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

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 garbage collection. This paper presents the design, implementation, and evaluation of MMTk, a Memory Management Toolkit for and in Java. MMTk is an efficient, composable, extensible, and portable framework for building garbage collectors. MMTk uses design patterns and compiler cooperation to combine modularity and efficiency. The resulting system is more robust, easier to maintain, and has fewer defects than monolithic collectors. Experimental comparisons with monolithic Java and C implementations reveal MMTk has significant performance advantages as well. 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 and high-level languages.

1 Introduction

The tension between *flexibility* and *performance* pervades systems development. Flexibility assists in rapidly realizing new ideas, and good base performance gives the realizations credibility. This paper is a case study in a systems research context that shows flexibility can actually improve rather than degrade performance.

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 [2, 19, 20, 25, 27, 34], this reliance on it and growing locality effects have made garbage collection research a high priority in academia [13, 17, 26, 30, 32] and industry [8, 9, 10]. Many collector implementations are monolithic and do not share reused components [1, 21]. Performance comparisons across a range of approaches is thus problematic and rare [4, 7, 21].

*This work is supported by NSF ITR CCR-0085792, NSF CCR-0311829, NSF EIA-0303609, DARPA 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 MMTk, a Memory Management Toolkit in Java for Java.[1] MMTk supports a wide range of collectors: *copying*, *mark-sweep*, *reference counting*, *copying generational*, *hybrid generational*, and new ones [13, 15, 30]. We show how MMTk combines good software engineering design with excellent performance by comparing MMTk's code and execution times in Jikes RVM [1], a Java-in-Java Virtual Machine, with monolithic Java and C collector implementations. MMTk is both more succinct and higher performing. Another benefit of good software engineering is a substantially more robust system—within six months MMTk had become more stable and exhibited a lower defect rate than its four year old monolithic counterparts.

MMTk addresses the tension between flexibility and performance with a combination of design features: 1) Java as a systems language, 2) well chosen design patterns, 3) a clean interface between the virtual machine and MMTk, and 4) the composition of policies and mechanisms to define collectors. For correctness, we extend the Java type system to implement memory addresses and operations on them. We use design patterns to efficiently implement succinct composition of policies and correctness in the face of concurrency. MMTk includes a narrow, portable interface between the runtime and memory manager, which abstracts VM-specific object and program representations.

Comparing MMTk to the original, highly tuned monolithic collectors in Jikes RVM [1] reveals a trade-off between flexibility and performance. Source code metrics show that MMTk has a substantially simpler and more modular design than the originals, but implements a wider range and number of collectors. On SPEC benchmarks, MMTk improves *total* performance 5 % to 20% on average over the monolithic collectors due to dynamic heap partitioning. On micro-benchmarks, MMTk performs between 5% and 25% worse on allocation and tracing speed. Together these results demonstrate that total perfomance is better optimized with modular components, even though in theory each com-

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

ponent could be faster in a monolithic system.

On a C implementation of the micro-benchmark, the GNU C allocator improves over MMTk on Java by 6% in the fairest comparison, but when Jikes RVM applies aggressive compiler inlining which the reduced impedance in a Java-in-Java implementation enables, MMTk outperforms the C allocator by 60%. These results suggest slightly augmented Java as a competitive systems language.

The background section next outlines the key mechanisms and policies for readers unfamiliar with garbage collection. Section 3 compares MMTk with other explicit and automatic memory management toolkits [13, 21, 23], none of which combine the diversity of implementation with performance of MMTk. Section 4 then discusses MMTk's design, followed by results, conclusions, and future work. The key contribution of this work is to describe a clean flexible design and implementation of a performance critical component, the memory manager for Java, that practices good software engineering. A surprising result is that this design approach also attains performance benefits.

2 Background

This section describes memory management terms and algorithms, and how MMTk organizes the heap to implement them. For a thorough treatment, see Jones and Lins [25]. 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, are mixed in with the mutator. Collection can run concurrently with mutation, but for simplicity our discussion assumes a separate collection phase.

MMTk groups regions of memory into *spaces* and implements garbage collection algorithms with a *policy* that couples a space with an allocation and collection mechanism. *Whole heap* collectors use one policy for most objects. *Generational* collectors divide the heap into age cohorts, and use one or more policies [2, 34]. 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, the write-barrier conditionally records pointers between independently collected spaces. MMTk 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.

MMTk forms *policies* with these mechanisms.

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

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

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

Immortal space: bump-pointer allocation and no collection.

Large object space: coarse-grained free-list of pages and *treadmill* collection [25].

These policies combine 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 [2] 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 [15] to combine a copying nursery with a RefCount mature space.

3 Related Work

This section compares MMTk with previous garbage collection toolkits [13, 23, 21] and explicit memory management [5, 6, 11, 35] 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 [23]. 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 recent copying collectors such as Older-First [32] or Beltway [13].

GCTk, a more general Garbage Collection Toolkit for Java addressed some of these shortcomings [13, 31]. To our knowledge, GCTk is the only other garbage collection toolkit implemented in Java. This framework provides a single shared implementation of key functions such as scanning and *remembered sets* which record write-barrier entries. GCTk implements copying age-based collectors by separating the collection increment from the heap position [13, 31], but it does not include free-list allocation, nor can it mix and match policies, and it was never intended to be portable. MMTk overcomes these limitations of GCTk. MMTk uses a composable design to mix and match policies and mechanisms. It has free-list memory managers, a large object space, and the composition of disparate policies, none of which GCTk supports. It also creates a portable language/compiler interface that researchers are porting to Rotor (a C# runtime), Open VM (another Java runtime), and a Haskell runtime.

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. MMTk generalizes this pattern and applies it more broadly. MMTk includes a much wider range of collectors and policies than Marmot, is modular and extensible, and works in both just-in-time and ahead-of-time settings.

A few researchers have also built memory management toolkits for explicit memory management of C/C++ programs [5, 6, 11, 35]. Heap layers is the most general and high performance of these frameworks [11]. 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 narrow interface that interacts with the program only through the malloc and free. MMTk has a more complex interface since it interacts with a managed runtime system on many mechanisms including scheduling, write barriers, object models, and root identification. C and C++ limit the memory management policies 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 [5, 6]. These frameworks thus are not and need not be as general as MMTk.

Yeates *et al.* [37] 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

MMTk'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. Our entire design process and the subsequent evolution of MMTk has focused on these goals. We use thin abstractions, target our optimizations carefully (and avoid making them prematurely!), and hold the pre-existing, highly tuned monolithic implementations as our benchmark for performance.

This remainder of this section discusses how the design of MMTk reaches both goals using Java as a systems implementation language. It begins with three issues for Java-in-Java implementations: extensions to the Java type system, compiler pragmas, and the problem of the collector executing within the heap that it is collecting. We also describe design patterns that attain correctness and performance, and then present the portable interface between MMTk and the virtual machine. Section 4.4 outlines MMTk'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 MMTk collectors to the original monolithic implementations. These results demonstrate the benefits of modularity and aggressive software reuse.

4.1 Java as a Systems Language

MMTk 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, 24]. We address these key issues: 1) extensions for manipulating memory directly, 2) exploiting VM compiler pragmas, and 3) safely implementing a collector that executes within the heap it collects (or, 'eating your own dog food').

New Types for Unsafe Operations

MMTk uses a modest extension of Java defined by and developed for Jikes RVM [1]. In order to access and modify memory, we require unsafe operations. MMTk 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 *unboxed* types (operations on them never result in allocation) provide methods for pointer arithmetic, pointer comparison, casts, and memory reads and writes including atomic operations. The memory manager implements its lowest-level operations, such as allocation, with them. A source code transformation ports these idioms to the gcc Java front end (gcj), and other targets such as OVM [29] and Rotor [33].

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 provides support for exact garbage collection which depends on compiler-generated maps that identify pointers stored on the stack.

Executing Within Its Own Heap

MMTk faces the problem that its code and state reside 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! The copying collectors thus must *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. 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 new algorithms from the burden of identifying and acting upon the crucial instances.

4.2 Design Patterns Reused in MMTk

MMTk uses design patterns for efficiency and reuse. The 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 synchronization for efficiency; 3) exploiting collection phases to simplify correct collector construction; and 4) delegating collector actions to specific policies.

Hot and Cold Paths

Wherever appropriate, MMTk applies a pattern that uses efficient, lightweight, unsynchronized mechanisms for frequently executed code (the *hot path*), and periodically uses more heavyweight mechanisms (the *cold path*) by marking the hot path with `PragmaInline` and the cold path with `PragmaNoInline`. MMTk uses this pattern extensively to reduce synchronization frequency, and to allow more complex heuristics.

For example, a copying nursery performs most allocation with an unsynchronized bump pointer, but periodically (every 128KB), the allocator takes the slow path. The slow path synchronously acquires another 128KB chunk of memory since multiple threads contend for a pool of such chunks, and polls the collection subsystem, giving it the opportunity to invoke a collection if necessary. MMTk reuses this pattern on write barriers and internal dynamic data structures such as queues and buffers.

Local and Global Scopes

The correctness and performance of a multi-threaded memory manager depends on a scalable division of the local and global context (i.e., exposure to synchronization or not). In MMTk, scope is overt through the use of classes. For example, MMTk associates an instance with each thread and uses the class to reflect global state. Threads are truly concurrent. Jikes RVM maps program, memory management, and VM threads to kernel threads which reflect the number of physical CPUs. Instance methods operate over their data without synchronization, and access shared state through synchronized global 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, MMTk 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. MMTk synchronizes only accesses to the global state. MMTk 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

MMTk 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*. MMTk 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 implements a prepare and release phases. For instance, a simple copying collector establishes all roots of collection in the prepare phase, and reclaims the space in the release phase. The local/global divisions ensure correct and efficient synchronization between phases.

Multiplexed Delegation

MMTk 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 the memory manager allocates or traces an object, it invokes the corresponding method in the plan, which then *delegates* responsibility to the appropriate policy depending on the object. We call this pattern *multiplexing delegation*. The pattern is reused in each plan for a number of different tasks, such as object allocation, object tracing, and testing object liveness. Figure 1 shows 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 analyze the cyclomatic complexity [28] of the plans (Section 4.5), we find that this pattern captures over 50% of the plans' complexity.

```
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. The Multiplexed Delegation Pattern: The traceObject Method for Generational Collectors.

4.3 The Virtual Machine Interface

Since one of MMTk'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. The interface is bidirectional across the VM (virtual machine) and MM (memory manager) and each side contains *requirements* and *features*. The `VM_Interface` class 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, 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. `VM_Interface` implements these requirements in terms of the VM's feature set. 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 MMTk's queue data structures. MMTk 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.

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. MMTk provides these with the `MM_Interface` class.

4.4 Composition: Mechanisms, Policies, and Plans

At the heart of MMTk 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 MMTk obtains these benefits together with excellent performance. We now outline the key compositional elements in MMTk: *mechanisms*, *policies*, and *plans*.

Mechanisms

MMTk implements a rich set of collector-neutral, highly-tuned mechanisms that it shares 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—55 classes in all.

Policies

MMTk 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. MMTk 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. Thus the key *spatial* and *temporal* elements of memory management policy are overtly captured in the design.

Plans

Most new memory management algorithms are compositions of a small set of well understood policies (Section 2). For example, Ulterior Reference Counting [15] composes reference counting with copying collection. A *Plan* is MMTk's highest level of composition, defining the rules by which policies are composed. Key among these are:

- the set of policies used by the plan,
- the allocation policy for each object, and
- the collection policy for each object (Figure 1).

Each of a plan's 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.

MMTk 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 [15] collector. Researchers are currently adding two more recently published collectors: Beltway [13] and Mark-Copy [30].

	m	NCSS	NCSS/m	BC	BC/m	ΣCCN	LCOM
Watson 2.0.0							
SemiSpace	50	1234	24.7	2223	44.5	325	0.97
MarkSweep	78	2288	29.3	3955	50.7	658	0.98
GenCopy	56	1597	28.5	2696	48.1	422	0.97
GenMS	90	2311	25.7	4719	52.4	633	0.98
Total	**274**	**7430**	**27.1**	**13593**	**49.6**	**2039**	
Watson 2.2.0							
SemiSpace	40	426	10.7	850	21.2	139	0.90
MarkSweep	31	346	11.2	574	18.5	105	0.93
GenCopy	42	659	15.7	1312	31.2	220	0.93
GenMS	40	531	13.3	1294	32.4	171	0.88
Total	**153**	**1962**	**12.8**	**4030**	**26.3**	**635**	
MMTk							
SemiSpace	30	237	7.9	463	15.4	98	0.93
MarkSweep	29	240	8.3	421	14.5	89	0.93
GenCopy	19	117	6.2	198	10.4	49	0.89
GenMS	18	104	5.8	158	8.8	43	0.86
Generational	36	279	7.8	434	12.1	102	0.96
Total	**132**	**977**	**7.4**	**1674**	**12.7**	**383**	

Table 1. Methods (m), Non-comment Source Statements (NCSS), Bytecodes (BC), Total Cyclomatic Complexity (ΣCCN), and Lack of Cohesion Of Methods (LCOM) for Four Garbage Collectors in Three Systems

Figure 2 illustrates the composition mechanism discussed in this section with a UML class diagram with multiplicity and association information. The bold box is the per-thread instance through which most memory-related requests are serviced. The central spine shows the inheritance relationship of the plans while the clusters of instances to the side represent different memory regions. For example, the cluster emanating from the BasePlan corresponds to the immortal memory region that holds objects with an immortal lifetime while the MarkSweep cluster coming from the GenerationalMS plan is the distinguishing feature of the GenMS collector. In each cluster, we see both the hot-cold pattern illustrated by the annotated paths and local-global pattern shown by the multiplicity annotation. The overall 4-way multiplexing or composition is handled by the plan hierarchy.

4.5 Exploiting Java's Features in MMTk

This section evaluates how well MMTk 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 MMTk 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, total cyclomatic complexity [28], and LCOM (Lack of Cohesion of Methods) [22] for each of the three systems. MMTk 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 MMTk attains an average method cyclomatic complexity [28] 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] 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 MMTk 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].

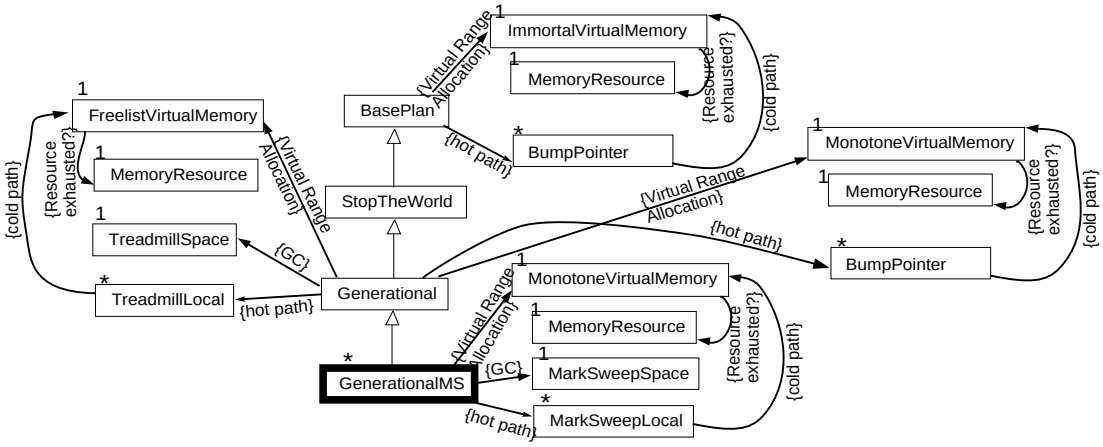


Figure 2. Composition of GenMS.

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 samples to select hot methods that it then optimizes [3]. Adaptive compilation introduces variations in the load on the garbage collector and program behavior due to its statistical choices. Different collectors compound this variation since the adaptive compilation of different write-barriers is part of the runtime system as well as the program [14].

We perform experiments on a 2.0 GHz Intel P4 Xeon, with a 64 byte L1 and L2 cache line size, 8KB 4-way set associative L1 data cache, 12K L1 instruction trace cache, 512KB unified 8-way set associative L2 on-chip cache, and 1 GB main 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 adaptive compiler's variability.

Table 2 shows benchmarks characteristics using fast adaptive compilation for eight SPEC JVM benchmarks, and pseudojbb, a variant of SPEC JBB2000 that executes a fixed number of transactions to hold constant the garbage collection load. The *alloc* column 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 MMTk 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. We configure the Watson collectors to allocate 1MB to immortal space and remaining space to the small and large spaces according to the ratio in column 3 for each heap size.

	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

6 Results

We first compare MMTk SemiSpace and MarkSweep with the original highly tuned Jikes RVM *Watson* collectors on micro-benchmarks and larger benchmarks. MMTk's abstractions cost about 5 to 25% compared to the Watson collectors in raw speed of the mechanisms. We also compare MMTk on Java micro-benchmark with a standard C malloc implementation on the same micro-benchmark written in C to reveal whether Java is a suitable systems language. MMTk actually attains better performance due to the Jikes RVM compiler's inlining, a Java in Java feature, without this advantage, C out performs MMTk by 6%. On standard benchmarks, MMTk attains significantly better total performance than the Watson collectors largely because all its collectors reuse a dynamic heap partitioning algorithm instead of the Watson collectors' static partitioning.

More broadly, MMTk's performance advantage is a direct result of good component design that it reuses among collectors, coupled with fast, but not the fastest mecha-

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

Table 3. Allocation and Tracing Rates

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

Table 4. Allocation Rates for MMTk and C

nisms. Building identical, faster monolithic collector is certainly possible, but in MMTk high-performance implementations are more likely.

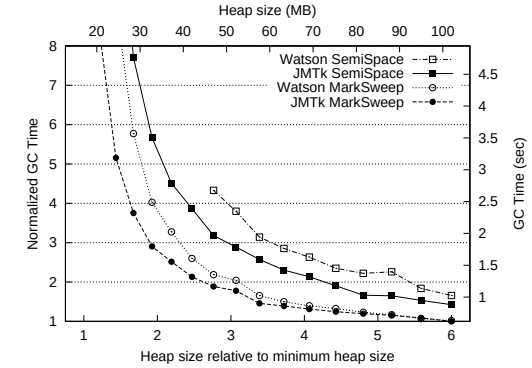
6.1 Raw Speed Comparisons

We measure throughput (raw speed) of allocation and tracing to reveal the runtime cost of our abstractions. Our Java 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 compares Watson and MMTk, SemiSpace and MarkSweep collectors. The MMTk 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 allocation difference on SemiSpace comes directly from the reuse in our abstraction: 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 MMTk and Watson discussed below.

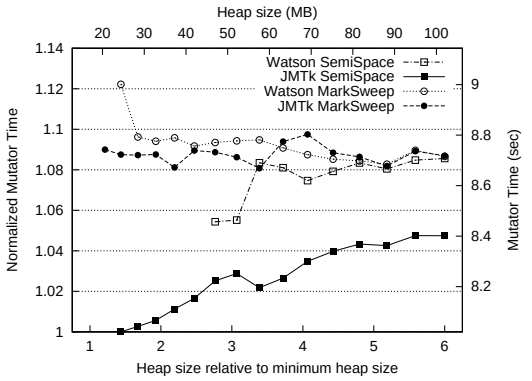
We also compare MMTk on the Java micro-benchmark to the GNU C library's malloc (ptmalloc version 1.108, which is based on version 2.7.0 of the Lea allocator in single threaded mode) on a C version of the same micro-benchmark. Since this version of malloc uses a function call, the fairest comparison is with no inlining in MMTk. 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 closest comparison (MMTk MarkSweep - noinline versus C calloc) shows that C has a slight advantage 6.8%.

6.2 MMTk versus Watson Collectors

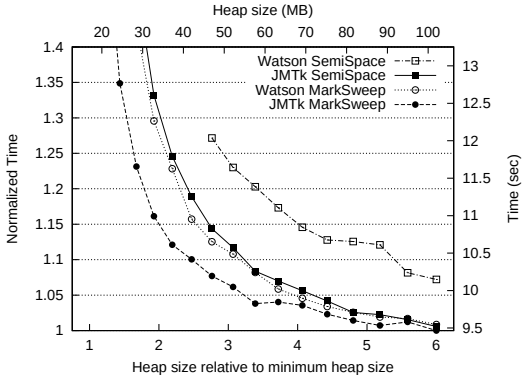
Figure 3 compares MMTk and Watson on the benchmarks from Table 2 using the geometric mean on garbage collection, mutator, and total time. Mutator time includes allocation, adaptive compilation, and application time. These results vary the heap size from the minimum in which any collector executes to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



(a) GC time



(b) Mutator time



(c) Total time

Figure 3. MMTk versus Watson

Although the MMTk and Watson collectors are similar in spirit, there are a few key differences. MMTk 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) and 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. MMTk uses a free list.

The Watson collectors *statically* divide the heap into areas for small, large, and immortal objects based on command line parameters. We experimentally determined the smallest 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 MMTk, a command line parameter sets the total heap size and then MMTk dynamically checks that the sum of the spaces does not exceed the specified heap size. MMTk thus enjoys a space advantage during the periods that the program is not using the maximum in the large or immortal space. This advantage accounts for much of the differences in garbage collection times for both SemiSpace and MarkSweep collectors in Figure 3(a). MMTk 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 implement dynamic heap partitioning, but MMTk's modular design provides feature to every collector without additional implementation effort.

Although MMTk and Watson SemiSpace are close algorithmically, Figure 3(b) shows a performance advantage in mutator time for MMTk. The advantage of MMTk 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). Watson uses the standard breadth-first copying order, whereas our other experiments show and these results confirm the superiority of a depth-first ordering used by MMTk [36].

Algorithmically, the two MarkSweep collector are similar, but Watson 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 Watson MarkSweep has a one word header, it enjoys a runtime advantage of on average 2% for our benchmarks over the two word header in MMTk. (We plan to implement this optimization.)

6.3 Variety of Implementation

Figure 4 presents total execution time for one application, _202_jess, with some of the collectors currently available in MMTk, but leaves for other work detailed performance analysis and comparisons [12]. The Figure shows MarkSweep, SemiSpace, classic generational (GenCopy, GenMS), reference counting (RefCount), and generational reference counting (GenRC) [15] collectors. The generational collectors uniformly enjoy a performance advantage over the full heap collectors.

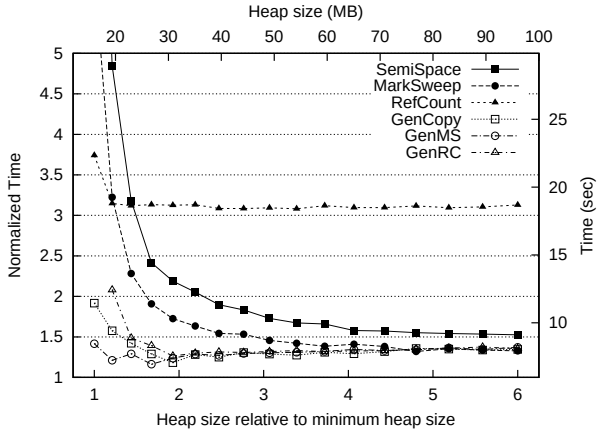


Figure 4. MMTk Collectors on _202_jess

7 Conclusions and Future Work

MMTk is a case study in mixing *performance* and *flexibility* in systems research where both are critical. The renewed interest in 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 MMTk's flexibility: 1) a wide range of collector implementations, 2) new collector implementations [15, 30], and 3) code metrics demonstrate a simple and modular design. Careful cooperation with an aggressive optimizing compiler couples this design with high-performance. On micro-benchmarks MMTk is 5% to 25% slower than a monolithic Java implementation, but 60% faster than standard non-inlined calloc in C. However, MMTk consistently improves total performance on real benchmarks by up to 20% over monolithic implementations because the clean design results in better memory management algorithms and mechanisms, and their broad application. Furthermore, the cleaner design yields a substantially more robust system that is easier to maintain and has fewer defects. This success refutes common wisdom about performance critical software in Java and suggests this approach can be more widely embraced.

Ongoing work on MMTk should fully evaluate its portability. In addition, preliminary evidence shows that the scalability of parallel collection in MMTk is sometimes noticeably worse than for the Watson collectors. Finally, MMTk is relatively immature and further tuning, including the addition of other object models, is needed.

References

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

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

[3] 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.

[4] 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.

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

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

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

[8] 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 *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 92–103, Snowbird, UT, June 2001.

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

[10] 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.

[11] 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.

[12] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *SIGMETRICS*, NY, NY, June 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] G. Bracha and W. Cook. Mixin-based inheritance. In *ACM Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, Ottawa, Canada, 1990.

[17] P. Cheng and G. Blelloch. A parallel, real-time garbage collector. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 125–136, Snowbird, UT, June 2001.

[18] J. Dean, G. DeFouw, D. Grove, V. Litinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 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-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, September 1978.

[21] 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.

[22] B. Henderson-Sellers. *Object-oriented metrics — measures of complexity*. Prentice-Hall, New Jersey, 1996.

[23] 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.

[24] 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 *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 318–326, Atlanta, GA, Oct. 1997.

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

[26] 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.

[27] 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.

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

[29] 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*, San Diego, CA, June 2003.

[30] N. Sachindran and J. E. B. Moss. Mark-Copy: Fast copying GC with less space overhead. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 326–343, Anahiem, CA, Oct. 2003.

[31] 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.

[32] D. Stefanović, K. McKinley, and J. Moss. Age-based garbage collection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 370–381, Denver, CO, Nov. 1999.

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

[34] 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.

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

[36] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 177–191, Toronto, Canada, June 1991.

[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) Pacifi c*, pages 77–92, Melbourne, 1997.