as an inter-generational barrier when the nursery is strictly higher or lower than the mature heap in the address space. If the nursery is in high memory then null references are automatically ignored by the barrier. At collection time all of the remembered reference fields are scanned by the collector. Unlike card marking and the object barrier, this barrier will log duplicates when the program changes a given field repeatedly. Table 4 shows that on average the boundary barrier remembers fields 19 times as often as the object barrier remembers objects.

Zone The zone barrier uses exclusive or to test for and remember all references that cross power-of-two-aligned regions of the address space. In our example we use a large 32 MB zone. So the address space is broken into 32 MB zones on 32 MB alignment boundaries. Before checking whether the reference crosses boundaries, we check whether the target is null. This is because 42% of references that cross boundaries are due to null-assignments to reference fields. Table 4 shows that the zone barrier remembers pointers very frequently, 390 times as often as the object boundary.

Read The unconditional read barrier is a very simple bit-masking barrier, masking the two low-order bits of the reference value before it is returned to the caller. The motivation for such a barrier is that it can be used to cleanse addresses that have had their low-order bits tainted. In a system such as a JVM where all addresses are guaranteed to be at least word-aligned this barrier safely allows 'stealing' of the low order bits. Although this barrier is very simple, reads occur an order of magnitude more frequently than writes

```
1 return value.and(^3);
2
3
4
```

```
1 AND     EDX     -4
2
3
4
5
```

(a) Read

```
1 Word old = slot.load();
2 if (!old.and(3).isZero())
3   slowPath(slot, tgt);
4
```

```
1 AND     EDI     -4
2 MOV     EAX     ESI
3 TEST    [EBX]   1
4 JNE     105
5
```

(b) Write

Figure 3: The bit-stealing barrier, showing Java source for the barrier and x86 assembler for the fast path.

(see Table 2), so it has far greater potential to slow the program down. This barrier has been implemented in hardware on a number of mainstream RISC architectures that require memory operations to be word-aligned, but is not supported on x86.

Conditional Read The conditional read barrier will remember the loaded reference if the reference has its low order bit set. When run in isolation, the test will always fail, so the slow path will never be executed. However, the compiler cannot identify this fact, so the test is not optimized away. We made an improvement over the conditional barrier used by Blackburn and Hosking [5]. They used `value.and(1).NE(1)` where we use `value.and(1).NE(0)`. Our code is faster for two reasons: a) it compiles down to the `TEST` instruction rather than an immediate mode comparison, and b) by reversing the sense of the comparison, the taken case (line 3 of the Java code) does not require a branch.

3.2 Compound Barriers

These barriers combine one or more of the primitive barriers to form more complex barriers.

Bit-Stealing The bit-stealing barrier 'steals' one or both of the low order bits of references in a context where the bits are unneeded because references are guaranteed to be word-aligned. The Java and assembler code for this barrier is shown in Figure 3. The barrier consists of both a write barrier that sets the bits under some condition, and a read barrier that masks the bits out before use. The read barrier is identical to the unconditional read barrier of Section 3.1. The bit-stealing barrier is analogous to the object barrier, but instead of conditionally remembering modified objects it conditionally remembers modified fields. It is therefore precise rather than conservative. Once the field is remembered, a low order bit is set ensuring that it is not re-remembered until the bit is cleared again at the next collection. Like the object barrier, the bit-stealing barrier does not remember duplicates.

Hybrid Object/Region The hybrid barrier presented by Blackburn and Hosking [5] simply uses the object barrier for stores to scalar objects and the boundary barrier for stores to arrays.

4. Methodology

We use similar methodology to that introduced by Blackburn and Hosking [5]. In particular, we use the ignore remsets feature they added to MMTk, which allows us to implement and measure the overhead of barriers. In this section, we present the software, hardware, and measurement methodologies we use, in particular highlighting areas where we differ from their previous barrier overhead study.

Measurement Methodology We implement all barriers in MMTk [10], based on JikesRVM's [1] production configuration that uses a generational Immix [6] collector. By default the generational collector relies on a write barrier to remember references from the mature space to the nursery, allowing an efficient partial trace of the heap. However, the need to gather this remembered set for correctness limits the experiments that can be performed. We use the approach of Blackburn and Hosking [5], where the effect of a nursery collection is simulated by performing a full trace and *only* collecting the nursery. This removes the requirement of gathering the remembered set for correctness, making it possible to measure other systems, including a baseline no-barrier system.

Our focus is on barrier overheads, and so we report mutator time, rather than total or garbage collection time. This is particularly important when using the ignore remset approach, because the full trace to simulate the remembered set may be quite expensive. We use a 32MB fixed size nursery, which performs well for our benchmarks. We execute with a generous heap size: $6\times$ the minimum required for each individual benchmark, so nursery allocation and collection dominates. We run each benchmark 20 times (20 invocations) and report the average. We also report 95% confidence intervals for the average using Student's t-distribution.

Controlling Non-Determinism To reduce perturbation due to dynamic optimization and to maximize the performance of the underlying system that we improve, we use a *warmup replay* methodology, which was recently committed to Jikes RVM, and is a refinement to the pseudoadaptive approach used by Blackburn and Hosking [5]. Before executing any experiments, we gathered compiler optimization profiles from the 10th iteration of each benchmark. When we perform an experiment, we execute one complete iteration of each benchmark without any compiler optimizations, so as to load all the classes and resolve methods. We next apply the benchmark-specific optimization profile after which no further compilation occurs. We then measure and report the subsequent iteration. This methodology greatly reduces non-determinism due to the adaptive optimizing compiler and improves underlying performance compared to the prior replay methodology which is used by Blackburn and Hosking [5].

To reduce the non-determinism introduced by the operating system's scheduler on multicore machines, we run using a single core (except for our microbenchmark results that investigate contention when using a card marking barrier).

Metrics We use performance counters to help understand barrier costs. We measure execution time, retired instructions, and instruction cache misses. We report percentage overhead for each of these measures (Δt, Δi, and Δi_{miss}) relative to the *no barrier* configuration. As an indicative measure of how costly the instructions added by each barrier are, we also report $\Delta t/\Delta i$.

Hardware and Software Environment We use three IA32 architectures to explore the role microarchitecture has on barrier overhead: 1) a recent Intel Core i7 2600 processor, 2) an in-order Atom D510, and 3) an older Pentium 4 (P4) D 820 machine (similar to the machine used in the previous study). The i7 represents the current mainstream multicore processor. Unlike the P4, which has a deep superscalar pipeline, the i7 has a more modest out-of-order pipeline with a powerful memory subsystem. The Atom tries to improve energy efficiency by using a simpler in-order pipeline. Table 1 shows the parameters of these three architectures.

Operating System We use Ubuntu 10.04.01 LTS server distribution running with a 64-bit (x86_64) 2.6.32-24 Linux kernel.

Benchmark Properties A key way we improve on previous work is the use of a more comprehensive, modern set of benchmarks. We draw the benchmarks from the DaCapo suite [11], the SPECjvm98 suite [23], and pjbb2005 [8] (a fixed workload version of SPECjbb2005 [22] with 8 warehouses that executes 10,000 transactions per warehouse). We use benchmarks from both 2006-10-MR2 and 9.12 Bach releases of DaCapo to enlarge our suite and because a few 9.12 benchmarks do not execute on Jikes RVM.

Table 2 shows the frequency of operations that may trigger barriers, expressed as a number of operations per millisecond. We include results for reference field and array load/store operations as well as array copy operations, which may invoke a special barrier to avoid performing a naïve element-by-element copy in a loop. The results show both that these statistics vary considerably between benchmarks, and also that the benchmarks used in the previous study are not representative. In particular, we see that the four benchmarks with the lowest reference `putfield` rates are all found within the seven SPECjvm98 benchmarks.

5. Results

We now report the barrier overheads, starting with a detailed evaluation of the costs on modern architectures before discussing microarchitectural sensitivity and examining a case study in pathological write barrier performance.

5.1 The Cost of Barriers on Modern Architectures

We start by examining the cost of read and write barriers on modern hardware. Table 3 summarizes these results and reproduces corresponding numbers from Blackburn and Hosking [5]. All numbers except for the right-most column ($\Delta t/\Delta i$) are expressed as percentage overhead compared to a base case with no barrier. We include average 95% confidence intervals in grey beneath the corresponding mean. We include a column P4$\star$ that uses a set of benchmarks similar to Blackburn and Hosking [5] as well as similar hardware.

Note that our data differs from Blackburn and Hosking [5] in at least three significant respects, each of which is covered in detail in Section 4: a) our benchmarks are larger and newer, b) our hardware is newer (excepting P4 which approximates the P4 used in the

Architecture	Pentium 4	Atom D510	i7-2600
Model	P4D 820	Atom D510	Core i7-2600
Technology	90nm	45nm	32nm
Clock	2.8GHz	1.66GHz	3.4GHz
Cores × SMT	2 × 2	2 × 2	4 × 2
L2 Cache	1MB × 2	512KB × 2	256KB × 4
L3 Cache	none	none	8MB
Memory	1GB DDR2-400	2GB DDR2-800	4GB DDR3-1066

Table 1: Processors used in our evaluation.

Benchmark	Reference Fields		Reference Arrays			
					Arraycopy	
	Get/μs	Put/μs	Load/μs	Store/μs	Call/μs	Elem/μs
compress	117.41	0.00	0.01	0.00	0.00	0.00
jess	61.98	2.08	42.87	1.99	1.33	9.34
db	75.10	0.66	26.03	5.77	0.00	0.60
javac	69.51	5.86	7.08	0.77	0.00	0.01
mpegaudio	37.34	0.92	60.15	0.00	0.00	0.00
mtrt	68.58	0.69	37.37	0.93	0.00	0.00
jack	52.60	8.86	12.86	2.06	0.01	0.09
SPECjvm mean	*68.93*	*2.72*	*26.62*	*1.64*	*0.19*	*1.43*
antlr	75.21	1.99	1.95	0.19	0.00	0.01
avrora	40.83	1.73	5.71	0.01	0.00	0.00
bloat	81.11	19.12	12.96	0.30	0.00	0.00
eclipse	51.42	2.18	11.97	2.93	0.08	1.75
fop	43.53	1.50	6.17	0.06	0.00	0.02
hsqldb	79.73	6.09	17.40	1.68	0.00	0.28
jython	64.88	6.48	18.45	3.13	0.11	1.07
luindex	61.77	4.90	18.79	0.46	0.00	0.03
lusearch	69.81	6.31	4.61	0.17	0.00	0.00
pmd	63.04	8.02	13.31	0.92	0.01	0.03
sunflow	69.49	3.18	19.13	0.01	0.00	0.00
xalan	60.27	2.82	5.28	1.49	0.00	0.01
DaCapo mean	*63.43*	*5.36*	*11.31*	*0.95*	*0.02*	*0.27*
pjbb2005	42.47	7.94	13.01	1.96	0.00	0.02
min	*37.34*	*0.00*	*0.01*	*0.00*	*0.00*	*0.00*
max	*117.41*	*19.12*	*60.15*	*5.77*	*1.33*	*9.34*
Total mean	***64.31***	***4.57***	***16.76***	***1.24***	***0.08***	***0.66***

Table 2: Frequency of reference field and reference array operations by benchmark and benchmark suite.

Barrier	*Prior [5]*		Current Overheads						
	P4	*AMD*	P4⋆	P4	Atom	i7			
	Δt	Δt	Δt	Δt	Δt	Δt	Δi	Δi_{miss}	Δt/Δi
Card	*0.8*	*1.0*	1.8 ±0.4	2.2 ±0.4	1.8 ±0.3	0.9 ±0.8	1.3 ±0.1	6.9 ±1.5	0.70
Object	*1.2*	*1.8*	0.8 ±0.4	1.8 ±0.4	1.3 ±0.3	1.6 ±0.7	2.0 ±0.1	5.7 ±1.2	0.81
Boundary	*1.3*	*2.2*	1.5 ±0.7	2.2 ±0.6	2.5 ±0.3	1.7 ±0.8	2.7 ±0.1	10.2 ±1.3	0.65
Zone	*4.8*	*5.1*	7.1 ±0.5	9.0 ±0.5	9.3 ±0.5	9.6 ±0.8	8.5 ±0.1	28.8 ±1.4	1.12
Read	*5.0*	*8.1*	4.1 ±0.5	4.6 ±0.5	5.5 ±0.3	5.4 ±0.8	9.3 ±0.2	11.9 ±2.4	0.64
Cond Read	*15.9*	*21.2*	9.2 ±0.6	9.4 ±0.6	9.1 ±0.5	10.1 ±0.8	20.9 ±0.1	37.2 ±1.7	0.48
Bit Steal			7.0 ±0.4	7.1 ±0.5	7.8 ±0.4	8.3 ±0.9	12.8 ±0.4	21.2 ±2.3	0.65
Hybrid	*1.3*	*1.8*	0.9 ±0.4	2.1 ±0.4	1.8 ±0.3	1.7 ±1.1	2.2 ±0.1	9.1 ±1.4	0.78

⋆ Running on current system with a subset of benchmarks similar to that used by Blackburn and Hosking [5].

Table 3: Summary of barrier overheads on various platforms, expressed in percentages in terms of time (Δt), instructions (Δi), and i-cache misses (Δi_{miss}). For comparison, measurements from Blackburn and Hosking [5] are reproduced in italics in the second and third columns. For each average, the corresponding 95% confidence intervals are also averaged and printed in small grey font.

Benchmark Suite	Object								Boundary								Zone							
	Take Rate		Original Slow			NOP Slow			Take Rate		Original Slow			NOP Slow			Take Rate		Original Slow			NOP Slow		
	/µs	%	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}	/µs	%	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}	/µs	%	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}
SPECjvm mean	0.019	0.1	1.0	1.4	4.6	0.8	1.0	11.6	0.0	0.4	0.7	1.7	7.5	1.3	1.4	12.6	7.3	51.7	7.6	7.5	38.6	3.1	3.6	18.5
geomean			0.9	1.4	4.1	0.8	1.0	9.0			0.7	1.7	6.9	1.3	1.4	12.0			7.5	7.3	36.5	3.1	3.5	16.8
DaCapo mean	0.010	0.1	1.8	2.2	6.7	2.0	1.7	7.3	0.4	2.8	2.3	3.1	12.6	3.0	2.5	12.7	6.0	31.4	11.4	9.8	28.0	5.3	4.5	18.6
geomean			1.8	2.2	6.4	2.0	1.7	6.8			2.3	3.1	12.1	2.9	2.4	12.5			10.7	9.3	26.0	5.2	4.4	17.3
Total mean	0.015	0.1	1.6	2.0	6.1	1.6	1.5	8.6	0.3	1.9	1.8	2.7	10.7	2.5	2.2	12.7	6.5	39.0	10.1	8.9	30.9	4.5	4.2	18.4
geomean			1.6	2.0	5.7	1.6	1.5	7.4			1.7	2.7	10.2	2.5	2.2	12.4			9.6	8.5	28.8	4.4	4.2	17.0

Table 4: Summary of take-rate and cost of slow path for conditional barriers on the i7.

prior work), and c) our JVM is newer and faster. Furthermore, on the i7 we present performance counter data for instructions retired (Δi), and i-cache misses (Δi_{miss}), and confidence intervals for our average of 20 runs. Each of these differences reflects the passage of eight years of improvements in software, hardware and evaluation methodology since the previous study and is a source of motivation for our study.

Tables 6 and 7, which appear at the end of the paper, provide substantially more detailed data, including per-benchmark results, all measured on the i7.

5.1.1 Primitive Barriers

Each primitive barrier is described in Section 3.1. The baseline for our comparison is a system with no barrier. The first six rows of Table 3 summarize performance results for each of the primitive barriers (detailed results are in Table 6). For each barrier, the tables report the percentage increase in execution time (Δt), retired instructions (Δi), and instruction cache (i-cache) misses (Δi_{miss}). Table 3 also presents $\Delta t/\Delta i$, which indicates each barrier's ability to be absorbed by instruction-level parallelism (ILP).

Card Card marking shows an average performance hit of just $0.9\% \pm 0.8\%$ on the i7 processor. Retired instructions increase by just 1.3% and i-cache misses are only 6.9% higher. The performance overhead is consistent with Blackburn and Hosking [5], and is explained by the increase in retired instructions. The increase in i-cache misses is higher, but not enough to result in a significant performance overhead. The performance overhead for card marking is significantly higher on the P4 ($2.2\% \pm 0.4\%$) and Atom ($1.8\% \pm 0.3\%$), which both have much lower memory bandwidth than the i7 and much smaller caches. This doubling in overhead between the i7 and Atom is the strongest architectural sensitivity we see for any of the barriers.

Unlike card marking, the object, boundary, and zone barriers are all conditional. Thus, their performance depends heavily on the rate at which their slow path is taken. Table 4 summarizes the take-rates of these write barriers and the impact of taking the slow path. Detailed results are presented in Table 7. For each of the three barriers, we present the rate at which the barrier is taken, both in terms of execution frequency and in percentage rate of slow paths taken per execution of the barrier. We also present overhead statistics as in Table 3 for the barrier with the regular slow path (Original Slow) and with a call to an empty function (NOP Slow).

Object The object barrier has an average performance overhead of $1.6\% \pm 0.7\%$ on the i7. Although a little higher than the card marking barrier, the 0.7% difference between the barriers is smaller than the confidence intervals on either barrier. On the i7, the number of retired instructions increased by 2.0% and the relative change in CPI is very similar to card marking. Perhaps unsurprisingly, we found that the i-cache locality is strongly affected by inlining of the slow path. When the slow path is forced inline the performance overhead grows to 2.6% and the i-cache misses increase by 20%, compared to 5.7% for the out of line case.

Boundary The boundary barrier has a performance overhead of $1.7\% \pm 0.8\%$ on the i7 — not statistically different from the object or card marking barriers. The take-rate for boundary is an order of magnitude higher than for object, but in absolute terms is low, at around 2%.

Each of these three primitive write barriers records mutated reference fields, but they represent different points in a mutator/collector tradeoff space. Card marking has the highest collection-time overhead, generates the most write traffic and, as we will show in Section 5.3, has some problematic performance pathologies. Yet its performance is not significantly better than either of the other barriers. In the case of card marking, the collector must scan each marked card. The object barrier is similar, but a little more targeted, because it requires each remembered object to be scanned at collection time. By contrast, the boundary barrier is precise — it remembers the address of each modified field, so it requires no scanning at collection time.

Zone The zone barrier is more general than the others because it records mutations crossing multiple boundaries (in both directions). However, this generality comes at a considerable cost, with a $9.6\% \pm 0.8\%$ performance overhead on the i7. This overhead is explained by its slow path take-rate of 39% — more than two orders of magnitude higher than the object barrier! As shown in Figure 2, we explicitly check whether the target address is null before checking whether the boundary is crossed (in a system where young objects are in high memory, a null looks like a mature object). This check is necessary because we found that 42% of taken slow paths were due to nulling of object fields! A standard generational barrier will only remember the mature-to-nursery pointers. Table 4 shows that when we replaced the slow path with a call to an empty function the average overhead reduced to 4.4%. This reduction is due both to the removal of parameter marshaling overhead and the cost of actually executing the code that stores the remembered pointer field. This result differs considerably from the overhead of just 5% reported by Blackburn and Hosking [5].

The overheads of these three conditional write barriers are quite different for different benchmark suites. The average overheads of DaCapo are 1.8%, 2.3%, 10.7% for object, boundary, and zone barriers, respectively. These are much higher than SPECjvm98, at 0.9%, 0.7%, and 7.5%, respectively. The explanation lies in the high rate of reference writes in DaCapo, revealed in Table 2 . The DaCapo reference write rate is about 2× higher than SPECjvm98. One DaCapo benchmark, bloat, has 7× more frequent reference writes than the average for SPECjvm98.

We now examine the two primitive read barriers.

Read The unconditional read barrier has an overhead of 5.4% on the i7 despite a 8.5% increase in retired instructions. The fraction $\Delta t/\Delta i$ is just 0.64, indicating that much of the instruction overhead is absorbed by instruction level parallelism (ILP). Interestingly, on the ILP-limited Atom the performance overhead is only 5.5%.

Conditional Read The conditional read barrier has a 10.1% overhead with a 20% increase in retired instructions. The fraction $\Delta t/\Delta i$ is 0.48, which is the lowest of all the barriers we measure. We see a 9.2% overhead on the P4, which is significantly lower than the 15.9% and 21.2% previously reported by Blackburn and Hosking [5] for the P4 and AMD, respectively. We attribute this improvement to our tuning of the barrier to use a compare to zero rather than immediate mode compare to 1 (cf. Section 3.1).

Summarizing our analysis of primitive barriers, we find that among the write barriers, card marking has the lowest overhead on i7, but only by 0.7%, which is not statistically significant. Furthermore, card marking does not perform well on the Atom or P4. The difference between them is very likely due to the substantially better memory subsystem of the i7. The object barrier is attractive because it outperforms card marking on the Atom and P4 and unlike the previous findings, it dominates boundary and hybrid.

5.1.2 Compound Barriers

Section 3.2 describes each of the compound barriers. The last two rows of Table 3 (and the right-most two column groups of Table 6: Bit Steal and Hybrid) give performance results for each of the compound barriers, relative to the base case where no barriers are used.

Bit Steal Recall from Section 3.2 that the bit steal barrier combines the (unconditional) read barrier with a write barrier that remembers any unlogged reference fields (determined by the 'stolen' low order bit of the reference), and marks the reference as logged by *atomically* updating it. Retired instructions increase by 12.8% while time increases by only 8.3%. I-cache miss rates also increase by 21%, in line with the overheads for the primitive conditional write and unconditional read barriers.

Hybrid The hybrid barrier shows an average performance overhead of 1.7% on the i7 processor. Retired instructions increase by 2.2% and i-cache misses increase by 9.1%. The performance overhead remains fairly consistent with Blackburn and Hosking [5], and is explained by the increase in retired instructions. However, unlike the prior findings, hybrid does not dominate object and boundary. The detailed results in Table 6 show that the worst case performance of hybrid (**max**) is worse than that for object and boundary. It is therefore hard to argue for hybrid over object or boundary on the basis of our analysis using modern benchmarks and modern machines.

5.2 Microarchitectural Sensitivity

We evaluated three different microarchitecture in this study: an aggressively out-of-order machine (P4), an in-order machine (Atom), and a modern out-of-order machine (i7). As shown in Table 1, compared with the i7, both the P4 and Atom have outdated memory subsystems. Not only do they have much smaller caches, but they also suffer low memory bandwidth. Nevertheless, it is surprising that the barrier performance is relative stable among the three diverse microarchitectures. However, there are a number of trends revealed in Table 3.

Among the barriers, card marking is the most memory intensive because it unconditionally executes a store (it does not have a slow path) which may cause a cache miss. Although the object barrier's fast path contains a load to check the source header, the header byte is likely already to be in the cache. Thus, the performance of card marking will depend more heavily on the memory subsystem. As shown in Table 3, the overhead of card marking on the more powerful i7 is much less than for the other two architectures.

Barrier	Overall	Pathology
Card Bytemap	0.88 ±0.78	378.0 ±51.6
Card Wordmap	0.91 ±0.75	372.4 ±48.4
Card Wordmap NT	3.58 ±0.77	40.1 ±24.7
Card Conditional Set	2.44 ±0.83	26.7 ±5.9
Object	1.57 ±0.74	1.0 ±5.1

Table 5: Card marking overheads on the i7 in average and pathological contexts, expressed as percentages. We show four variations of card marking barriers, plus an object barrier. The barriers have successively lower overhead in the pathological setting.

In contrast, read barriers have different behavior. On the slow in-order Atom, read barrier overheads are always marginally better than that of the state-of-the-art i7. This is also due to the differences in memory subsystem. Unlike the write barriers, our read barriers do not have any memory accesses, so their cost should be similar on all three machines. But, the Atom's cache is 16 times smaller than the i7's, so the impact of the extra read barrier instructions is buried in the higher cache miss rates for the Atom. The P4 has a similar memory subsystem to the Atom (for similar miss rates), but it is much more aggressively out-of-order so it can more readily hide the read barrier overheads. The i7 can similarly hide the read barrier overhead, but its lower cache miss rate offers less opportunity to hide read barriers behind cache misses. Indeed, memory-bound operations like the write barriers benefit more from the i7's larger caches.

The Atom is noticeably worse on the boundary barrier than the other architectures. We speculate that the cause of this slowdown may be that the boundary barrier uses two `CMP REG IMM` instructions where the immediate value is 32 bits long. The Atom is known to perform poorly with instructions greater than 4 bytes because of its low instruction fetch-rate [15].

5.3 A Case Study in Barrier Pathology

We now explore and evaluate a known barrier pathology along with several potential solutions to the pathology. Table 5 shows the performance of four different card marking implementations and the object barrier. We show the overall performance when run against our suite of 20 benchmarks and the performance when run on a microbenchmark designed to highlight the pathology.

Card marking is often advocated as a very low-cost unconditional write barrier (indeed it is cheap according to our results and those of Blackburn and Hosking [5]), but it does suffer from at least two known pathologies. The first is that the work of scanning the cards is a function of heap size and thus small nurseries perform poorly because of the fixed cost of scanning the cards dominates. The second arises when multiple threads perform frequent concurrent updates to the same or adjacent cards, resulting in cache contention on the cache line holding the metadata for those cards [14]. We focus now on this second pathology.

One proposed solution is to use conditional card marking, marking a card only if it is not already marked. This eliminates unnecessary writes (and any associated contention) at the expense of an additional read. Table 5 shows that whereas the average overhead of card marking is $0.88\% \pm 0.78\%$, conditional card marking is more than twice as expensive at $2.4\% \pm 0.83\%$. The simple object barrier

discussed earlier is cheaper at $1.57\% \pm 0.74\%$ overhead. Another suggested solution is to use a non-temporal store instruction when marking a card, allowing the write to occur without affecting the cache. Unfortunately, Table 5 reveals that this has unacceptably high overhead of $3.58\% \pm 0.77\%$. Thus, using non-temporal instructions to avoid cache contention in card marking is less preferable than conditional card marking, but using a cheaper alternative such as the object barrier also avoids the pathology.

To illustrate the gains to be had, we use a synthetic microbenchmark designed to trigger card mark contention. This benchmark creates multiple worker threads, each of which has a thread-local buffer that is continually written to in a tight loop. If the buffer objects themselves are located on different cache lines — but lie within the same or adjacent cards — then the card marking barrier can introduce contention. This microbenchmark is drawn from the real world example of a work-stealing scheduler, where each worker thread is managing task objects in a thread-local buffer.[1] Table 5 shows that this pathology can cause dramatic slowdowns, with the byte-map card barrier suffering 378% overhead, meaning that the system runs nearly $5\times$ slower. While all of the proposals to reduce contention are reasonably effective when compared to the original byte-map card marking, in practice the object barrier handles this case better than any of the card approaches.

6. Conclusion

This paper presents a detailed and up-to-date quantitative study of barrier costs. Because barriers are a key building block for memory management algorithms and enable opportunities for algorithmic creativity, properly understanding their cost is essential. Reprising the previous study of Blackburn and Hosking [5], we deepen and renew the results using modern workloads, modern hardware, and more rigorous methodology. The use of modern workloads is critical because the prior study was done with the simplest of Java workloads. The use of modern hardware is important because there has been substantial upheaval in computer architecture in the eight years since the previous work. Applying more rigorous methodology is essential because it clarifies the significance of some important results. By using performance counters we were able to shed more light on why barriers perform differently. We have also examined an important write barrier pathology and evaluated four alternative solutions.

Significant changes in computer architecture and increasing demand for managed languages are likely to put renewed pressure on researchers to develop interesting memory management solutions for diverse settings. Our work fortifies the algorithmic toolkit available to researchers embarking on this route by quantifying barrier costs in detail.

References

[1] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), Feb. 2000. doi: 10.1147/sj.391.0211.

[2] A. W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, Feb. 1989. doi: 10.1002/spe.4380190206.

[3] A. Azagury, E. K. Kolodner, E. Petrank, and Z. Yehudai. Combining card marking with remembered sets: How to save scanning time. In *ACM International Symposium on Memory Management*, pages 10–19, Vancouver, Canada, Oct. 1998. doi: 10.1145/286860.286862.

[4] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the Thirtieth Annual ACM Symposium on the Principles of Programming Languages*, pages 285–294, New Orleans, LA, Jan. 2003. doi: 10.1145/604131.604155.

[5] S. M. Blackburn and A. Hosking. Barriers: Friend or foe? In *ACM International Symposium on Memory Management*, pages 143–151, Vancouver, Canada, Oct. 2004. doi: 10.1145/1029873.1029891.

[6] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator locality. In *ACM Conference on Programming Language Design and Implementation*, pages 22–32, Tuscon, AZ, June 2008. doi: 10.1145/1375581.1375586.

[7] S. M. Blackburn and K. S. McKinley. In or out? Putting write barriers in their place. In *ACM International Symposium on Memory Management*, pages 175–184, Berlin, Germany, June 2002. doi: 10.1145/512429.512452.

[8] S. M. Blackburn, M. Hirzel, R. Garner, and D. Stefanović. pjbb2005: The pseudoJBB benchmark. URL http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005.

[9] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *ACM Conference on Programming Language Design and Implementation*, pages 153–164, Berlin, Germany, June 2002. doi: 10.1145/512529.512548.

[10] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Scotland, UK, May 2004. doi: 10.1109/ICSE.2004.1317436.

[11] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, Oct. 2006. doi: 10.1145/1167515.1167488.

[12] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *ACM Conference on Lisp and Functional Programming*, pages 256–262, Austin, Texas, Aug. 1984. doi: 10.1145/800055.802042.

[13] P. J. Caudill and A. Wirfs-Brock. A third-generation Smalltalk-80 implementation. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 119–130, Portland, OR, Nov. 1986. doi: 10.1145/28697.28709.

[14] D. Dice. False sharing induced by card table marking, Feb. 2011. URL https://blogs.oracle.com/dave/entry/false_sharing_induced_by_card.

[15] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs. An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering, June 2011.

[16] L. Hellyer, R. E. Jones, and A. L. Hosking. The locality of concurrent write barriers. In *ACM International Symposium on Memory Management*, pages 83–92, Toronto, Canada, June 2010. doi: 10.1145/1806651.1806666.

[17] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 359–373, Anaheim, California, Nov. 2003. doi: 10.1145/949305.949337.

[18] A. L. Hosking and R. L. Hudson. Remembered sets can also play cards. In J. E. B. Moss, P. R. Wilson, and B. Zorn, editors, *OOPSLA Workshop on Garbage Collection in Object-Oriented*

[1] Indeed, Doug Lea has reported this pathology in his work on concurrent data structures for the `java.util.concurrent` library.

Systems, Oct. 1993. URL ftp://ftp.cs.utexas.edu/pub/garbage/GC93/hosking.ps.

[19] A. L. Hosking and J. E. B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 106–119, Asheville, North Carolina, Dec. 1993. doi: 10.1145/168619.168628.

[20] A. L. Hosking, J. E. B. Moss, and D. Stefanović. A comparative performance evaluation of write barrier implementations. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, Oct. 1992. doi: 10.1145/141936.141946.

[21] P. Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, Feb. 1988. Bachelor's thesis.

[22] SPEC. *SPECjbb2005 (Java Server Benchmark), Release 1.07*. Standard Performance Evaluation Corporation, 2006. URL http://www.spec.org/jbb2005.

[23] SPEC. *SPECjvm98, Release 1.03*. Standard Performance Evaluation Corporation, Mar. 1999. URL http://www.spec.org/jvm98.

[24] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Apr. 1984. doi: 10.1145/800020.808261.

[25] P. R. Wilson and T. 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, May 1989. doi: 10.1145/66068.66077.

[26] B. Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, Nov. 1990.

Benchmark	Card			Object			Boundary			Zone			Read			Cond Read			Bit Steal			Hybrid		
	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}
compress	0.0 ±0.0	0.0 ±0.0	-0.8 ±0.5	-0.1 ±0.0	0.0 ±0.0	-1.5 ±0.6	0.0 ±0.0	0.0 ±0.0	2.4 ±0.6	-0.1 ±0.1	0.0 ±0.0	7.3 ±0.7	-9.3 ±0.0	13.7 ±0.0	4.4 ±0.7	-4.6 ±0.0	37.3 ±0.0	14.4 ±0.8	-9.2 ±0.0	16.0 ±4.5	9.5 ±1.2	0.0 ±0.0	0.0 ±0.0	-0.8 ±0.5
jess	2.8 ±0.2	1.2 ±0.1	15.5 ±0.6	1.3 ±0.2	1.5 ±0.1	16.5 ±1.1	1.0 ±0.2	2.0 ±0.1	21.5 ±0.7	12.4 ±0.3	10.1 ±0.2	90.6 ±2.2	20.9 ±0.3	17.0 ±1.5	17.8 ±1.4	23.7 ±1.0	32.9 ±0.1	56.8 ±2.4	31.9 ±0.2	28.6 ±0.1	41.6 ±2.3	2.3 ±2.2	1.8 ±0.1	25.8 ±1.9
db	1.3 ±0.4	1.5 ±0.0	3.1 ±0.6	1.4 ±0.3	2.3 ±0.0	4.1 ±0.6	1.3 ±0.4	3.3 ±0.0	6.7 ±0.7	15.6 ±0.4	19.7 ±0.0	52.2 ±1.1	9.9 ±0.4	25.0 ±0.0	12.1 ±0.7	13.2 ±0.4	43.6 ±0.0	14.0 ±0.7	11.0 ±0.3	28.8 ±0.0	10.4 ±0.7	1.5 ±0.3	3.3 ±0.0	2.5 ±0.6
javac	2.1 ±0.1	1.8 ±0.0	5.7 ±0.2	3.6 ±0.1	3.1 ±0.0	6.5 ±0.2	3.0 ±0.1	3.0 ±0.0	10.4 ±0.2	8.9 ±0.1	6.8 ±0.0	19.5 ±0.3	6.4 ±0.1	6.9 ±0.0	13.2 ±0.3	10.3 ±0.1	14.4 ±0.0	40.1 ±0.2	9.7 ±0.1	10.5 ±0.0	18.8 ±0.2	3.1 ±0.1	2.9 ±0.0	5.8 ±0.2
mpegaudio	0.4 ±0.1	0.2 ±0.0	-0.4 ±1.6	0.2 ±0.0	0.3 ±0.0	-2.0 ±1.1	-0.2 ±0.0	0.3 ±0.0	-0.1 ±1.3	3.9 ±0.1	2.5 ±0.0	31.1 ±1.6	3.2 ±0.0	6.7 ±0.0	17.7 ±1.5	8.2 ±0.0	19.2 ±0.0	41.1 ±1.6	3.5 ±0.8	7.1 ±0.0	13.1 ±1.3	0.3 ±0.0	0.3 ±0.0	-2.0 ±1.2
mtrt	0.0 ±0.2	0.4 ±0.0	21.2 ±7.8	-0.3 ±0.1	0.5 ±0.0	19.4 ±2.1	0.0 ±0.2	0.6 ±0.0	23.6 ±2.3	1.6 ±0.2	1.3 ±0.0	37.2 ±1.7	8.5 ±0.2	9.6 ±0.0	73.5 ±27.1	13.6 ±0.2	27.3 ±0.0	141 ±2.0	9.2 ±0.2	10.6 ±0.0	81.0 ±16.9	0.4 ±0.1	0.6 ±0.0	24.6 ±2.1
jack	0.5 ±0.4	1.6 ±0.0	6.7 ±1.0	0.5 ±0.7	2.4 ±0.0	-11.0 ±0.5	0.1 ±0.4	2.8 ±0.0	-11.8 ±0.6	11.1 ±0.5	12.4 ±0.0	32.0 ±0.8	1.5 ±0.5	4.3 ±0.0	-12.5 ±0.5	4.2 ±0.6	10.9 ±0.0	7.8 ±1.0	5.4 ±2.3	7.7 ±0.0	3.8 ±0.6	1.5 ±2.5	2.5 ±0.0	0.1 ±1.3
SPECjvm mean	**1.0** ±0.2	**1.0** ±0.0	**7.3** ±1.8	**1.0** ±0.2	**1.4** ±0.0	**4.6** ±0.9	**0.7** ±0.2	**1.7** ±0.0	**7.5** ±0.9	**7.6** ±0.2	**7.5** ±0.0	**38.6** ±1.2	**5.9** ±0.2	**11.9** ±0.2	**18.0** ±4.6	**9.8** ±0.3	**26.5** ±0.0	**45.0** ±1.2	**8.8** ±0.6	**15.6** ±0.7	**25.5** ±3.3	**1.3** ±0.8	**1.6** ±0.0	**8.0** ±1.1
geomean	**1.0**	**1.0**	**7.0**	**0.9**	**1.4**	**4.1**	**0.7**	**1.7**	**6.9**	**7.5**	**7.3**	**36.5**	**5.5**	**11.7**	**15.8**	**9.5**	**26.0**	**39.9**	**8.2**	**15.3**	**23.2**	**1.3**	**1.6**	**7.5**
antlr	-3.3 ±4.4	0.7 ±0.1	4.9 ±0.9	-2.4 ±4.4	1.2 ±0.1	2.8 ±0.6	-5.0 ±3.4	1.2 ±0.1	5.1 ±0.5	-4.3 ±3.4	1.6 ±0.1	10.6 ±0.6	-2.5 ±3.4	5.6 ±0.1	9.1 ±0.4	1.0 ±3.6	10.9 ±0.1	25.0 ±0.5	-0.8 ±3.5	7.1 ±0.1	13.0 ±0.4	-4.9 ±3.9	1.2 ±0.1	0.1 ±0.6
avrora	0.4 ±0.9	0.7 ±0.1	-0.8 ±0.7	2.6 ±0.7	0.8 ±0.1	7.0 ±0.6	0.0 ±0.8	1.6 ±0.1	-0.3 ±1.0	5.0 ±1.0	3.1 ±0.1	13.5 ±1.0	3.5 ±0.7	8.2 ±0.1	10.7 ±0.8	6.1 ±0.9	18.7 ±0.1	32.1 ±1.5	3.2 ±0.9	9.5 ±0.1	9.5 ±1.2	-1.0 ±1.1	0.8 ±0.1	-5.4 ±0.6
bloat	1.0 ±0.4	4.2 ±0.1	3.8 ±0.7	6.8 ±0.3	8.2 ±0.2	6.1 ±0.5	8.3 ±0.4	11.8 ±0.1	7.4 ±0.6	52.7 ±0.4	43.8 ±0.3	37.2 ±0.5	5.8 ±0.4	11.6 ±0.0	7.9 ±0.4	21.1 ±0.5	35.0 ±0.1	32.7 ±0.8	18.7 ±0.4	25.3 ±0.0	20.3 ±0.5	8.6 ±1.6	9.1 ±0.1	9.2 ±0.8
eclipse	0.3 ±3.6	0.6 ±1.9	1.4 ±4.4	-0.7 ±3.4	1.5 ±1.8	-2.6 ±3.6	-0.5 ±3.3	1.3 ±1.9	6.0 ±4.3	7.0 ±3.5	8.1 ±1.6	3.2 ±3.7	3.6 ±3.9	6.1 ±2.1	2.0 ±3.8	4.9 ±4.2	12.1 ±2.2	22.3 ±5.3	3.7 ±3.5	7.7 ±2.2	9.1 ±4.3	2.0 ±3.7	1.2 ±1.8	5.3 ±4.9
fop	1.8 ±0.8	0.6 ±0.0	8.2 ±0.4	5.1 ±0.8	1.3 ±0.0	6.1 ±0.4	2.0 ±0.8	1.3 ±0.0	8.1 ±0.3	6.9 ±0.9	3.0 ±0.0	11.8 ±0.4	2.8 ±0.8	5.3 ±0.0	7.8 ±0.3	7.8 ±1.0	18.0 ±0.0	24.5 ±0.4	5.9 ±1.0	7.1 ±0.0	12.9 ±0.4	1.7 ±0.7	1.4 ±0.0	0.2 ±0.3
hsqldb	2.8 ±0.3	1.8 ±0.0	8.0 ±1.3	3.6 ±0.2	2.5 ±0.0	6.2 ±0.9	5.4 ±0.2	4.4 ±0.0	7.3 ±0.8	9.8 ±0.2	7.8 ±0.0	15.8 ±0.6	7.8 ±0.3	7.5 ±0.0	14.4 ±1.2	12.6 ±0.3	16.9 ±0.0	38.8 ±5.8	12.0 ±0.2	11.2 ±0.0	17.6 ±0.9	5.4 ±0.2	3.4 ±0.0	7.8 ±0.9
jython	1.0 ±0.4	1.6 ±0.0	16.4 ±4.8	2.2 ±0.4	2.4 ±0.1	22.3 ±4.3	2.7 ±0.3	3.5 ±0.0	40.4 ±6.9	16.0 ±0.7	13.0 ±0.3	105 ±8.6	13.8 ±0.5	17.7 ±0.3	23.5 ±4.9	18.8 ±0.3	28.8 ±0.1	66.2 ±6.4	17.1 ±1.9	21.4 ±0.1	66.7 ±10.7	2.9 ±0.4	2.7 ±0.0	60.4 ±7.1
luindex	0.7 ±0.2	1.0 ±0.0	12.8 ±0.5	0.9 ±0.2	0.5 ±0.0	9.3 ±1.1	0.6 ±0.2	0.9 ±0.0	7.1 ±1.0	3.9 ±0.2	2.5 ±0.0	26.5 ±0.6	5.2 ±0.2	7.7 ±0.0	18.0 ±0.4	8.0 ±0.3	14.2 ±0.0	49.0 ±0.6	7.1 ±0.2	8.9 ±0.0	20.7 ±0.5	0.9 ±0.2	0.5 ±0.0	9.8 ±0.5
lusearch	1.7 ±0.3	2.1 ±0.0	7.0 ±0.7	1.4 ±0.3	2.6 ±0.0	7.7 ±2.3	5.1 ±2.1	3.6 ±0.0	17.3 ±0.7	12.8 ±0.4	9.1 ±0.0	24.5 ±1.0	3.7 ±0.3	7.7 ±0.1	8.3 ±0.3	9.0 ±0.3	15.6 ±0.0	29.6 ±0.7	6.9 ±0.4	10.8 ±0.0	18.6 ±0.5	3.1 ±1.5	2.5 ±0.1	10.6 ±0.8
pmd	1.4 ±0.1	2.3 ±0.1	6.5 ±0.7	3.3 ±0.2	3.0 ±0.1	17.5 ±0.7	4.9 ±0.2	4.1 ±0.1	24.9 ±1.1	13.4 ±2.3	13.4 ±0.1	28.3 ±1.4	6.7 ±0.2	6.0 ±0.1	10.5 ±1.4	9.5 ±0.2	16.9 ±0.1	36.1 ±1.5	10.1 ±0.2	11.0 ±0.1	27.8 ±1.1	4.4 ±0.2	3.4 ±0.1	24.0 ±0.8
sunflow	0.1 ±0.0	-0.7 ±0.0	17.4 ±0.7	1.5 ±0.0	0.9 ±0.0	0.1 ±0.7	2.5 ±0.0	1.2 ±0.0	22.3 ±0.5	11.5 ±0.2	10.7 ±0.1	39.8 ±1.2	4.3 ±0.0	7.2 ±0.0	0.8 ±0.4	14.8 ±0.1	17.9 ±0.0	62.6 ±0.7	7.2 ±0.0	8.1 ±0.0	30.6 ±0.4	1.5 ±0.0	0.9 ±0.0	9.1 ±0.6
xalan	1.3 ±0.3	1.8 ±0.0	1.6 ±0.1	-2.7 ±0.3	1.5 ±0.0	-2.4 ±0.1	1.9 ±0.5	2.6 ±0.0	6.1 ±0.1	1.9 ±0.0	1.3 ±0.0	19.6 ±1.0	13.6 ±1.3	5.7 ±0.0	12.9 ±0.1	16.5 ±0.3	19.5 ±0.0	26.6 ±0.2	12.7 ±0.3	11.7 ±0.0	11.9 ±0.1	-1.1 ±0.2	2.5 ±0.0	1.7 ±0.1
DaCapo mean	**0.8** ±1.0	**1.4** ±0.2	**7.3** ±1.3	**1.8** ±0.9	**2.2** ±0.2	**6.7** ±1.3	**2.3** ±1.0	**3.1** ±0.2	**12.6** ±1.5	**11.4** ±1.1	**9.8** ±0.2	**28.0** ±1.7	**5.7** ±1.0	**8.0** ±0.2	**10.5** ±1.2	**10.8** ±1.0	**18.7** ±0.2	**37.1** ±2.0	**8.7** ±1.0	**11.6** ±0.2	**21.6** ±1.8	**2.0** ±1.1	**2.5** ±0.2	**11.1** ±1.5
geomean	**0.8**	**1.4**	**7.1**	**1.8**	**2.2**	**6.4**	**2.3**	**3.1**	**12.1**	**10.7**	**9.3**	**26.0**	**5.6**	**8.0**	**10.3**	**10.7**	**18.5**	**36.4**	**8.5**	**11.5**	**20.7**	**1.9**	**2.5**	**10.0**
pjbb2005	1.5 ±2.6	1.9 ±0.0	2.8 ±1.1	3.8 ±2.2	3.0 ±0.0	9.1 ±1.3	2.4 ±3.0	4.1 ±0.0	9.5 ±1.6	11.1 ±1.5	7.5 ±0.0	12.0 ±0.1	3.0 ±2.6	8.9 ±0.0	4.9 ±1.1	6.5 ±2.2	15.3 ±0.1	28.0 ±1.6	6.0 ±2.6	10.6 ±0.0	12.4 ±1.3	2.3 ±2.9	3.2 ±0.1	9.5 ±1.5
min	-3.3	-0.7	-0.8	-2.7	0.0	-11.0	-5.0	0.0	-11.8	-4.3	0.0	3.2	-9.3	4.3	-12.5	-4.6	10.9	7.8	-9.2	7.1	3.8	-4.9	0.0	-5.4
max	2.8	4.2	21.2	6.8	8.2	22.3	8.3	11.8	40.4	52.7	43.8	105	20.9	25.0	73.5	23.7	43.6	141	31.9	28.8	81.0	8.6	9.1	60.4
Total mean	**0.9** ±0.8	**1.3** ±0.1	**7.0** ±1.5	**1.6** ±0.7	**2.0** ±0.1	**6.1** ±1.2	**1.8** ±0.8	**2.7** ±0.1	**10.7** ±1.3	**10.1** ±0.8	**8.9** ±0.1	**30.9** ±1.4	**5.6** ±0.8	**9.4** ±0.2	**12.8** ±2.4	**10.3** ±0.8	**21.3** ±0.1	**39.4** ±1.7	**8.6** ±0.9	**13.0** ±0.4	**22.5** ±2.3	**1.7** ±1.1	**2.2** ±0.1	**9.9** ±1.4
geomean	**0.9**	**1.3**	**6.9**	**1.6**	**2.0**	**5.7**	**1.7**	**2.7**	**10.2**	**9.6**	**8.5**	**28.8**	**5.4**	**9.3**	**11.9**	**10.1**	**20.9**	**37.2**	**8.3**	**12.8**	**21.2**	**1.7**	**2.2**	**9.1**

Table 6: Overheads (%) in time, instructions, and i-cache misses for the primitive barriers on the i7. The the first six column groups summarize the performance for each barrier, showing percentage increase in execution time (Δt), retired instructions (Δi), and instruction cache misses (Δi_{miss}) compared to the base case where no barriers are used. The right-most two column groups give results for the compound barriers. The figures in grey beneath the corresponding arithmetic mean report 95% confidence intervals.

Benchmark	Object								Boundary								Zone							
	Take Rate		Original Slow			NOP Slow			Take Rate		Original Slow			NOP Slow			Take Rate		Original Slow			NOP Slow		
	/µs	%	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}	/µs	%	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}	/µs	%	Δt	Δi	Δi_{miss}	Δt	Δi	Δi_{miss}
compress	0.000	0.2	-0.1	0.0	-1.5	0.0	0.0	-1.2	0.0	1.3	0.0	0.0	2.4	0.0	0.0	1.8	0.0	39.8	-0.1	0.0	7.3	-0.2	0.0	1.7
jess	0.001	0.0	1.3	1.5	16.5	0.7	1.0	10.0	0.1	0.2	1.0	2.0	21.5	2.5	1.5	19.9	18.9	52.4	12.4	10.1	90.6	9.2	7.1	60.7
db	0.000	0.0	1.4	2.3	4.1	1.9	1.7	4.9	0.0	0.0	1.3	3.3	6.7	1.2	2.8	5.6	15.0	86.1	15.6	19.7	52.2	2.7	7.2	9.0
javac	0.124	0.8	3.6	3.1	6.5	3.1	2.1	9.4	0.1	0.3	3.0	3.0	10.4	4.4	2.5	15.1	3.7	24.5	8.9	6.8	19.5	5.1	4.0	15.7
mpegaudio	0.000	0.0	0.2	0.3	-2.0	0.1	0.2	-1.0	0.0	0.0	-0.2	0.3	-0.1	-0.2	0.2	1.1	3.6	98.5	3.9	2.5	31.1	0.5	0.8	0.1
mtrt	0.003	0.0	-0.3	0.5	19.4	0.2	0.3	74.6	0.0	0.0	0.0	0.6	23.6	0.2	0.6	38.6	1.1	14.6	1.6	1.3	37.2	0.6	0.7	37.8
jack	0.001	0.0	0.5	2.4	-11.0	-0.3	1.8	-15.4	0.2	0.8	0.1	2.8	-11.8	1.1	2.2	6.1	8.9	46.4	11.1	12.4	32.0	3.9	5.1	4.2
SPECjvm mean	**0.019**	**0.1**	**1.0**	**1.4**	**4.6**	**0.8**	**1.0**	**11.6**	**0.0**	**0.4**	**0.7**	**1.7**	**7.5**	**1.3**	**1.4**	**12.6**	**7.3**	**51.7**	**7.6**	**7.5**	**38.6**	**3.1**	**3.6**	**18.5**
geomean			**0.9**	**1.4**	**4.1**	**0.8**	**1.0**	**9.0**			**0.7**	**1.7**	**6.9**	**1.3**	**1.4**	**12.0**			**7.5**	**7.3**	**36.5**	**3.1**	**3.5**	**16.8**
antlr	0.003	0.1	-2.4	1.2	2.8	-4.0	0.9	2.6	0.0	0.2	-5.0	1.2	5.1	-5.4	1.0	4.0	0.2	3.7	-4.3	1.6	10.6	-3.9	1.4	5.5
avrora	0.000	0.0	2.6	0.8	7.0	-0.2	0.7	-1.6	0.2	8.7	0.0	1.6	-0.3	2.8	1.2	7.4	0.9	33.4	5.0	3.1	13.5	3.1	1.4	14.4
bloat	0.009	0.0	6.8	8.2	6.1	7.9	7.6	2.2	0.0	0.0	8.3	11.8	7.4	10.7	10.2	11.3	26.3	61.1	52.7	43.8	37.2	17.6	18.2	19.5
eclipse	0.004	0.0	-0.7	1.5	-2.6	1.0	1.7	-5.5	0.0	0.4	-0.5	1.3	6.0	-0.8	1.3	3.0	4.2	34.2	7.0	8.1	3.2	5.3	5.0	0.3
fop	0.005	0.2	5.1	1.3	6.1	6.6	0.9	5.4	0.0	0.5	2.0	1.3	8.1	1.4	1.1	5.8	0.8	28.9	6.9	3.0	11.8	7.1	1.8	7.3
hsqldb	0.025	0.1	3.6	2.5	6.2	5.1	1.8	10.5	2.1	8.9	5.4	4.4	7.3	4.5	2.9	14.9	5.2	19.6	9.8	7.8	15.8	6.7	4.9	18.0
jython	0.001	0.0	2.2	2.4	22.3	1.1	1.5	40.7	1.0	3.7	2.7	3.5	40.4	2.0	2.3	27.6	16.5	68.4	16.0	13.0	105.2	6.2	5.7	80.3
luindex	0.000	0.0	0.9	0.5	9.3	1.1	0.2	7.1	0.1	0.6	0.6	0.9	7.1	1.0	0.6	19.5	2.7	18.8	3.9	2.5	26.5	2.1	1.1	9.6
lusearch	0.001	0.0	1.4	2.6	7.7	1.4	1.8	7.0	0.5	3.5	5.1	3.6	17.3	4.8	3.0	16.1	6.1	42.2	12.8	9.1	24.5	5.9	4.3	20.5
pmd	0.058	0.3	3.3	3.0	17.5	2.9	2.5	11.0	0.7	3.3	4.9	4.1	24.9	5.2	3.4	18.9	5.5	27.5	13.4	13.4	28.3	6.2	6.2	23.7
sunflow	0.014	0.1	1.5	0.9	0.1	1.1	-0.2	7.2	0.0	0.2	2.5	1.2	22.3	1.0	0.4	14.6	0.1	1.1	11.5	10.7	39.8	1.9	0.5	12.9
xalan	0.003	0.0	-2.7	1.5	-2.4	0.0	1.4	1.3	0.4	3.9	1.9	2.6	6.1	8.4	2.2	9.9	3.6	38.3	1.9	1.3	19.6	5.7	4.0	10.9
DaCapo mean	**0.010**	**0.1**	**1.8**	**2.2**	**6.7**	**2.0**	**1.7**	**7.3**	**0.4**	**2.8**	**2.3**	**3.1**	**12.6**	**3.0**	**2.5**	**12.7**	**6.0**	**31.4**	**11.4**	**9.8**	**28.0**	**5.3**	**4.5**	**18.6**
geomean			**1.8**	**2.2**	**6.4**	**2.0**	**1.7**	**6.8**			**2.3**	**3.1**	**12.1**	**2.9**	**2.4**	**12.5**			**10.7**	**9.3**	**26.0**	**5.2**	**4.4**	**17.3**
pjbb2005	0.054	0.3	3.8	3.0	9.1	1.1	2.2	3.0	0.4	1.9	2.4	4.1	9.5	3.0	3.4	12.5	7.0	40.6	11.1	7.5	12.0	4.5	5.4	16.4
min	0.000	0.0	-2.7	0.0	-11.0	-4.0	-0.2	-15.4	0.0	0.0	-5.0	0.0	-11.8	-5.4	0.0	1.1	0.0	1.1	-4.3	0.0	3.2	-3.9	0.0	0.1
max	0.124	0.8	6.8	8.2	22.3	7.9	7.6	74.6	2.1	8.9	8.3	11.8	40.4	10.7	10.2	38.6	26.3	98.5	52.7	43.8	105.2	17.6	18.2	80.3
Total mean	**0.015**	**0.1**	**1.6**	**2.0**	**6.1**	**1.6**	**1.5**	**8.6**	**0.3**	**1.9**	**1.8**	**2.7**	**10.7**	**2.5**	**2.2**	**12.7**	**6.5**	**39.0**	**10.1**	**8.9**	**30.9**	**4.5**	**4.2**	**18.4**
geomean			**1.6**	**2.0**	**5.7**	**1.6**	**1.5**	**7.4**			**1.7**	**2.7**	**10.2**	**2.5**	**2.2**	**12.4**			**9.6**	**8.5**	**28.8**	**4.4**	**4.2**	**17.0**

Table 7: Effect of take-rate and slow path cost for conditional barriers on the i7.