Improving Locality with Parallel Hierarchical Copying GC *

David Siegwart
IBM Software Group
Hursley (UK)
david_siegwart@uk.ibm.com

Martin Hirzel
IBM Watson Research Center
Hawthorne, NY (USA)
hirzel@us.ibm.com

Abstract

This paper shows how to reduce cache and TLB misses by changing the order in which a parallel garbage collector copies heap objects. Reducing cache and TLB misses improves program run time. Parallel garbage collection improves scaling on multi-processor machines. Technology trends indicate that both memory locality and multi-processor scaling increase in importance. Our new algorithm is based on the earlier single-threaded "hierarchical decomposition" algorithm by Wilson, Lam, and Moher. This paper presents a thorough evaluation of parallel hierarchical copying, showing that it improves spatial locality, reduces cache and TLB misses, and speeds up 14 out of 26 benchmarks.

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

General Terms Languages, Performance, Experimentation, Algorithms

Keywords parallel, generational, cache locality

1. Introduction

Programs spend a lot of time stalled in cache and TLB misses, because computation tends to be faster than memory access. For example, Adl-Tabatabai et al. report that the SPECjbb2000 benchmark spends 45% of its time stalled in misses on an Itanium processor [1]. Better locality reduces misses, and thus improves performance. For example, techniques like prefetching or cache-aware memory allocation improve locality, and speedups such as 14% [1], 25% [21], and 21% [29] have been measured.

Locality is in part determined by the order of heap objects in memory. If two objects reside on the same cache line or page, then an access to one causes the system to fetch this cache line or page. A subsequent access to the other object is fast. Copying garbage collection (GC) can change the order of objects in memory. To improve locality, copying GC should strive to colocate related objects on the same cache line or page. This paper presents and evaluates a GC algorithm that improves locality by colocating related objects.

* This research was funded in part by DARPA contract No. NBCH30390004

ISMM'06 10–11 June, Ottawa, Ontario, Canada.

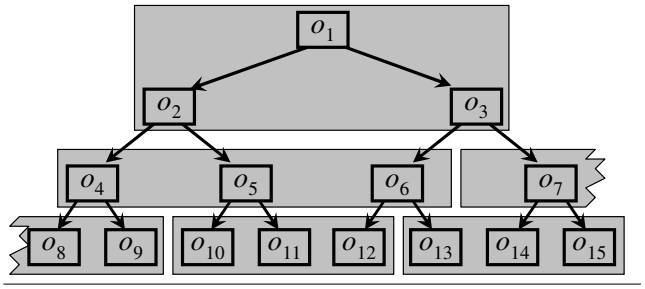


Figure 1. Breadth first copy order.

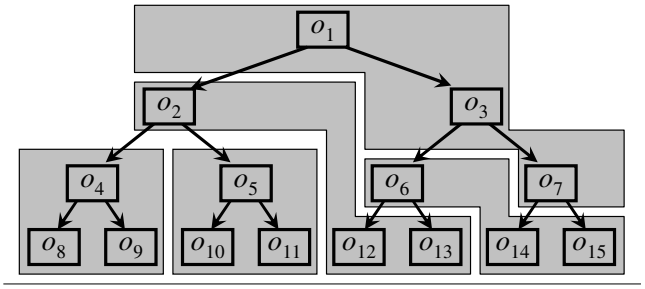


Figure 2. Depth first copy order.

Copying GC traverses the graph of heap objects, copies objects when it reaches them, and recyles memory of unreachable objects afterwards. Consider copying a binary tree of objects, where each cache line can hold three objects. When the traversal uses a FIFO queue, the order is breadth-first and results in the cache line layout in **Figure 1**. When the traversal uses a LIFO stack, the order is depth-first and results in the cache line layout in **Figure 2**. In both cases, most cache lines hold unconnected objects. For example, breadth-first order colocates o_{10} and o_{11} with o_{12}, even though o_{12} will usually not be accessed together with o_{10} or o_{11}.

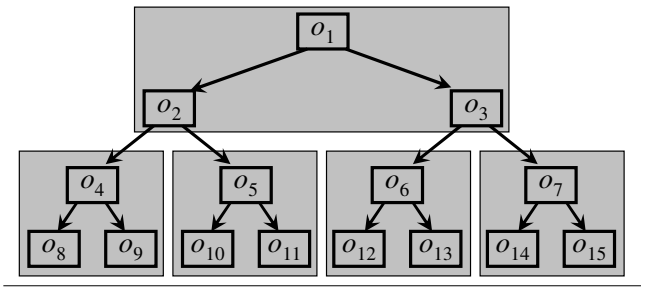


Figure 3. Hierarchical copy order.

Intuitively, it is better if an object occupies the same cache line as its siblings, parents, or children. Hierarchical copy order achieves this (**Figure 3**). Moon invented a hierarchical GC in 1984, and Wilson, Lam, and Moher improved it in 1991 [31], calling it "hierarchical decomposition". The algorithms by Moon and by Wilson, Lam, and Moher use only a single GC thread. Using multiple parallel GC threads reduces GC cost, and most product GCs today are parallel.

Despite the fact that hierarchical GC is well known (Jones discusses it in the GC book [22]), to our knowledge, no GC for Java uses it today. We speculate that this is because it has been unclear how to achieve hierarchical order in a parallel GC, and how beneficial it is. This paper answers both questions.

This paper presents parallel hierarchical copying GC, a new parallel copying GC algorithm that achieves hierarchical copy order. The algorithm is non-trivial, but its implementation is simple, and needs no online profiling or compilation. This paper presents experimental results for using generational parallel hierarchical copying GC for 26 Java programs running in a product Java virtual machine on IA32 hardware. Besides wall-clock time, the results include cache and TLB miss counts, as well as a metric for spatial locality, all measured at various heap sizes. Of the 26 programs, 14 speed up, 4 are unaffected, and 8 slow down.

Section 2 sets the stage, Section 3 presents the algorithm, Section 4 provides experimental results, Section 5 discusses related work, and Section 6 concludes.

2. Background

Section 2.1 describes Cheney's copying GC. Cheney's algorithm copies in breadth-first order; Sections 2.2 and 2.3 describe algorithms based on Cheney that copy in hierarchical order instead. Hierarchical order improves locality, and thus reduces memory stalls in the mutator. Cheney's algorithm (as well as its hierarchical variants) are sequential; Sections 2.4 and 2.5 describe parallel GC algorithms based on Cheney. Parallelism multiplies GC throughput.

2.1 Cheney's copying GC

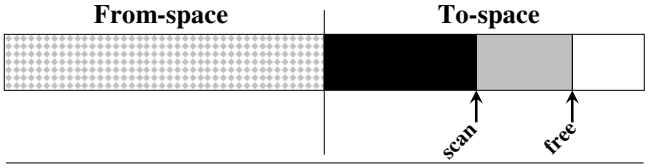


Figure 4. Cheney's copying GC.

Figure 4 illustrates Cheney's copying GC algorithm [6]. Memory has two semi-spaces, from-space and to-space. At GC start, all heap objects are in from-space, and all of to-space is empty □. GC first scans the program variables for pointers to heap objects, and copies their target objects from from-space to to-space. Copied objects are gray, and a "free" pointer keeps track of the boundary between gray objects ▒ and the empty part of to-space □. Next, GC scans copied objects for pointers to from-space, and copies their target objects to to-space. Scanned objects are black, and a "scan" pointer keeps track of the boundary between black objects ■ and gray objects ▒. When the scan pointer catches up to the free pointer, GC has copied all heap objects that are transitively reachable from the program variables. From-space is discarded, and the program continues, using the objects in to-space.

The previous paragraph introduced terminology for the rest of this paper: *black* ■ means copied and scanned, *gray* ▒ means copied but not yet scanned, and *empty* □ means free available memory. (We avoid referring to □ as white, since white has a different meaning in the GC literature [12]).

Cheney's algorithm copies in breadth-first order (see Figure 1), because it scans gray objects ▒ first-in-first-out. One advantage of Cheney's algorithm is that it requires no separate stack or queue to keep track of its progress, saving space and keeping the implementation simple. Cheney published his GC in 1970 for Lisp [6] without offering performance results. Cheney's algorithm uses only one thread for garbage collection, it is not parallel.

2.2 Moon's hierarchical copying GC

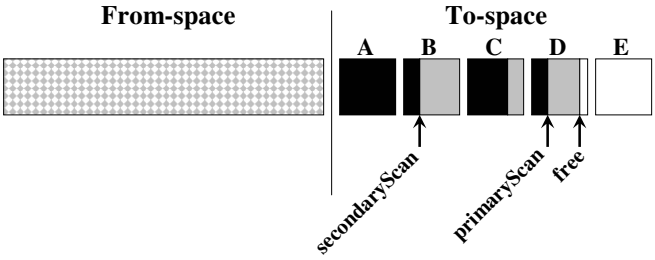


Figure 5. Moon's hierarchical copying GC.

Moon modified Cheney's algorithm to improve locality by copying in hierarchical order instead of breadth-first. **Figure 5** illustrates Moon's algorithm [27]. To-space is now divided into blocks. As before, objects are copied by bumping the free pointer, which separates gray objects ▒ from empty space □. But instead of just one scan pointer, Moon maintains two scan pointers. The primary scan pointer is always in the same block as the free pointer. For example, in Figure 5, both the primary scan pointer and the free pointer point into block D.

If there are gray objects at the primary scan pointer, Moon scans them. If the free pointer reaches the next block (for example E), Moon advances the primary scan pointer to the start of that block, even though there may still be gray objects in the previous block (for example D). The secondary scan pointer keeps track of the earliest gray objects (for example, in block B). If the primary scan pointer catches up with the free pointer, Moon scans from the secondary scan pointer, until the primary scan pointer points to gray objects again. If the secondary scan pointer catches up with the free pointer as well, GC is complete.

Moon's algorithm copies objects in hierarchical order. For example, in Figure 3, Moon's algorithm first copies o_1 and its children, o_2 and o_3, into the same block. Next, it copies o_4 (the first child of o_2) into a different block. At this point, the block with o_4 has a gray object at the primary scan pointer, so Moon proceeds to copy the children of o_4 into the same block as o_4. Only when it is done with that block does it continue from the primary scan pointer, which still points into o_2.

The *mutator* is the part of an executing program that is not part of the GC: the user program, and run time system components such as the JIT compiler. Moon's GC is concurrent to the mutator, but there is only one active GC thread at a time, no parallel GC threads.

One problem with Moon's algorithm is that it scans objects twice when the secondary scan pointer advances through already black objects ■ (for example in block C in Figure 5). Moon published his GC in 1984 for Lisp [27]. Moon's paper evaluates performance of two Lisp benchmarks on a Lisp machine.

2.3 Wilson, Lam, and Moher's hierarchical copying GC

Wilson, Lam, and Moher improve Moon's algorithm by avoiding re-scanning of black objects. **Figure 6** illustrates Wilson, Lam, and Moher's algorithm [31]. It keeps track of the scan pointers in all partially scanned blocks. When the block with the free pointer contains gray objects (for example block D), scanning proceeds in

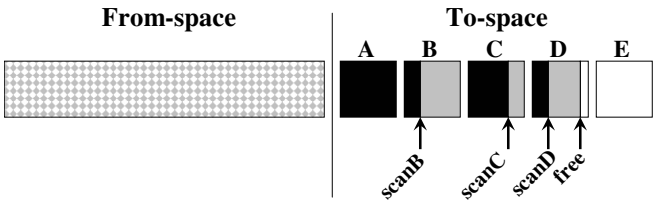


Figure 6. Wilson, Lam, and Moher's hierarchical copying GC.

that block; otherwise, it proceeds from the earliest block with gray objects (for example block B). The copy order of Wilson, Lam, and Moher's algorithm is identical to that of Moon's algorithm (see Figure 3). It was published in 1992, and evaluated by collecting page access traces for three Lisp programs, and using those traces to drive a paging simulator. The hierarchical copying GC algorithm by Wilson, Lam, and Moher is neither parallel nor concurrent.

2.4 Halstead's parallel copying GC

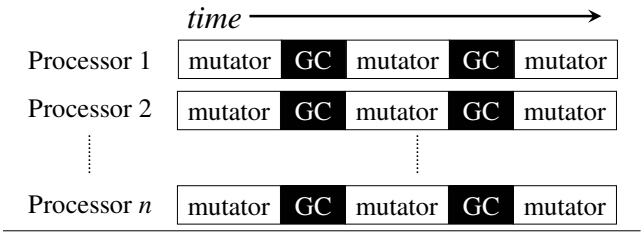


Figure 7. Parallel GC.

In 1985, Halstead published the first parallel GC algorithm [16]. It is based on Baker's GC [3], which is an incremental variant of Cheney's GC [6]. The incremental aspect is not relevant for this paper. Halstead's GC works on shared-memory multiprocessor machines with uniform access time to the shared memory. The garbage collector works in SIMD (single instruction, multiple data) style: each worker thread performs the same GC loop on different parts of the heap. The mutator may be SIMD or MIMD (multiple instruction, multiple data). As illustrated in **Figure 7**, at any given point in time, either GC threads are running or mutator threads are running, but not both. The GC is parallel, but not concurrent.

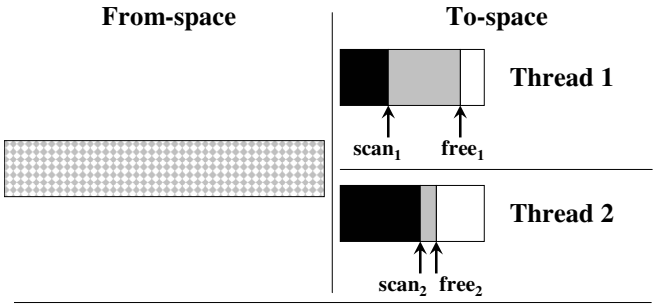


Figure 8. Halstead's parallel copying GC.

Halstead's algorithm partitions to-space into n equally sized parts on an n-processor machine. **Figure 8** illustrates the heap organization for $n = 2$. Worker thread i has a scan pointer scan_i and a free pointer free_i, which point to gray objects ▒ and empty space □ in their respective parts of to-space. Termination detection is simple: when $\text{scan}_i = \text{free}_i$ for all i, then there are no more gray objects to scan anywhere. Since each thread has its own private part of to-space, the threads do not need to synchronize when scanning objects in to-space or allocating memory in to-space. But they do need to synchronize on individual objects in from-space: if two worker threads simultaneously encounter pointers to the same object in from-space, only one of them should copy it and install a forwarding pointer.

Like Cheney, Halstead has the advantage of requiring no separate queue or stack to keep track of gray objects, because within the part of to-space that belongs to a thread, the objects themselves are layed out contiguously and form an implicit FIFO queue. The algorithm therefore copies in breadth-first order (Figure 1). Unfortunately, the static partitioning of to-space into n parts for n processors leads to work imbalance. This imbalance causes two problems: overflow and idleness. Overflow occurs when a worker thread runs out of empty space □ to copy objects into. Halstead solves this problem by providing additional empty space to worker threads on demand. Idleness occurs when one thread runs out of gray objects ▒ to scan while other threads are still busy. Halstead does not address the idleness problem caused by work imbalance [16].

2.5 Imai and Tick's parallel copying GC

In 1993, Imai and Tick published the first parallel GC algorithm with load balancing [20]. Their algorithm extends Halstead's algorithm by overpartitioning: on an n-processor machine, it partitions to-space into m blocks, where $m > n$.

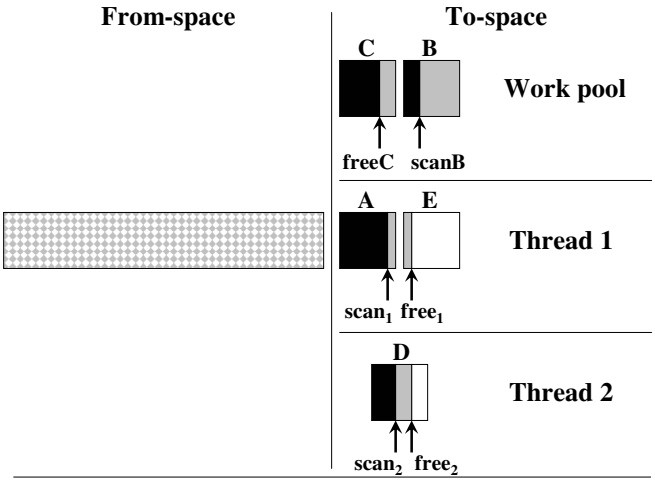


Figure 9. Imai and Tick's parallel copying GC.

Figure 9 illustrates Imai and Tick's GC. Each GC worker thread has one *scan* block with gray objects ▒ to scan, and one *copy* block with empty space □ to copy objects into. These blocks may be *separate* (A and E in Thread 1) or *aliased* (D in Thread 2). A shared work pool holds blocks currently unused by any thread. When a copy block has no more empty space, it is completely gray ▒ or black and gray ◧. The thread puts the copy block into the work pool for future scanning, replacing it with a new empty block □. When the scan block has no more gray objects, it is completely black ■, and thus done for this garbage collection: the thread gets rid of it. Then, the thread checks whether its private copy block has any gray objects (coloring ◧ or ◧). If yes, it aliases the copy block as scan block. Otherwise, it obtains a new scan block from the shared work pool. In addition to having to synchronize on from-space objects like Halstead's algorithm, the algorithm by Imai and Tick also has to synchronize operations on the shared work pool.

The aliasing between copy and scan blocks avoids a possible deadlock where the only blocks with gray objects also have empty space. In addition, it reduces contention on the shared work queue when there are many GC threads. Imai and Tick's GC only checks

for an aliasing opportunity when it needs a new scan block because the old scan block is completely black. Imai and Tick evaluated their algorithm on 14 programs written in a logic language. They report parallel speedups of 4.1× to 7.8× on an 8-processor machine. Their metric for speedup is not based on wall-clock time, but rather on GC "work" (number of cells copied plus number of cells scanned); it thus does not capture synchronization overhead or locality effects.

3. Algorithm

This section contains the first main contribution of this paper: a new GC algorithm that achieves hierarchical copy order with parallel GC threads. The following description assumes that the reader is familiar with the background from Section 2.

3.1 Baseline garbage collector

Our implementation of parallel hierarchical copying GC is based on the generational GC [22] implemented in IBM's J9 JVM. It uses parallel copying for the young generation and concurrent mark-sweep with occasional stop-the-world compaction for the old generation. This is a popular design point in products throughout the industry. The baseline GC has exactly two generations, and young objects remain in the young generation for a number of birthdays that is adapted online based on measured survival rates. We are only concerned with copying of objects within the young generation or from the young generation to the old generation.

The baseline GC uses Imai and Tick's algorithm [20] for the young generation. To accommodate tenuring, each worker thread manages two copy blocks: one for objects that stay in the young generation, and another for objects that get tenured into the old generation. Either block may be aliased as scan block.

3.2 Parallel hierarchical GC

Parallel hierarchical GC achieves hierarchical copy order by aliasing the copy and scan blocks whenever possible. That way, it usually copies an object into the same block that contains an object that points to it. This is the parallel generalization of the single-threaded algorithm by Wilson, Lam, and Moher that uses the scan pointer in the block with empty space whenever possible (Section 2.3). Blocks serve both as the work unit for parallelism and as the decomposition unit for hierarchical copying.

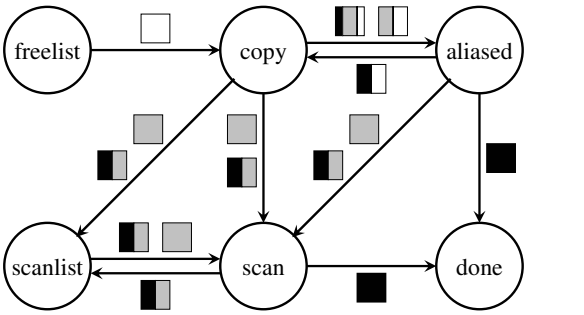


Figure 10. Block states and transitions.

Figure 10 shows the possible states of a block in to-space as circles. Transitions labels denote the possible coloring of the block when a GC thread changes its state. Blocks in states *freelist*, *scanlist*, and *done* belong to the shared work pool. No GC thread scans them or copies into them, and thus, their coloring can not change. Blocks in states *copy*, *scan*, and *aliased* belong to a GC thread. These states have been described in Section 2.5.

For example, a copy block must have room to copy objects into; therefore, all incoming transition labels to state *copy* are at least partially empty □. If the copy block has some gray objects and some empty space, then it can serve both as copy block and as scan block simultaneously, and the GC aliases it; therefore, the transition from state *copy* to state *aliased* is labeled with colorings that include both gray and empty (◧ or ◨). The state machine in Figure 10 is non-deterministic: the state and coloring of a block alone do not determine which transition it takes. Rather, the transitions depend on the colorings of both the copy block and the scan block of the worker thread.

copy	scan aliased	scan ◧ or ▩	scan ■
◧ or ◨	(no action)	scan → scanlist copy → aliased	scan → done copy → aliased
□ or ◧	aliased → copy scanlist → scan	(no action)	scan → done scanlist → scan
◧ or ▩	aliased → scan freelist → copy	copy → scanlist freelist → copy	scan → done copy → scan freelist → copy
■	aliased → done freelist → copy scanlist → scan	(can't happen)	(can't happen)

Table 1. Transition logic in GC thread.

Table 1 shows the actions that the GC thread performs after scanning a slot in an object. For example, if the copy block contains both gray slots and empty space (row copy $\in \{◧, ◨\}$), and the scan block is already aliased with the copy block (column scan = aliased), no action is necessary before the next scanning operation. If copy $\in \{◧, ◨\}$, and the scan block is not aliased, the thread transitions the copy block to the aliased state, and either puts the scan block back on the scanlist if it still has gray slots (scan $\in \{◧, ▩\}$), or transitions it to the done state if it is completely black (scan = ■).

As described in Table 1, parallel hierarchical GC leads to increased contention on the scanlist. To avoid this, our implementation caches up to one block from the scanlist with each thread. Thus, if there is a cached block, the action scanlist → scan really obtains that cached block instead. Likewise, the transition scan → scanlist really caches the scan block locally, possibly returning the previously cached block to the scanlist in its stead.

3.3 Discussion

Like Cheney's algorithm and the other Cheney-based algorithms in Section 2, parallel hierarchical GC requires no separate mark stack or queue of objects. Instead, the gray objects are consecutive in each block, thus serving as a FIFO queue. On the other hand, like Imai and Tick's algorithm, our GC requires a shared work pool of blocks to coordinate between GC threads. In addition, it requires per-block data to keep track of its state and coloring.

After scanning a gray slot, parallel hierarchical GC checks immediately whether it became possible to alias the copy block and the scan block. Since this check happens on the innermost loop of the GC algorithm, it must be fast. The immediacy of this check is what leads to hierarchical order like in the algorithms by Moon and by Wilson, Lam, and Moher.

The goal of hierarchical copy order is improved mutator locality. But of course, it also affects GC locality and load balancing. This effect can be positive or negative.

As mentioned earlier, in our implementation, each GC thread actually manages two copy blocks, one each for young and old objects. Only one of them can be aliased at a time.

4. Results

This section contains the second main contribution of this paper: a thorough evaluation of parallel hierarchical copying GC (PH), compared to parallel breadth-first copying GC (BF). Section 4.1 describes the experimental setup. Section 4.2 shows how hierarchical copy order affects overall run time. Section 4.3 breaks down its effects on mutator and garbage collection performance. Section 4.4 shows how the algorithm scales to different numbers of threads and processors. Section 4.5 investigates time-space tradeoffs. Section 4.6 shows cache and TLB misses, and Section 4.7 quantifies colocation.

4.1 Experimental setup

All experiments for this paper were conducted with a modified version of IBM J2SE 5.0 J9 GA Release (IBM's product JVM), running on real hardware in common desktop and server operating systems. This section discusses the methodology.

The platform for Sections 4.2, 4.3, 4.5, and 4.6 was a dual-processor IA32 SMT system running Linux. The machine has two 3.06 GHz Pentium 4 Xeon processors with hyperthreading. The memory hierarchy consists of an 8 KB L1 data cache (4-way associative, 64 Byte cache lines); a 512 KB combined L2 cache (8-way associative, 64 Byte cache lines); a 64 entry data TLB (4 KB pages); and 1 GB of main memory. The platforms for other sections are described there.

Name	Suite	Description	MB
SPECjbb2005	jbb05	business benchmark	149.3
antlr	DaCapo	parser generator	1.4
banshee	other	XML parser	84.6
batik	DaCapo	movie renderer	15.0
bloat	DaCapo	bytecode optimizer	11.5
chart	DaCapo	pdf graph plotter	25.0
compress	jvm98	Lempel-Ziv compressor	8.8
db	jvm98	in-memory database	13.6
eclipse	other	development environment	4.8
fop	DaCapo	XSL-FO to pfd converter	8.5
hsqldb	DaCapo	in-memory JDBC database	22.6
ipsixql	Colorado	in-memory XML database	2.5
jack	jvm98	parser generator	1.5
javac	jvm98	Java compiler	13.3
javalex	other	lexer generator	1.0
javasrc	Ashes	code cross-reference tool	61.3
jbytemark	other	bytecode-level benchmark	6.5
jess	jvm98	expert shell system	2.3
jpat	Ashes	protein analysis tool	1.1
jython	DaCapo	Python interpreter	2.1
kawa	other	Scheme compiler	3.1
mpegaudio	jvm98	audio file decompressor	1.0
mtrt	jvm98	multi-threaded raytracer	10.4
pmd	DaCapo	source code analyzer	7.0
ps	DaCapo	postcript interpreter	229.3
soot	DaCapo	bytecode analyzer	33.0

Table 2. Benchmarks.

Table 2 shows the benchmark suite, consisting of 26 Java programs: SPECjbb2005[1], the 7 SPECjvm98 programs[2], the 10 DaCapo benchmarks[3], 2 Ashes benchmarks[4], and 6 other big Java programs. Column "MB" gives the minimum heap size in which the program runs without throwing an OutOfMemoryError. The rest of this paper reports heap sizes as $n\times$ this minimum heap size.

All timing numbers in this paper are relative, we avoid publishing absolute numbers to protect IBM's business interests.

To reduce the effect of noise on the results, all experiments consist of at least 9 runs (JVM process invocations), and usually several iterations (application invocations within one JVM process invocation). For each SPECjvm98 benchmark, a run contains around 10 to 20 iterations at input size 100. Each run of a DaCapo benchmark contains two or more iterations on the largest input.

4.2 Speedups

This section shows the effect of hierarchical copying on runtime for 25 Java programs. Section 4.4 studies a 26th program, SPECjbb2005, in more detail.

Benchmark	% Speedup (1-$\frac{PH}{BF}$) at heap size 1.33×	2×	4×	10×	C.I. (4×)	# GCs (10×)
db	+21.9	+22.9	+23.5	+20.5	0.6	40
javasrc	0	+3.5	0	+3.0	2.5	110
mtrt	0	0	0	+3.4	4.6	482
jbytemark	+3.3	0	0	0	1.6	1,761
javac	+2.8	+0.9	+1.6	+3.0	0.5	309
chart	0	+3.0	0	0	3.0	126
jpat	0	0	0	+2.6	0.7	14,737
banshee	0	+2.1	0	0	3.7	6
javalex	+1.0	+1.0	+1.7	+1.6	0.6	201
jython	0	+1.3	0	0	2.3	893
eclipse	0	0	+1.2	0	1.0	9
mpegaudio	0	0	0	+1.0	0.9	15
compress	0	0	0	+1.0	1.8	142
fop	0	0	0	0	1.1	391
hsqldb	0	0	0	0	1.1	239
kawa	0	0	0	0	0.0	13
soot	0	0	0	0	1.1	237
batik	0	0	0	-1.4	0.7	89
jack	0	-1.4	-0.6	0	0.4	1,440
antlr	-1.9	-1.3	-1.0	-1.1	0.9	3,070
jess	-2.8	-2.4	-1.5	0	0.7	3,558
ps	-3.0	-2.7	-2.2	-1.3	0.8	59
bloat	0	-1.7	0	-4.7	1.1	341
pmd	-1.8	0	0	-5.1	3.3	775
ipsixql	-6.0	-6.5	-8.7	-5.9	0.7	3,433

Table 3. Speedups for all benchmarks except SPECjbb2005.

The speedup columns of **Table 3** show the percentage by which parallel hierarchical copying (PH) speeds up (+) or slows down (-) run time compared to the baseline parallel breadth-first copying (BF). They are computed as 1-$\frac{PH}{BF}$, where PH and BF are the respective total run times. For example, at a heap size of 4× the minimum, parallel hierarchical copying speeds up db's run time by 23.5% compared to breadth-first. When the speedup or slowdown is too small to be statistically significant (based on Student's t-test at 95% confidence), the table shows a "0". Column "C.I." shows the confidence intervals for the 4× numbers as a percentage of the mean run time. The confidence intervals at other heap sizes are similar. Finally, Column "#GCs" shows the number of garbage collections in the runs at heap size 10×; smaller heaps cause more garbage collections.

None of the benchmarks experienced speedups at some heap sizes and slowdowns at others. The benchmarks are sorted by their maximum speedup or slowdown at any heap size. Out of these 25 programs, 13 speed up, 4 are unaffected, and 8 slow down.

[1] http://www.spec.org/jbb2005/

[2] http://www.spec.org/osg/jvm98/

[3] http://www-ali.cs.umass.edu/DaCapo/gcbm.html

[4] http://www.sable.mcgill.ca/ashes/

Section 4.4 will show that SPECjbb2005 also speeds up. While speedups vary across heap sizes, we observed no pattern.

The program with the largest slowdown is ipsixql, which maintains a software LRU cache of objects. Because the objects in the cache survive long enough to get tenured [18], but then die, ipsixql requires many collections of the old generation. The program with the largest speedup is db, which experiences similar speedups from depth-first copy order [19]. Depth-first copy order requires a mark stack, hence we do not consider it in this paper.

Parallel hierarchical copy order speeds up the majority of the benchmarks compared to breadth-first copy order, but slows some down. It may be possible to avoid the slowdowns by deciding the copy order based on runtime feedback; this is future work.

4.3 Mutator vs. collector behavior

Parallel hierarchical copying GC tries to speed up the mutator by improving locality. Section 4.2 showed that most programs speed up, but some slow down. This section explores how mutator and garbage collection contribute to the overall performance.

Benchmark	Mutator			Collector		
	Time	TLB misses		Time	TLB misses	
	$1-\frac{PH}{BF}$	BF	PH	$1-\frac{PH}{BF}$	BF	PH
db	+24.3	7.0	5.5 (-)	-37.6	0.6	0.6 (0)
javasrc	0	1.0	1.0 (0)	0	0.6	0.5 (-)
mtrt	0	2.4	2.5 (0)	-15.4	0.6	0.5 (-)
jbytemark	0	0.3	0.3 (+)	+9.4	0.6	0.6 (0)
javac	+2.0	1.6	1.5 (-)	0	0.6	0.5 (-)
chart	0	0.8	0.8 (0)	0	0.7	0.6 (0)
jpat	0	2.6	2.7 (0)	0	0.8	0.8 (0)
banshee	0	0.4	0.4 (0)	-3.3	1.0	1.0 (0)
javalex	+1.7	0.7	1.2 (+)	0	0.5	0.5 (0)
jython	0	1.5	1.5 (0)	-9.0	0.7	0.7 (-)
eclipse	+3.1	0.9	0.8 (-)	0	0.7	0.5 (-)
mpegaudio	0	0.4	0.4 (0)	-5.7	0.8	0.7 (-)
compress	0	1.2	1.1 (0)	0	1.0	1.0 (0)
fop	+1.3	1.4	1.2 (0)	0	0.5	0.4 (-)
hsqldb	0	1.2	1.1 (-)	0	0.5	0.5 (0)
kawa	+0.4	1.3	1.3 (0)	-9.6	0.6	0.5 (-)
soot	0	1.7	1.7 (0)	-3.9	0.5	0.5 (0)
batik	0	0.8	0.8 (0)	0	0.6	0.6 (0)
jack	0	1.2	1.2 (0)	-9.2	0.6	0.4 (-)
antlr	0	0.8	0.8 (0)	-6.5	0.6	0.6 (0)
jess	0	2.1	2.1 (0)	-7.2	0.5	0.4 (-)
ps	0	1.3	1.7 (+)	-25.6	0.5	0.4 (-)
bloat	0	1.2	1.1 (0)	-2.7	0.6	0.5 (-)
pmd	0	1.6	1.7 (0)	-13.5	0.6	0.5 (-)
ipsixql	-2.9	0.8	0.8 (0)	-13.2	0.5	0.4 (-)

Table 4. Mutator and collector behavior at heap size 4×.

Table 4 breaks down the results of running in 4× the minimum heap size into mutator and collector. The "Time" columns show improvement percentages of parallel hierarchical copying (PH) compared to breadth-first (BF); higher numbers are better, negative numbers indicate degradation. The "TLB misses" columns show miss rates per retired instruction, in percent (lower is better; Section 4.6 will explore TLB and other hardware characteristics in more detail). A (+) indicates that PH has a higher miss rate than BF, a (-) indicates that it has a lower miss rate, and a (0) indicates that there is no statistically significant difference. The benchmarks are ordered by the total speedup from Table 3.

When there is a measurable change, with few exceptions, the mutator speeds up and the collector slows down. Even fop and kawa, which experienced no overall speedup, experience a small mutator speedup. Usually, TLB miss rates decrease both in the mutator and in the GC. For the mutator, this explains the speedup; for the GC, this does not prevent the slowdown caused by executing more instructions to achieve hierarchical order. The large reduction in mutator TLB misses for db (from 7% to 5.5%) leads to an overall speedup despite having the largest GC slowdown (of 37.6%). Hierarchical copying only slows down collections of the young generation, but since most objects in db die young, collections of the young generation dominate GC cost.

To conclude, parallel hierarchical copying trades GC slowdown for mutator speedup. This is a reasonable tradeoff as long as GC scaling on multiprocessors is not impacted.

4.4 Scaling on multi-processor systems

This paper shows how to achieve hierarchical copy order in a parallel GC. The goal of parallel GC is to scale well in multi-processor systems by using all CPUs for collecting garbage. This is necessary to keep up with the mutator, since it uses all CPUs for allocating memory and generating garbage. This section investigates how well parallel hierarchical copying GC scales.

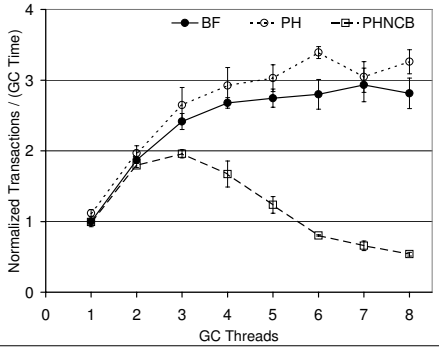
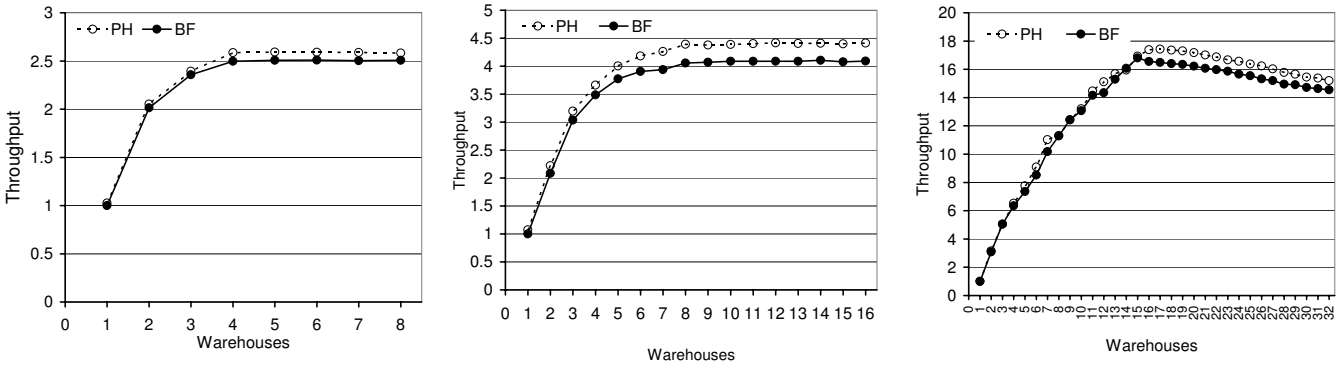


Figure 11. GC scaling for SPECjbb2005 on 4 IA32 SMT Processors running Windows.

Figure 11 shows how the collector scales for SPECjbb2005. SPECjbb2005, the SPEC Java business benchmark, models a server that uses multiple parallel mutator threads to service transactions against a database. For this experiment, the number of mutator threads is fixed at 8, and when the mutator threads are stopped for collection, the GC uses between 1 and 8 threads. The platform is an IA32 Windows system with four 1.6GHz Pentium 4 Xeon processors with hyperthreading (i.e. 8 logical CPUs), 256KB of L2 cache, 1MB of L3 cache, and 2GB of RAM. The heap size is 1GB, out of which the young generation uses 384MB.

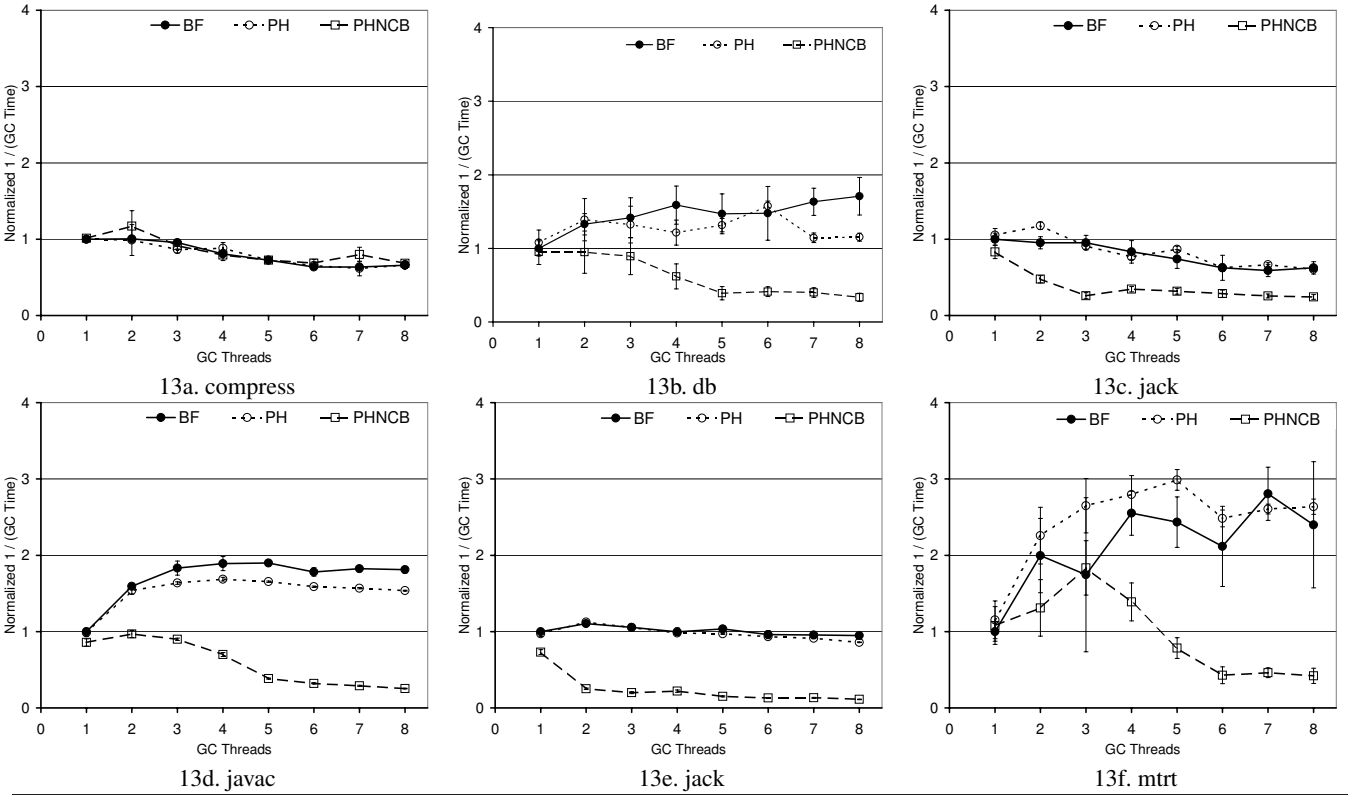
All numbers in Figure 11 are mutator transactions per GC time. Higher numbers indicate that the mutator gets more mileage out of each second spent in GC, indicating better GC scaling. There are curves for parallel breadth-first (BF) copying, parallel hierarchical (PH) copying, and PH with no cached block (PHNCB). All numbers are normalized to BF at 1 thread. The error bars show 95% confidence intervals. With 8 GC worker threads, both BF and PH run around 3 times faster than with 1 thread. Without the cached block optimization from Section 3.2, PH would not scale: it would run 46% slower with 8 threads than with only 1 thread (PHNCB).

Whereas Figure 11 shows how SPECjbb2005's GC time scales, **Figure 12** shows how its total throughput scales on three hardware platforms. SPECjbb2005 measures throughput as transactions per second, which should increase with the number of parallel mutator threads ("warehouses"). The three platforms are:



12a. 2 EM64T SMT Processors, Linux | 12b. 4 IA32 SMT Processors, Windows | 12c. 8 Power SMT Processors, AIX

Figure 12. Throughput for SPECjbb2005.



13a. compress | 13b. db | 13c. jack

13d. javac | 13e. jack | 13f. mtrt

Figure 13. GC scaling on 4 IA32 SMT Processors, Windows.

a. A 2-processor EM64T system running Linux. The machine has two 3.4GHz Pentium 4 Xeon processors with hyperthreading, with 1MB of L2 cache and 4GB of RAM. On this machine, SPECjbb2005 used a 1GB heap with a 384MB young generation.

b. The 4-processor IA32 Windows system from Figure 11.

c. An 8-processor Power system running AIX. The machine has eight 1.5GHz Power 5 processors with hyperthreading, with a total of 144MB of L3 cache and 16GB of RAM. On this machine, we ran SPECjbb2005 in a 3.75GB heap with a 2.5GB young generation.

In each of the graphs 12a-c, the x-axis shows the number of warehouses (parallel mutator threads), and the y-axis shows the throughput (transactions per second) relative to the BF throughput with 1 warehouse. Higher is better in thes graphs, because it means that more transactions complete per second.

On all three platforms, throughput increases until the number of warehouses reaches the number of logical CPUs, which is twice the number of physical CPUs due to hyperthreading. At that point, parallel hierarchical GC has a 3%, 8%, and 5% higher throughput than the baseline GC. Increasing the number of threads further does not increase the throughput, since there are no additional hardware resources to exploit. But hierarchical GC sustains its lead over the baseline GC even as threads are increased beyond the peak.

Figure 13 shows GC scaling for the SPECjvm98 benchmarks except mpegaudio (which does very little GC). The platform is the same as for Figure 11, and the heap size is 64MB. Except for

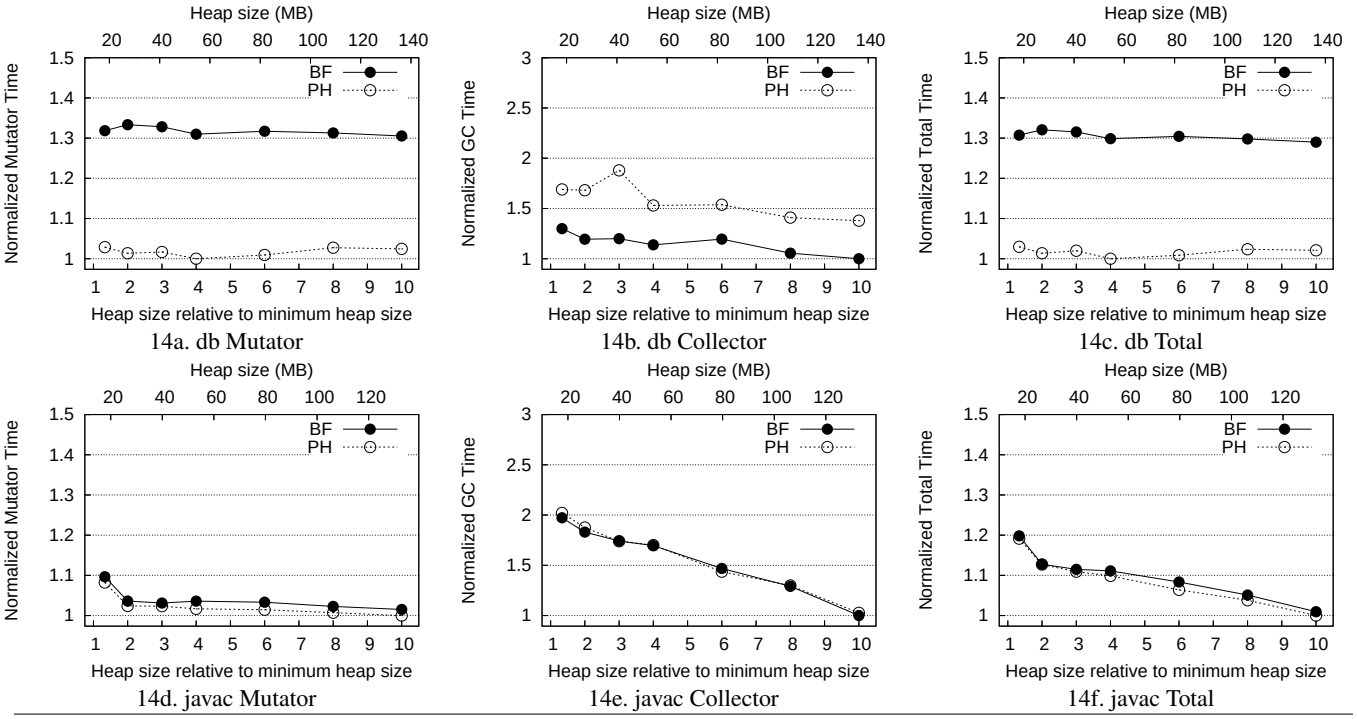


Figure 14. Time for SPECjvm98 db and javac.

mtrt, all of these programs are single-threaded. Since the amount of mutator work is constant between the different collectors, Figure 13 measures parallel GC scaling as the inverse of GC time, normalized to GC throughput for BF with 1 thead. For most SPECjvm98 benchmarks, neither PH nor BF scale well. This is in part due to their small memory usage compared to SPECjbb2005: there is not enough work to distribute on the parallel GC worker threads. As for SPECjbb2005, PH with no cached block (PHNCB) scales worse than either PH or BF.

To conclude, parallel hierarchical copying GC scales no worse with increasing load caused by parallel applications than parallel breadth-first copying GC. A single-threaded GC, on the other hand, would have a hard time keeping up with the memory demands of several parallel mutators.

4.5 Time-space tradeoffs

In a small heap, GC has to run more often, because the application exhausts memory more quickly. This increases the cumulative cost of GC. On the other hand, in a small heap, objects are closer together, which should intuitively improve locality. This section investigates how these competing influences play out.

Figure 14 shows the run times of two representative benchmarks, SPECjvm98 db and javac, at 6 different heap sizes from 1.33× to 10× (occupancy 75% to 10%). The x-axis shows the heap size; each graph carries labels for absolute heap size at the top and labels for relative heap size at the bottom. The y-axis shows run time relative to the best data point in the graph. In these graphs, lower is better, since it indicates faster run time. There are three graphs for each benchmark, one each for total time, mutator time, and GC time. While the y-axis for total and mutator time goes to 1.5, the y-axis for GC time goes to 3.

Figures 14a+d show that parallel hierarchical copying (PH) speeds up the mutator for both db and javac. Figures 14b+e show that, as expected, total GC cost is higher in smaller heaps. But this effect is more significant for javac than for db, because javac has a higher nursery survival rate [18]. That is also the reason why PH slows down the collector for db, while causing no significant change in collector time for javac. The overall behavior of db is dominated by the mutator speedup caused by PH (Figure 14c), whereas the overall behavior of javac is dominated by the decrease of GC cost in larger heaps (Figure 14f).

This confirms the conclusions from Section 4.2: parallel hierarchical GC performs well in both small and large heaps.

4.6 Cache and TLB misses

The goal of hierarchical copying is to reduce cache and TLB misses by colocating objects on the same cache line or page. This section uses hardware performance counters to measure the impact of hierarchical copying on misses at different levels of the memory subsystem.

Pentium processors expose hardware performance counters through machine specific registers (MSRs), and many Linux distributions provide a character device, `/dev/cpu/*/msr`, to access them. Doing `modprobe msr` ensures the presence of this device; for experiments in user mode, the files must be readable and writeable for users. The JVM sets up the registers for collecting the desired hardware events at the beginning of the run, and reads them at the beginning and end of GC, accumulating them separately for the mutator and the GC.

Figure 15 shows the results. The x-axis shows the heap size; each graph carries labels for absolute heap size at the top and labels for relative heap size at the bottom. The y-axis shows the hardware metric; each graph carries labels for relative miss rate at the left and labels for absolute miss rate at the right. In these graphs, lower is better, since it indicates fewer misses. The denominator of all ratios is retired instructions. See Table 4 for statistical confidence on the TLB miss rates; there are some variations due to noise. The "Bus cycles" event measures for how many cycles the bus between the L2 cache and main memory was active. This indicates L2 misses, for which Pentium 4 does not provide a reliable direct counter. Note

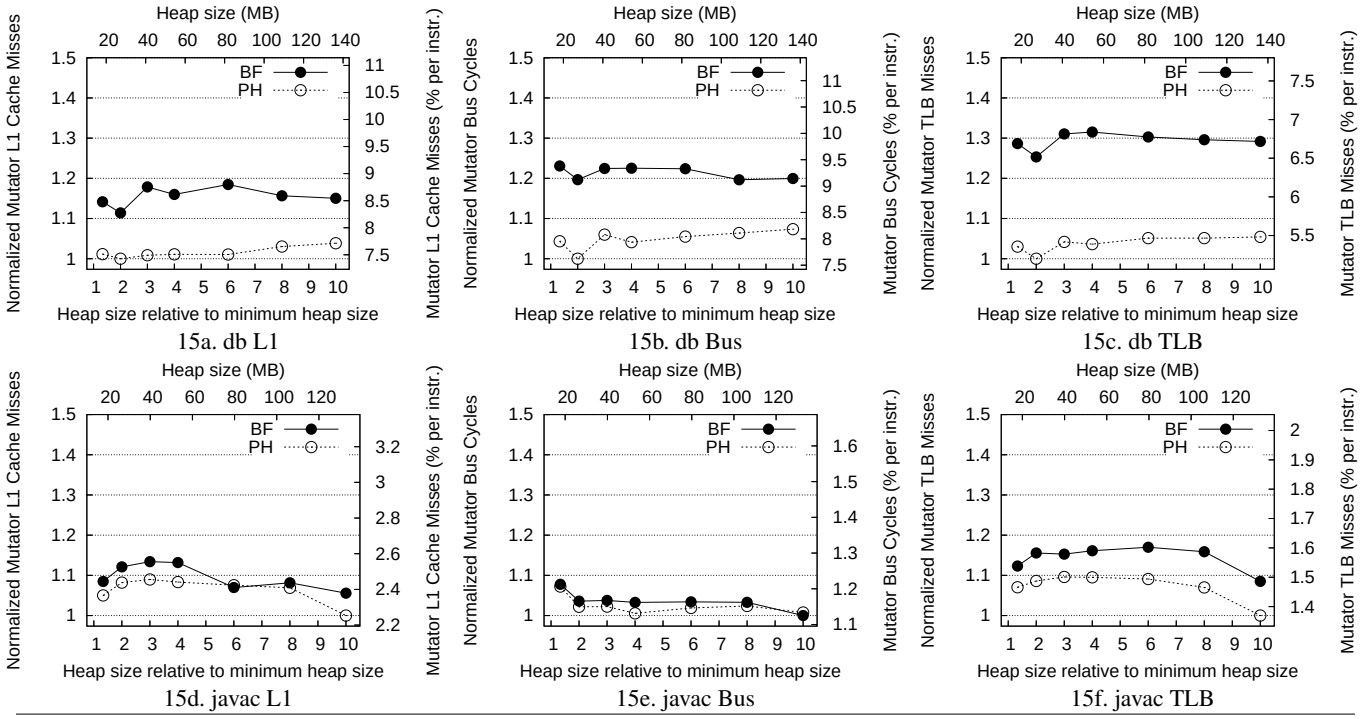


Figure 15. Cache and TLB misses for SPECjvm98 db and javac.

that bus clock speeds are usually an order of magnitude slower than processor clock speeds.

Parallel hierarchical copying reduces mutator misses on all measured levels of the memory subsystem: L1 data cache, combined L2 cache, and TLB. It reduces misses for both db and javac, at all heap sizes. As expected, the reduction in TLB misses is the most significant, because the hierarchical GC uses 4KB blocks as the decomposition unit, which coincide with the page size. With BF, db has high L1 and TLB miss rates, and PH reduces the miss rates significantly. That explains the large speedup that Sections 4.2 and 4.5 report for db.

To conclude, parallel hierarchical copying GC reduces TLB misses most, while also reducing L1 and L2 cache misses significantly. These reduced miss rates translate into reduced run time.

4.7 Pointer distances

Section 4.6 already demonstrated that hierarchical copying reduces cache and TLB misses. This section validates that it achieves that by colocating objects on the same cache line or page.

For this experiment, the GC records the distance between the address of a pointer and the address of the object it points to just after a copying or forwarding operation. Pointers with an absolute distance under 64B are classified as "Line", and pointers with an absolute distance between 64B and 4KB are classified as "Page". The numbers only consider pointers from objects in the young generation to other objects in the young generation, and from newly tenured objects in the old generation to other newly tenured objects in the old generation. Among other things, this disregards pointers between young and old objects; those have longer distances, but are rare, and hierarchical copying can not colocate them on the same page.

Table 5 shows pointer distances. For example, db with breadth-first copying yields 9.4% pointers that are longer than 64 bytes but under 4KB, whereas parallel hierarchical copying improves that to

Benchmark	BF		PH	
	Line	Page	Line	Page
db	0.0	9.4	23.6	65.1
SPECjbb2005	0.0	0.5	6.8	72.4
javasrc	0.3	20.8	17.6	32.2
mtrt	2.2	28.5	24.1	46.7
jbytemark	0.1	4.0	11.2	11.6
javac	1.2	33.9	33.1	29.1
chart	0.1	4.9	58.0	5.6
jpat	0.1	7.2	46.3	5.3
banshee	0.6	28.1	15.9	46.8
javalex	1.6	19.6	21.9	17.2
jython	0.2	11.1	4.5	35.6
eclipse	1.9	25.9	28.6	37.3
mpegaudio	0.9	33.0	16.7	50.8
compress	5.4	33.2	23.2	40.1
fop	0.2	32.8	11.9	52.0
hsqldb	0.1	28.9	20.3	64.9
kawa	3.0	28.0	23.4	32.1
soot	6.8	30.1	21.5	38.1
batik	1.4	32.9	20.5	45.5
jack	0.4	35.5	26.4	49.4
antlr	2.0	32.8	20.1	44.4
jess	0.3	6.7	8.0	6.5
ps	0.1	24.1	32.2	33.9
bloat	4.0	24.5	34.5	26.5
pmd	1.7	28.5	27.4	29.0
ipsixql	1.0	20.2	32.6	21.1

Table 5. Pointer distances.

65.1%. Except for SPECjbb2005, all runs used heaps of 4× the minimum size.

These numbers show that parallel hierarchical copying succeeds in colocating objects on the same 4KB page for the majority of the pointers. This explains the reduction in TLB misses observed in Table 4. Also, parallel hierarchical copying colocates objects on the same 64 byte cache line much more often than the baseline garbage collector. This explains the noticeable reduction in L1 and L2 cache misses observed in Section 4.6.

While hierarchical copying is tremendously successful at improving spatial locality of connected objects, wall-clock numbers from a real system (Table 3) paint a more sober picture. This discrepancy underlines three points: (i) Hierarchical copying trades GC slowdown for mutator speedup. The result of this tradeoff is determined by the concrete benchmark, GC implementation, and platform. (ii) Hierachical copying aims at decreasing TLB and cache miss rates. When the application working set is small compared to the memory hierarchy of the machine, miss rates are already so low that decreasing them further helps little. (iii) Hierarchical copying optimizes for the "hierarchical hypothesis" that connectivity predicts affinity. In other words, it assumes that objects with connectivity (parents or siblings in the object graph) also have affinity (the application accesses them together). Not all applications satisfy the hierarchical hypothesis.

5. Related work

Section 5.1 discusses related work on using GC to improve mutator locality, Section 5.2 discusses related work on parallel GC, and Section 5.3 discusses other locality optimizations.

5.1 Locality optimization with copying garbage collection

This section reviews, in chronological order, papers on how GC can automatically improve mutator locality.

Section 2.2 describes Moon's hierarchical copying GC [27]. Moon's GC is incremental, and relies on a read barrier that copies objects before the mutator accesses them, so the mutator can never see objects in from-space, only in to-space. Special-purpose Lisp hardware makes this read barrier efficient. Courts used Moon's GC to improve locality further by letting the mutator drive the copy order [11]. To do this, Courts first enables the read barrier, then allows the mutator to trigger copies for a while, and only then starts scanning objects from the GC.

Section 2.3 describes how Wilson, Lam, and Moher extend Moon's hierarchical copying GC to avoid rescanning black objects [31]. That paper also advocates that even when the runtime system stores roots in a hash table, GC should scan roots in declaration order. If GC were to scan roots in order of their hash keys, object layout would get randomized, leading to poor locality.

Chilimbi and Larus [10] use a software read barrier to exhaustively profile memory accesses in Cecil. They construct an affinity graph, where nodes are objects, and edge weights denote co-access frequency. At beginning of GC, Chilimbi and Larus greedily precopy the affinity graph (in order of decreasing affinity).

Kistler and Franz reorder fields inside an object to group the hottest fields on the same cache line [23]. They perform this as an online feedback-directed optimization in an Oberon system [24]. To support field reordering, the compiler either has to generate code that accesses object fields via a level of indirection, or must invalidate compiled code when field offsets change. Our algorithm, as well as all the other algorithms in this section, use online object reordering rather than field reordering.

Shuf et al. copy objects close together if they were already close together before the garbage collection, and vice versa [29]. Thus, they preserve the locality of the allocator. Shuf et al. implemented their algorithm in Jikes RVM, but disabled parallel GC, so their algorithm is sequential. The same paper also presents a technique for improving locality at object allocation time (see Section 5.3).

Huang et al.'s GC scans the hottest pointer fields of an object first, based on a low-overhead field heat profile [19]. Scanning the hottest fields first increases the likelihood that the hot successor objects are copied close to the scanned object. Besides scanning hot fields first, Huang et al. also mention trying "partial depth-first using the first two children (a hierarchical order)". This partial depth-first order copies the first two children of each node before traversing the first child depth-first. That is hierarchical in the sense that it increases the likelihood of colocating heap objects with their parents, siblings, or children, even though it does not necessarily achieve the copy order in Figure 3. Huang et al.'s results show that partial depth-first order is not very effective. Our results indicate that truly hierarchical order helps more.

Our GC algorithm is parallel, enabling it to keep up with the memory demands of parallel mutator threads. Except for [19], all algorithms discussed in this section use only a single GC worker thread. Our technique requires no profiler, making it simple to engineer and avoiding profiling overhead. Except for [27, 29, 31], all algorithms in this section require online profilers.

5.2 Parallel garbage collection

This section reviews, in chronological order, papers on parallel GC for shared memory multiprocessors (see Figure 7), with special emphasis on load balancing.

Section 2.4 describes Halstead's algorithm, the first parallel GC [16]. Section 2.5 describes Imai and Tick's algorithm, the first parallel GC with load balancing [20]. Imai and Tick maintain a shared pool of fixed-size blocks of to-space. Each GC worker thread has a private copy block and a private scan block.

Endo, Taura, and Yonezawa introduce work stealing for parallel GC [13]. Each GC worker thread provides a stealable queue of pointers to gray objects. When a thread runs out of work, it attempts to steal half of the stealable queue of one of its peers. Endo, Taura, and Yonezawa achieve impressive scaling results: they report GC speedups of 6.2× at 8 processors, 10.9× at 16 processors, 17.8× at 32 processors, and 28.3× at 64 processors.

Flood et al. present a mechanism for handling overflow in the meta-data required to maintain work-stealing queues [14].

Cheng and Blelloch present the first copying GC that is parallel and incremental as well as concurrent, thus achieving both high throughput and high responsiveness [7]. Cheng and Blelloch balance work using a shared pool, but unlike Imai and Tick, there is no fixed work packet size; instead, variable sized chunks of work are copied into and out of a shared array.

Attanasio et al. implemented and compared a variety of different parallel GC algorithms, both copying and non-copying, generational and non-generational [2]. They balance work with a shared pool of fixed-size work packets. Attanasio et al.'s copying GC is not based on Cheney's algorithm; thus, a work packet is just an array of pointers to gray objects.

Ossia et al. present a non-copying GC that is parallel and incremental as well as concurrent [28]. They use a similar approach to load balancing as Attanasio et al. [2], but also discuss issues with weak memory ordering.

A major distinction between the algorithms in this section is whether they partition work into fixed-size [2, 20, 28] or variable-size [7, 13, 14] units. Fixed-size work units tend to get used with a shared pool, whereas variable-size work units tend to get used with work stealing, though there are exceptions [7]. One advantage of variable-sized work units is that the parallel GC can adapt more flexibly when there is little work but many processors.

Unlike the algorithms discussed in this section, a main design goal of our algorithm is to improve mutator locality.

5.3 Other automatic software locality optimizations

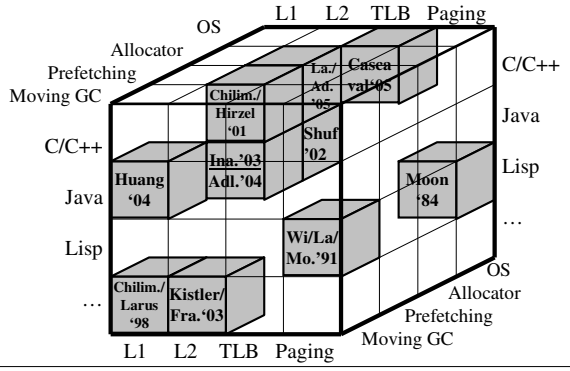


Figure 16. Automatic mutator locality optimizations.

This section reviews other software techniques for improving locality; for hardware techniques, see VanderWiel and Lilja's survey [30]. **Figure 16** shows optimizations that are fully automatic, requiring no user intervention such as performing a training run or modifying application code. Automatic optimizations include improving locality during garbage collection [10, 11, 19, 23, 24, 31], automatically injecting prefetching code into the mutator [1, 9, 21], using locality-aware memory allocation [25, 29], and using operating system support [5, 27]. Automatic locality optimizations have been evaluated for C/C++, Java, Lisp, and other programming languages such as Oberon or Cecil. We have taken an educated guess at the memory hierarchy level targetted by each optimization, but this is debatable, and not intended to be authoritative. Indeed, those papers that offer experimental results for this usually find improvements on multiple levels.

Except for [27, 31], all automatic mutator locality optimizations in Figure 16 are complicated to engineer and rely on profiling. Our technique, on the other hand, is simple to engineer and needs no profiling. Except for [1, 21], none of the optimizations have been implemented or evaluated in a product JVM. We implemented our technique in a product JVM, and compare it against the highly tuned GC it ships with. Many of these techniques may be complementary with our algorithm. Our algorithm occupies the coordinates Java/MovingGC/TLB in Figure 16, though it also helps the L1/L2 cache.

Some techniques are almost automatic, except for the fact that they require an offline training run [4, 26, 32]. Some techniques improve locality in the GC rather than the mutator, which is important when memory is so constrained that GC dominates execution time [8, 17]. Some techniques require more heavy user intervention; for example, users can rewrite their application with cache-oblivious algorithms [15].

6. Conclusions

This paper introduces a new algorithm: parallel hierarchical copying GC. It combines parallel GC, which is important for scaling on multiprocessor machines, with hierarchical decomposition, which is important for mutator locality. The key idea behind this new algorithm is to use blocks both as the work unit for parallelism and as the decomposition unit for hierarchical copying. This is facilitated by the fact that in both cases, each block has its own scan pointer. It turns out that the logic for hierarchical copying is non-trivial, but simple to engineer once all cases have been figured out. The technique works in the real world, and requires no profiler.

While the new algorithm is the first contribution of this paper, the second contribution is a thorough evaluation of its properties. The single-threaded variant of hierarchical decomposition has been known for decades, yet has not been adopted in product virtual machines, since its concrete benefits have remained unclear. This paper offers an evaluation on 26 Java programs, executing on a product JVM on stock hardware. The results show that hierarchical copying colocates pointers and their targets on the same page in the majority of the cases, that it reduces TLB misses dramatically, and that it also reduces L1 and L2 cache misses. For the majority of our benchmarks, this translates into an overall run time improvement.

Acknowledgments

We thank Perry Cheng, Michael Hind, Ryan Sciampacone, Xipeng Shen, and the anonymous reviewers for their feedback.

References

[1] Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, and Sreenivas Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Programming Language Design and Implementation (PLDI)*, 2004.

[2] Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith. A comparative evaluation of parallel garbage collector implementations. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, August 2001.

[3] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM (CACM)*, 21(4), 1978.

[4] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[5] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimizations. In *Parallel Architectures and Compilation Techniques (PACT)*, 2005.

[6] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM (CACM)*, 13(11), 1970.

[7] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[8] Chen-Yong Cher, Antony L. Hosking, and T. N. Vijaykumar. Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[9] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Programming Language Design and Implementation (PLDI)*, 2002.

[10] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *International Symposium on Memory Management (ISMM)*, 1998.

[11] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM (CACM)*, 31(9), 1988.

[12] E. W. Dijkstra. Solution to a problem in concurrent programming control. *Communications of the ACM (CACM)*, 8(9), 1965.

[13] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *High Performance Computing and Networking (SC)*, 1997.

[14] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Java Virtual Machine Research and Technology Symposium (JVM)*, April 2001.

[15] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science (FOCS)*, 1999.

[16] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 7(4), 1985.

[17] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Programming Language Design and Implementation (PLDI)*, 2005.

[18] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *International Symposium on Memory Management (ISMM)*, 2002.

[19] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

[20] Akira Imai and Evan Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(9), 1993.

[21] Tatsushi Inagaki, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Stride prefetching by dynamically inspecting objects. In *Programming Language Design and Implementation (PLDI)*, 2003.

[22] Richard Jones and Rafael Lins. *Garbage collection: Algorithms for automatic dynamic memory management*. John Wiley & Son Ltd., 1996.

[23] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *Transactions on Programming Languages and Systems (TOPLAS)*, 22(3), 2000.

[24] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *Transactions on Programming Languages and Systems (TOPLAS)*, 25(4), 2003.

[25] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout on the heap. In *Programming Language Design and Implementation (PLDI)*, 2005.

[26] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[27] David A. Moon. Garbage collection in a large Lisp system. In *LISP and Functional Programming (LFP)*, 1984.

[28] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *Programming Language Design and Implementation (PLDI)*, 2002.

[29] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[30] Steven P. VanderWiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2), 2000.

[31] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in a garbage-collected system. In *Programming Language Design and Implementation (PLDI)*, 1991.

[32] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Programming Language Design and Implementation (PLDI)*, 2002.