Oil and Water? High Performance Garbage Collection in Java with JMTk

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

Increasingly popular languages such as Java and C# require efficient and flexible garbage collection. This paper presents the design, implementation, and evaluation of JMTk, a Java Memory Management Toolkit in Java. We show a software engineering success story with an efficient, composable, extensible, and portable framework for quickly building and evaluating collectors. JMTk attains its modularity and efficiency using a few design patterns and compiler cooperation. Experimental comparisons with monolithic Java and C implementations reveal significant performance advantages from our design. Performance critical system software typically uses monolithic C at the expense of flexibility. Our results refute common wisdom that only this approach attains efficiency, and suggest that performance critical software can embrace modular design.

1 Introduction

The tension between *flexibility* and *performance* pervades systems development, and one goal usually gives way to the other. Flexibility assists in rapidly realizing new ideas, and good performance gives the realizations credibility. This paper is a case study in mixing performance and flexibility in a systems research context where both are critical.

Programmers are increasingly choosing object-oriented languages with automatic memory management (*garbage collection*) because of their software engineering benefits. Although researchers have studied garbage collection for a long time [3, 19, 20, 24, 26, 35], this reliance on it and growing locality effects have made garbage collection research a high priority academically [13, 25, 17, 29, 33] and industrially [10, 11, 9]. Many collector implementations are monolithic and do not share reused components [2, 21]. Performance comparisons across a range of approaches is thus problematic and rare [5, 8, 21].

*This work is supported by NSF ITR grant CCR-0085792, NSF CCR grant CCR-0311829, DARPA grant F33615-03-C-4106, and IBM. Any opinions, findings and conclusions or recommendations expressed in this material are the authors and do not necessarily reflect those of the sponsors.

This paper presents the design, implementation, and evaluation of JMTk, a Java Memory Management Toolkit in Java for Java.[1] JMTk's flexibility is evidenced by support for a wide range of collectors: *copying, mark-sweep, reference counting, copying generational, hybrid generational*, and new ones [13, 14, 29]. We show how careful software engineering combines good design with excellent performance by comparing code and execution times of JMTk in Jikes RVM [1, 2], a Java-in-Java Virtual Machine, with monolithic Java and C implementations. JMTk is both more succinct and higher performing.

We address the tension between flexibility and performance with a combination of design features: 1) the use of Java as a systems language, 2) well chosen design patterns, 3) a clean interface between the virtual machine and JMTk, and 4) the composition of policies and mechanisms to define collectors. For correctness, we extend the Java type system to implement physical addresses and operations to move and update them. Design patterns provide performance and correctness in the face of concurrency by teasing out lightweight *hot* paths from heavyweight less frequently executed *cold* paths, together with *local* and *global* scopes. Additional patterns delineate collector phases and policies for correctness and extensibility. JMTk includes a narrow, portable interface between the runtime and memory manager, which abstracts VM-specific object and program representations. Researchers are porting JMTk to another JVM, a C# runtime, and a Haskell runtime.

We evaluate JMTk's trade-off between flexibility and performance by comparing with the original, highly tuned monolithic collectors in Jikes RVM [2]. JMTk's flexibility is evidenced two ways: 1) source code metrics reflect a substantially simpler and more modular design, and 2) JMTk implements a much wider range and number of collectors. We measure the performance using micro-benchmarks for raw allocation and collection speeds, and using the SPEC JVM benchmarks and a variant of the SPEC JBB2000 for

[1] JMTk is publicly available as part of Jikes RVM at: `http://www-124.ibm.com/developerworks/oss/jikesrvm/`.

performance. On micro-benchmarks, JMTk performs on average about 5% worse on allocation and tracing speed than comparable monolithic Jikes RVM collectors. We also implemented the micro-benchmark in C. JMTk outperforms the GNU C library's allocation implementation on average by 60%, due to aggressive compiler inlining and reduced impedance in a Java-in-Java implementation. On real benchmarks, JMTk consistently improves total performance by up to 20% over the original monolithic collectors, largely due to JMTk's more flexible space management.

The background section next outlines the key mechanisms and policies for readers unfamiliar with memory management. Section 3 compares JMTk with other memory management toolkits [13, 22, 21] none of which attain the diversity and composability of implementation, nor the performance of JMTk. Section 4 then discusses JMTk's design, followed by results and conclusions. The key contribution of this work is to describe a clean flexible design and a fully fleshed out implementation of a performance critical system component, the memory manager, that practices good software engineering. A surprising result is that this design approach attains performance benefits as well.

2 Background

This section describes memory management terms and algorithms, and how JMTk organizes the heap to implement them. For a thorough treatment, see Jones and Lins [24]. 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, mix in with the mutator. Collection can run concurrently with mutation, but for simplicity our discussion assumes a separate collection phase.

JMTk groups regions of memory into *space* organizations and implements garbage collection algorithms with a *policy* which couples a space with an allocation and collection mechanism. *Whole heap* collectors use one policy. *Generational* collectors divide the heap into age cohorts, and use one or more policies [3, 35]. For generational and other incremental algorithms, a *write barrier* remembers pointers into independently collected spaces. For every pointer store, the compiler inserts write-barrier code. At execution time, pointers into an independently collected space are conditionally recorded by the write-barrier. JMTk implements the following standard allocation and collection mechanisms.

A Bump Pointer 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 a size-segregated *free-list* that divides memory into blocks of size *k*. New objects are allocated into a free cell of a block whose size can just accommodate that object.

A Tracing Collector identifies live objects by computing a transitive closure from the *roots* which include stacks, registers, and 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.

JMTk forms *policies* with these mechanisms.

Copy space has Bump-pointer allocation and tracing collection that copies live objects out of the space.

MarkSweep space has free-list allocation and tracing collection that returns dead objects to the free-list.

RefCount space has free-list allocation and reference counting collection that returns dead objects to the free-list.

Immortal space has bump-pointer allocation and no collection.

Large object space has a coarse-grained free-list of pages and *treadmill* collection [24].

These policies are combined to form the following collectors.

SemiSpace: SemiSpace uses two copy spaces. It allocates into one. When full, it copies live objects into the other, and then swaps them.

MarkSweep: MarkSweep uses one mark-sweep space. It traces and marks the live objects, and lazily finds free slots during allocation.

RefCount: The deferred reference-counting collector uses a free-list allocator. During mutation, it buffers counts. The collector periodically processes the counts, introduces *temporary* increments for deferred objects (e.g., roots), and then deletes objects with a zero count.

GenCopy: The classic copying generational collector [3] allocates into a young (*nursery*) Copy space, and promotes survivors into an old SemiSpace. The write barrier records pointers from old to nursery objects. It collects when the nursery is full, and reduces the nursery size by the size of the survivors. When the old space is full, it collects the entire heap.

GenMS: This hybrid generational collector is like GenCopy except it uses a MarkSweep old space.

GenRC: This hybrid generational collector uses Ulterior Reference Counting [14] to combine a copying nursery with a RefCount mature space.

3 Related Work

This section compares JMTk with previous garbage collection toolkits [13, 22, 21] and explicit memory management [6, 7, 12, 36] toolkits. The UMass Language Independent GC Toolkit was the first garbage collection toolkit to tease apart the language and collector interface in order to build portable garbage collectors [22]. Systems for Smalltalk, Modula-3, Cecil, and Java [18] use the UMass GC Toolkit. It provides generational copying collectors, and manages memory in fixed-size blocks. It manages each large object directly, using a list associated with each generation. It does not include free lists, so does not support mark-sweep or reference counting collectors. Its design is not general enough to include even some recent copying collectors such as Older-First [33] nor Beltway [13].

GCTk, a more general Garbage Collection Toolkit for Java addressed some of these shortcomings [13, 32]. GCTk is the only other garbage collection toolkit we know of that was implemented in Java. This framework provides a single shared implementation of key functions such as scanning and *remembered sets* which record write barrier entries. In addition, GCTk implements copying age-based collectors by separating the collection increment from the heap position [13, 32], but it did not include free-list allocation, nor could it mix and match policies, and it was not intended to be portable. JMTk improves upon GCTk in a number of ways. It creates a cleaner, and we believe portable language/compiler interface. JMTk uses a composable design to easily mix and match policies and mechanisms. It also adds free-list memory managers (e.g., mark-sweep and reference counting), a large object space, the composition of disparate policies, and accounting for physical memory that is independent of virtual memory use and uniform across policies.

The Marmot system [21] is an ahead-of-time compiler and runtime system for Java written in C. It provides semi-space and copying generational collectors. It produces very efficient allocation and write-barrier sequences using compiler cooperative inlining. JMTk generalizes this pattern and applies it more broadly. JMTk includes a much wider range of collectors and policies than Marmot, is modular and extensible, and works in a just-in-time Java VM.

A few researchers have also built memory management toolkits for explicit memory management of C/C++ programs [6, 7, 12, 36]. Heap layers is the most general and high performance of these frameworks [12]. It provides multiple and composable heaps. It achieves the performance of existing custom and general-purpose allocators in a flexible, reusable framework. It attains this combination through the use of mixins [16]. Our framework could also probably benefit from mixins that statically express multiple possible class hierarchies, but we have not investigated it here. Explicit memory managers for C/C++ have a very small interface and interact with the program only through the malloc and free. JMTk has a more complex interface since it interacts with a managed runtime system on many mechanisms including scheduling, write barriers, object model, and root identification. Because of the small interface, C/C++ limits the memory manager to free-lists, since objects cannot move. The Customizable Memory Management (CMM) framework focuses on automatically collecting these heaps, but uses virtual method calls, thus sacrificing performance [6, 7]. These frameworks thus are not and need not be as general as JMTk.

Yeates *et al.* [38] analyzed four tracing collectors and identified six design patterns, of which two were new domain-specific patterns. Their work did not include reference counting nor collector toolkits where multiple instances of the same pattern might occur.

4 Design

JMTk's highest level design goals are *flexibility* and *performance*. Flexibility assists in rapidly realizing new research ideas, and good performance gives the realizations credibility. Common wisdom holds that these goals are incompatible. As a result, prior memory management systems focus on high performance using monolithic and inflexible C and assembly implementations that curtail creative research.

This section discusses how the design of JMTk reaches both goals using Java as a systems implementation language. It begins with three factors of our Java-in-Java implementation: extensions to the Java type system, pragmas for guiding the compiler, and the problem of the collector executing within the heap that it is collecting. We then describe design patterns used to attain correctness and performance. Then the interface between JMTk and the virtual machine and its role in portability is discussed. Section 4.4 outlines JMTk's reusable memory management mechanisms and policies, and how to compose them to yield new systems. Finally we measure the reuse of components and compare collectors in JMTk to the original monolithic implementations. These results demonstrate the benefits of modularity and aggressive software reuse.

4.1 Java as a Systems Language

JMTk grew out of Jikes RVM and thus, like Jikes RVM, is implemented in Java. Implementing a language runtime in itself presents some well known problems [1, 23]. We address the key issues for Java here: 1) extensions for manipulating memory directly, 2) exploiting VM compiler pragmas, and 3) how to safely implement a collector that executes within the heap it collects (or, 'eating your own dog food').

New Types for Unsafe Operations

JMTk uses a modest extension of Java defined by and developed for Jikes RVM [1, 2]. In order to access and

modify memory, we require unsafe operations. JMTk requires that the VM defines three special types: `VM_Address`, `VM_Offset`, and `VM_Word`. `VM_Address` corresponds to a hardware-specific notion of memory address. `VM_Offset` expresses the distance between two addresses, and `VM_Word` corresponds to the value returned by dereferencing an address. These types provide methods for pointer arithmetic, pointer comparison, casts, and memory reads and writes including atomic operations. The memory manager uses these methods to perform its lowest-level operations, such as allocating and moving objects.

Since Java does not provide extensible primitive types, these special types are Java classes and we rely on the underlying VM to treat these types specially. The intended behavior is for values of such types to be *unboxed*, i.e., operations on these types never result in allocation. We currently express these extensions using the idioms defined by Jikes RVM. A source code transformation makes JMTk portable to the gcc Java front end (gcj) and other targets such as OVM [28] and Rotor [34].

Compiler Pragmas

We use Jikes RVM's pragmas for controlling *inlining* and *interruptibility*. In Jikes RVM, pragmas are scoped across classes and methods using the **`implements`** and **`throws`** idioms with suitably named interfaces and exceptions. For example, the method qualifier '**`throws`** `PragmaInline`' directs the compiler to inline a method. Inlining is only scoped with respect to individual methods. More specific pragma scopes (such as method) override broader scopes (such as class), allowing interruptible methods to exist within otherwise uninterruptible classes. Control over inlining helps improve efficiency for system-level code written in an object-oriented style (see Section 4.2 and Section 6).

Specifying when the program can be interrupted is important because exact garbage collection depends on compiler-generated maps to identify pointers stored on the stack. To ensure that all suspended threads have accurate stack maps, the virtual machine makes sure that thread switches occur only at garbage collection *safe-points*, which are automatically generated by the compiler. JMTk needs further control over this behavior and uses pragmas to mark certain code as uninterruptible. For example, JMTk implements write barriers with a call to a Java write-barrier method (see Section 2). Pointer states at intermediate points during a write-barrier sequence are not necessarily accurate, and thus should not be exposed to the collector. To attain atomicity on these operations, write barrier methods use the *uninterruptible* compiler pragma to ensure that thread switching never happens within the barrier.

Executing Within Its Own Heap

JMTk faces the problem of executing with all of its code and state residing within the heap it is collecting (in Jikes RVM code, threads and stacks all exist as heap objects due to the Java in Java implementation). The collector must not scavenge itself! More subtly, it is essential that copying collectors *pre-copy* any GC-related instances and execute with respect to that state in order to avoid referencing an out-of-date snapshot of the collector's state—such a snapshot will lead to catastrophic time-warp once execution eventually switches to the copied instance. We address these issues by allocating certain objects in an immortal (unmoving and uncollected) space and by providing a generic feature that pre-copies crucial state for any copying collector, relieving the implementer of the collector from the burden of identifying and acting upon the crucial instances.

4.2 Design Patterns Used in JMTk

JMTk uses design patterns for efficiency and reuse. JMTk patterns include those identified in prior work (*TriColor*, *RootSet*, *Adapter*, *Facade*, *Iterator* and *Proxy*) [37], and four additional patterns: 1) exploiting the behavior of the most heavily executed code for efficiency; 2) minimizing contention and the synchronization for efficiency; 3) exploiting collection phases to simplify the construction of collectors; and 4) delegating collector actions to specific policies.

Hot and Cold Paths

Wherever appropriate, JMTk applies a pattern that uses very efficient, lightweight mechanisms for frequently executed code (the *hot path*), and periodically uses more heavyweight mechanisms (the *cold path*). JMTk usually marks the hot path with the compiler inline pragma. JMTk uses this pattern extensively to reduce synchronization frequency, and to allow more complex heuristics.

For example, a copying nursery performs most allocation with a very fast, unsynchronized 'bump pointer', but periodically (every 128KB), the allocator takes the slow path, and acquires another 128KB chunk of memory. The acquisition of each 128KB chunk is synchronized (multiple threads contend for a pool of such chunks), and includes a poll of the collection subsystem, giving it the opportunity to invoke a collection if necessary. JMTk also applies this pattern to write barriers and internal dynamic data structures such as queues and buffers, which support concurrent allocation.

Local and Global Scopes

In a concurrent system, determining the local or global context of memory management functions (i.e., exposure to contention or not) is essential to both correctness and performance. In JMTk, scope is overt through the use of classes. For example, JMTk typically associates an instance with each thread[2] and uses the class to reflect global state.

[2]Here we mean truly concurrent threads, which in Jikes RVM map to kernel threads and typically reflect the number of physical CPUs.

Instance methods operate over their data without synchronization, and access shared state through synchronized class methods. This model assumes a single global state. More generally, N global instances may exist, over each of which P threads operate concurrently. In this case, JMTk provides 'local' and 'global' variants of a class, with N instances of the global class and $N \times P$ instances of the local class, each mapped to one global instance. JMTk synchronizes only accesses to the global state.

JMTk uses this pattern to build load balancing shared data structures (such as work queues and sequential store buffers), to build multi-threading mechanisms, and to operate over free lists, bump pointers, mark-sweep collection, reference counting, and other functions.

Prepare and Release Phases

JMTk uses a simple high level algorithm to implement all of the stop-the-world (i.e., non-concurrent) collectors. The algorithm has three phases: *prepare*, *process all work*, and *release*. JMTk splits the phases into global and local steps and performs them in the following order: `prepareGlobal`, `prepareLocal`, `processAllWork`, `releaseLocal`, and `releaseGlobal`. The `processAllWork` method is common to all collectors, and consists of transitively processing a collection work queue which is primed in the prepare phase. Each new collector need only implement the respective prepare and release phases. For instance, a simple copying collector will establish all roots of collection in the prepare phase, and reclaim the space in the release phase. The infrastructure ensures proper synchronization between phases and that the global versions are called by exactly one thread.

Multiplexed Delegation

JMTk builds collectors through the composition of policies and mechanisms. Plans (discussed in more detail in Section 4.4 below) perform this composition. For example, when an object needs to be allocated or traced for liveness, an appropriate method will be invoked on the plan, which will *delegate* responsibility to the appropriate policy depending on the object. This is performed using a pattern we call '*multiplexing delegation*'. In Figure 1 we see the `traceObject()` method of the `Generational` class, which delegates tracing to a range of policies depending on the space in which the object resides. When we analyzed the cyclomatic complexity [27] of the plans (Section 4.5), we found that the application of this pattern captured over 50% of the complexity of the plans. This result is not surprising since the role of the plan is composition and delegation.

4.3 The Virtual Machine Interface

Since one of JMTk's goals is to be portable, the interface between it and the rest of the virtual machine must be as clear and thin as possible without compromising on design flexibility. Researchers have ported JMTk to a C interface, using the gcj ahead-of-time compiler to build a static JMTk library, and are porting it to Rotor [34], Microsoft's open C# runtime, among others.

```
1  public static VM_Address traceObject(VM_Address obj) {
2    if (obj.isZero()) return obj;
3    VM_Address addr = VM_Interface.refToAddress(obj);
4    byte space = VMResource.getSpace(addr);
5    if (space == NURSERY_SPACE)
6      return CopySpace.traceObject(obj);
7    if (!fullHeapGC)
8      return obj;
9    switch (space) {
10   case LOS_SPACE:
11     return losSpace.traceObject(obj);
12   case IMMORTAL_SPACE:
13     return ImmortalSpace.traceObject(obj);
14   case BOOT_SPACE:
15     return ImmortalSpace.traceObject(obj);
16   case META_SPACE:
17     return obj;
18   default:
19     return Plan.traceMatureObject(space, obj, addr);
20   }
21 }
```

Figure 1. A Multiplexed Delegation Pattern: The traceObject Method for Generational Collectors.

We think of the interface in terms of the *requirements* and *features* of the VM (virtual machine) and MM (memory manager). The interface is bi-directional, reflected in the classes `VM_Interface` and `MM_Interface`. `VM_Interface` implements the requirements of the MM in terms of the VM's feature set, while `MM_Interface` implements the requirements of the VM in terms of the MM's feature set.

The key requirements of the MM include identifying the sources of pointers and providing access to per-object GC state. In addition to these, the MM requires housekeeping functionality such as low level memory operations (`mmap`, `bzero`, `memcpy`, etc.), hardware timers, atomic memory operations, error handling, I/O, and option processing. These requirements are implemented in terms of the VM's feature set through `VM_Interface`. Garbage collection typically begins at the root set (global variables and local variables on the threads' stacks and registers). The VM enumerates these roots objects into JMTk's queue data structures. JMTk enumerates all pointers in these objects and performs a transitive closure over them. Some collectors maintain state on a per-object basis (in the object header, for example). The `VM_Interface` provides this abstractly, giving portability across VMs, languages, and object models.

Currently, JMTk allocates memory from the OS via `mmap` in 4MB chunks when needed. Depending on a flag, some of the policies will return memory back to the operating system by unmapping the memory. Whether this is desirable is dependent on the scarcity of memory and the frequency of memory-mapping operations.

On the other side, the VM requires that the MM provide allocation; finalization; soft, weak and phantom references; write barrier implementations; and basic statistics such as heap size and GC count. JMTk provides these with the `MM_Interface` class.

4.4 Composition: Mechanisms, Policies, and Plans

At the heart of JMTk are the software engineering benefits of composition. These benefits include reuse, quick development of new collectors, robustness, fair comparisons of algorithms by holding the underlying mechanisms constant, and the opportunity for performance tuning. Section 6 shows that JMTk obtains these benefits together with excellent performance. We now outline the key compositional elements in JMTk: *mechanisms*, *policies*, and *plans*.

Mechanisms

JMTk implements a rich set of collector-neutral, highly-tuned mechanisms that are shared among collectors, including bump pointer allocation; free list allocation; large object allocation; finalization; soft, weak, and phantom references; parallel load balancing data structures; template-driven command line parsing; trial-deletion cyclic garbage collection; a generic, abstract free list; and thread-safe and GC-safe I/O routines.

Policies

JMTk currently implements five policies: immortal allocation, copying collection, mark-sweep collection, reference counting collection, and treadmill collection. These policies are each expressed succinctly in terms of the above mechanisms. JMTk maps policies to *spaces*, which are contiguous regions of virtual memory managed by a single policy. The same policy can manage multiple spaces within an address space. For example, in the GenCopy collector the copying collection policy manages multiple spaces that correspond to generations. Each policy follows the local/global pattern (Section 4.2), implemented in terms of a `Space` and `Local` pair for each policy (for example `MarkSweepSpace` and `MarkSweepLocal`). Each instance of a policy space maps to a single virtual memory space, and has associated with it *P* instances of the 'local' class, where the collector is *P*-way parallel.

Plans

JMTk uses the term *plan* to define the highest level of composition. A plan composes policies to build a memory management algorithm. For example, the GenMS plan composes a variety of policies. Each of these policies is manifest as a space declared within the plan, which binds each space to a region of virtual memory. Virtual and physical memory resources are associated with spaces and the multiplex pattern (Section 4.2) ensures that allocation and tracing use the appropriate policy depending on the space.

JMTk implements a growing number of plans that include SemiSpace, MarkSweep, RefCount, CopyMS, GenCopy, GenMS, NoGC, and GenRC which implements the recently published Ulterior Reference Counting [14] collector. Researchers are currently adding two more recently published collectors: Beltway [13], and Mark-Copy [29].

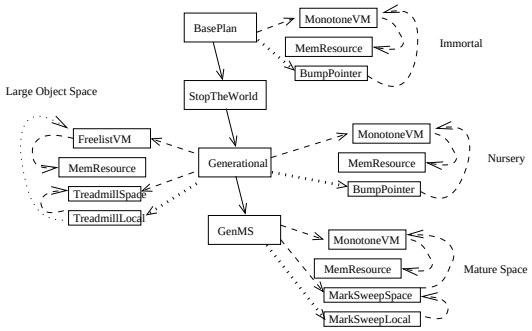


Figure 2. Composition of GenMS.

	m	NCSS	NCSS/m	BC	BC/m	CCN	CCN/m
Watson 2.0.0							
SemiSpace	50	1234	24.7	2223	44.5	325	6.5
MarkSweep	78	2288	29.3	3955	50.7	658	8.4
GenCopy	56	1597	28.5	2696	48.1	422	7.5
GenMS	90	2311	25.7	4719	52.4	633	7.0
Total	**274**	**7430**	**27.1**	**13593**	**49.6**	**2039**	**7.4**
Watson 2.2.0							
SemiSpace	40	426	10.7	850	21.2	139	3.5
MarkSweep	31	346	11.2	574	18.5	105	3.4
GenCopy	42	659	15.7	1312	31.2	220	5.2
GenMS	40	531	13.3	1294	32.4	171	4.3
Total	**153**	**1962**	**12.8**	**4030**	**26.3**	**635**	**4.2**
JMTk							
SemiSpace	30	237	7.9	463	15.4	98	3.3
MarkSweep	29	240	8.3	421	14.5	89	3.1
GenCopy	19	117	6.2	198	10.4	49	2.6
GenMS	18	104	5.8	158	8.8	43	2.4
Generational	36	279	7.8	434	12.1	102	2.9
Total	**132**	**977**	**7.4**	**1674**	**12.7**	**383**	**2.9**

Table 1. Methods (m), Non-comment Source Statements (NCSS), Bytecodes (BC), and Cyclomatic Complexity (CCN) for Four Garbage Collectors in Three Systems

Figure 2 illustrates the composition mechanism discussed in this section with a diagram of the GenMS collector. The unbold boxes represent the mechanisms and these form four groups, each representing a different space. For example, the mature space is managed with the mark-sweep collection (i.e. MarkSweep) policy. The bold boxes represent various plan classes with the GenMS being the most specific one. The solid arrows represent class inheritance, the dashed arrows point to static fields, and the dotted arrows point to instance fields. Each of the four groups show the local-global and hot-cold pattern. The 4-way multiplexing is demonstrated by the connection between the plan components and the four spaces.

4.5 Exploiting Java's Features in JMTk

This section evaluates how well JMTk attains its software engineering goals with reuse and inheritance. Table 1 compares the implementation of a classic copying generational collector (GenCopy) and a hybrid copying, mark-sweep generational collector (GenMS) in JMTk written in Java with an object-oriented style and two releases of the Watson collectors written in Java with a monolithic style. Watson 2.0.0 was the first public release of the Watson collectors in Jikes RVM, and Watson 2.2.0 reflected a major clean up and refactoring and was the *last* public release.

Table 1 reports the number of methods, lines of code, and number of bytecodes, and method cyclomatic complexity [27] for each of the three systems. JMTk uses a common superclass `Generational` to implement most of the functionality of the two generational collectors. Command line parameters select among multiple nursery sizing polices (fixed, bounded, flexible) in these collectors. The Watson collectors implement only the fixed nursery policy (the Watson 2.0.0 code base included a *distinct* collector with 1850 lines of code which implemented a variable nursery generational collector). In addition, there is an enormous reduction in overall complexity, the object-oriented style in JMTk attains an average method cyclomatic complexity [27] substantially lower than in the Watson implementations. Cyclomatic complexity measures the complexity of the branching and looping. This approach reflects our faith in the capacity of Jikes RVM's aggressive optimizing compiler [1, 2] to produce high quality code from strongly objected-oriented source.

5 Methodology

This section briefly describes Jikes RVM, our experimental platform, and key characteristics of our benchmarks.

We use JMTk in Jikes RVM version 2.3.0.1 (formerly known as Jalapeño). 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 (the *Fast* build-time configuration), and turn off assertion checking. For our micro-benchmarks, we use the highest level of optimization and run the benchmark multiple times to exclude compiler activity. For all other experiments, we use the adaptive compiler which uses sampling to select methods that it then optimizes [4]. Adaptive compilation introduces variations in the load on the garbage collector due to its statistical choices and because 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 [15].

We perform all of our experiments on a 2 GHz Intel Xeon, with 16KB L1 data cache, a 16K L1 instruction cache, a 512KB unified L2 on-chip cache, and 1GB of memory running Linux 2.4.20. We run each benchmark at a particular parameter setting five times and use the fastest of these. The variation between runs is low, and we believe this number is the least disturbed by other system factors and the natural variability of the adaptive compiler with respect to its heap allocation and time.

	alloc (MB)	alloc:min	Watson small:large
_202_jess	403	25:1	6:5
_213_javac	593	23:1	16:5
_228_jack	307	22:1	2:1
_205_raytrace	215	12:1	8:5
_227_mtrt	224	11:1	11:5
_201_compress	138	8:1	1:3
pseudojbb	339	7:1	36:5
_209_db	119	6:1	2:1
_222_mpegaudio	51	4:1	3:4

Table 2. Benchmark Characteristics

Table 2 shows key characteristics of each of our benchmarks using fast adaptive compilation. We use the eight SPEC JVM benchmarks, and pseudojbb, a variant of SPEC JBB2000 [30, 31] that executes a fixed number of transactions to perform comparisons under a fixed garbage collection load. The *alloc* column in Table 2 indicates the total number of megabytes allocated. The second column lists the ratio of total allocation to the minimum heap size for the GenMS collector in JMTk to quantify the garbage collection load. Watson collector users must statically partition the heap into 3 parts: immortal, large, and small objects (see Section 6.2 for additional discussion). For all of our experiments, the Watson collectors use 1 MB of immortal space. We determined experimentally the minimum size for the small and large object spaces and show this ratio in third column. When varying heap sizes, we allocate 1MB to Watson's immortal space and allocate the remaining space to the small and large spaces according to the ratio in column 3.

6 Results

We first compare our implementations of SemiSpace and MarkSweep with the original highly tuned Jikes RVM collectors called the *Watson* collectors to illustrate that our collector design has not come at a performance penalty. We compare on micro-benchmarks to reveal raw throughput. JMTk's abstractions cost about 5% compared to the Watson collectors. We also compare JMTk with a standard C malloc implementation on micro-benchmarks to reveal whether Java is a suitable systems language, where JMTk actually attains better performance due to the Jikes RVM compiler's inlining, a Java in Java feature. To measure what users will see in practice, we compare JMTk and Watson collectors on standard benchmarks, measuring garbage collection, mutator, and total time. JMTk attains significantly better performance largely because it dynamically partitions the heap

	Allocation Rate (MB/s)	Tracing Rate (MB/s)
Watson SemiSpace	690	58
JMTk SemiSpace	610	55
Watson MarkSweep	600	93
JMTk MarkSweep	575	69

Table 3. Allocation and Tracing Rates

	No Inlining	Inlining
JMTk SemiSpace	396	610
JMTk MarkSweep	315	575
C malloc	478	—
C calloc	338	—

Table 4. Allocation Rates for JMTk and C

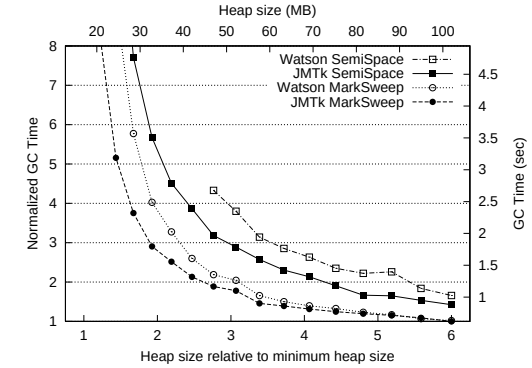
based on usage. Finally, we compare six of JMTk's collectors as a proof of concept indication of JMTk's utility as a memory management research platform.

6.1 Raw Speed Comparisons

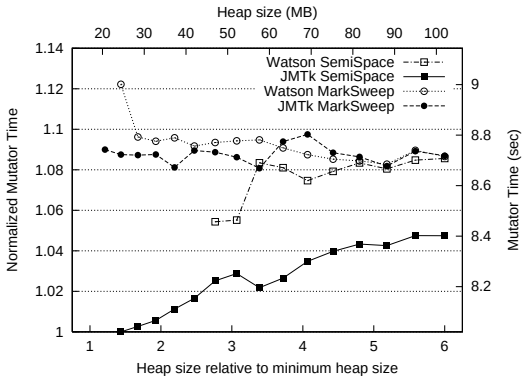
We measure throughput (raw speed) of allocation and tracing on JMTk and the Watson collectors to reveal the runtime cost of our abstractions. The micro-benchmark constructs a binary tree whose nodes have two references fields for the children and two data fields. We compute the allocation rate by allocating 100 MB of unconnected binary nodes, and the tracing rate by tracing 100 MB of a balanced binary tree. Table 6.1 shows the results of these tests on the SemiSpace and MarkSweep as implemented by JMTk and the Watson collectors. The JMTk collectors are slower by 11% and 4% in allocation speed for SemiSpace and MarkSweep, respectively. On tracing rates, the slowdown is 4.7% and 25.8% The slowdown in allocation rates comes directly from the reuse in our abstraction. In particular, the SemiSpace allocation code contains an extra load instruction to retrieve the bump pointer object whereas the corresponding fields in the Watson collectors are manually inserted in an unrelated class as an optimization. The most serious discrepancy in the MarkSweep tracing rate comes from algorithmic differences between JMTk and Watson discussed below. JMTk's more efficient use of space in the MarkSweep policy and in all plans more than makes up for this discrepancy.

We also tested the allocation rates of JMTk and the GNU C library's malloc (ptmalloc version 1.108, which is based on version 2.7.0 of the Lea allocator). Since this version of malloc uses a function call, the fairest comparison is with no inlining in JMTk.[3] On the flip side, since Java returns zeroed memory, calloc rather malloc should be used. Table 4 shows that inlining gives a significant advantage (about 35% to 45%) and zeroing memory does have a significant cost (29%). The best comparison (JMTk MarkSweep - noinline versus C calloc) shows that C has a slight advantage 6.8%.

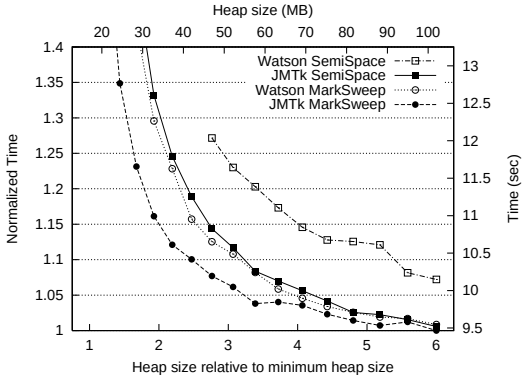
[3]We will manually implement an inlined version of the ptmalloc allocator and compare it with JMTk for the final version of this paper.



(a) GC time



(b) Mutator time



(c) Total time

Figure 3. JMTk versus Watson

6.2 JMTk versus Watson Collectors

Figure 3 compares JMTk and Watson on the benchmarks from Table 2 using the geometric mean on garbage collection, mutator, and total time. Mutator time includes allocation, adaptive compilation, and application time. These results vary the heap size from the minimum in which any collector executes to 6 times the minimum at 16 different points, and are normalized to the best result.

Although the JMTk and Watson collectors are similar in spirit, there are a few key differences. JMTk stores collector meta-data in the heap, whereas Watson collectors do not,

enjoying a small space advantage. Both families directly manage objects larger than 8 KB with a large object space (LOS). These collectors trace the LOS on every collection. Watson's LOS is a next-fit algorithm with page alignment. It does not maintain a free list. To satisfy a request, it sequentially scans through the LOS memory until it finds sufficient contiguous free pages. JMTk uses a free list.

The Watson collectors *statically* divide the heap into areas that hold small, large, and immortal objects based on command line parameters. We experimentally determined the smallest possible parameter for the small, large and immortal spaces. We use the ratio between large and small and a constant immortal setting to give the Watson collectors the best possible command-line parameters at any heap size. In JMTk, a command line parameter sets the total heap size and then JMTk dynamically checks that the sum of the three spaces (and others depending on the collector) does not exceed the specified heap size. JMTk thus enjoys a space advantage during the periods that the program is not using the maximum in the large and immortal space. This advantage accounts for much of the differences in the garbage collection times for both SemiSpace and MarkSweep collectors in Figure 3(a). JMTk SemiSpace runs in smaller heaps than Watson SemiSpace for the same reason. This result is also reflected but dampened in Figure 3(c) since collection time is a fraction of total time. Of course, each of the Watson collectors could have implemented dynamic heap partitioning, but JMTk's modular design provides this feature to every collector without additional implementation effort.

Although JMTk and Watson SemiSpace are nearly equal algorithmically, Figure 3(b) shows a performance advantage in mutator time for JMTk. The advantage of JMTk SemiSpace is strongly correlated with smaller heap sizes, which suggests collection-induced locality as the cause (collection occurs more frequently at smaller heap sizes, compacting the space and improving locality). The different traversal orders used by the JMTk and Watson implementations most likely account for JMTk's advantage.

Algorithmically, the two MarkSweep collector are similar, but the Watson one uses different size classes. It uses powers of two and some additional ones: 12, 20, 84, and 524. This results in worst case internal fragmentation of 50%. Since most objects are small, this worst case is unlikely. However our size classes obtain a perfect fit on all objects less than 64 bytes and have a worst case fragmentation of $1/8$. Because the Watson MarkSweep collector has a one word header, it enjoys a runtime advantage of on average 2% for our benchmarks over the two word header in JMTk. (We plan to implement this optimization in JMTk.)

6.3 Variety of Implementation

Figure 4 presents total execution time for one application, _202_jess, with some of the collector implementations currently available in JMTk, but leaves for future work detailed performance analysis and comparisons. It adds the classic generational (GenCopy, GenMS) and reference counting (RefCount) collectors, as well as a more recent generational reference counting (GenRC) [14]. The generational collectors uniformly enjoy a performance advantage.

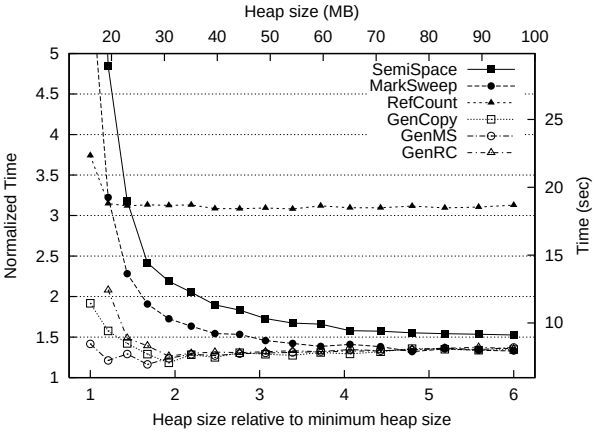


Figure 4. JMTk Collectors on _202_jess

6.4 Detailed Results

7 Conclusion

JMTk is a case study in mixing *performance* and *flexibility* in a systems research context where both are critical. The renewed industrial and academic importance of garbage collection highlights the need for a memory management toolkit where ideas can be rapidly realized and compared without sacrificing performance. Three factors point to JMTk's flexibility: 1) it implements a wide range of collectors, 2) it is being used to develop significant new collectors [14, 29], and 3) code metrics show it is dramatically simpler and more modular than previous implementations. Careful co-operation with an aggressive optimizing compiler allows this flexible design to attain performance benefits as well. On micro-benchmarks JMTk is only 5% slower than a monolithic Java implementation and 60% faster than standard non-inlined malloc in C, while JMTk consistently improves total performance on real benchmarks by up to 20%. This success in mixing flexibility and performance through strong object-oriented design in Java refutes common wisdom about performance critical software and suggests that such an approach can be more widely embraced.

While JMTk has met its initial design goals, there are several areas where further work is necessary. First, the portability of JMTk has to be tested and the work to port it to other systems is ongoing. Second, the parallelism of JMTk is untested and preliminary evidence shows that, at least in some cases, the scalability is noticeably worse than for the Watson collectors. Third, while the design of JMTk is general, the code is somewhat immature and further tuning, in-

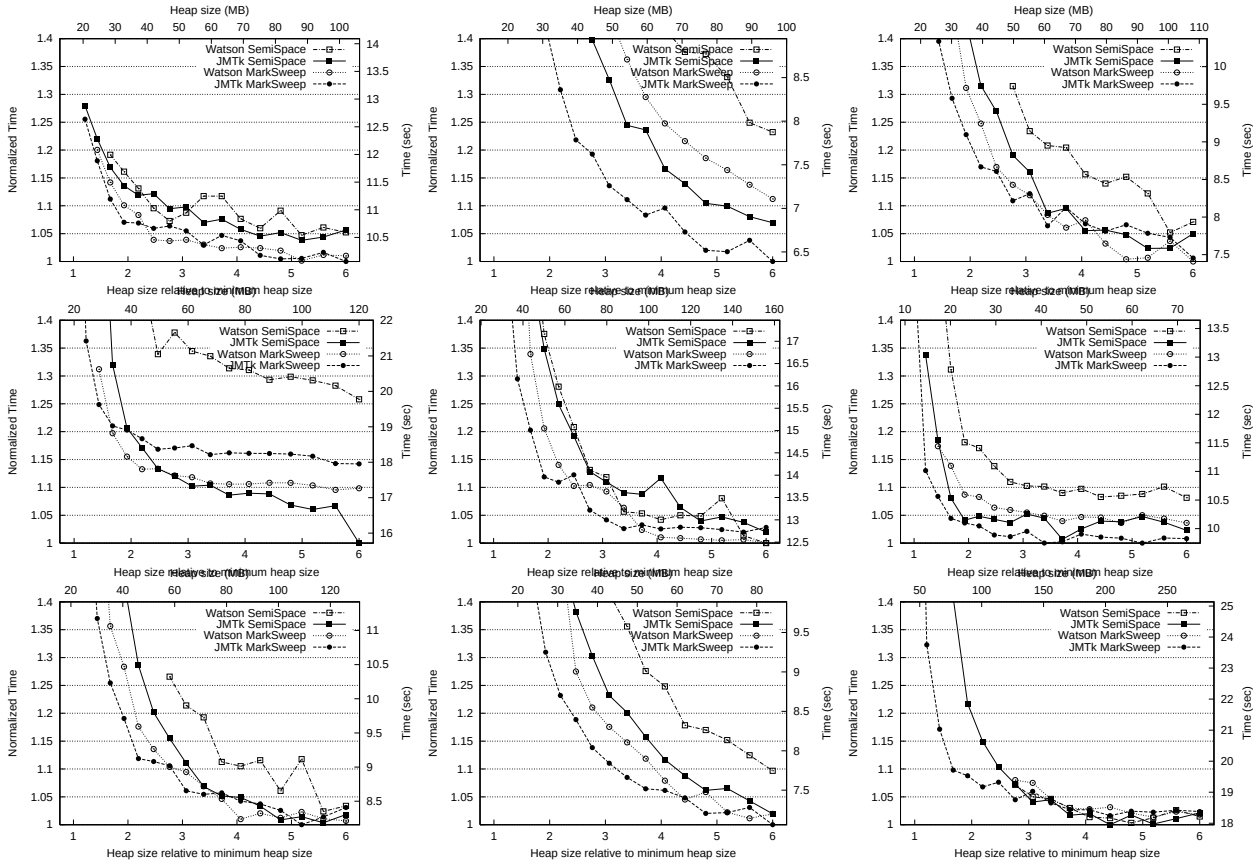


Figure 5. Total Time

cluding the addition of other object models, is needed.

References

[1] B. Alpern et al. Implementing Jalapeño in Java. In *Proc. of the 1999 ACM SIGPLAN Conf. on OOPSLA*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324, Oct. 1999.

[2] B. Alpern et al. The Jalapeño virtual machine. *IBM System 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 *Conf. on OOPSLA*, volume 35(10) of *ACM SIGPLAN Notices*, pages 47–65, 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, 14th International Workshop, LCPC 2001*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[6] G. Attardi and T. Flagella. A customizable memory management framework. In *Proceedings of the USENIX C++ Conference*, Cambridge, Massachussetts, 1994.

[7] G. Attardi, T. Flagella, and P. Iglio. A customizable memory management framework for C++. *Software Practice & Experience*, 28(11):1143–1183, 1998.

[8] H. Azatchi and E. Petrank. Integrating generations with advanced reference counting garbage collectors. In *International Conference on Compiler Construction*, Warsaw, Poland, Apr. 2003. To Appear.

[9] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proc. of the ACM SIGPLAN'01 Conference on PLDI*, volume 36(5) of *ACM SIGPLAN Notices*, June 2001.

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

[11] 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.

[12] E. D. Berger, B. G. Zorn, and K. S. McKinley. Building high-performance custom and general-purpose memory allocators. In *Proceedings of the SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 114–124, Salt Lake City, UT, June 2001.

[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*, volume 37(5) of *ACM SIGPLAN Notices*, Berlin, Germany, June 2002.

[14] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Systems, Anaheim, CA, Oct, 2003*.

[15] S. M. Blackburn and K. S. McKinley. In or out? Putting write barriers in their place. In *Proc. of the Third Int'l Symposium on Memory*

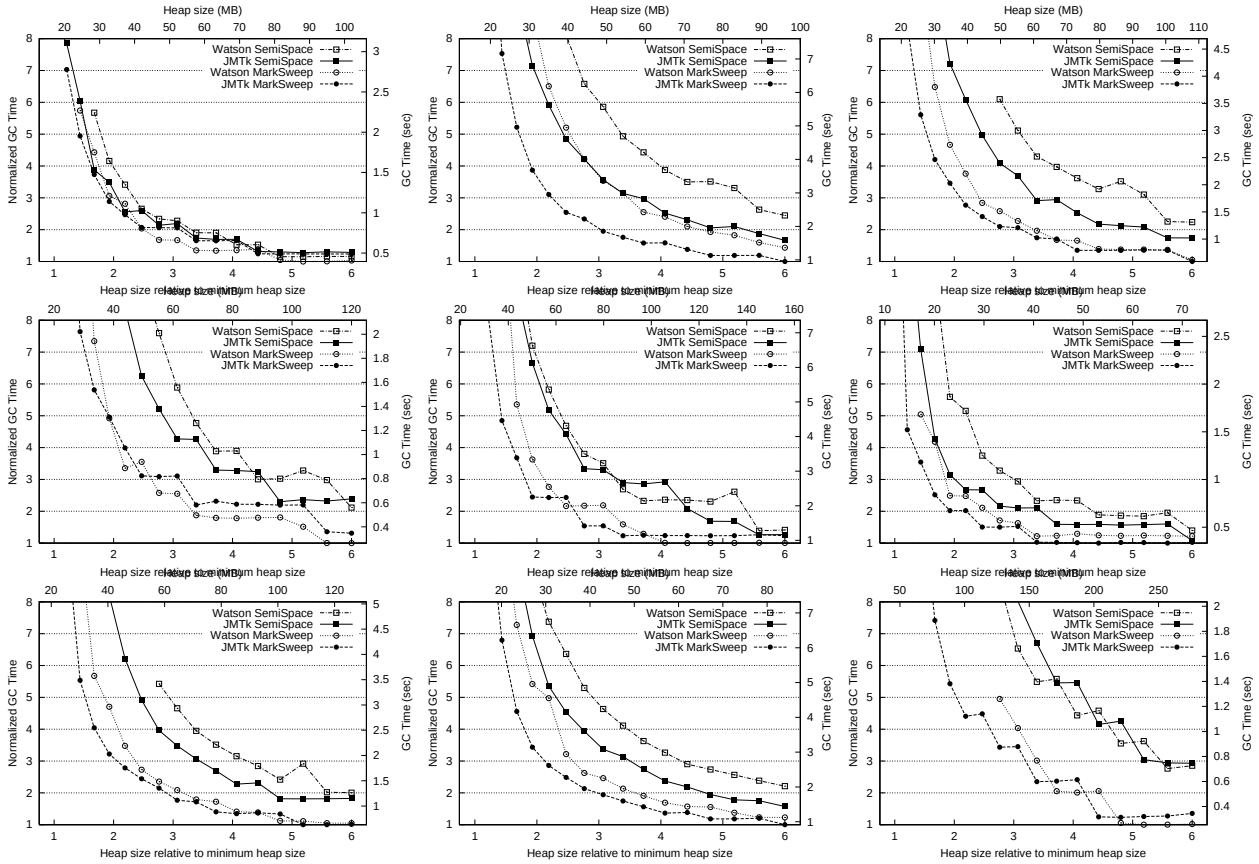


Figure 6. GC Time

Management, volume 37 of *ACM SIGPLAN Notices*. ACM Press, June 2002.

[16] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proc. of the Conf. on OOPSLA / Proc. of the European Conf. on ECOOP*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[17] P. Cheng and G. Belloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN'01 Conference on PLDI*, volume 36(5) of *ACM SIGPLAN Notices*, pages 125–136, June 2001.

[18] J. Dean, G. DeFouw, D. Grove, V. Litinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *ACM Conf. Proc. OOPSLA*, pages 83–100, San Jose, CA, Oct. 1996.

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

[20] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-thefly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, September 1976.

[21] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In T. Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, pages 111–120, Minneapolis, MN, Oct. 2000.

[22] R. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. Technical Report TR-91-47, Dept. of Computer Science, University of Massachusetts, Amherst, Sept. 1991.

[23] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 318–326. ACM Press, 1997.

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

[25] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM Conf. Proc. OOPSLA*, pages 367–380, Tampa, FL, Oct. 2001.

[26] 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.

[27] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[28] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME 03)*, 2003.

[29] N. Sachindran and J. E. B. Moss. Mark-copy: Fast copying GC with less space overhead. In *The ACM SIGPLAN Conference on Object-Oriented Systems, Languages, and Applications*, Oct. 2003.

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

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

[32] 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*, June 2002.

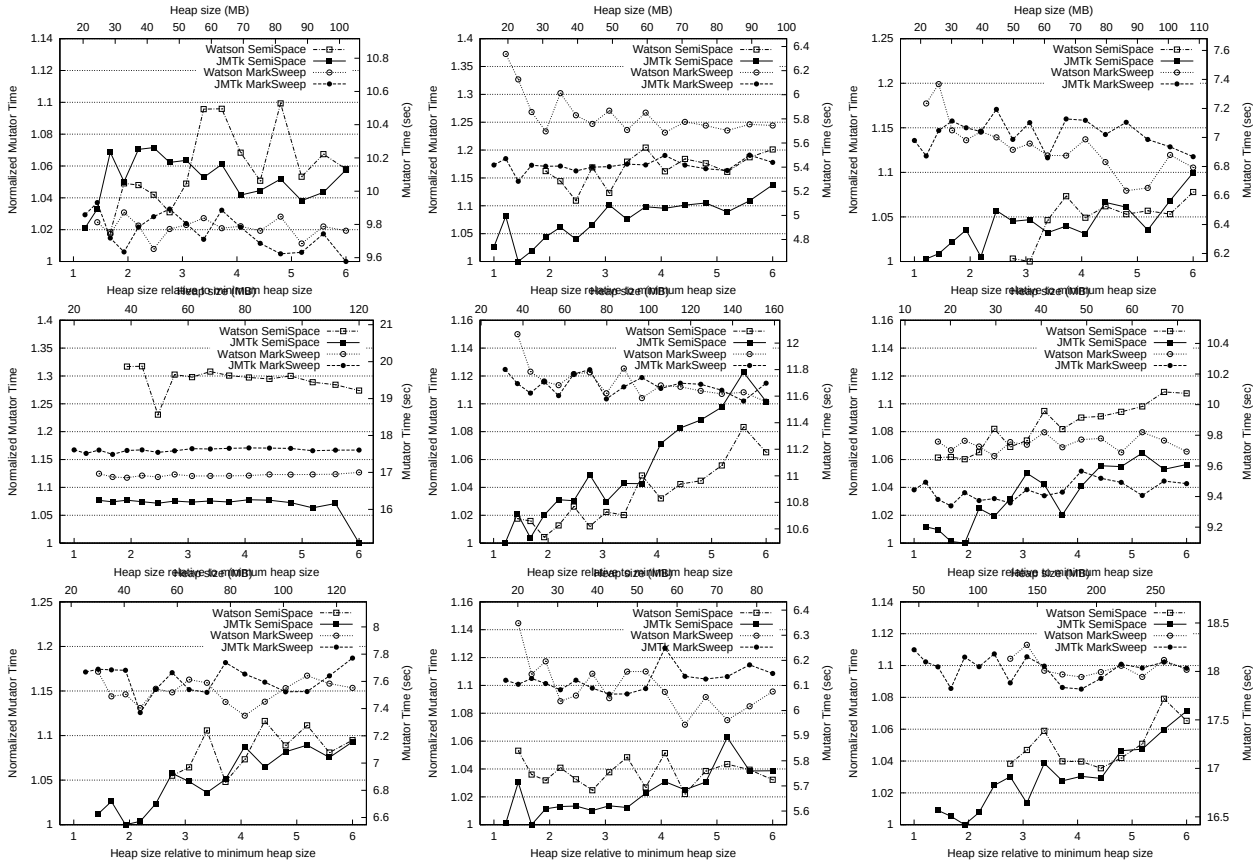


Figure 7. Mutator Time

[33] D. Stefanović, K. McKinley, and J. Moss. Age-based garbage collection. In *ACM Conf. Proc. OOPSLA*, Denver, CO, Nov. 1999.

[34] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly, 2003.

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

[36] K.-P. Vo. Vmalloc: A general and efficient memory allocator. *Software Practice & Experience*, 26(3):1–18, 1996.

[37] S. A. Yeates and M. de Champlain. Design of a garbage collector using design patterns. In C. Mingins, R. Duke, and B. Meyer, editors, *Proceedings of the Twenty-Fifth Conference of (TOOLS) Pacific.*, pages 77–92, Melbourne, 1997.

[38] S. A. Yeates and M. de Champlain. Design patterns in garbage collection. In *Proc. of the 4th Annual Conf. on the Pattern Languages of Programs*, volume 6 "General Techniques", 2-5 1997.