Myths and Realities: The Performance Impact of Garbage Collection[*]

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

Perry Cheng
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY, 10598, USA
perryche@us.ibm.com

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

ABSTRACT

This paper explores and quantifies garbage collection behavior for three whole heap collectors and generational counterparts: *copying semi-space*, *mark-sweep*, and *reference counting*, the canonical algorithms from which essentially all other collection algorithms are derived. Efficient implementations in MMTk, a Java memory management toolkit, in IBM's Jikes RVM share all common mechanisms to provide a clean experimental platform. Instrumentation separates collector and program behavior, and performance counters measure timing and memory behavior on three architectures.

Our experimental design reveals key algorithmic features and how they match program characteristics to explain the direct and indirect costs of garbage collection as a function of heap size on the SPEC JVM benchmarks. For example, we find that the contiguous allocation of copying collectors attains significant locality benefits over free-list allocators. The reduced collection costs of the generational algorithms together with the locality benefit of contiguous allocation motivates a copying *nursery* for newly allocated objects. These benefits dominate the overheads of generational collectors compared with non-generational and no collection, disputing the myth that "no garbage collection is good garbage collection." Performance is less sensitive to the mature space collection algorithm in our benchmarks. However the locality and pointer mutation characteristics for a given program occasionally prefer copying or mark-sweep. This study is unique in its breadth of garbage collection algorithms and its depth of analysis.

Categories and Subject Descriptors

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

General Terms

Design, Performance, Algorithms

Keywords

Java, Mark-Sweep, Semi-Space, Reference Counting, Generational

[*]This work is supported by the following grants: ARC DP0452011, NSF ITR CCR-0085792, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, and IBM. Any opinions, findings and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

1. Introduction

Programmers are increasingly choosing object-oriented languages such as Java with automatic memory management (*garbage collection*) because of their software engineering benefits. Although researchers have studied garbage collection for a long time [3, 21, 23, 29, 34, 41], few detailed performance studies exist. No previous study compares the effects of garbage collection algorithms on instruction throughput and locality in the light of modern processor technology trends to explain how garbage collection algorithms and programs can combine to yield good performance.

This work studies in detail the three canonical garbage collection algorithms: *semi-space*, *mark-sweep*, and *reference counting*, and three generational counterparts. These collectors encompass the key mechanisms and policies from which essentially all garbage collectors are composed. Our findings therefore have application beyond these algorithms. We conduct our study in the Java memory management toolkit (MMTk) [12] in IBM's Jikes RVM [2, 1]. The collectors are efficient, share all common mechanisms and policies, and provide a clean and meaningful experimental platform [12].

The results use a wide range of heap sizes on SPEC JVM Benchmarks to reveal the inherent space-time trade-offs of collector algorithms. For fair comparisons, each experiment fixes the heap size, and triggers collection when the program exhausts available memory. We use three architectures (Athlon, Pentium 4, PowerPC) and find the same trends on all three. Each experiment divides total program performance into *mutator* (application code) and collection phases. The mutator phase includes some memory management activity, such as the allocation sequence and for the generational collectors, a *write barrier*. Hardware performance counters measure the L1, L2, and TLB misses for collector and mutator phases. The experiments reveal the direct cost of garbage collection and its indirect effects on mutator performance and locality.

Our first set of experiments confirm the widely held, but unexamined hypothesis, that the locality benefits of contiguous allocation improves the locality of the mutator. For the whole heap collectors in small heaps, the more space efficient *free-list* mark-sweep collector performs best because collection frequency dominates the locality benefit of contiguous allocation. As heap size increases, the mutator locality advantage of contiguous allocation with copying collection outweighs the space efficiency of mark-sweep. Contiguous allocation provides fewer misses at all levels of the cache hierarchy (L1, L2 and TLB). These results are counter to the myth that collection frequency is always the first order effect that determines total program performance. Further experiments reveal that most of these locality benefits are for the young objects which motivates a contiguous allocation for them in generational collectors.

The generational collectors divide newly allocated *nursery* objects from mature objects that survive one or more collections, and collect the nursery independently and more frequently than the mature space [34, 41]. They work well when the rate of death among the young objects is high. In order to collect the nursery independently, the generational collectors use a *write barrier* which records any pointer into the nursery from the mature objects. During a nursery collection, the collector assumes the referents of these pointers are live to avoid scanning the entire mature generation. To implement the write barrier, the compiler generates a sequence of code for *every* pointer store that at runtime records only those pointers from the mature space into the nursery. The write barrier thus induces direct mutator overhead between programs that use whole heap versus generational collection.

Our experiments show that the generational collectors provide better performance than the whole heap collectors in virtually all circumstances. They significantly reduce collection time itself, and their contiguous nursery allocation has a positive impact on locality. We carefully measure the impact of the write barrier on the mutator and find that their mutator cost is usually very low (often 2% or less), and even when high (14%), the cost is outweighed by the improvements in collection time.

Comparing the generational collectors against each other, performance differences are typically small. Two factors contribute to this result. First, allocation order provides good spatial locality for young objects even if the program briefly uses and discards them. Second, the majority of reads are actually to the mature objects, but caching usually achieves good temporal locality for these objects regardless of mature space policy. Some object demographics do however have a preference. For instance, generational collection with a copying mature space works best when the mature space references are dispersed and frequent. The mark-sweep mature space performs best, sometimes significantly, in small heaps when its space efficiency reduces collector invocations.

The next section compares our study to previous collector performance analysis studies, none of which consider this variety of collectors in an apples-to-apples setting, nor do any include a similar depth of analysis or vary the architecture. We then overview the collectors, a number of key implementation details, and the experimental setting. The results section studies the three base algorithms separating allocation and collection costs (as much as possible), compares whole heap algorithms and their generational counterparts, and examines the cost of the generational write barrier. We examine the impact of nursery size on performance and debunk the myth that the nursery size should be tied to the L2 cache size. We also examine mature space behaviors using a fixed-size nursery to hold the mature space work load constant. We perform every experiment on the nine benchmarks and three architectures, but select representative results for brevity and clarity.

2. Related Work

To our knowledge, few studies quantitatively compare uniprocessor garbage collection algorithms [5, 13, 26, 27, 39], and these studies evaluate various copying and generational collectors. Our results on copying collectors are similar to theirs, but they do not compare with free-list mark-sweep or reference counting collectors, nor explore memory system consequences.

Attanasio et al. [5] evaluate parallel collectors on SPECjbb, focusing on the effect of parallelism on throughput and heap size when running on 8 processors. They concluded that mark-sweep and generational mark-sweep with a fixed-size nursery (16 MB or 64 MB) are equal and the best among all the collectors. Our data shows that the generational are superior to whole heap collectors especially with a variable-size nursery.

A few recent studies explore heap size effects on performance [13, 17, 31, 39], and as we show here, garbage collectors are very sensitive to heap size, and in particular to tight heaps. Diwan et al. [24, 40], Hicks et al. [27], and others [14, 28] measure detailed, specific mechanism costs, and architecture influences [24], but do not consider a variety of collection algorithms. Many researchers have evaluated a range of memory allocators for C/C++ programs [9, 10, 11, 16, 20, 42], but this work does not include copying collectors since C/C++ programs may store pointers arbitrarily.

Java performance analysis work either disabled garbage collection [22, 36] which introduces unnecessary memory fragmentation, or hold it constant [31]. Kim and Hsu measure similar details as we do, with simulation of IBM JDK 1.1.6, a Java JIT, using whole heap mark-sweep algorithm with occasional compaction. Our work thus stands out as the first thorough evaluation of a variety of different garbage collection algorithms, how they compare and affect performance using execution measurements and performance counters. The comprehensiveness of our approach reveals new insights, such as the most space efficient collection algorithms and the distinct locality patterns of young and old objects, suggests mechanisms for matching algorithms to object demographics, and reveals performance trade-offs each strategy makes.

We evaluate the reuse, modularity, portability, and performance of MMTk in a separate publication [12]. In that work we do not explore generational collectors, nor measure and explain performance differences between collectors. However, we do demonstrate that MMTk combines modularity and reuse with high performance, and we rely on that finding here. For example, collectors that share functionality, such as root processing, copying, tracing, allocation, or collection mechanisms, use the exact same implementation in MMTk. In addition, the allocation and collector mechanisms perform as well as hand tuned monolithic counterparts written in Java or C. The experiments in this paper thus offer true policy comparisons in an efficient setting.

3. Background

This section presents the garbage collection terminology, algorithms, and features that this paper compares and explores. It first presents the algorithms, and then enumerates a few key implementation details. For a thorough treatment of algorithms, see Jones and Lins [29], and Blackburn et al. for additional implementation details [12].

In MMTk, a *policy* pairs one allocation mechanism with one collection mechanism. *Whole heap* collectors use a single policy. *Generational* collectors divide the heap into age cohorts, and use one or more policies [3, 41]. For generational and incremental algorithms, such as reference counting, a *write barrier* remembers pointers. For every pointer store, the compiler inserts write-barrier code. At execution time, this code conditionally records pointers depending on the collector policy. Following the literature, the execution time consists of the *mutator* (the program itself) and periodic *garbage collection*. Some memory management activities, such as object allocation and the write barrier, mix in with the mutator. Collection can run concurrently with mutation, but this work uses a separate collection phase. MMTk implements the following standard allocation and collection mechanisms.

A Contiguous Allocator appends new objects to the end of a contiguous space by incrementing a *bump pointer* by the size of the new object.

A Free-List Allocator organizes memory into *k* size-segregated *free-lists*. Each free list is unique to a size class and is composed of blocks of contiguous memory. It allocates an object into a free cell in the smallest size class that accommodates the object.

A Tracing Collector identifies live objects by computing a transitive closure from the *roots* (stacks, registers, and class variables/statics) and from any remembered pointers. It reclaims space by copying live data out of the space, or by freeing untraced objects.

A Reference Counting Collector counts the number of incoming references for each object, and reclaims objects with no references.

3.1 Collectors

All modern collectors build on these mechanisms. This paper examines the following whole heap collectors, and a generational counterpart for each. The generational collectors use a copying *nursery* for newly allocated objects.

SemiSpace: The semi-space algorithm uses two equal sized copy spaces. It contiguously allocates into one, and reserves the other space for copying into since in the worst case all objects could survive. When full, it traces and copies live objects into the other space, and then swaps them. Collection time is proportional to the number of survivors. Its throughput performance suffers because it reserves half of the space for copying and it repeatedly copies objects that survive for a long time, and its responsiveness suffers because it collects the entire heap every time.

Implementation Details: Copying tracing implements the transitive closure as follows. It enqueues the locations of all root references, and repeatedly takes a reference from the locations queue. If the referent object is uncopied, it copies the object, leaves a forwarding address in the old object, enqueues the copied object on a gray object queue, and adjusts the reference to point to the copied object. If it previously copied the referent object, it instead adjusts the reference with the forwarding address. When the locations queue is empty, the collector scans each object on the gray object queue. Scanning places the locations of the pointer fields of these objects on the locations queue. When the gray object queue is empty, it processes the locations queue again, and so on. It terminates when both queues are empty. These experiments use a depth-first order, because our experiments show it performs better than the more standard breadth-first order [18]. MMTk supports other orderings. SemiSpace has no write barrier.

MarkSweep: Mark-sweep uses a free-list allocator and a tracing collector. When the heap is full, it triggers a collection. The collection traces and marks the live objects using bit maps, and lazily finds free slots during allocation. Tracing is thus proportional to the number of live objects, and reclamation is incremental and proportional to allocation. The tracing for MarkSweep is exactly the same as SemiSpace, except that instead of copying the object, it marks a bit in a live object bit map. Since MarkSweep is a whole heap collector, its maximum pause time is poor and its performance suffers from repeatedly tracing objects that survive many collections.

Implementation Details: The free-list uses segregated-fits with a range of size classes similar to the Lea allocator [32]. MMTk uses 51 size classes that attain a worst case internal fragmentation of 1/8 for objects less than 255 bytes. The size classes are 4 bytes apart from 8 to 63, 8 bytes apart from 64 to 127, 16 bytes apart from 128 to 255, 32 bytes apart from 256 to 511, 256 bytes apart from 512 to 2047, and 1024 bytes apart from 2048 to 8192. Small, word-aligned objects get an exact fit—in practice, these are the vast majority of all objects. All objects 8KB or larger get their own block (see Section 3.2.3). MarkSweep has no write barrier. The collector keeps the blocks of a size class in a circular list ordered by allocation time. It allocates the first free element in the first block. Finding the right fit is about 10% slower [12] than bump-pointer allocation. The free-list stores the bit vector for each block together with the block. Since block sizes vary from 256 bytes to 8K bytes, this organization may be a source of some conflict misses, but we leave that investigation for future work.

RefCount: The deferred reference-counting collector uses a free-list allocator. During mutation, the write barrier ignores stores to roots and logs mutated objects. It then periodically updates reference counts for root referents and generates reference count increments and decrements using the logged objects. It then deletes objects with a zero reference count and recursively applies decrements. It uses trial deletion to detect cycles [7]. Collection time is proportional to the number of dead objects, but the mutator load is significantly higher than other collectors since it logs every mutated heap object.

Implementation Details: RefCount uses *object logging* with *coalescing* [33]. RefCount thus records objects only the first time the program modifies it, and buffers decrements for all its referent objects. At collection time, it (1) generates increments for all root and modified object referents, thus *coalescing* intermediate updates, (2) introduces *temporary* [7] increments for deferred objects (e.g., roots), and (3) deletes objects with a zero count. When a reference count goes to zero, it puts the object back on the free-list by setting a bit and it decrements all its referents. On the next collection, it includes a decrement for all temporary increments from the previous collection.

GenCopy: The classic copying generational collector [3] allocates into a young (*nursery*) space. The write barrier records pointers from mature to nursery objects. It collects when the nursery is full, and promotes survivors into a mature semi-space. When the mature space is exhausted, it collects the entire heap. When the program follows the weak generational hypothesis [34, 41], i.e., many young objects die quickly and old objects survive at a higher rate than young, GenCopy attains better performance than SemiSpace. GenCopy improves over SemiSpace in this case because it repeatedly collects the nursery which yields a lot of free space, it compacts the survivors which can improve mutator locality, and incurs the collection cost of the mature objects infrequently. It also has better average pause times than SemiSpace, since the nursery is typically smaller than the entire heap.

GenMS: This hybrid generational collector uses a copying nursery and the MarkSweep policy for the mature generation. It allocates using a bump pointer and when the nursery fills up, triggers a nursery collection. The write barrier, nursery collection, nursery allocation policies, and mechanisms are identical to those for GenCopy. The test for an exhausted heap must accommodate space for copying an entire nursery full of survivors into the MarkSweep space. GenMS should be better than MarkSweep for programs that follow the weak generational hypothesis. In comparison with GenCopy, GenMS can use memory more efficiently, since GenCopy reserves half the heap for copying space. However, both MarkSweep and GenMS can fragment the free space when objects are distributed among size classes.

Infrequent collections can contribute to spreading consecutively allocated (or promoted) objects out in memory. Both sources of fragmentation can reduce locality. Mark-compact collectors can reduce this fragmentation, but need one or two additional passes over the live and dead objects [19].

GenRC This hybrid generational collector uses a copying nursery and RefCount for the mature generation [15]. It ignores mutations to nursery objects by marking them as logged, and logs the addresses of all mutated mature objects. When the nursery fills, it promotes nursery survivors into the reference counting space. As part of the promotion of nursery objects, it generates reference counts for them and their referents. At the end of the nursery collection, GenRC computes reference counts and deletes dead objects, as in RefCount. Since GenRC ignores the frequent mutations of the nursery objects, it performs much better than RefCount. Collection time is proportional to the nursery size and the number of dead objects in the RefCount space. With a small nursery and other collection triggers, pause times are very low [15]. RefCount and GenRC are subject to the same free-list fragmentation issues as MarkSweep and GenMS. However, since GenRC collects the mature space on every collection, it is likely to maintain a smaller memory footprint.

3.2 Implementation Details

This section adds a few more implementation details about shared mechanisms including the nursery size policies, inlining write barriers and allocation, reference counting header, the large object space, and the boot image.

3.2.1 Nursery size policies

By default, the generational collectors implement a *variable* nursery [3] whose initial size is half of the heap, the other half is reserved for copying. Each nursery collection reduces the nursery by the size of the survivors. When the available space for the nursery is too small (256KB by default), it triggers a mature space collection. MMTk also provides a *bounded* nursery which takes a command line parameter as the initial nursery size, collects after the nursery is full, and resizes the nursery below the bound only when the mature space cannot accommodate a nursery of survivors. It shrinks using the above variable nursery policy with the same lower bound. The *fixed* nursery never reduces the size of the nursery, and thus triggers a whole heap collection sooner than the bounded nursery of the same size. The bounded nursery triggers more collections than the variable nursery which uses space more efficiently, but when the variable nursery is large, pause time suffers.

3.2.2 Write-barrier and allocation inlining

For the generational collectors, MMTk inlines the write-barrier fast path which filters stores to nursery objects and thus does not record most pointer updates, i.e., ignores between 93.7% to 99.9% of pointer stores. The slow path makes the appropriate entries in the remembered set. Since the write barrier for RefCount is unconditional, it is fully inlined but forces the slow path object remembering mechanism out-of-line to minimize code bloat and compiler overhead [14]. SemiSpace and MarkSweep have no write barrier.

MMTk inlines the fast path for the allocation sequence. For the copying and generational allocators, the inlined sequence consists of incrementing a bump pointer and testing it against a limit pointer. If the test fails (failure rate is typically 0.1%), the allocation sequence calls an out-of-line routine to acquire another block of memory, which may trigger a collection.

For the MarkSweep and RefCount free-list allocators, the inline allocation sequence consists of establishing the size class for the allocation (for non-array types, the compiler statically evaluates the size), and removing a free cell from the appropriate free-list, if such a cell is available. If there is no available free cell, the allocation path calls out-of-line to move to another block, or if there are no more blocks of that size class, to acquire a new block.

3.2.3 Header, large objects, and boot image

MMTk has a two word (8 byte) header for each object, which contains a pointer to the TIB (type information block located in the immortal space, see below), hash bits, lock bits, and GC bits. A one word header for MarkSweep collectors is possible, but not yet implemented. Bacon et al. found that a one word header yields an average of 2-3% improvement in overall execution [6]. RefCount and the mature space in GenRC have an additional word (4 bytes) in the object headers to accommodate the reference count.

MMTk allocates all objects 8KB or larger separately into a large object space (LOS) using an integral number of pages. The generational collectors allocate large objects directly into this space. The LOS uses the treadmill algorithm [8]. It records a pointer to each object in a list. During whole heap collections, all of the collectors but RefCount and GenRC trace the live large objects, placing them on another list. They then reclaim any objects left on the original list. RefCount and GenRC reference count the large objects at each collection. MMTk does not a priori reserve space for the LOS, but allocates it on demand.

The boot image contains various objects and precompiled classes necessary for booting Jikes RVM, including the compiler, classloader, the garbage collector, and other essential elements of the virtual machine as part of the Java-in-Java design. MMTk puts these objects in an *immortal* space, and none of the collectors *collect* them. All except RefCount and GenRC trace through the boot image objects whenever they perform a while heap collection. RefCount and GenRC assume all pointers out of the boot image are live to avoid a priori assigning reference counts at boot time.

4. Methodology

This section describes Jikes RVM, our experimental platform, and key benchmark characteristics.

4.1 IBM Jikes RVM

We use MMTk in Jikes RVM version 2.3.1+CVS[1] [2, 1], with patches to support performance counters and *pseudo-adaptive* compilation. Jikes RVM is a high-performance VM written in Java with an aggressive optimizing compiler [1, 2]. We use configurations that precompile as much as possible, including key libraries and the optimizing compiler and turn off assertion checking (the *Fast* build-time configuration). The adaptive compiler uses sampling to select methods to optimize, leading to high performance [4], but a lack of determinism. Eechout et al. use statistical techniques to show that including the adaptive compiler for short running programs skews the results to measure the virtual machine [25]. In addition, adaptive compiler variations result in changes to allocation behavior and running time of the same run or runs with different heap sizes. For example, sampling triggers compilation in different methods, and the compilation of different write barriers for each collector is part of the runtime system as well as the program and induces both different mutator behavior and collector load [14].

Since our goal is to focus on application and garbage collection interactions, our *pseudo adaptive* approach deterministically mimics adaptive compilation.[2] First we profile each benchmark five times and select the best, collecting a log of the methods that the adaptive compiler chooses to optimize. This log is then used as deterministic compilation advice for the performance runs. For our *performance* runs, we run two iterations of each benchmark. In the first iteration, the compiler optimizes the methods in the advice file on demand, and base compiles the others. Before the second iteration, we perform a whole heap garbage collection to flush the heap of compiler objects. We then *measure* the second iteration which uses optimized code for hot methods and whose heap includes only application objects. We perform this experiment five times and report the fastest time. Our methodology thus avoids variations due to adaptive compilation.

4.2 Experimental Platform

We perform our experiments on three architectures: Athlon, Pentium 4, and Power PC. We present the Athlon results because it performs the best and it has a relatively simpler memory hierarchy that is easier to analyze.

[1]A 2.3.2 pre-release, cvs timestamp '`2004/03/25 05:11:47 UTC`'.
[2]Xianglong Huang and Narendran Sachindran jointly implemented the pseudo adaptive compilation mechanism.

	alloc	alloc:	% GC	% Nur	Source Field (p)					Target Object (o = *p)				
					% Read %			Focus		% Read %			Focus	
	MB	min	ß	srv	Nur	Mat	Imm	Nur	Mat	Nur	Mat	Imm	Nur	Mat
_202_jess	261	17:1	63	1	29	44	27	0.4	69	18	62	20	0.2	97
_228_jack	231	17:1	53	3	25	39	36	0.1	6	21	50	28	0.1	7
_205_raytrace	135	8:1	46	2	19	75	6	0.3	48	18	78	4	0.3	49
_227_mtrt	142	7:1	51	5	20	75	6	0.3	21	19	77	5	0.3	21
_213_javac	185	7:1	29	23	30	46	24	0.4	2	25	55	21	0.3	3
_201_compress	99	6:1	8	0	97	0	3	11.0	3	61	39	0	6.9	712
pseudojbb	216	5:1	21	32	16	59	25	0.2	2	14	72	15	0.1	2
_209_db	82	4:1	11	9	5	69	26	0.3	49	1	89	9	0.1	63
_222_mpegaudio	3	1:1	0	–	–	–	–	–	–	–	–	–	–	–

Table 1: Benchmark Characteristics

We use a 1.9GHz AMD Athlon XP 2600+. It has a 64 byte L1 and L2 cache line size. The data and instruction L1 caches are 64KB 2-way set associative. It has a unified, *exclusive* 512KB 16-way set associative L2 cache, and an 8 entry victim buffer [30] between the two caches. The L2 holds only replacement victims from the L1, and does not contain copies of data cached in the L1. When the L1 data cache evicts a line, it goes to the victim buffer, which in turn evicts the LRU line in the victim buffer into the L2. The Athlon has 1GB of dual channel 333MHz DDR RAM configured as 2 × 512MB DIMMs with an nForce2 K7N2G motherboard and 333MHz front side bus. This machine is marketed by AMD as being comparable to a 2.6GHz Pentium 4.

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

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

All three platforms run the same configuration of Debian Linux with a 2.6.0 kernel. We run all experiments in a standalone mode with all non essential daemons and services (including the network interface) shut down. We instrument MMTk and Jikes RVM to use the AMD and Intel performance counters to measure cycles, retired instructions, L1 cache misses, L2 cache misses, and TLB misses of both the mutator and collector as the collector algorithm, heap size, and other features vary. Because of hardware limitations, each performance counter requires a separate execution. We use version 2.6.5 of the *perfctr* Intel/x86 hardware performance counters for Linux with the associated kernel patch and libraries [35]. At the time of writing, perfctr was unavailable for the PowerPC 970.

4.3 Benchmarks

Table 1 shows key characteristics of each of our benchmarks. We use the eight SPEC JVM benchmarks, and pseudojbb, a variant of SPEC JBB2000 [37, 38] that executes a fixed number of transactions to perform comparisons under a fixed garbage collection load. The *alloc* column in Table 1 indicates the total number of megabytes allocated. Our prior work reports on the adaptive compiler activity [12] and thus shows more allocation and higher ratios of live data to allocation. However, Eeckhout et al. show that the adaptive compiler swamps program behaviors, and thus the methodology we use here exposes variations due to the program instead of the VM. The *alloc:min* column quantifies the garbage collection load with the ratio of total allocation to the minimum heap size in which GenMS executes. For a heap size of 2 × the minimum, the *% GC SemiSpace* shows the percentage of time SemiSpace spends performing GC work. The *Nur srv* quantifies generational behavior for a 4MB fixed size nursery using the percentage of allocated data that the collector copies out of the nursery.

The remaining columns indicate access patterns for object accesses. We instrument every pointer read '`o = *p`' and count the dereferenced field, p (columns 6–11), and the referent object, o (last five columns). Table 1 includes the percentage of reads from nursery (Nur), mature (Mat) and immortal (Imm) spaces. The *focus* presents the accesses in the nursery and mature space divided by the number of bytes allocated in the nursery and mature space respectively. For example, in _202_jess, 29% of the time p is the nursery, and 18% of the time, the dereferenced object o is in the nursery. The focus of accesses to p in the mature space (69) was more than 100 times greater than accesses to p in the nursery (0.4). A higher number reflects higher temporal locality. _202_jess promotes only around 1% of data into the mature space, and yet 44% of _202_jess's field reads are to this 1%, while 29% are to the 99% of objects that never survive the nursery.

We group programs according to Table 1. _202_jess, _228_jack, _205_raytrace, and _227_mtrt exhibit low nursery survival and high ratios of total allocation to minimum live size. _213_javac, pseudojbb, and _209_db have higher nursery survival, but a relatively high heap turnover. Two programs have high nursery survival and do not exercise collection much: _201_compress and _222_mpegaudio. _201_compress allocates large objects, and requires little garbage collection. _222_mpegaudio allocates less than 4MB, and thus the generational collectors never collect it. The first two groups of programs are thus better tests of memory management influences and policies and we focus on them. The results section presents representative benchmarks which we discuss in detail. Other benchmarks follow the same trends, except when noted. Complete results included in a technical report [**?**].

5. Results

This section examines collector performance and its influence on mutator and total performance using the Athlon. We first explain how occasionally small changes in heap sizes cause variations in collection time. We then compare the whole heap and generational collectors, validating the uniform performance benefits of the weak generational hypothesis [34, 41]. We then tease apart the influences of allocation and collection mechanisms. Contiguous allocation yields better mutator locality than free-list allocation, but the space-efficient free-list reduces total collector load. For most programs, cache measurements reveal that the spatial locality of objects allocated close together in time is key for nursery objects, but not as important for mature objects. A fixed nursery isolates the influence of the mature space collection policy, showing that mutator performance is usually agnostic to mature space policies with a few notable exceptions that need copying to achieve locality. However, when the mature space benefits from less frequent collection in GenMS, total time improves. Varying the nursery size reveals that frequent GC's in the small nursery degrade collector performance, and nursery sizes well above the L2 cache size perform best. We then show that the same trends hold across the Athlon, P4, and PPC architectures.

Figure 1 and subsequent figures plot total time, garbage collection (GC) time, mutator time, and cache statistics for different benchmarks as a function of heap size. The right y-axis expresses time in seconds and the left normalizes to the fastest time. Heap size is shown as a multiple of the smallest heap size in which the particular application executes using GenMS on the bottom x-axis, and in mega-bytes (MB) on top.

5.1 Collector Sensitivity to Heap Size

Figure 1 shows the general trend that up to some point increases in heap size tend to decrease the frequency of garbage collection and thus total time (see Figure 1). Each heap size is an independent trial. In all our experiments, the variation between runs on the same heap size is less than 1%. However, small changes in heap size can produce what seem like chaotic behavior, such as the differences in total and GC time between heap sizes 1 and 1.3 the minimum for GenMS on _213_javac. The reason is that a small change in heap size triggers collections at different points which changes which objects a collector promotes. For instance, consider a program that builds a large but relatively short lived pointer data structure. In a small heap, the generational collection point happens just prior to when the program builds the data structure, and in a slightly larger heap it happens in the middle. In the second case, the collector promotes the data structure, which dies shortly thereafter, but it does not detect the death until a whole heap collection. In the meantime, the increased heap occupancy triggers the next nursery collection sooner, and so on. The exact timing of a collection can thus have cascading positive as well as negative effects, and explains variations between nearby heap sizes.

5.2 Evaluating Generational Behavior

This section compares whole heap collectors to their generational counterparts and explores the generational write-barrier cost. Figure 1 shows that for _202_jess, _209_db, and _213_javac the generational collectors perform much better than their whole heap variants. This result holds on all the benchmarks, although the low-mortality, low GC load programs such as _201_compress only benefit in small heaps. Generational collectors reduce GC time for _202_jess by an order of magnitude, and even for _213_javac, where 23% of nursery objects survive, GenCopy improves GC time over SemiSpace by a factor of two, and GenMS improves over MarkSweep. The generational collectors reduce GC time by reducing the cost of each collection through only examining the nursery. Counting the number of collections (unshown) shows the reductions come from dramatically fewer collections as well. Because collection costs are heap-size dependent, the impact of GC time on total time is greatest in small to modestly sized heaps.

Examining mutator performance reveals that heap size does not systematically influence mutator time. Although the application itself is unchanged by heap size, larger heap sizes will tend to spread objects out more which makes this result counter intuitive. The mutator time is however strongly correlated with the GC algorithm, where SemiSpace usually performs best. SemiSpace benefits from no write barrier and faster allocation than MarkSweep. The generational collectors benefit from contiguous allocation. GenCopy and SemiSpace perform about the same for _213_javac and _209_db, whereas mutator performance in GenCopy is around 20% slower than SemiSpace on _202_jess. We now show that this difference is mostly due to the write barrier.

	% Overhead %
_202_jess	13.6
_228_jack	1.7
_205_raytrace	0.9
_227_mtrt	2.9
_213_javac	4.6
_201_compress	0
pseudojbb	3.1
_209_db	2.4
_222_mpegaudio	0
Geometric mean	3.2

Table 2: Write Barrier Mutator Overhead For 4MB Nursery

5.2.1 The Write Barrier: Friend or Foe?

To examine the cost of the write barrier, we use a new collector which has the same heap organization, write barrier, and promotion policies as GenCopy, but *traces* (but does not collect) the whole heap at each collection. It *collects* the whole heap only when the mature space is full. Because it always traces the entire heap, it establishes liveness of nursery objects by reachability, so the write barrier is not required for correctness. The garbage collection overhead of this collector is substantial, so we do not recommend it, but it yields an experimental platform in which we can include or exclude the write barrier while holding all other factors constant, such as the heap organization and promotion policy.

Table 2 shows the overhead of the standard MMTk generational write barrier on mutator performance with a 4MB nursery and a moderate heap (3 $\times$ minimum) on the Athlon platform. We show the percentage slowdown in the mutator when using the write barrier relative to mutator performance without the barrier. The overhead is low, 3.2% on average (3.1% for the P4 and 1.9% for the PPC). _202_jess suffers a substantial mutator slowdown. Table 1 indicates the high mortality rate and concentration of accesses to the few objects that do survive as the cause of the heavy write barrier traffic for _202_jess. However, the previous section shows that the massive reduction in collection costs swamps the mutator overhead in such a setting. Other benchmarks show very low overheads. For example in _222_mpegaudio, it never collects, thus no objects are ever in the large space, and the write barrier test never adds to the remembered sets. The multi-issue architecture thus completely hides its cost in unused issue slots. So while the write barrier has the potential to be expensive, its overhead is usually very low, and the advantages seen at collection time far outweigh the cost.

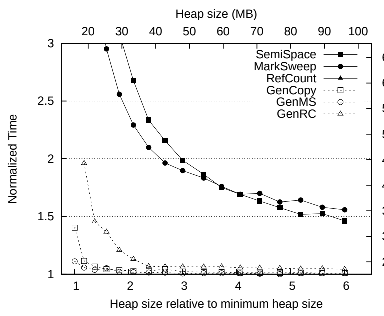
The combination of good mutator performance and outstanding GC performance is clear in the total time results. Even in _213_javac which has low infant mortality and in _209_db which has low GC work load, the generational collectors perform better than the whole heap collectors. In _202_jess, the advantage for the generational collectors is dramatic. This data supports the weak generational hypothesis, and indicates even when it is less true, generational collectors offer benefits.

5.3 Allocation: Free List versus Contiguous

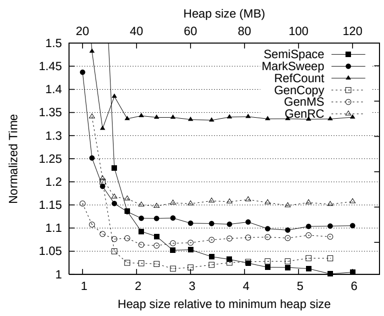
The essential allocator choice is free-list or contiguous, which in turn dictates the choice of collection algorithm. Free-list allocation is more expensive than contiguous allocation, but permits incremental freeing and obviates the need for a copy reserve. Contiguous allocations provide spatial locality for objects allocated close together in time, whereas free-list allocation may spread out these objects. To reveal the allocation time trade-offs, we examine their impact on the mutator. Since both RefCount and MarkSweep use the same free-list allocator, our analysis focuses on MarkSweep and GenMS, which are simpler than RefCount and GenRC.

5.3.1 Mutator costs in whole heap collectors

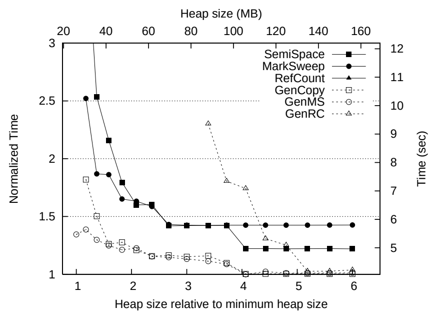
The contiguous and free-list allocators directly impact mutator performance as a consequence of the mutator allocation cost and the collection policies they impose. They also impact on the mutator through their locality effects.



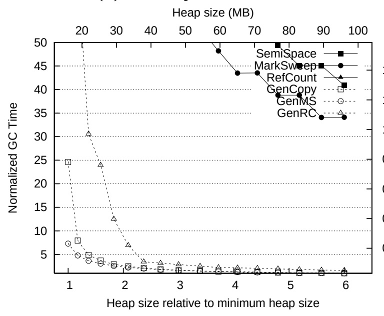
(a) _202_jess Total Time



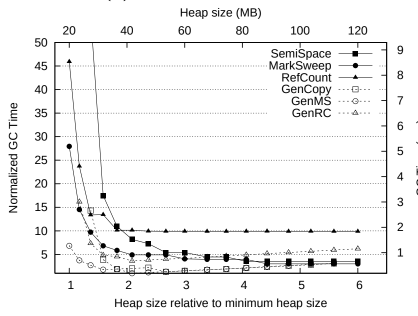
(b) _209_db Total Time



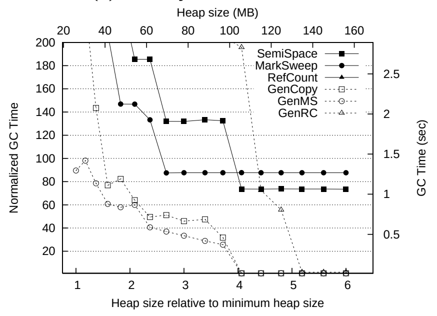
(c) _213_javac Total Time



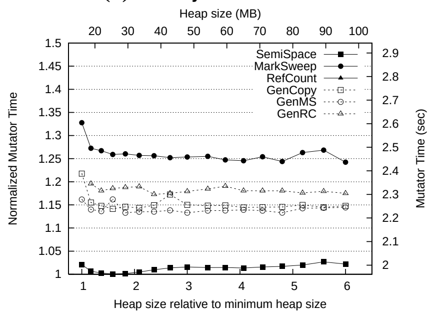
(d) _202_jess GC Time



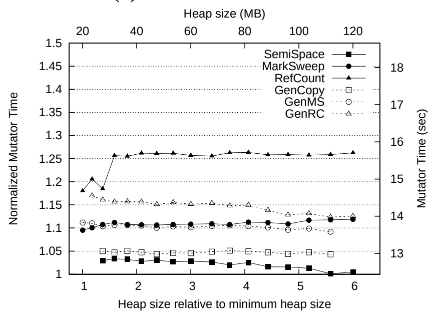
(e) _209_db GC Time



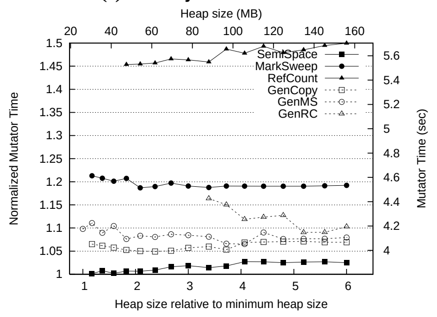
(f) _213_javac GC Time



(g) _202_jess Mutator Time



(h) _209_db Mutator Time



(i) _213_javac Mutator Time

Figure 1: Total, Mutator and GC Performance of All Six Collectors

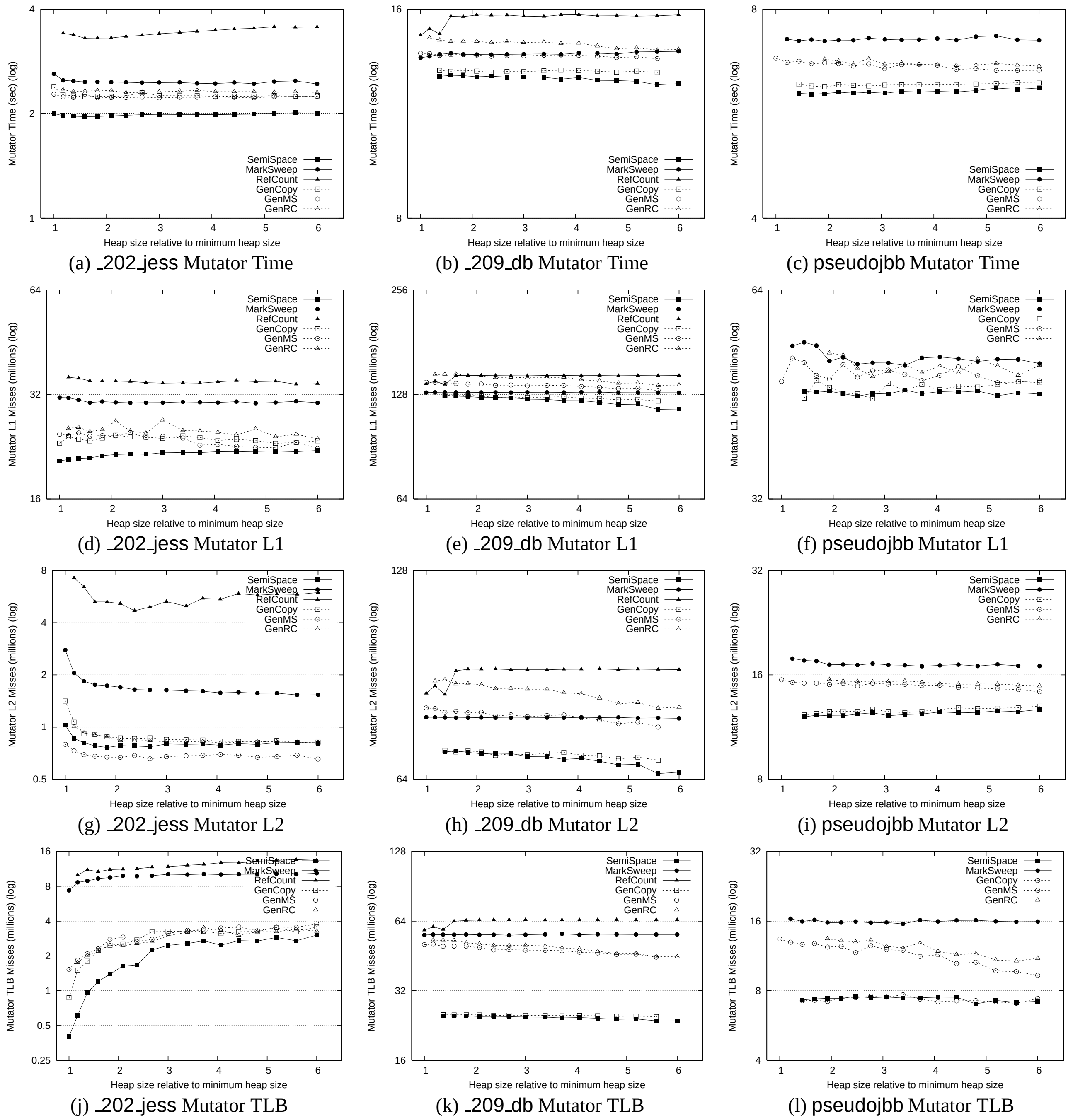
We first measure an upper bound on the time the program spends in contiguous allocation by pushing the allocation sequence out-of-line. This cost typically ranges from 1% to at most 10% of total time. We then use a micro benchmark to establish the relative costs of the two mechanisms. The benchmark allocates objects of the same size in a tight loop. Contiguous allocation is 11% faster than the free-list allocation, allocating at 726 MB/s and 654 MB/s respectively. (We recently reported slower times on an older architecture [12].) Since allocation time is less than 10%, this small difference between the mechanisms reduces to less than 1% of total time, and excludes the allocation sequence as a major source of variation.

Figure 2 examines mutator time and memory hierarchy performance for _202_jess, _209_db, and pseudojbb which have representative behaviors, plotting mutator time, L1 misses, L2 misses, and TLB misses as a function of heap size on a log scale. First consider SemiSpace and MarkSweep. SemiSpace mutator performance improvements range from 7 to 15% over MarkSweep (only on _201_compress and _222_mpegaudio is free-list allocation within 5%). The limit analysis above indicates that the direct effect of the allocator is typically 1% or less of this difference. Since the application code is otherwise identical, second order effects must dominate. _202_jess, _209_db, and pseudojbb each show a strong and consistent correlation between cache memory and mutator performance, where SemiSpace always improves over MarkSweep. Contiguous allocation in SemiSpace thus offers locality from two sources: allocation order and copying compaction. Free-list allocation in MarkSweep degrades program locality. The mutator benefit of SemiSpace over MarkSweep is relatively insensitive to heap size, thus suggesting that this benefit is from allocation locality rather than mature object compaction. An exception is the TLB performance on _202_jess, where the four copying collectors show a sharp reduction in TLB misses at smaller heap sizes, presumably due to collection-induced locality. However, L1 misses appear to dominate, so the reduction in TLB misses does not translate to a reduction in mutator time.

5.3.2 *Mutator costs in generational collectors*

We perform the following experiment to examine more closely whether SemiSpace locality is mostly due to the allocation order or the copying compaction of mature objects. We hold the work load on the mature space constant with a fixed-size nursery variant of the generational collectors. The young objects thus are all in allocation order. Since young objects are collected at the same frequency, only the mature space collection policies differ. Figure 3 shows the geometric mean of mutator performance across all benchmarks.

When the nursery size is fixed, GenCopy and GenMS have very similar mutator performance. The locality of mature space objects is thus not a dominant effect on mutator performance. As Sec-



(a) _202_jess Mutator Time (b) _209_db Mutator Time (c) pseudojbb Mutator Time

(d) _202_jess Mutator L1 (e) _209_db Mutator L1 (f) pseudojbb Mutator L1

(g) _202_jess Mutator L2 (h) _209_db Mutator L2 (i) pseudojbb Mutator L2

(j) _202_jess Mutator TLB (k) _209_db Mutator TLB (l) pseudojbb Mutator TLB

Figure 2: Mutator time and L1, L2 and TLB misses for all six collectors collectors (log scale).

tion 5.4.1 discusses in more detail, the variable-size nursery attains a space advantage when combined with GenMS which reduces the number of nursery collections, a direct benefit. The indirect benefit is slightly improved locality for nursery objects since they stay in allocation order in the nursery for longer. Figure 3 suggests that mature object compaction in the free-list will be of little use for these programs. However, Figure 2 reveals the two exceptions to this rule: _209_db and pseudojbb.

The most striking counterpoint is _209_db, where the generational collectors make little impact on mutator time. Even the copying nursery in GenMS provides no advantage over MarkSweep. GenCopy slightly degrades mutator locality compared with SemiSpace, due to the write barrier (see Table 2). Section 4.3 shows that _209_db is dominated by mature space accesses, and thus nursery locality is immaterial for _209_db.

In pseudojbb, the copying nursery benefits GenMS compared to MarkSweep, but GenCopy still performs significantly better than both. This suggests that pseudojbb has mature space access patterns which are locality sensitive. The access pattern statistics in Section 4.3 confirm this result. Mature space is accessed more heavily by pseudojbb, but the accesses are relatively unfocused.

Together the whole heap and generational results indicate that free-list allocation significantly degrades locality, whereas contiguous allocation achieves locality on young objects from allocation

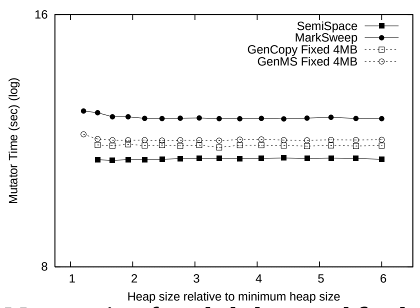


Figure 3: Mutator time for whole heap and fixed-size nursery collectors, geometric mean across all benchmarks

	MarkSweep mutator			SemiSpace mutator		
	1.5× min		ratio	1.5× min		ratio
	GCs	time (s)	∞/1.5×	GCs	time (s)	∞/1.5×
_202_jess	27	2.48	0.97	50	1.97	1.18
_228_jack	25	2.39	0.97	49	2.11	1.08
_205_raytrace	10	2.35	0.98	25	2.04	1.06
_227_mtrt	9	2.38	1.04	26	2.1	1.07
_213_javac	12	4.57	0.99	28	3.8	1.03
_201_compress	7	5.47	1.00	7	5.41	0.99
pseudojbb	9	7.21	1.00	32	6.04	1.07
_209_db	5	13.78	1.01	22	12.81	0.86
_222_mpegaudio	0	10.57	0.93	0	9.77	1.00
Geometric mean	8	4.57	0.99	18	4.02	1.03

Table 3: Impact of very large heap size on mutator time

order. Furthermore, a copying nursery ameliorates the locality penalty of the mature space free-list in all but _209_db and pseudojbb, where mature-space reads play a large role.

5.4 Collection: How, when, and whether?

The choice of allocation mechanism also governs the choice of collection mechanisms. We now examine the time and space overheads of the collection algorithms, and their influence on mutator locality. We consider how frequently to collect. We also show that our results are consistent across architectures, and then discuss if we should choose garbage collection at all.

5.4.1 Garbage collection costs

Contiguous allocation dictates copying collection which requires a copy reserve. The SemiSpace, GenCopy, and GenMS collector performance graphs reflect this copying space overhead which leads to many more collections than pure MarkSweep—SemiSpace typically collects between 1.5 and 2 times as often as MarkSweep for a given heap size. For example, GC time in Figure 2 for SemiSpace is typically at least 50% worse than MarkSweep. We measured the tracing rates for SemiSpace and MarkSweep on a micro benchmark: they are very close (59.5MB/sec and 59.2MB/sec) which means that the frequency of collection is the source of the overhead. In addition, GenMS with a variable nursery reduces the number of nursery collections over GenCopy because it is more space efficient. The first order effect of fewer collections is reduced collection time. A second order effect could be fewer cache line displacements to collector invocations. The stability of the mutator cache performance as a function of heap size in the face of dramatic differences in numbers of collections dissuades us of this hypothesis.

5.4.2 Trading off collection cost and mutator locality

Total performance is of course a function of the mutator and collector performance. While contiguous allocation offers a significant mutator advantage, its copy reserve requirement results in a substantial overhead. In small heap sizes, collection time typically swamps total performance and overwhelms mutator locality differences; MarkSweep outperforms SemiSpace. In large heaps, mutator time dominates and SemiSpace outperforms MarkSweep. Figure 1 illustrates the crossovers in total performance for MarkSweep and SemiSpace on _213_javac and _202_jess.

As Sections 5.3.1 and 5.3.2 establish, the locality advantage of contiguous allocation is greatest among the young objects. These results indicate that the copying nursery combined with a space efficient MarkSweep mature space offers a good combination of locality benefits and reduced collection costs. However, when mature space locality dominates, such as in _209_db, GenCopy can perform best.

5.4.3 Tracing or Reference Counting?

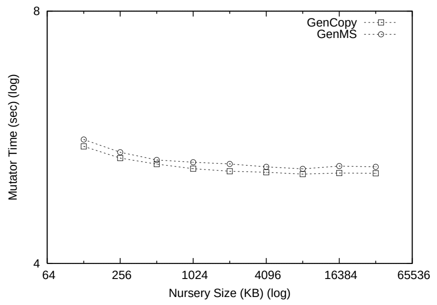
With a free-list, the collector can either trace the live objects from the roots or count references. Continuously tracking the number of references to each object is expensive, even with aggressive optimizations [21, 33], which the MMTk implementation also uses. This result is evident in Figure 1, where RefCount performs dramatically worse than MarkSweep for _202_jess and _213_javac. RefCount performs well on _201_compress, but this application is atypical. As discussed in Sections 5.2 and 5.3, there is compelling evidence for a generational policy with a copying nursery and a free-list in the mature space. The distinctly different demographics of young and old objects further motivate a hybrid generational reference counting policy [15]. Figure 1 shows that GenRC performs similar to the other generational collectors, except in _213_javac, which has an unusually large amount of cyclic data structures [7]. The performance of GenRC is sensitive to the frequency of cycle detection, which we did not tune in these experiments. GenRC holds a potential locality and space advantage over GenMS because it *promptly* reclaims dead mature space objects, and thus can more tightly pack the free-list. GenRC performs reference counting at every nursery collection whereas GenMS infrequently performs whole heap collections. This promise is not borne out in Figure 1, but may be a reflection of the immaturity of the GenRC implementation rather than on the fundamentals of the algorithm.

5.4.4 How often?

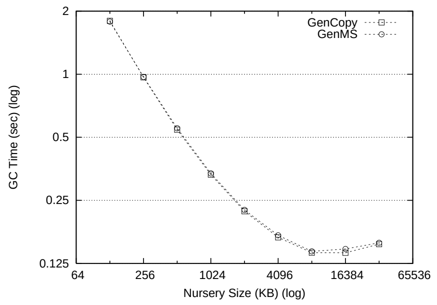
We now examine the limits of not collecting, and then examine how often to collect the nursery.

If the heap is never collected and memory is monotonically consumed, the spatial locality of older objects should gradually degrade as neighboring objects die. Assuming an approximately uniform death rate over time, fragmentation will be an exponential function of age—older objects being the most fragmented, and the very most recently allocated objects suffering no fragmentation. To examine this effect, Table 3 compares the mutator time for each benchmark using contiguous and free-list allocation with a modest heap (1.5× minimum), and an uncollected heap, large enough to avoid triggering any collection. For these benchmarks, 900MB is adequate. Only _202_jess follows the hypothesis that never collecting degrades performance. Since _202_jess has a high heap turn over and some accesses to mature space, it does suffer some fragmentation that degrades mutator performance when the heap is never collected.

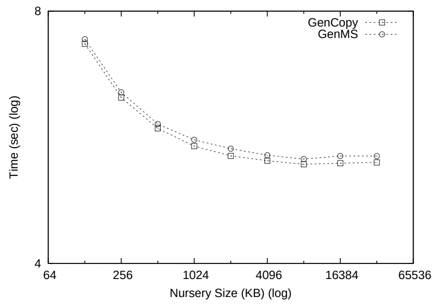
Most of the other benchmarks have about the same mutator performance in the uncollected heap (∞) as in the modest heap. At first this result seems a little surprising in light of the inevitable degradation in locality among the older objects. However, as Section 5.3 showed, the spatial locality of the mature objects is not a dominant factor for these benchmarks. _209_db actually achieves better performance without collection because it attains good locality from contiguous allocation and it has low GC work load. Blackburn et al. found for a more memory constrained machine, never collecting caused severe degradations in _209_db due to paging [13]. Table 1 together with mutator locality results indicate that all of the other programs have a slight majority of accesses to a few mature objects



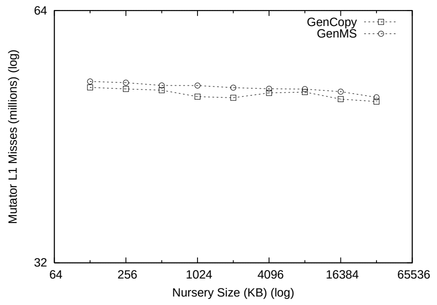
(a) Mutator Time



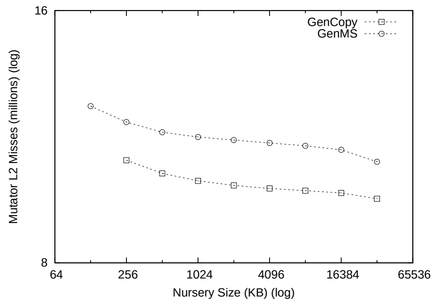
(b) GC Time



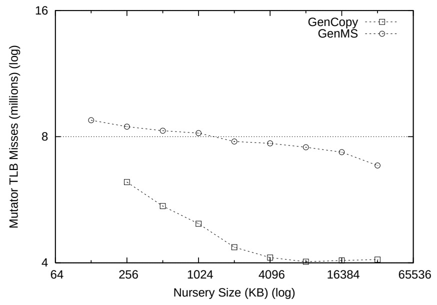
(c) Total Time



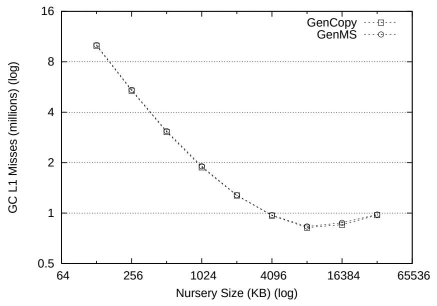
(d) L1 Mutator Misses



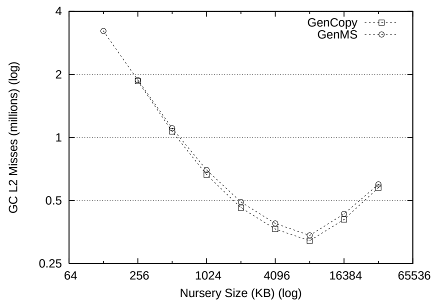
(e) L2 Mutator Misses



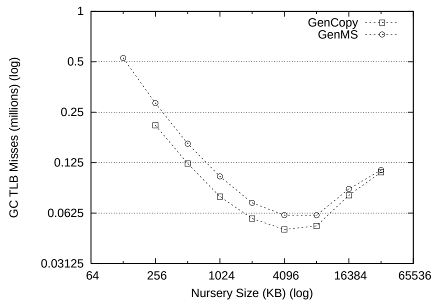
(f) TLB Mutator Misses



(g) L1 GC Misses



(h) L2 GC Misses



(i) TLB GC Misses

Figure 4: Performance Effect of Nursery Size, 128KB to 32MB (log scale)

with good temporal locality, and accesses to a very large number of young objects with poor temporal locality (typically used briefly then discarded). Thus, compression of mature space objects is not an important source of locality in these programs. We expect that server applications, and others with large memory usage and foot prints will follow _202_jess more than these results.

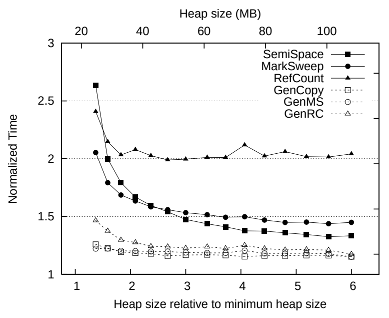
5.4.5 Sizing the nursery

Given the performance advantages of generational collection, we now examine the influence of the nursery size. Figure 4 shows the performance of GenMS and GenCopy over a wide range of bounded nursery sizes (128KB to 32MB), running in a very large heap (900MB). Note the x-axis in this figure is nursery size, rather than heap size as in all the other figures in this paper. Figure 4(a) shows a small improvement with larger nurseries in mutator performance due to fewer L2 (Figure 4(e)) and TLB misses (Figure 4(f)). However, the difference in GC time dominates: smaller nurseries demand more frequent collection and thus a substantially higher load. We measured the fixed overhead of each collection and found that each invocation of a collection scanned around 64KB of roots. These fixed costs become significant when the nursery is as small as 128KB. The garbage collection cost tapers off between 4MB and 8MB as the fixed collection costs become insignificant. These results debunk the myth that the nursery size should be matched to the L2 cache size (512KB on all three architectures).

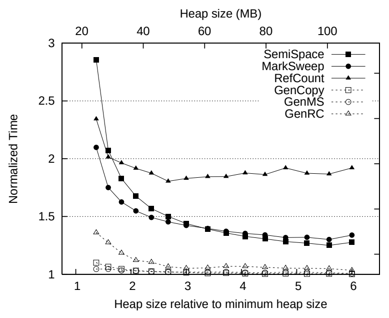
5.5 Architecture influences

Figure 5 compares the geometric mean of the benchmarks for all 6 collectors on the P4, Athlon, and PPC. The x-axis is heap size, and the y-axis is time. The P4 has the fastest clock speed, followed by the Athlon, and then the PPC. Intel would like us to believe that this ordering means the P4 will perform the best. Instead, the Athlon performs about 20% better. For the generational collectors, even the PPC is close to the P4. The Athlon's advantage comes from substantially fewer cache misses than the P4 (compare Figures 2 and 6). Due to the Athlon's exclusive cache architecture, substantially larger L1 and higher associativity L2, it simply has more effective cache and this advantage dominates clock speed.

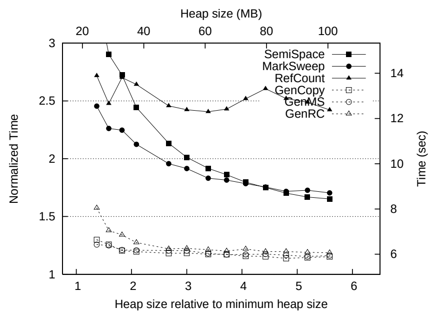
The collectors follow the same trends discussed above on all of the architectures. The generational collectors perform best on all architectures due to reductions in collection time and locality from contiguous nursery allocation. However the difference is more pronounced on the PPC than the P4 or Athlon which suggests reductions in the influence of collection time on faster processors. The space advantage of MarkSweep over SemiSpace, and the locality advantage of SemiSpace over MarkSweep show different cross-over points on each architecture. The faster the clock speed, the closer the cross-over point moves towards the minimum heap size, i.e., the cross-over where SemiSpace improves over MarkSweep is 2.2 for the P4, 3.4 for the Athlon, and 4 for the PPC. This trend



(a) P4

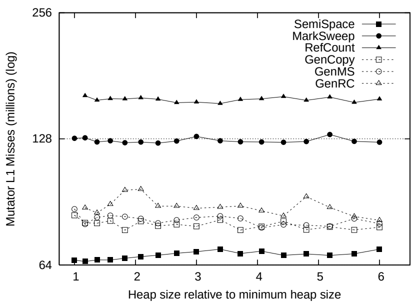


(b) Athlon

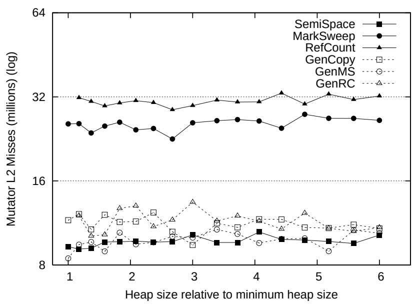


(c) PPC

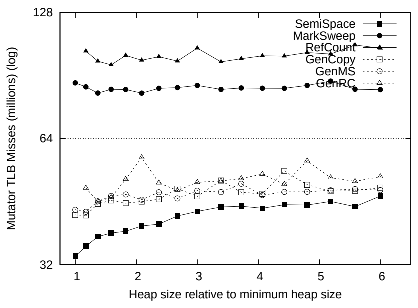
Figure 5: Total time on three architectures



(a) _202_jess Mutator L1



(b) _202_jess Mutator L2



(c) _202_jess Mutator TLB

Figure 6: P4 mutator L1, L2 and TLB misses for _202_jess (log scale). Compare with Figures 2(d), 2(g) and 2(j).

suggests that for future processors that the locality advantages of contiguous allocation will become even more pronounced.

5.6 Is garbage collection a good idea?

The software engineering benefits of garbage collection over explicit memory management are widely accepted, but the performance trade-off in languages designed for garbage collection is unexplored. Section 5.3 shows a clear mutator performance advantage for contiguous over free-list allocation, and the architectural comparison shows that architectural trends should make this advantage more pronounced. The traditional explicit memory management use of `malloc()` and `free()` is tightly coupled to the use of a free-list allocator—in fact the MMTk free-list allocator implementation is based on Lea allocator [32], which is the default allocator in standard C libraries. Standard explicit memory management is thus unable to exploit the locality advantages of contiguous allocation. It is therefore possible that garbage collection presents a performance *advantage* over explicit memory management on current or future architectures. A striking example of this is seen in Figures 1(a) and 1(g), where the *total* time for GenMS matches or betters the *mutator* time for MarkSweep. Further explortation of this is unfortunately beyond our scope. Another alternative—not reclaiming memory at all—is unsustainable.

6. Conclusion

This study examines the implications of the key policy choices in memory management on collection time, space, mutator locality, mutator performance, and total performance. A few key observations emerge. First, even if programs *do not* follow the generational hypothesis, the contiguous allocation of a copying nursery offers locality benefits that indicate the weak generational collectors are always the collectors of choice. As a corollary, although many accesses go to mature objects, their performance relies on temporal locality, whereas in the nursery, allocation order provides good spatial locality for young objects that die quickly. We also show that the cost of the generational write barrier is usually low. Secondly, the choice of mature space collector should not only be dictated by the space efficiency, which would always prefer MarkSweep, but should also include the rate of death among the mature objects, and the access and mutation rate of the mature space. If these rates are high, a copying mature space can attain better mutator locality that in the end overcomes its higher collection time penalty. These results can guide users to the right collector for their program, and offer insights to memory management designers for future collectors that could tune themselves on long running applications.

7. REFERENCES

[1] B. Alpern et al. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, Denver, CO, Nov. 1999.

[2] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

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

[4] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, October 2000.

[5] C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collectors. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[6] D. Bacon, S. Fink, and D. Grove. Space- and time-efficient implementations of the Java object model. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 111–132. ACM Press, June 2002.

[7] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in

reference counted systems. In J. L. Knudsen, editor, *Proc. of the 15th ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235. Springer-Verlag, 2001.

[8] H. G. Baker. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, 1992.

[9] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000.

[10] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 114–124, Salt Lake City, UT, June 2001.

[11] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–12, Seattle, WA, Nov. 2002.

[12] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *ICSE*, Scotland, UK, May 2004.

[13] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Proc. of SIGPLAN 2002 Conference on PLDI*, pages 153–164, Berlin, Germany, June 2002.

[14] S. M. Blackburn and K. S. McKinley. In or out? Putting write barriers in their place. In *ACM International Symposium on Memory Management*, pages 175–183, Berlin, Germany, June 2002.

[15] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 244–358, Anaheim, CA, Oct. 2003.

[16] H.-J. Boehm. Space efficient conservative garbage collection. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 197–206, 1993.

[17] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 353–366, Tampa, FL, 2001.

[18] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, Nov. 1970.

[19] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, Oct. 1983.

[20] D. L. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice & Experience*, 24(6):527–542, June 1994.

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

[22] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 92–115, June 1999.

[23] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, September 1978.

[24] A. Diwan, D. Tarditi, and J. E. B. Moss. Memory subsystem performance of programs using copying garbage collection. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages*, pages 1–14, Portland, OR, Jan. 1994.

[25] L. Eeckhout, A. Georges, and K. D. Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 244–358, Anaheim, CA, Oct. 2003.

[26] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In *ACM International Symposium on Memory Management*, pages 111–120, Minneapolis, MN, Oct. 2000.

[27] M. W. Hicks, J. T. Moore, and S. Nettles. The measured cost of copying garbage collection mechanisms. In *ACM International Conference on Functional Programming*, pages 292–305, 1997.

[28] A. L. Hosking and R. L. Hudson. Remembered sets can also play cards, Oct. 1993. Position paper for OOPSLA '93 Workshop on Memory Management and Garbage Collection.

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

[30] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990.

[31] J. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 264–274, Santa Clara, CA, June 2000.

[32] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[33] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 367–380, Tampa, FL, Oct. 2001.

[34] H. Lieberman and C. E. Hewitt. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

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

[36] Y. Shuf, M. J. Serran, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 194–205, Cambridge, MA, June 2001.

[37] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[38] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[39] D. Stefanović, M. Hertz, S. M. Blackburn, K. McKinley, and J. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Memory System Performance*, pages 175–184, June 2002.

[40] D. Tarditi and A. Diwan. Measuring the cost of storage management. *Lisp and Symbolic Computation*, 9(4), Dec. 1996.

[41] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, April 1984.

[42] B. G. Zorn. The measured cost of conservative garbage collection. *Software Practice & Experience*, 23(7):733–756, 1993.