The Case for Profile-Directed Selection of Garbage Collectors

Robert Fitzgerald and David Tarditi
Microsoft Research
Redmond, WA 98052, USA
bobfitz,dtarditi@microsoft.com

ABSTRACT

Many garbage-collected systems use a single garbage collection algorithm across all applications. It has long been known that this can produce poor performance on applications for which that collector is not well suited. In some systems, such as those that execute stand-alone compiled executables, an appropriate collector for each application can be selected from a pool of available collectors and tuned by using profile information. In a study of 20 benchmarks and several collectors, compiled with the Marmot optimizing Java-to-native compiler, for every collector there was at least one benchmark that would have been at least 15% faster with a more appropriate collector. The collectors are a copying collector, a generational copying collector, which is combined with each of 4 different write barriers, and the null collector, which allocates but never collects. A detailed analysis of storage management costs shows how they vary by application and collector.

1. INTRODUCTION

Automatic storage management eliminates a significant source of software defects by freeing programmers from the obligation to release dynamically allocated storage explicitly. An enormous number of algorithms and variations have been proposed in the 40-year history of automatic storage management [21]. Basic algorithms include mark-and-sweep collection [24], semispace copying collection [11], and generational collection [22, 25, 32].

Although no one algorithm has proven to be uniformly better than all the others, most systems use only one. This "one size fits all" approach is appropriate for systems that must handle applications whose characteristics are not known ahead of time. These include environments for Lisp [30], Smalltalk [14], and Java [23] that outlive individual applications. It can be difficult or impossible for a running environment to replace its garbage collector, especially if performance-critical parts of the implementation such as allocation sequences and write barrier checks have been inlined into executable code.

The use of a single algorithm, however, can hurt the performance of applications that are not well matched to that algorithm. Each algorithm has characteristic strengths and weaknesses that make it better suited to some applications and more poorly suited to others [2]. For example, some applications run faster with generational garbage collection than with copying garbage collection while others run slower. The latter can happen when the cost of the write barrier checks exceeds the reduction in collection time or when generational collection causes excessive copying [32].

Other systems, such as those that produce stand-alone executables, can pair each application with an appropriate collector. These include the increasingly prevalent optimizing compilers for safe languages such as Java [20, 26, 31]. Such systems might reasonably use profile information from prior executions of an application to select and tune the collector for subsequent executions.

Early Lisp machines had a simple all-or-nothing variation on this theme. Moon [25] reports that users often disabled the garbage collector and rebooted when the virtual address space filled. Rebooting was faster than waiting while the garbage collector thrashed virtual memory. Newer systems can provide a greater variety of interchangeable garbage collector components.

Moss et al. [19] describe a language-independent garbage collector toolkit that demonstrated that garbage collector components could be replaced easily at compile time. The toolkit includes a generational collector that allows the write barrier to be replaced. The collector can also be configured at run-time in a variety of ways, including changing the number of generations, generation sizes, and the promotion policy.

There is a rich tradition of tuning the parameters of a particular garbage collector by using profile information and feedback. For example, Ungar and Jackson [35] describe an adaptive policy for deciding when to move objects into the older generation of a generational copying collector. Barrett and Zorn [3] describe a 2-generation collector with a threatening boundary between the two generations that allows data to be moved back into the younger generation. Cheng et al. [6] identify allocation sites that consistently produce long-lived data and pretenure data from those sites into the older generation of a 2-generation collector.

This paper investigates the value of pairing applications with appropriate collectors. 20 Java benchmarks are combined with each of several garbage collectors in an environment that builds standalone compiled executables. The

collectors are the "null" collector, which allocates but never collects, a copying collector, and a generational copying collector. Furthermore, the generational collector is combined with each of 4 different write barriers. The best collector for each benchmark is selected. This application-specific strategy is compared against a fixed strategy of using each collector across all benchmarks. The costs of each collector are examined in detail to show how application behavior affects performance.

The measurement environment consists of largely idle PCs with fast processors and large memories. For these machines, elapsed program execution time is an adequate measurement of performance and it can be reasonable to trade memory for time. Other environments may have different measures of performance and constraints.

The remainder of the paper is organized as follows. Section 2 describes the benchmarks and the Marmot system used to build them. Section 3 shows that all of the collectors perform well on some of the benchmarks and poorly on others. Section 4 details the application- and collector-specific nature of the storage management costs of several benchmarks. Section 5 compares the costs of zeroing memory incrementally at allocation time and of getting zero-filled virtual memory pages on demand from the operating system. Section 6 compares the average performance of using the same collector for all benchmarks with that of pairing each benchmark with the collector best suited to it. Section 7 summarizes our conclusions.

2. THE EXPERIMENTAL TESTBED

The experimental testbed used in this paper consists of a Java compiler, a collection of automatic storage management components, a set of benchmark programs, and the hardware they ran on.

2.1 The Marmot Java-to-Native-Code Optimizing Compiler

Marmot [12] is a whole-program Java-to-native-machine-code optimizing compiler developed as a research platform. It includes standard scalar optimizations, basic object-oriented optimizations, and advanced optimizations. Standard optimizations include constant and copy propagation, common subexpression elimination, dead-assignment elimination, loop-invariant code motion, induction variable elimination, strength reduction, and inlining of small, statically called functions. Object-oriented optimizations include virtual call rebinding based on class hierarchy analysis, null check elimination, type test elimination, and removal of uninvoked methods. Advanced optimizations include stack allocation of objects [13], elimination of redundant array-bounds checks, and elimination of redundant synchronization operations [27].

Marmot produces compiled Java executables that sometimes rival C++ performance, but are on average 20-30% slower than corresponding C++ binaries compiled with a compiler product such as Microsoft Visual C++. The proportion of time that Marmot-compiled executables spend in storage-management-related operations is reasonably indicative of the costs that might be expected for Java programs compiled with a product-quality compiler.

2.2 Automatic Storage Management Components

The Marmot system includes fast allocation, several automatic storage management algorithms, and several write barrier implementations for generational collection.

2.2.1 Allocation

Memory is allocated from a compacted free space using a short inline instruction sequence [1]. The sequence is inlined for both structures and arrays because both are allocated frequently. Figure 1 shows a typical sequence for a 12-byte structure in a single-threaded program. The allocation and allocation limit pointers are stored in global variables because the x86 has so few registers.

Each newly allocated object must also be initialized: a vtable pointer is stored at offset 0 and the rest of the object is zeroed. The manner in which memory is zeroed has a significant effect on performance and is discussed in Section 5.

2.2.2 Pointer tracking

All objects allocated on the heap are valid Java objects. Each object has at offset 0 a pointer to a vtable that also contains a 32-bit tag describing offsets to pointers in the object. The tag describes whether the object is an array of pointers, an array of non-pointers, or a structure. There are three kinds of tags for structures: a sparse tag that contains up to 7 4-bit offsets, a dense tag that contains a 28-bit bitmap for offsets 1 to 28, and for large structures, an escape tag that points to an array of offsets. All pointers are aligned on 4-byte boundaries, so the offsets are in units of 4-bytes.

2.2.3 Automatic storage management algorithms

Marmot includes several automatic storage management algorithms. The *null garbage collector (nullgc)* allocates but never collects. A program starts with a large empty heap from which it allocates until the program either exits or fails of heap exhaustion.

The *semispace copying garbage collector (copygc)* copies live data between two semispaces using a Cheney scan [5]. Virtual memory for the unused semispace is invalidated between garbage collections so that it ties up neither physical memory nor paging space on disk. There is no large object space [34].

The *2-generation copying garbage collector* divides memory into a nursery and an older generation composed of two semispaces. Most objects are allocated in the nursery. When the nursery fills, live objects are tenured into the older generation using a Cheney scan. There is no large object space, but large objects are pretenured into the older generation. A write barrier partitions the heap so that the nursery can be collected without collecting the older generation.

2.2.4 Write barrier implementations

Marmot includes both filtering and non-filtering write barriers, all of which record slot pointers rather than object pointers. Non-filtering write barriers record all pointer stores. Filtering write barriers check whether a stored pointer is cross-generational before recording it. Filtering can be worthwhile because cross-generational pointer stores can be rare and because it can take less time to filter than to record a pointer store and to scan the recorded information during collection.

```
mov eax,allocationPtr      ; load allocation pointer into a register
add eax,12                 ; add amount of space to be allocated
cmp eax,allocationLimitPtr ; compare against limit pointer
ja  needgc                 ; call gc if necessary
mov allocationPtr,eax      ; update the allocation pointer
```

Figure 1: Allocation sequence for a 12-byte structure

Figure 2 shows the filtering code sequence shared by all the filtering write barriers. It uses a table of ages indexed by the higher-order 16 bits of pointer addresses [29]. The ages are inverted (older generations are assigned lower numbers) and 0 is used for non-heap memory such as stacks and statically-allocated data. Pointer stores into non-heap memory are not recorded.

An explicit check for pointer stores into non-heap memory is needed because Marmot allocates objects statically and on the stack [13] in addition to allocating objects on the heap. Thus, a pointer store to an object field may store into a non-heap-allocated object.

Marmot includes several filtering write barriers, all of which are called out-of-line from the above filtering code. Recording is often rare enough that the call overhead is minimal.

The *sequential store buffer (ssb)* appends destination addresses to a buffer [1, 19, 17]. Buffer overflow triggers a collection of the nursery. Buffer overflow is detected by an explicit check, though a guard page could be used instead.

The *card table (cards)* divides memory into cards that are a power of 2 in size and alignment [25, 29, 36]. The card table stores one bit of information per card. Initially unmarked, a card table entry is marked to indicate that the corresponding card may contain a cross-generational pointer. Each card table entry is put in a separate byte so that it can be set atomically by a byte store of 1 [4, 17]. There is no cost difference between storing 0 and 1 on the x86, so 0 represents "unmarked" to avoid the cost of initializing the table with nonzero entries.

Each marked card is processed during garbage collection by scanning the pointer-containing fields of each object in the card. A card offset table contains the location of the start of the last object in a card [17]; the offset for the previous card is used to find the start of the first object for the current card. Once the first object has been found, vtable fields are used to find pointer locations in the first object and to calculate the location of the next object. The start-up cost of card scanning is amortized by scanning ranges of contiguously-marked cards.

The *2-level card table (cards2)* is similar to the word-marking write barrier of the Lucid Common LISP ephemeral garbage collector [29]. It has two levels: a coarse grain level that has 1 bit per 4 KB of virtual memory and a fine grain level that has 1 bit per pointer (word). Like the ssb, the fine grain word-marking level precisely identifies potential cross-generational pointers, avoiding the object scanning and card offset table required with larger cards. The coarse grain level allows large unmarked areas to be scanned quickly: 128 KB per 32-bit test for 0.

Marmot also includes an *unfiltered card table (cardsnf)* that is optimized for fast inline recording [28, 29, 36, 19], as shown in Figure 3. Because there is no filtering to eliminate stores outside of the older heap generations, the card table must cover all user-addressable memory. Objects may be stack allocated and the OS controls the placement of thread stacks, so in general an object may be placed anywhere in user-addressible memory. A 31-bit user address space with 512-byte cards and 1 card table byte per card requires a 4 MByte unfiltered card table. The lack of filtering can also increase card scanning overhead during garbage collection and the volume of tenured garbage [33].

2.3 Java Benchmarks

Table 1 describes the 20 benchmark programs used in this study.[1] For each benchmark, the table describes the number of instructions executed, the amount of data heap allocated, and the allocation rate in bytes per instruction. The benchmarks vary widely both in the number of instructions executed and in allocation rates, which range from almost no allocation (impgo and docgo) to 0.26 bytes/instruction (cn2).

The benchmarks are divided into four groups. The SPEC JVM98 group includes six Java programs from the SPEC JVM98 benchmark suite [8]. Dieckmann and Hölzle have described the allocation behavior of these programs in detail [9]. The MiscJava group includes a variety of other Java programs, including the Marmot compiler itself.

The IMPACT group includes a set of C programs translated to Java by the IMPACT/NET project [18]. Several of these originally came from the SPEC95 suite [7]. The DOCTOR group includes several programs from the IMPACT group that have been modified slightly. The 099.go program was changed to be more faithful to the original C version: many static arrays that had been transcribed as instance variables were redeclared as static variables. The 130.li program had its own garbage collector; it was altered to use the Java garbage collector instead.

2.4 Hardware Platform

Benchmarks were run on a 300 MHz dual Pentium II (x86 Family 6 Model 5) processor Gateway2000 E-5000 running Windows NT Version 4.0, Service Pack 3. It had data caches of 8 KB (L1, split) and 256 KB (L2, combined), and 512 MB of 60 ns physical memory organized into 4 KB virtual memory pages.

3. EACH COLLECTOR IS SOMETIMES SLOW

Table 2 shows that none of our 3 collectors is always better than the others. For each collector there are some benchmarks that perform significantly better with a different collector.

The percentages in parentheses give the increase in overall program execution time for a given collector relative to the best collector for that benchmark. Any collector used across

[1] Another 20–30 benchmarks, many transcribed from C, were rejected as having too little storage management overhead to be interesting.

```
    mov  destSegment,destAddr          ; calculate 64K segment number
    shr  destSegment,16
    mov  destAge,ageTable[destSegment] ; get inverted age
    mov  srcSegment,src                ; calculate 64K segment number
    shr  srcSegment,16
    cmp  destAge,ageTable[srcSegment]  ; compare inverted ages
    jb   mayRecord                     ; if dest is older than source
doneRecording:

. . .

mayRecord:
    cmp  destAge,0                     ; check that dest is a heap address
    je   doneRecording                 ; if dest is not a heap address
    call recordPointerInWriteBarrier   ; record pointer store
    jmp  doneRecording
```

Figure 2: Filtering code for detecting cross-generational pointers

```
    mov cardIndex,destAddr
    shr cardIndex,9
    mov dword ptr [_unfilteredCardTableBase+cardIndex*1], 1
```

Figure 3: Recording code sequence of the unfiltered card table

Name	Instrs (10^9)	Alloc (MBs)	Bytes / Instr	Description
The SPECJVM98 group				
_201_compress	8.674	113.7	0.01	An LZW compression program
_202_jess	3.999	265.6	0.07	The Java Expert System Shell
_209_db	2.852	85.3	0.03	An in-memory database
_213_javac	3.295	187.7	0.06	Java bytecode compiler
_222_mpegaudio	11.236	3.2	0.00	MPEG-3 audio decoding algorithm
_228_jack	1.564	147.3	0.09	A parser generator
The Misc Java group				
cn2	2.102	544.2	0.26	CN2 induction algorithm
javacup	0.387	35.5	0.09	JavaCup generating a Java 1.1 parser
jlex	0.075	1.7	0.02	JLex generating a lexer from sample.lex
jessword	0.998	10.6	0.01	JESS solving a word puzzle
jessmab	0.373	18.4	0.05	JESS solving the monkeys and banana problem
marmot0	27.828	1855.2	0.06	Marmot compiling _213_javac
marmot3	186.711	11867.7	0.07	Marmot compiling itself
parser	1.021	31.5	0.03	Java 1.1 parser generated by JavaCup parsing itself
qsort0	1.785	19.1	0.01	Sorting 1,000,000 objects using quicksort
The IMPACT group				
impgo	21.749	0.7	0.00	The SPEC95 099.go benchmark
impijpeg	1.066	41.2	0.04	The SPEC95 132.ijpeg benchmark
impli	85.771	1964.3	0.02	The SPEC95 130.li benchmark using its own heap
The DOCTOR group				
docgo	17.247	0.3	0.00	The SPEC95 099.go benchmark
docli	57.009	6445.5	0.11	The SPEC95 130.li benchmark using the Marmot heap

Table 1: The Java benchmark programs.

GC + WB	Fastest	Slowest benchmarks
nullgc	qsort0 javacup	cn2 docli impli marmot0 marmot3 (did not finish)
copygc	_213_javac	marmot3 (51.7%) cn2 (26.6%) docli (6.9%)
gengc + ssb	cn2 marmot3	javacup (18.0%) _213_javac (16.9%) _209_db (15.5%) docli (14.7%)
gengc + cards2	_201_compress	qsort0 (19.7%) _213_javac (17.8%) javacup (17.7%) _209_db (15.9%)
gengc + cards	_228_jack	_213_javac (20.2%) javacup (18.3%) _209_db (15.8%)
gengc + cardsnf	_202_jess docli	_213_javac (18.6%) javacup (18.5%) _209_db (15.2%) cn2 (14.5%)

Table 2: The fastest and slowest benchmarks for each garbage collector.

all benchmarks had at least 2 benchmarks that ran more than 15% slower than was necessary. The generational collectors did poorly on benchmarks that had too little garbage collection overhead to recover the up front cost of the write barrier. The semispace collector did poorly on the benchmarks that rapidly allocate and discard large amounts of data. The null collector simply failed of heap exhaustion on some benchmarks.

Table 2 also shows some benchmarks for which each collector is well suited. Among the generational collectors, the ssb and unfiltered card table each had benchmarks for which they were clearly fastest. The 1- and 2-level filtered card tables never did better than tie for fastest. When the nullgc was not exhausted, it typically tied with the copygc.

The performance of some benchmarks changed little with changing collectors. For example, the SPECJVM benchmarks _201_compress and _222_mpegaudio spend their entire time in numeric computation, and varying the collector has little effect.

4. STORAGE MANAGEMENT COSTS ARE APPLICATION AND COLLECTOR SPECIFIC

Figure 4 shows how storage management costs vary by benchmark and by collector. It decomposes benchmark elapsed time into:

- *non-heap time*, spent doing real work including allocating heap objects,
- *filtering time*, spent determining whether a pointer crosses into a younger generation,
- *recording time*, spent remembering locations containing a cross–generation pointer,
- *scanning time*, spent examining remembered locations,
- *nursery gc time*, spent collecting the younger generation excluding scanning time, and
- *older gc time*, spent collecting the older generation with the generational gc or a semi-space with the copying gc.

Total benchmark elapsed time, write barrier scanning time, nursery collection time and older generation collection time are measured directly by subtracting entry times from exit times. Filtering and recording time are measured indirectly as the increase in elapsed time when write barrier code is added to the non-generational collectors.

Figure 4 is scaled to emphasize storage management costs. Results are grouped by their proportion of storage management overhead (note the differing maximum values of the horizontal axis). For each benchmark, results are scaled so that the lowest non-heap time observed across all collectors is 100%. Thus, most non-heap time is elided to the left of the figure.

Surprisingly, non-heap times sometimes vary significantly across collectors (_202_jess, _209_db, _213_javac, _228_jack). The SPECJVM benchmark _209_db is often stalled on L2 D-cache misses, resulting in a large CPI (4-5), so the extra memory traffic generated by any of the write barriers slows all the generational collectors by 15-20%. The variation in non-heap times of the other 3 SPECJVM benchmarks and cn2 results from the way that the different collectors zero memory, as will be discussed Section 5.

The largest collector-dependent performance changes in these benchmarks occur when alternating between generational and non-generational collectors. For some benchmarks with significant collection costs, a generational collector reduces collection costs enough to recover the up-front cost of the write barrier (marmot3 by 51%, cn2 by 26%). Other benchmarks are faster with the copying collector than with any generational collector (javacup by 18%, _213_javac by 17%, _209_db by 15%). The collection costs are so low that the cost of the write barrier is never recovered.

Among generational collectors, there is sometimes a more modest performance difference between write barriers. For example, the ssb was 14% faster than the unfiltered card table for cn2, but 13% slower for docli. For 15 benchmarks, however, the overall times for each of the several write barrier schemes fell within 2% of each other. For these benchmarks, although the time may be distributed significantly differently among filtering, reporting, scanning, etc., the choice of write barrier makes little overall difference. This is consistent with the measurements reported by Hosking et al. [17].

The qsort0 benchmark is a pathological case for the write barriers. It has a large, pretenured array of pointers that point at objects in the nursery. As the array is sorted, each assignment into the array is recorded in the write barrier. By contrast, fewer than 1% of pointer stores record in more than half of the other benchmarks. The large number of records overruns the ssb, triggering minor collections that tenure the objects, eliminating the cost of subsequent records but adding the cost of the scans and minor collections. Lacking any of these costs, the semispace collector is faster than any of the generational collectors on qsort0.

Write barrier costs further illustrate the application-dependent nature of storage management costs. The filtering cost may be offset by corresponding reductions in recording and scanning cost (cn2) or may not (docli). Sometimes the filtering cost can exceed the recording cost of the unfiltered card table (docli).

The scanning times of the filtered and unfiltered card tables may be significantly greater than those of the ssb and 2-level card table (cn2, _213_javac, marmot0) or lower (docli for the ssb). None of the benchmarks show an increase in

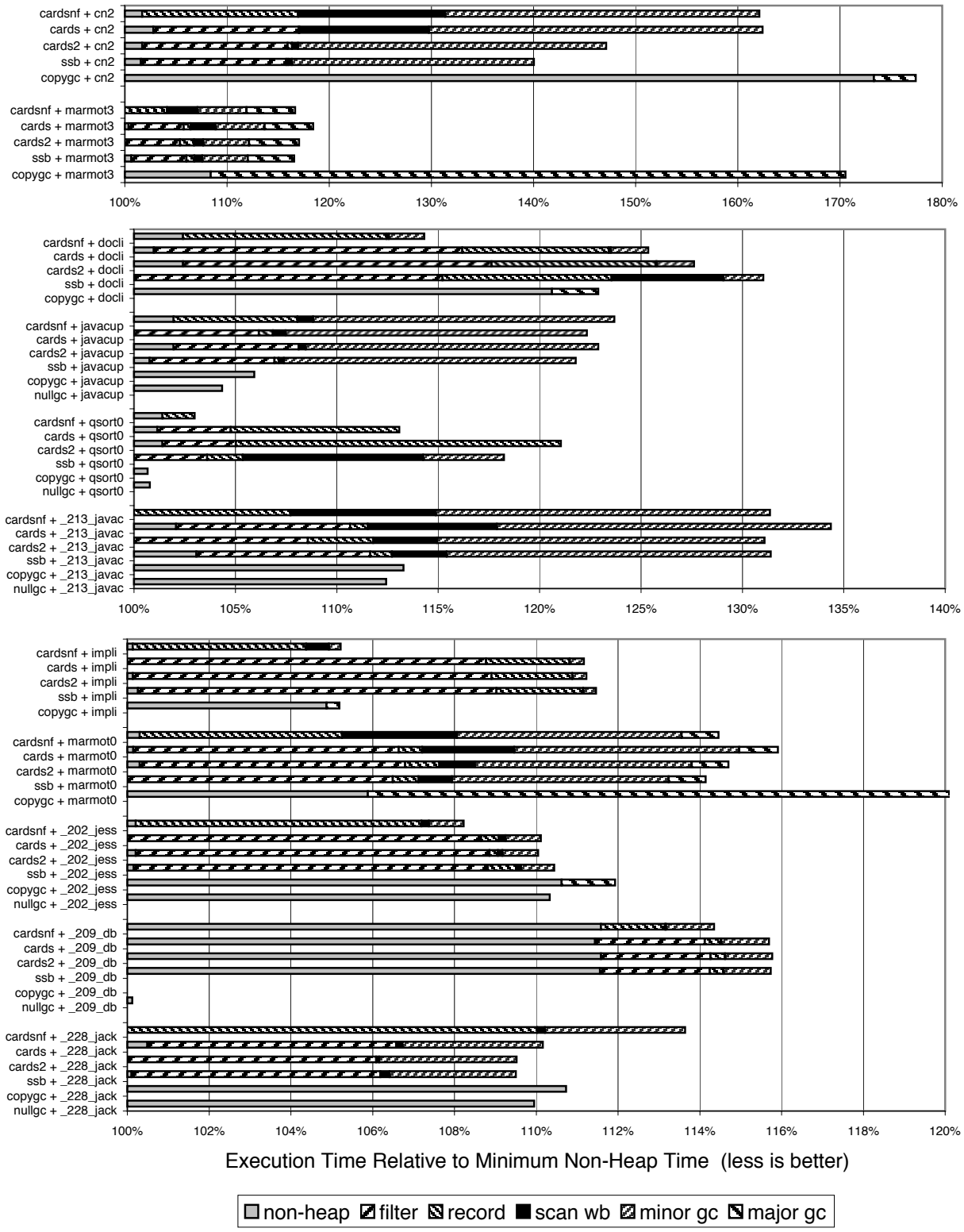

Figure 4: Detailed Costs of Storage Management

untenured garbage due to the conservative scanning forced by larger cards.

The 2-level card table may be faster than the corresponding single-level card table (cn2, _213_javac) or slower (docli), depending on whether the reduction in scanning time offsets the increase in recording time.

5. ZEROING MEMORY INCREMENTALLY CAN BE FASTER THAN ZERO-FILL FAULTS

Newly allocated memory in Java must be zeroed [15]. This is done in two distinct ways by the various collectors. The generational collectors normally zero the nursery incrementally as objects are allocated. The semispace and null collectors obtain zero-filled memory on demand from the operating system via zero-fill page faults. The semispace collector and the older generation of the generational collectors invalidate the virtual memory for the unused semispace and revalidate the virtual memory for the current semi-space.

Figure 5 compares the costs of incrementally zeroing memory and of zero-fill page faults. For each of the three SPECJVM benchmarks for which zeroing is a significant cost, it shows heap times (the sum of filter, record, and collect times) and non-heap times for each of seven collectors. The collectors include the semispace collector (copygc), the null collector (nullgc), generational collectors (cardsnf, cards2, cards, and ssb), and a hybrid collector (ssb_os_zero). The hybrid collector is a generational collector (ssb) modified to invalidate and revalidate the nursery during minor collections, so it gets zero-filled memory from the operating system instead of zeroing it during allocation. The generational collectors that incrementally zero memory have uniformly lower non-heap costs. The hybrid generational collector has higher non-heap costs that are comparable to those seen in the semispace and null collectors as well as the heap costs of the generational collectors, demonstrating that incrementally zeroing memory can be faster than getting zero-filled VM pages.

There are several reasons why zeroing memory incrementally during allocation can be faster than getting zero-filled pages from the operating system. The latter has the obvious costs of invalidating and revalidating virtual memory and of handling zero-fill page faults, but potentially reduces the demand for physical memory by releasing physical memory pages until they are needed. The former can also have better D-cache locality than the latter. Incremental zeroing often moves memory through the D-cache only once because the newly allocated memory is used shortly after it has been allocated. Zero-fill page faults can move memory through the D-cache twice: once when the page is zeroed and flushed to physical memory[2] and again when the memory is used after it has been allocated.

Note that benchmark–collector pairs sometimes have approximately the same elapsed times for different reasons. The non-hybrid SPECJVM benchmarks _202_jess and _228_jack have comparable elapsed times although the generational collectors have about 10% heap overhead while the null and semispace heaps have no heap overhead but pay about 10% in increased memory zeroing costs.

[2]Dougan et al. [10] recommend that at least some operating system page zeroing go around the D-cache to avoid cache pollution. Uncached stores may have their own costs.

6. PROFILE-SELECTED COLLECTORS IMPROVE AVERAGE SPEED

Figure 6 shows that the average speed of using a "profile selected" collector for each benchmark is better than that of using any of several fixed collectors across all benchmarks. The former strategy uses a garbage collector that was selected as being most appropriate for each benchmark, e.g. that gave the shortest elapsed time on some prior training run. Benchmarks whose behavior varies based on input may in general require some sort of averaged selection, although that was not necessary in this collection of 20 benchmarks. The fixed collectors include the 2-generation collector with unfiltered and filtered card tables of 128, 256 and 512 byte cards, with the 2-level card table and with sequential store buffers of 64 KB, 1 MB and 16 MB.

The average speed of the benchmark suite for each collector is calculated as the harmonic mean of rates [16]. First, for each benchmark, the average elapsed time using that collector is normalized to the lowest average elapsed time using any collector. Each benchmark–collector pair was run repeatedly until the standard deviation of the average elapsed time was nominal. Second, the normalized elapsed times across all the benchmarks using that collector are averaged and inverted. The error bars show the standard deviation of the average. They become larger (worse) when a collector performs relatively worse for some benchmarks than for others.

A new "profile selected" binary was rebuilt and remeasured once a suitable collector had been chosen for each benchmark. That composite collector differs from 100% in Figure 6 because of measurement noise. In general, it could also differ because of input-dependent variations in behavior.

Figure 6 also shows that the choice of write barrier makes little difference in average speed. Moderate card sizes (around 512 bytes) have good average speed among the 1-level card tables. The averages of larger and smaller card sizes are disproportionately influenced by a few extreme outliers. If those few outliers are discarded, the resulting average speed is even more insensitive to the choice of card size. This is consistent with the measurements reported by Hosking et al. [17], even though their interpreted Smalltalk environment is quite different from a compiled Java environment.

Because it pays neither for a write barrier nor for garbage collection, the nullgc can be surprisingly competitive on benchmarks that do not fail of heap exhaustion. To get the same benefits, the copygc is configured to collect as rarely as possible, i.e. as a null garbage collector with a parachute. Doubling the interval between collections roughly halves the number of collections and thus halves the total collection cost. These benchmarks also run faster if they disregard explicit requests to collect, e.g. calls to `java.lang.System.gc()`.

7. SUMMARY

Automatic storage management costs can vary considerably by application and collector. For some systems such as those that execute stand-alone compiled executables, profile information can be used to pair an application with a suitable garbage collector chosen from a pool of collectors. This can significantly improve the performance of applications that would otherwise have been paired with an unsuitable collector. For 20 benchmarks and 3 garbage collectors, it

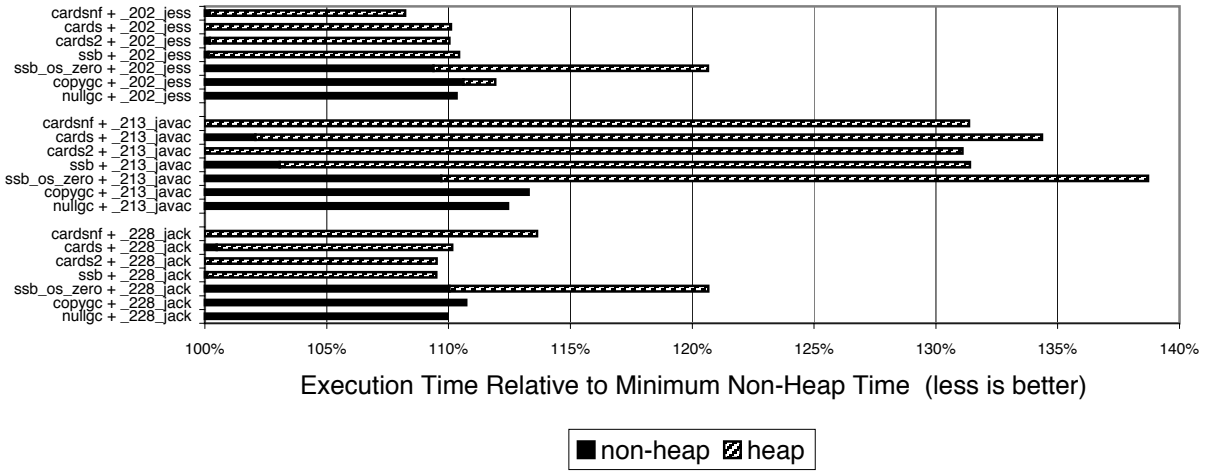

Figure 5: The cost of zeroing memory

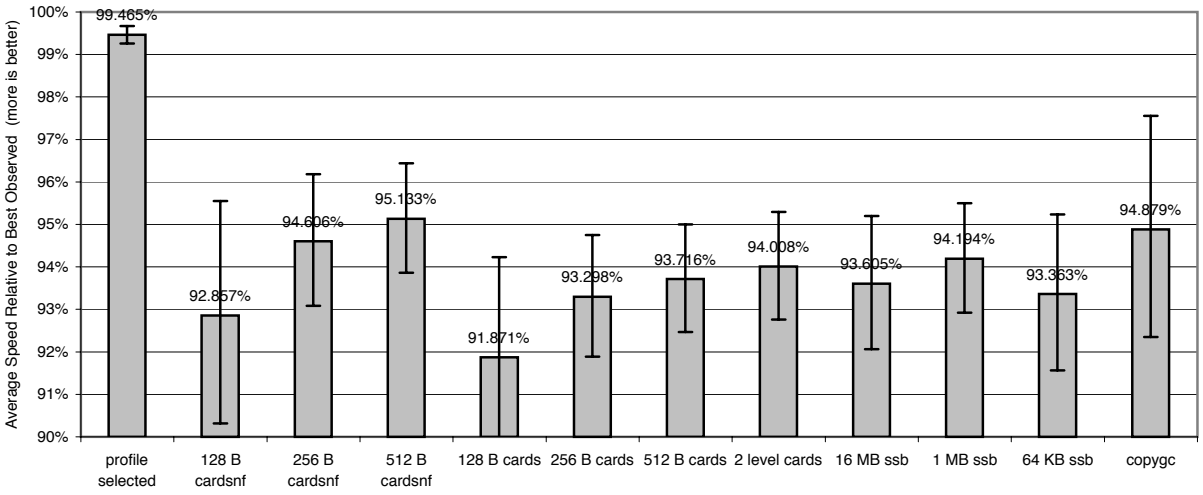

Figure 6: The average speed of the benchmark suite using profile-directed collector selection compared to using any of several fixed collectors.

improved the overall performance of individual applications by more than 15%.

No single collector performed uniformly better than all the others. The most significant choice affecting performance was whether to use a generational collector. Generational collectors did well on benchmarks that had high collection costs, improving performance by about 50% for the marmot3 benchmark and about 25% on the cn2 benchmark. They did poorly on benchmarks that had low collection costs and high write barrier costs. For those benchmarks, the cost of the write barrier was higher than the reduction in collection costs. The _209_db, _213_javac and javacup benchmarks were 15-20% faster with a non-generational semispace collector.

The choice of write barrier sometimes made a significant difference for individual benchmarks (up to 14% for cn2 and 13% for docli), even though the average performance across all benchmarks of the different write barriers fell within 2% of each other.

No write barrier performed uniformly better than the others. For some benchmarks, filtering before recording improved performance. For others, it was faster to record everything. Write barrier scan times were lower with a card table for some benchmarks and lower with a sequential store buffer for others. The 2-level card table typically decreased the cost of scanning but increased the cost of recording, making some benchmarks faster and others slower.

In all, storage management costs were highly application dependent. This creates an opportunity to improve the performance of some applications significantly by pairing each application with an appropriate collector.

8. REFERENCES

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

[2] Andrew W. Appel. Book review: "Garbage Collection: Algorithms for Automatic Storage Management", by Richard Jones and Rafael Lins. *Journal of Functional Programming*, 7(12):219–225, 1997.

[3] David A. Barrett and Benjamin G. Zorn. Garbage collection using a dynamic threatening boundary. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 301–314, La Jolla, California, June 1995. *SIGPLAN Notices* 30(6).

[4] Craig Chambers. The design and implementation of the Self compiler, an optimizing compiler for object-oriented programming languages. Ph.D. dissertation, Stanford University, March 1992.

[5] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[6] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 162–173, Montreal, Canada, June 1998. ACM Press.

[7] Standard Performance Evaluation Corporation. SPEC CPU 95 benchmarks. Online version at http://www.spec.org/osg/cpu95, 1995.

[8] Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks. Online version at http://www.spec.org/osg/jvm98, 1998.

[9] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 92–115, Lisbon, Portugal, June 1999. Springer-Verlag.

[10] Cort Dougan, Paul Mackerras, and Victor Yodaiken. Optimizing the idle task and other MMU tricks. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation, Feb 22–25, 1999, New Orleans, Louisiana*, pages 229–237, February 1999.

[11] R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[12] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. *Software—Practice and Experience*, 30(3):199–232, March 2000.

[13] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *9th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 82–93. Springer-Verlag, 2000.

[14] Adele Goldberg and David Robson. *Smalltalk-80: the language*. Addison-Wesley, Reading, MA, USA, 1989.

[15] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, California, second edition, 1996.

[17] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic̀. A comparative performance evaluation of write barrier implementations. In *Proceedings of the ACM '92 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 92–109, Vancouver, British Columbia, October 1992. ACM Press.

[18] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In IEEE, editor, *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture, December 2–4, 1996, Paris, France*, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[19] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, June 1991.

[20] Instantiations, Inc. Jove: Super optimizing deployment environment for Java. http://www.instantiations.com/javaspeed/jovereport.htm, July 1999.

[21] Richard Jones and Rafael Lins. *Garbage Collection: algorithms for automatic dynamic memory management*. John Wiley and Sons, 1996.

[22] Henry Lieberman and Carl Hewitt. A real–time garbage–collector based on the lifetimes of objects. *Communications of the ACM*, 23(6):419–429, 1983.

[23] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[24] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(1):184–195, 1960.

[25] David A. Moon. Garbage collection in a large Lisp system. In Guy L. Steele, editor, *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246, Austin, Texas, August 1984. ACM Press.

[26] NaturalBridge. Bullettrain compiler. http://www.naturalbridge.com/, 1999.

[27] Erik Ruf. Removing synchronization operations from Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 208–218, Vancouver, Canada, June 2000. ACM Press.

[28] Robert Shaw. Empirical analysis of a LISP system. Ph.D. dissertation, Stanford University, Cambridge, MA, 1988. Also Computer Systems Laboratory Technical Report CSL-TR-88-351.

[29] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers. B.S. thesis, Dept. of EECS, Massachusetts Institute of Technology, Cambridge, MA, 1988. Also Technical Report AITR-1417, MIT, AI Lab, Feb 1988.

[30] Guy L. Steele, Jr. *Common LISP: the language*. Digital Press, 1990.

[31] TowerJ Technology Corporation. TowerJ 3 – a new generation native Java compiler and runtime

environment. http://www.towerj.com/, 1999.

[32] David Ungar. Generation scavenging: a non–disruptive high performance storage reclamation algorithm. *SIGPLAN Notices (Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments)*, 19(5):157–167, 1984.

[33] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[34] David M. Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *Proceedings of the ACM '88 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–17, September 1988.

[35] David M. Ungar and Frank Jackson. Outwitting gc devils: A hybrid incremental garbage collector. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. Available for anonymous FTP from cs.utexas.edu in /pub/garbage/GC91.

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