Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance *

Stephen M. Blackburn
Australian National University
Steve.Blackburn@anu.edu.au

Kathryn S. McKinley
The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Programmers are increasingly choosing managed languages for modern applications, which tend to allocate many short-to-medium lived small objects. The garbage collector therefore directly determines program performance by making a classic space-time tradeoff that seeks to provide space efficiency, fast reclamation, and mutator performance. The three canonical tracing garbage collectors: semi-space, mark-sweep, and mark-compact each sacrifice one objective. This paper describes a collector family, called *mark-region,* and introduces *opportunistic* defragmentation, which mixes copying and marking in a single pass. Combining both, we implement *immix*, a novel high performance garbage collector that achieves all three performance objectives. The key insight is to allocate and reclaim memory in contiguous regions, at a coarse *block* grain when possible and otherwise in groups of finer grain *lines*. We show that immix outperforms existing canonical algorithms, improving total application performance by 7 to 25% on average across 20 benchmarks. As the mature space in a generational collector, immix matches or beats a highly tuned generational collector, e.g. it improves jbb2000 by 5%. These innovations and the identification of a new family of collectors open new opportunities for garbage collector design.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)
General Terms Algorithms, Experimentation, Languages, Performance, Measurement
Keywords Fragmentation, Free-List, Compact, Mark-Sweep, Semi-Space, Mark-Region, Immix, Sweep-To-Region, Sweep-To-Free-List

1. Introduction

Modern applications are increasingly written in managed languages and make conflicting demands on their underlying memory managers. For example, real-time applications demand pause-time guarantees, embedded systems demand space efficiency, and servers demand high throughput. In seeking to satisfy these demands, the literature includes reference counting collectors and three canonical tracing collectors: semi-space, mark-sweep, and mark-compact. These collectors are typically used as building blocks for more sophisticated algorithms. Since reference counting is incomplete, we omit it from further consideration here. Unfortunately, the tracing collectors each achieve only two of: *space efficiency*, *fast collection*, and *mutator performance* through contiguous allocation of contemporaneous objects.

Figure 1 starkly illustrates this dichotomy for full heap versions of mark-sweep (MS), semi-space (SS), and mark-compact (MC) implemented in MMTk [**?**], running on a Core 2 Duo. It plots the geometric mean of total time, collection time, mutator time, and mutator cache misses as a function of heap size, normalized to the best, for 20 DaCapo, SPECjvm98, and SPECjbb2000 benchmarks, and shows 99% confidence intervals. The crossing lines in Figure 1(a) illustrate the classic space-time trade-off at the heart of garbage collection. Mark-compact is uncompetitive in this setting due to its overwhelming collection costs. In smaller heap sizes, the space and collector efficiency of mark-sweep perform best since the overheads of garbage collection dominate total performance. Figures 1(c) and 1(d) show that the primary advantage for semi-space is 10% better mutator time compared with mark-sweep, due to better cache locality. Once the heap size is large enough, garbage collection time reduces, and the locality of the mutator dominates total performance so semi-space performs best.

To explain this tradeoff, we need to introduce and slightly expand memory management terminology. A tracing garbage collector performs *allocation* of new objects, *identification* of live objects, and *reclamation* of free memory. The canonical collectors all identify live objects the same way, by marking objects during a transitive closure over the object graph.

Reclamation strategy dictates allocation strategy, and the literature identifies just three strategies: (1) *sweep-to-free-list*, (2) *evacuation*, and (3) *compaction*. For example, mark-sweep collectors allocate from a free list, mark live objects, and then *sweep-to-free-*

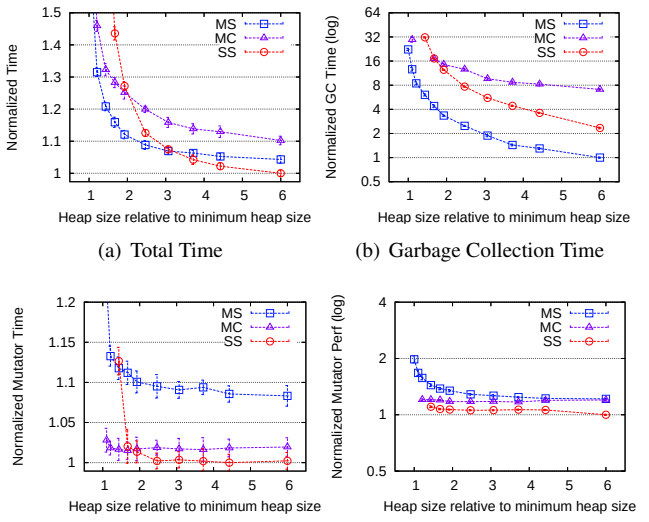


Figure 1. Performance Tradeoffs For Canonical Collectors: Geometric Mean for 20 DaCapo and SPEC Benchmarks.

* This work is supported by ARC DP0666059, NSF CNS-0719966, NSF CCF-0429859, NSF EIA-0303609, DARPA F33615-03-C-4106, Microsoft, Intel, and IBM. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

[Copyright notice will appear here once 'preprint' option is removed.]

list puts reclaimed memory back on the free list [**?**, **?**, **?**]. Because mark-sweep collection is non-moving, it is time and space efficient, but cannot provide locality for contemporaneously allocated objects. Semi-space, older-first, garbage-first, and others *evacuate* by moving all live objects to a *new* space, reclaiming the old space en masse [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. Mark-compact, the compressor, and others *compact* by moving all live objects to one end of the *same* space, reclaiming the unused portion en masse [**?**, **?**, **?**, **?**]. Evacuation and compaction strategies provide *contiguous allocation*, which puts contemporaneously allocated objects next to each other in space, and thus offer good mutator locality. However, evacuation incurs a 2 $\times$ space overhead and in-place compaction is time inefficient because it requires multiple passes over the heap.

This paper identifies the *mark-region*family of non-moving tracing collectors. Mark-region uses the *sweep-to-region*strategy, which reclaims contiguous free regions and thus provides contiguous allocation. To reduce fragmentation in mark-region and other non-moving collectors, this paper introduces *opportunistic* defragmentation which mixes copying and marking in a single pass. Using these building blocks, we introduce *immix*, a novel high performance garbage collector that combines mark-region with opportunistic defragmentation to achieve space efficiency, fast collection, and contiguous allocation for mutator performance.

In a mark-region collector, region size embodies the collector's space-time tradeoff. Large contiguous regions maximize mutator performance and minimize collection cost, but are space inefficient since a single object can withhold an entire region. Small regions increase space efficiency but increase collection time, reduce mutator performance, and make placement of large objects harder. Immix balances these concerns by preferentially managing memory at a coarse *block* grain and falling back to a fine *line* grain when necessary. Immix reclaims completely free 32KB fixed size blocks, and groups of free 128B lines in partially populated blocks. Immix contiguously allocates objects in empty and partially filled blocks.

Immix addresses fragmentation, a problem for all non-moving collectors, with demand-driven, *opportunistic* evacuation. When defragmentation is necessary, immix chooses source blocks (to evacuate) and target blocks (to allocate into) prior to collection. It evacuates objects from source blocks, leaving forwarding pointers. Evacuation is opportunistic because if immix runs out of space, it stops evacuating. If an object is pinned by the application or is on a target page, immix simply marks the object, leaving it in place.

We evaluate immix as a full heap collector and in two composite collectors. Comprehensive performance measurements show that immix consistently outperforms the best of the canonical collectors by 5 to 25% in total performance on average across many heap sizes, rarely degrades performance, and improves one benchmark by 75%. To measure space efficiency, we compute the minimum heap size in which each algorithm executes. Immix requires on average only 3% more memory than mark-compact, and it requires 15% *less* than mark-sweep. Furthermore, it matches the mutator locality of semi-space and the collection time of mark-sweep.

We build a generational collector with an evacuating semi-space nursery and an immix mature space. Although we did not tune immix for the mature space, this collector performs slightly better on 20 benchmarks than an evacuating semi-space nursery and mark-sweep mature space, Jikes RVM's best performing production generational collector. However, it significantly improves several interesting benchmarks, e.g., total time for SPECjbb2000 improves by 5% or more on all heap sizes we tested. We also design and implement an in-place generational variant based on the sticky mark bits algorithm of Demmers et al. to support pinning [**?**]. We made these collectors publicly available in Jikes RVM.

These results show that mark-region collectors coupled with defragmentation offer an interesting direction for further exploration.

2. Related Work

Researchers have previously addressed the tension between mutator performance, collection speed, and space efficiency. However, we are unaware of any published work which describes or evaluates a mark-region collector. We define mark-region and the sweep-to-region strategy on which it is based. We implement and provide detailed analysis of immix, a high performance collector which combines mark-region collection and opportunistic defragmentation.

Mark-Region Collectors. We believe that there are two previous implementations of mark-region collectors, a JRockit [**?**] collector and an IBM [**?**] collector. Our descriptions here are based on web sites where neither are described in detail or evaluated, and the original JRockit web site is no longer available. The JRockit collector uses a range of coarse grain block sizes, and only recycles completely free blocks for allocation. To limit fragmentation, it compacts a fraction of the blocks at every garbage collection, incurring multi-pass overheads for that fraction of the heap.

The IBM collector uses 'thread local heaps' (TLH), which are unsynchronized, variable-sized, bump-allocated regions with a minimum size of 512 bytes [**?**]. The allocator uses a TLH for objects smaller than 512 bytes, and for larger objects only when they fit in the current TLH. The collector uses a global synchronized free list to allocate both TLHs and large objects. The allocation and mark phase each use bit vectors at an 8 byte resolution. When the allocator fills a TLH, it scans the TLH and records the start positions of each of the objects in the 'allocbits' bit vector. Immix does not use allocation bits, which impose avoidable overhead. The mark phase records live objects in a 'markbits' vector, which requires synchronization that immix avoids. The sweep phase compares the markbits and allocbits vectors to identify free memory. The collector places contiguous free memory greater than 512 bytes on the free list for subsequent use as either a TLH or large object. It does not recycle 'dark matter', regions less than 512B.

Immix differs in a number of regards. Our metadata overhead is lower (0.8% compared to 3.1%). The IBM collector requires 8-byte alignment and an extra header word in each object that stores the object size (typically around 10% space overhead). Immix is structured around aligned 128B lines and 32KB blocks, and can reclaim space at 128B granularity. The IBM collector does not reclaim regions smaller than 512B, but can apparently reclaim space at 8 byte alignment. The IBM collector addresses fragmentation using compaction, described by Borman [**?**] as 'complex', while immix uses evacuation to achieve lightweight and opportunistic defragmentation. The IBM mark-region collector and its derivatives have been referred to in publications [**?**, **?**], but to date have not been described or evaluated in a publication.

Mark-Sweep Variants. Spoonhower et al. [**?**] take a different approach, developing a collector which dynamically applies mark-sweep and semi-space policies to regions of the heap, using page residency counts. Immix applies mark-region uniformly to all of memory. Vam [**?**] is designed to provide contiguous allocation when possible for non-moving explicit memory management. If a block becomes fragmented, Vam must revert to size-segregated free-lists which make no guarantees about locality. The Metronome real-time garbage collector attains pause time guarantees, space guarantees, and collector efficiency using mark-sweep, but not mutator locality [**?**, **?**]. Metronome's mostly non-copying on-demand defragmentation is most similar to immix's defragmentation. However, it requires exact fragmentation statistics for each size-segregated block from a previous collection to evacuate objects from one block to another. Whereas, opportunistic defragmentation adds the ability to evacuate as dynamically possible and without first computing precise fragmentation statistics.

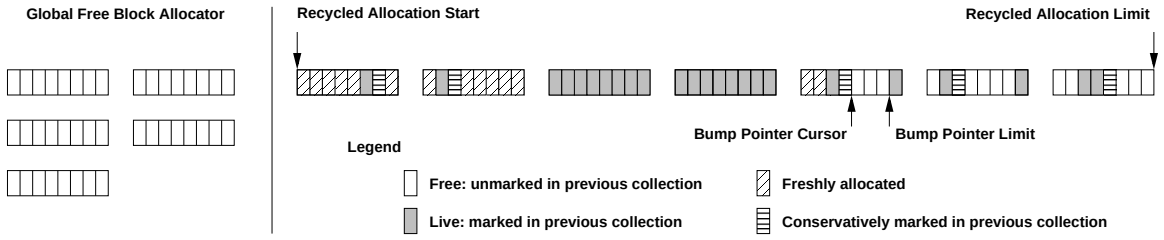


Figure 2. Immix Heap Organization

Space Efficiency and Contiguous Allocation. Approaches to improve the space efficiency of contiguous heap organization, such as mark-compact, MC^2, and copy size reduction [**?**, **?**, **?**, **?**], all may substantially increase garbage collection costs. The garbage-first collector [**?**] bump allocates into empty 1MB fixed sized regions. Each collection selectively evacuates the most fragmented regions. It relies on a concurrent tracing collector to identify garbage and bi-directional remembered sets to evacuate regions in arbitrary order. Although Kermany and Petrank introduced an efficient two pass compaction algorithm [**?**] which uses a single compact pass *after* a mark pass, immix opportunistic evacuation performs defragmentation in one pass, the mark pass. Immix then performs a separate sweep pass to find free blocks. Immix always performs fine-grained line sweeps lazily. Kermany and Petrank implemented in Jikes RVM, but their source is not available. They show improvements over mark-sweep but are not competitive with Jikes RVM's high performance generational production collector, except on one benchmark, jbb2000. We match and sometimes outperform the Jikes RVM production collector even with our full heap algorithm. Three immix variants, including the most simple full heap algorithm, substantially outperform Jikes RVM's production collector on jbb2000 (see Figure 4(c)).

Generational Collectors. High performance generational collectors resolve the tensions between reclamation strategies with composite algorithms. State-of-the-art collectors use an evacuating nursery that provides mutator locality to the young objects, and typically a mark-compact or mark-sweep mature space for collection and space efficiency, together with occasional mature space compaction [**?**]. However, some application domains, such as real-time, have yet to benefit much from generational collection [**?**].

3. Immix: Mixing Locality, Speed, and Efficiency

A naive mark-region implementation is straightforward. Memory is divided into fixed sized regions, each of which is either free or unavailable. The allocator bump allocates into free regions until all free regions are exhausted, triggering a collection. The collector marks any region containing a live object as unavailable and all others as free. Objects may not span regions.

This simple algorithm motivates the design of immix and raises two questions: (1) *How big should the regions be?* Large regions are space-inefficient since a single small object can withhold an entire region, while small regions increase space efficiency but increase collection time. (2) *How to defragment?* An entire region is unavailable as long as any object within it remains alive, so defragmentation is essential.

Immix addresses the dilemma of region sizing by operating at two levels, coarse grained *blocks* and fine grained *lines*. By reclaiming at line granularity, it recycles partially used blocks. When the allocator recycles a block, it skips over unavailable lines, allocating into contiguous free lines. Objects may span lines, but cannot span blocks. Immix addresses fragmentation by using lightweight opportunistic evacuation, which is folded into marking when defragmentation is necessary, as described in Section 3.2.

3.1 Efficient Mark-Region Design

The base mark-region algorithm used by immix is quite simple.

Initial Allocation. All blocks are initially empty. A thread-local allocator obtains an empty block from the global allocator, and bump allocates objects into the block. When the allocator exhausts the block, it requests another block and repeats until the entire heap is exhausted, which triggers a collection.

Identification. The collector traces the live objects by performing a transitive closure starting with the application roots. As immix traces the object graph, it marks objects and lines in a line map.

Reclamation. When immix completes the trace, it performs a *coarse-grained sweep*, linearly scanning the line map and identifying entirely free blocks and free lines in partially free blocks. It returns entirely free blocks to a global pool, and it *recycles* partially free blocks for the next phase of allocation.

Steady State Allocation. Threads resume allocation into recycled blocks, provided on demand, in address order, and skip over completely full and completely empty blocks. The thread-local allocator linearly scans the line map of each recycled block until it finds a hole (one or more contiguous free lines). The allocator then bump allocates into that hole. Once it exhausts the hole, the it continues its linear scan for another hole until it exhausts the recyclable block. It then requests another block. Once the allocators have exhausted all recyclable blocks, they allocate into empty blocks until the entire heap is exhausted.

Figure 2 illustrates the basic immix heap organization during steady state allocation. Just like a semi-space allocator, the allocator maintains a cursor and a limit, as shown in the figure, pointing into block five. The limit for immix is either the next occupied line, or the end of the block. When the cursor would exceed the limit, immix finds the next hole in the block, and resets the cursor and the limit. If there is no hole, it requests another block.

3.1.1 Mark-Region Details and Policies

The above algorithm is relatively easy to implement. However, we found that for high performance over a wide range of workloads, we needed to consider and tune the following policies and mechanisms.

Recycling Policy. At the end of a collection, each block is either *free*, *recyclable*, or *unavailable*. The collector marks a partly used block with at least F free lines as recyclable. We explored various thresholds for immix but found that the most simple policy of $F = 1$ worked best. Section 5.4 shows that the overhead of recycling blocks with few available lines is only incurred regularly when space is very scarce, which is precisely when such diligence is rewarded.

Allocation Policy. Immix allocates into recyclable blocks in address order. It saves allocating into completely free blocks until last because multiple consumers may compete for free blocks and they

offer more flexibility. For instance, the thread-local allocators and the LOS (large object space) compete for pages. The LOS requires completely free pages. Inspired by prior work [**?**, **?**, **?**], we explored other allocation orderings in an effort to maximize the likelihood of reclaiming completely free blocks at collection time. In the full heap setting, we found the simple address order processing worked best. We did not revisit these policies for a generational setting and believe they are an interesting avenue for future work.

Parallelism. Our immix implementations are parallel. They support multiple collector threads and multiple mutator threads. Our implementations however do not perform concurrent collection in which the mutator and collector execute in parallel. To achieve scalable parallel allocation and collection throughout immix, we follow a design pattern common to many systems. The design pattern maximizes fast, unsynchronized thread-local activities and minimizes synchronized global activities. Thus, the synchronized global allocator gives blocks to the thread-local allocators (TLA), which are unsynchronized. Similarly, our base mark-region collector requires hardly any synchronization. The transitive closure is performed in parallel. It allows races to mark objects since at worst an object will be marked and scanned more than once. Since updating nearby bits in a bitmap can generate a race, we use bytes to represent line marks, incurring a 1/128 metadata overhead, but avoiding synchronization. Immix performs coarse-grained sweeping in parallel by dividing the heap among the available collector threads. Our evaluation demonstrates that immix performs very well on uniprocessors and a two-way machine, but we leave to future work a detailed study of parallelism.

Demand Driven Overflow Allocation. We found in an early design that the allocator occasionally skipped over and wasted large numbers of holes when trying to allocate *medium* objects into fragmented blocks. We define a medium object as greater than a line. To address this problem, we implement *demand driven overflow allocation*. We pair each immix allocator with an overflow allocator that is also a contiguous allocator, however it always uses empty blocks. If immix cannot accommodate a medium object in the current block, but there are one or more free lines between the bump pointer cursor and limit, immix allocates it with the overflow allocator. Thus, immix allocates medium objects in the overflow blocks only on demand. For example in Figure 2, there are three free lines between the cursor and the limit. If the mutator requests an object whose size exceeds three lines, immix triggers overflow allocation to avoid wasting three lines. Since the vast majority of objects in Java programs tend to be small [**?**] and lines accommodate several objects on average, this optimization improves space efficiency. It is extremely effective at combating, dynamically on demand, the pathological effect of occasional medium object allocations into a recycled block dominated by small holes. Section 5.4 shows that overflow allocation improves performance in memory constrained heaps that tend to fragment more.

3.2 Defragmentation: Lightweight Opportunistic Evacuation

A pure mark-region collector is non-moving and thus subject to fragmentation. Both *evacuation* [**?**, **?**] and *compaction* [**?**] can be effective defragmentation mechanisms. Immix uses *opportunistic evacuation*. At the start of each collection, immix determines whether to defragment, e.g., based on fragmentation levels. If so, immix uses statistics from the previous collection to select defragmentation candidates and evacuation targets.

When the collector encounters a live object in a candidate block, it *opportunistically* evacuates the object. It only evacuates an object if the application has not pinned it and prior evacuation has not exhausted all target space. If an object is unmovable when the collector encounters it, the collector marks the object and line as live and leaves it in place. Otherwise, the collector evacuates the object to a new location on a target block and leaves a forwarding pointer which records the address of the new location. If the collector encounters subsequent references to a forwarded object, it replaces them with the value of the object's forwarding pointer.

To evacuate an object, the collector uses the same allocator as the mutator, continuing allocation right where the mutator left off. Once it exhausts any unused recyclable blocks, it uses any completely free blocks. The collector of course does not allocate into defragmentation candidate blocks. By default, immix sets aside a small number of free blocks that it never returns to the global allocator and only ever uses for evacuating. This *headroom* eases defragmentation and is counted against immix's overall heap budget. By default immix reserves 2.5% of the heap as compaction headroom, but Section 5.4 shows immix is fairly insensitive to values ranging between 1 and 3%.

3.2.1 Defragmentation Policies and Details

This section describes additional details of opportunistic defragmentation, including parallelism, and pinning.

Candidate Selection. A variety of heuristics may select defragmentation candidates. A simple policy would target fractions of the heap in a round-robin manner like JRockit. Similar to Metronome [**?**, **?**], our implementation instead performs defragmentation on demand. If there are one or more recyclable blocks that the allocator did not use or if the previous collection did not yield sufficient free space, immix triggers defragmentation at the beginning of the current collection.

If immix chooses to defragment, it selects candidate blocks with the greatest number of holes since holes indicate fragmentation. It uses conservative statistics from the previous collection and allocation statistics from the last mutator period to select as many blocks as the available space allows. To compute these estimates efficiently, immix uses two histograms indexed by hole count; a *mark histogram* estimating required space and an *available histogram* reflecting available space. The mark histogram is constructed eagerly during the coarse-grain sweep at the end of each collection. Immix marks each block with its number of holes and updates the mark histogram to indicate the distribution of marked lines in the heap as a function of the number of holes in the associated block. For example, if a block has thirty marked lines and four holes, we increment the fourth bin in the histogram by thirty. Since these operations are cheap, we can afford to perform them at the end of every collection, even if there is no subsequent defragmentation. Immix creates the available histogram lazily, once it decides to defragment. Each bin in the available histogram reflects the number of available lines within the blocks with the given number of holes.

To identify candidates, immix walks the histograms, starting with the most fragmented bin, increments the *required* space by the volume in the mark histogram bin, and it decrements the *available* space by the volume in the available histogram bin. Immix decrements the bin number, moving from most fragmented to least. When including the blocks in the bin would exceed the estimated available space, immix selects as defragmentation candidates all blocks in the previous bin and higher. Section 5.4 shows this policy works very well for our full-heap collector, but we have not tuned it for a generational setting, which we leave to future work.

Mixing Marking and Evacuation. The collector mixes evacuation and marking, combining a mark bit, orthodox forwarding bits, and forwarding pointer in the object header. Even without defragmentation, a per-object mark is necessary in addition to a line mark, to ensure the transitive closure terminates. Objects on candidate blocks are movable, except if they are pinned. Pinned objects are not movable. Remember that immix: a) only triggers defragmenta-

tion if there is available memory, and b) when selecting candidates, tries to ensure that it can accommodate their evacuation. However, since candidate selection is based entirely on space *estimates*, the collector may consume all the unused memory before evacuating all the candidate objects. At this point, all subsequently processed objects become unmovable.

Initially all objects are neither marked or forwarded. When the collector processes a reference to an object, if the object is unmarked and not movable, it marks the object and enqueues the object for processing of its children. If the object is unmarked and movable, it evacuates the object, installs a forwarding pointer, sets the forwarded bits, and then enqueues the forwarded object for processing of its children. If an object is marked and not forwarded, the collector performs no additional actions. If the object is forwarded, the collector updates the reference with the address stored in the forwarding pointer.

This design combines mark-region and evacuation on a per-object basis. It also uses the same allocator for the mutator and collector. These innovations achieve opportunistic, single-pass evacuation-based defragmentation.

Parallel Defragmentation. During defragmenting collection, as with any evacuating garbage collection, some form of synchronization is essential to avoid evacuating an object into two locations due to a race. We use a standard compare and exchange approach to manage the race to forward an object. We exploit MMTk's facility for trace specialization to separately specialize the collector code for defragmentation and normal scenarios. This specialization gives us all the benefits of defragmentation and only pays for the synchronization overhead when it is actually required.

Pinning. In some situations, an application may request that an object not be moved. This language feature is required for C#, and although not directly supported by Java, some JVM-specific class library code in Jikes RVM optimizes for the case when an object is immovable, for example, to improve buffer management for file I/O. We therefore leverage immix's opportunism and offer explicit support for pinning. When the application requests pinning for an object, immix sets a *pinned* bit in the object header. Defragmentation never moves pinned objects.

3.3 Further Implementation Details

This section describes a few more implementation details.

Block and Line Size. Key parameters for immix are the block and line size. We experimentally selected a line size of 128B and a block size 32KB, as shown in Section 5.4. We roughly size the blocks to match the operating system page size. For simplicity, we choose a uniform block size and do not permit objects to span blocks. We use the default large object size of 8K currently implemented in MMTk. Thus, immix only considers objects smaller than 8K. A 32KB block can accommodate at least four immix objects, bounding worst-case block-level fragmentation to about 25%. Blocks are also the unit of coarse-grain space sharing among threads, which has two consequences: (1) each block acquisition and release must be synchronized, and (2) threads cannot share space at a finer granularity than a block. Smaller block sizes would have higher synchronization overheads and a higher worst case fragmentation bound, but more space sharing. Larger blocks would have less synchronization and lower worst case fragmentation, but worse space sharing. Section 5.4 shows that immix performance is not very sensitive to block size, but large block sizes can reduce memory efficiency in small heap sizes.

We roughly size lines to match the architecture's cache line and to accommodate more than one object. The 128B line size reflects a tension between metadata overhead for small lines (since we require one mark byte per line), and higher fragmentation for large lines (since we only reclaim at line granularity). Section 5.4 shows that immix is more sensitive to line size than block size. The smallest object size in Jikes RVM is 8B—the size of the standard header with no payload—for an object with no fields. In theory, if all objects are 8B and only one object per two lines survives a collection and each survivor straddles two lines, worst-case line fragmentation is 97%. Given the allocation and live object size demographics for these benchmarks, a line size of 128B accommodates one to four objects on average [**?**] with an average worst-case internal fragmentation of 25% and a lower expected fragmentation. Tuning line size seems most closely tied to the programming language's influence on object size demographics, rather than the architecture.

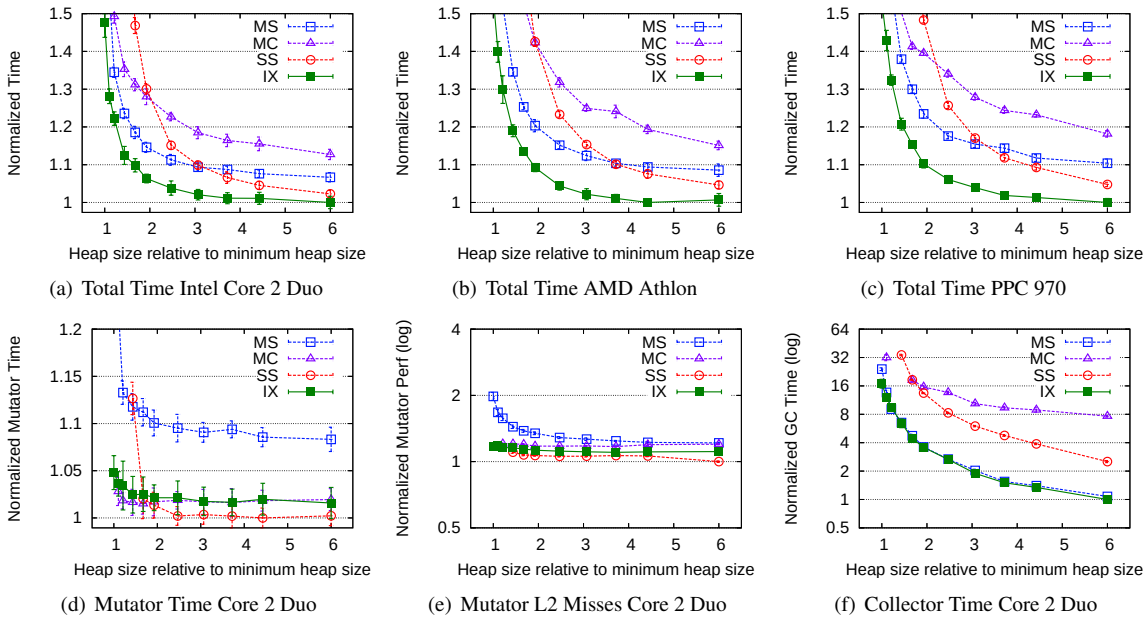
Size and Accounting of Metadata. We embed all immix metadata in the heap, leveraging a mechanism provided by MMTk to interleave metadata at 4MB intervals. Thus, immix, like all other MMTk collectors, is correctly charged for its metadata space overhead. Immix requires one byte per line and four bytes per block, totaling 260 bytes per 32KB block, which amounts to 0.8%.

Conservative Marking. Immix may allocate objects across line boundaries. This choice impacts marking, allocation, and space efficiency. Since immix cannot reclaim space at a finer granularity than a line, the collector must mark all lines on which a live object resides. A simple exact marking scheme interrogates the object's type information to establish its size, and then iterates through the address range, marking the relevant lines. We found experimentally that these operations were quite expensive, especially when compared to mark-sweep which simply sets a bit in the object header.

We instead use *conservative line marking*. We leverage the observation that the overwhelming majority of objects are less than 128 bytes [**?**], so can span at most two lines. Conservative marking explicitly marks just the line associated with the start of each small object, independent of the object's exact size and placement, making marking simple and cheap. Since a small object may span two lines, conservative marking *implicitly* marks the second line by skipping the first line in each hole at allocation time. In the worst case, conservative marking could waste nearly a line for every hole. Since medium objects are uncommon, we perform exact marks for them at collection time. However, our optimization requires that we differentiate small and medium objects, which requires interrogating each object's size, defeating much of the effect of the optimizations. We solve this problem by using a single header bit to mark each object as small (0) or medium (1) when the object is allocated. Since in Java, the size of non-array objects is statically known, Jikes RVM's optimizing compiler can statically evaluate the conditional and elide the setting of the size bit for small objects, which form the overwhelming majority of allocated objects. Figure 2 shows example line marks due to conservative marking. Conservative marking significantly speeds up marking of small objects compared with exact marking.

Optimization of Hot Paths. We took great care to ensure the allocation and tracing hot paths were carefully optimized. As we experimented with various algorithmic alternatives, we evaluated the performance of the resulting mechanism, ensuring the allocator matched semi-space performance, and the tracing loop was as close as possible to mark-sweep tracing. For example, we examined the compiled IR for the allocation sequence, confirming the compiler eliminated setting a bit in the header of small objects.

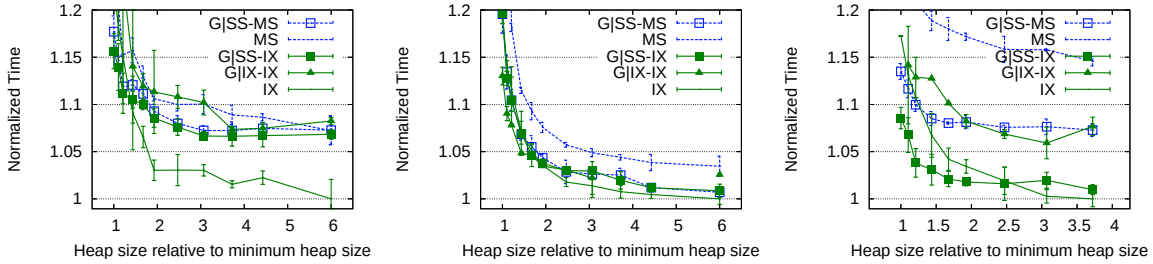
We found that tracing performance was noticeably better if the collector performs the line mark operation when it scans an object, rather than when it marks an object. Because the scanning code is longer and loops over and examines any child references, we believe that it provides better latency tolerance for hiding the cost of the line mark operation.



(a) Total Time Intel Core 2 Duo (b) Total Time AMD Athlon (c) Total Time PPC 970

(d) Mutator Time Core 2 Duo (e) Mutator L2 Misses Core 2 Duo (f) Collector Time Core 2 Duo

Figure 3. Geometric Mean Full Heap Algorithm Performance Comparisons for Total, Mutator, and Collector Time



(a) javac Core 2 Duo (b) luindex Core 2 Duo (c) pjbb2000 Core 2 Duo

Figure 4. Selected Benchmarks Showing Immix, Mark-Sweep and Composite Performance

4. Methodology

We use the following experimental methodology.

Benchmarks. We use all the benchmarks from SPECjvm98 and DaCapo (v. 2006-10-MR2) suites, and pseudojbb2000. The DaCapo suite [**?**] is a recently developed suite of real-world open source Java applications which are substantially more complex than the SPECjvm98. The characteristics of both are described elsewhere [**?**]. Pseduojbb2000 is a fixed workload version of SPEC jbb2000 [**?**]. We configure it with 8 warehouses and 12500 transactions per warehouse. Of these benchmarks, jbb2000, hsqldb, luse-arch, and xalan are multi-threaded.

Hardware and Operating System We use three hardware platforms: (1) *Intel Core 2 Duo* with a 2.4GHz clock, a 800MHz DDR2 memory, a 32KB, 64B line 8-way L1, and a 4MB, 64B line 16-way L2; (2) *AMD Athlon 3500+* with a 2.2GHz clock, a 400MHz DDR2 memory, a 64KB, 64B line 2-way L1, and a 512B, 64B line 16-way L2; and (3) *IBM PowerPC 970 G5* with a 1.6GHz clock, a 333MHz DDR memory, a 32KB, 128B line 2-way L1, and a 512KB, 128B line 8-way L2. All the systems run Linux 2.6.20 kernels from Ubuntu 7.04. We use the perfctr [**?**] library to gather hardware performance counters on the Core 2 Duo. All CPUs operate in 32-bit mode, and use 32-bit kernels. We use all available processors in our performance evaluations, so our main results, from the Core 2 Duo use two processors. We use separate uniprocessor runs to gather performance counter data due to perfctr limitations. Our AMD and PPC results are uniprocessor.

Jikes RVM and MMTk. We implement our algorithm in the memory management toolkit (MMTk) [**?**, **?**] in revision 13767 of Jikes RVM [**?**, **?**] (October 2007). Jikes RVM is an open source high performance [**?**] Java virtual machine (VM) written almost entirely in a slightly extended Java. Jikes RVM *does not have* a bytecode interpreter. Instead, a fast template-driven baseline compiler produces machine code when the VM first encounters each Java method. The adaptive compilation system then optimizes the frequently executed methods [**?**]. Using a timer-based approach, it schedules periodic interrupts. At each interrupt, the adaptive system records the currently executing method. Using a threshold, the optimizing compiler selects and optimizes frequently executing methods at increasing levels of optimization. Since the interrupts are not deterministic, the level of compiler activity and final code quality are non-deterministic.

We use four standard MMTk collectors in our experiments. MS is a mark-sweep implementation that uses a Lea-style segregated fits free list [**?**]. SS is a classic semi-space garbage collector, although it uses a mark stack rather than a Cheney scan [**?**]. MC is a Lisp-2 [**?**] style mark-compact collector which unconditionally

benchmark	time (ms)	MS	SS	MC	IX
compress	3305	1.00	1.01	1.02	1.00
jess	1405	1.00	1.06	1.20	0.90
raytrace	1034	1.00	1.33	1.32	0.92
db	6256	1.00	1.06	1.04	0.96
javac	2616	1.00	1.03	1.03	0.93
mpegaudio	2554	1.00	0.98	0.94	0.97
mtrt	771	1.00	1.44	1.28	0.97
jack	2319	1.00	1.17	1.22	0.92
antlr	2117	1.00	1.21	1.16	0.93
bloat	8038	1.00	1.27	1.27	0.87
chart	7009	1.00	1.04	1.14	0.93
eclipse	33412	1.00	1.07		0.93
fop	1795	1.00	1.02	1.03	0.95
hsqldb	1562	1.00	1.14	0.87	0.88
jython	7459	1.00	1.40		0.82
luindex	9830	1.00	1.11	1.11	0.96
lusearch	10463	1.00	1.25	1.39	0.95
pmd	4775	1.00	1.19	1.39	1.01
xalan	4773	1.00	1.11	1.14	0.90
pjbb2000	16510	1.00	0.99	0.95	0.88
min		*1.00*	*0.98*	*0.87*	*0.82*
max		*1.00*	*1.44*	*1.39*	*1.01*
geomean		*1.00*	*1.14*	*1.13*	*0.93*

Table 1. Full Heap Algorithm Performance at 2× Minimum Heap

benchmark	time (ms)	Copying (G|SS-⋆)			In-Place (G|⋆-⋆)		
		MS	SS	IX	MS	IX	IX^{nm}
compress	3297	1.00	1.00	0.99	1.00	0.99	1.00
jess	1116	1.00	1.04	1.02	1.19	1.01	1.10
raytrace	884	1.00	1.93	0.96	1.18	1.02	1.01
db	6362	1.00		1.01	1.02	0.99	0.97
javac	2650	1.00	1.02	0.99	1.06	1.02	1.02
mpegaudio	2487	1.00	0.97	1.00	1.00	1.00	1.00
mtrt	670	1.00	1.81	1.01	1.21	1.07	1.05
jack	2156	1.00	1.12	0.96	1.19	1.06	1.06
antlr	1950	1.00	1.20	0.96	1.06		0.99
bloat	6371	1.00	1.09	1.00	1.24	1.03	1.08
chart	6657	1.00	1.03	0.99	1.06	0.98	0.98
eclipse	31588	1.00	1.02	0.99	1.10	0.99	1.01
fop	1722	1.00	0.99	0.99	1.02	1.00	0.99
hsqldb	1756	1.00	2.04	1.11	1.00	1.16	
jython	5322	1.00	1.17	0.98	1.19	0.99	1.01
luindex	9763	1.00	1.47	1.00	1.02	0.99	0.98
lusearch	8088	1.00	0.99	0.97	1.15	0.98	1.01
pmd	4799	1.00	1.18	1.02	1.23	1.18	1.27
xalan	4999	1.00		0.93	1.11	1.01	
pjbb2000	15288	1.00		0.95	1.05	0.97	1.04
min		*1.00*	*0.97*	*0.93*	*1.00*	*0.97*	*0.97*
max		*1.00*	*2.04*	*1.11*	*1.24*	*1.18*	*1.27*
geomean		*1.00*	*1.20*	*0.99*	*1.10*	*1.02*	*1.03*

Table 2. Generational Performance at 1.5× Minimum Heap

compacts the entire heap. G|SS-MS is a generational collector with a variable sized evacuating nursery and mark-sweep mature space. All collectors, including immix, use MMTk's large object space (LOS) for objects greater than 8KB.

Experimental Design and Data Analysis. We conduct all of our comparisons across a range of heap sizes from one to six times the minimum heap in which mark-sweep will run (see Figure 5). To limit experimental noise, machines are stand-alone with all unnecessary daemons stopped and the network interface down. We ran each experiment six times, with each execution interleaved among the systems being measured. We discard the fastest and slowest experiments and report here the mean of the remaining four experiments. Each graph shows 99% confidence intervals.

We use Jikes RVM's *replay* feature to factor out non-deterministic allocation into the heap by the compiler due to timer-based adaptive optimization decisions [**?**]. For collector experiments, the allocation load must remain constant. Replay uses a profile of all the dynamic information gathered for compilation decisions: edge frequencies, the dynamic call graph, and method optimization levels. When executing, the system lazily compiles, as usual, but under replay uses the profile to immediately compile the method to its final level of optimization. We gathered five sets of profiles for each benchmark using a build of Jikes RVM with the mark-sweep collector, running each benchmark for ten iterations to ensure the profile captured steady state behavior. We selected profile information from the fastest of the five trials and then used that profile for all experiments reported in this paper. We choose the fastest since slower mutator performance obscures differences due to the garbage collector. We also use Jikes RVM's new profiling mechanism to build optimized, profiled images for each collector. Because the first iteration is dominated by compilation and startup costs, we time the second iteration of the benchmark. We measured the performance of second iteration replay and found that it outperformed a tenth iteration run using the default adaptive optimization system. We also compared our replay configuration with Sun's 1.5 and 1.6 production JVMs in server mode, timing the second iteration on the DaCapo benchmarks, and found Jikes RVM with replay outperformed JDK 1.5 by 5% while JDK 1.6 outperformed replay by 12%. We are therefore confident that our experiments are conducted in a highly optimized environment.

5. Results

We first evaluate immix (IX) against three canonical collectors representing the three previously published reclamation strategies, mark-sweep ('MS', *sweep-to-free-list*), semi-space ('SS', *evacuate*), and mark-compact ('MC', *compact*) on three architectures. We break down performance by mutator and collector. We use hardware performance counters to reveal the role of locality and minimum heap sizes to show space efficiency. Section 5.2 measures immix as a component of a composite generational collector and compares against Jikes RVM's production collector, a high performance generational mark-sweep composite. Section 5.3 evaluates an in-place generational algorithm that handles pinning with respect to both mark-sweep and immix full heap algorithms. Section 5.4 teases apart the contributions of various aspects of immix. Together the results show that a full heap implementation of immix outperforms existing canonical full heap collectors and occasionally outperforms a production generational collector. A generational composite based on immix modestly but consistently outperforms the production collector on average and on some benchmarks significantly outperforms the production collector.

5.1 Full Heap Immix Performance

Figure 3 shows performance as the geometric mean for all 20 benchmarks for the three canonical collectors and immix. Each graph normalizes performance to the best result on the graph and covers heap sizes from 1× to 6× the minimum heap in which mark-sweep will run each benchmark. Figures 3(a)-(c) show total performance for the three architectures, and Figures 3(d)-(f) show the Core 2 Duo mutator time, mutator L2 cache misses, and garbage collection time. Table 1 presents a detailed performance slice at 2× the minimum heap size.

Figures 3(a)-(c) show that immix (IX) outperforms each of the collectors on all heap sizes on all three architectures, typically by around 5-10%. Immix achieves this result by breaking the performance dichotomy illustrated in Figure 1. Figures 3(d) and (e) show that immix matches the superior mutator times of contiguous allocation by providing comparable numbers of L2 misses on the Core 2 Duo. Figures 3(f) shows that immix matches the superior garbage collection times of mark-sweep. Figures 3(b)-(c) show that immix performs even better on the AMD and PPC than it does on the Core 2. The rest of our analysis focuses on the Core 2.

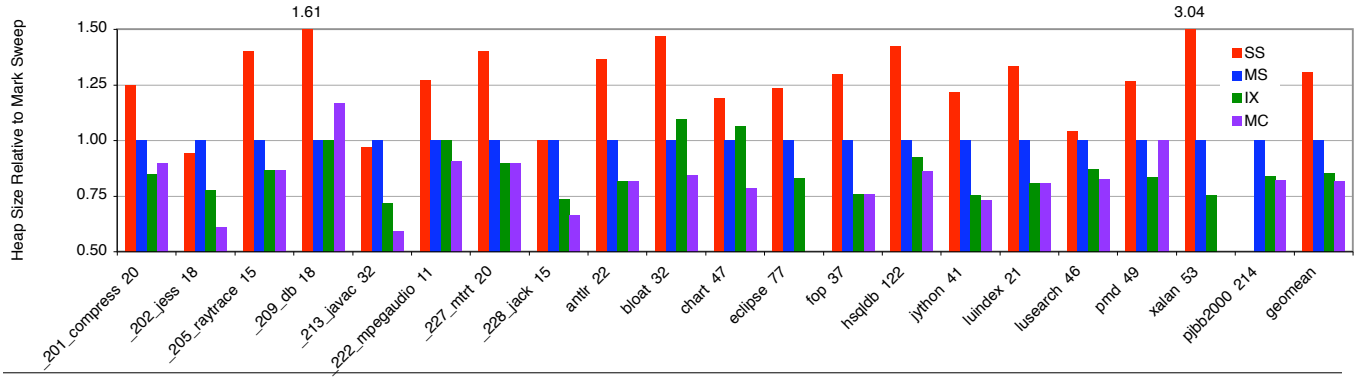


Figure 5. Minimum Heap Sizes for Canonical Algorithms, Normalized to Mark-Sweep. Mark-Sweep Minimum Shown in Label (in MB).

Figure 5 shows the minimum heap sizes for each of the canonical algorithms and immix, normalized to mark-sweep (MS). The legend contains the heap sizes for mark-sweep for each benchmark in MB. The minimums reported here are total heap sizes, inclusive of LOS, metadata, etc. Immix (86%) is 14% more space efficient than mark-sweep on average, and is close to the efficiency of mark-compact (83%). Immix is occasionally more space efficient than mark-compact, e.g., see pmd and db. We speculate that one cause may be pathological interactions between mark-compact (which may move every object at every collection), and address-based hashing, which pays a one word header penalty for any object that is moved after being hashed. Immix minimizes moving, so is much less exposed to this problem.

Table 1 shows that immix achieves its improvements reliably and uniformly, improving over or matching the best of the other collectors on nearly every benchmark. The worst degradation suffered by immix is 1%, and its greatest improvement is 18% at this heap size. Across the 11 heap sizes and 20 benchmarks we measured, the worst result for immix was a 4.9% slowdown on hsqldb at a 1.1× heap. The best result for immix was a 74% (a factor of four) improvement for jython compared to MS, at a 1× heap. These results indicate that immix not only performs very well, but is remarkably consistent and robust. Since performance deteriorates rapidly for all collectors at very small heap sizes, we highlight 2× heap size in Table 1, which in Figure 3(a) is approximately the 'knee' in all four performance curves.

In some cases, immix significantly outperformed Jikes RVM's high performance production generational collector ('GenMS', G|SS-MS), with a semi-space nursery ('G|SS') and mark-sweep old space ('MS'). Figure 4(a) shows that for javac, immix is by far the best choice of collector, outperforming mark-sweep (MS) and three generational collectors. Figure 4(b) shows that for luindex, immix is again the best performing collector, except in a very tight heap where the immix in-place generational collector (G|IX-IX, Section 5.3) performs slightly better. In Figure 4(c), the immix full heap algorithm eclipses mark-sweep (MS) and the Jikes RVM production collector (G|SS-MS) in all but the tightest heaps. Figure 6(c) shows that in larger heaps immix outperforms G|SS-MS on average across all benchmarks.

5.2 A Generational Composite

We now examine the performance of immix in a composite generational collector. We implemented our composite following the template of Jikes RVM's production collector (G|SS-MS), which is a highly tuned collector representative of many high performance production collectors. This collector allocates into a variable-sized evacuating nursery and promotes survivors of nursery collections into a mark-sweep mature space [**?**]. Thus it is a generational semi-space, mark-sweep composite. Because it uses a semi-space for young objects, this collector does not support pinning. It uses an efficient 'boundary' write barrier [**?**], identifying nursery objects as those lying above a certain (constant) address boundary. We created a semi-space, immix composite (G|SS-IX), which mirrors G|SS-MS in all regards except for the change in mature space algorithm. We also compare with Jikes RVM's 'GenCopy' collector, a semi-space, semi-space generational composite (G|SS-SS), which also differs only in its mature space algorithm. We did not compare against a semi-space, mark-compact composite (G|SS-MC) because Jikes RVM does not currently have such a collector.

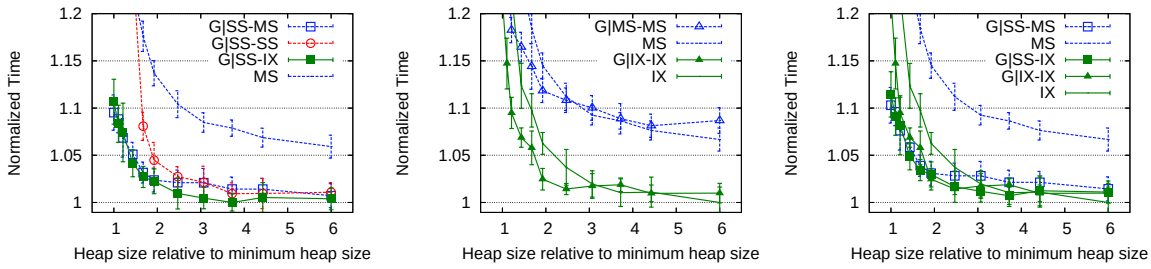
Figure 6(a) shows the total performance of each of the three generational semi-space composites using a geometric mean of our 20 benchmarks, and includes mark-sweep (MS) as a reference. First, we note that the generational collectors significantly and consistently outperform MS in this geometric mean, explaining the wide-spread use of such collectors. We see that G|SS-IX performs similar to and slightly better than G|SS-MS. It is interesting to note that G|SS-IX performs well even at larger heap sizes, where the mutator performance will tend to dominate. Since each of the three collectors shares *exactly* the same write barrier, allocator, and nursery implementation, the observed performance difference is most likely due to better mature space locality offered by immix as compared to mark-sweep.

Table 2 shows a performance slice of the generational semi-space composites at a 1.5× heap, which Figure 6(a) indicates is the 'knee' in the performance curves for these collectors. G|SS-IX performs slightly better than Jikes RVM's production collector (G|SS-MS) on average, with five good results, only one poor result (hsqldb), and the remainder neutral. The few empty cells indicate a collector failing to successfully complete either due to memory exhaustion or some other failure.

Figures 4(a) and (b) show that for some benchmarks, including javac and luindex, there is little difference between G|SS-IX and G|SS-MS. However, Figure 4(c) shows that for some important benchmarks, including jbb2000, G|SS-IX consistently outperforms G|SS-MS. Our implementation of G|SS-IX is untuned—we use exactly the same parameters and configuration in the immix mature space as for the full heap algorithm. We believe there is room for improvement, but leave that to future work.

5.3 Sticky Mark Bit In-Place Collector

We now evaluate mark-sweep (MS) and immix (IX) based implementations of a 'sticky mark bit' in-place generational algorithm [**?**] (G|MS-MS and G|IX-IX). The sticky mark-bit is a simple and elegant extension of a full heap collector to perform partial collections, collecting only the newly allocated objects. The primary difference between our implementation and Demmers et al.'s orig-



(a) Copying Generational (b) In-Place Generational (c) Immix Variants

Figure 6. Generational Collector Performance on Core 2 Duo. Geometric Mean of 20 DaCapo and SPEC Benchmarks.

inal by is that we use an efficient object-remembering barrier [**?**] to identify modified mature objects rather than page protection and card marks. Our collectors are trivial extensions over their canonical full heap counter-parts. An early implementation of Jikes RVM had an in-place generational collector which did not use sticky mark bits, but stored extra state on the side. In Attanasio et al.'s garbage collector review [**?**], they mention this collector, but do not evaluate it. Domani et al. [**?**] build and evaluate an on-the-fly generational collector using the sticky mark bits algorithm. Aside from these collectors, we are unaware of in-place generational collection finding use outside of the conservative setting of Demmers et al.'s work, where a sticky mark bit collector is the only way to achieve generational scavenging since copying is not possible.

Figure 6(b) shows that each of these in-place collectors improves over the canonical algorithms in tighter heaps with minimal degradation in large heaps. In particular, G|IX-IX almost uniformly improves over IX. However, G|MS-MS does not improve sufficiently over MS to justify its use, given the option of a regular copying generational collector. In Figure 6(c), we see G|IX-IX compared to the other immix collectors and Jikes RVM's production collector (G|SS-MS). These results show that G|IX-IX performs very competitively. Since in-place generational collection is trivial to implement, and unlike composites with evacuating nurseries, does not detract from the mostly non-moving properties of immix, the in-place immix collector may have interesting application opportunities.

Columns 6, 7 and 8 of Table 2 show a performance slice at a 1.5× heap for the mark-sweep and immix in-place collectors, with results normalized to Jikes RVM's production collector, G|SS-MS. Here we include two variants on G|IX-IX: G|IX-IX will opportunistic evacuate during nursery collections as well as during defragmentation, while G|IX-IX^{nm} will only opportunistically evacuate during defragmentation time. We found that these two variants performed about the same. We spent more time tuning this collector than we did G|SS-IX, but it remains a fairly naive implementation, and similar to G|SS-IX, can likely be improved.

The result of the experiment with in-place generational collection highlights the importance of significantly improving the performance of the full heap algorithm. While the mark-sweep in-place collector is 'interesting' and perhaps useful in a conservative collection context, changing the base from mark-sweep to immix transforms the idea into a serious proposition for a performance-oriented setting.

5.4 Understanding Immix Performance

Each of the preceding sections and Figure 6(c) show how the immix achieves all three goals: space efficiency (Figure 5), fast collection (Figure 3(f)) and mutator performance (Figure 3(d)). This section examines the individual features and policy sensitivities presented in Tables 3 and 4.

Block Utilization To understand immix's allocation behavior, columns 2–6 of Table 3 show how allocation was distributed among blocks, in terms of the fullness of the blocks. At the nominal 2× heap size, immix allocates 79% of data into completely free or mostly free blocks (< 25% marked), on average across our benchmark suite. Only occasionally, always less than 5%, does immix allocate into mostly full blocks with > 75% marked. We also measured how these statistics vary with heap size. Immix allocates more from recyclable blocks in small heaps than large. For example, compared to 43% of allocation to completely free blocks at 2× heap size, immix allocates 76% at a 6× heap size. This trend is because more frequent collection tends to fragment the heap more; given longer to die more objects die together, whereas more frequent collection exposes more differences in object lifetimes.

Overflow Allocation We found overflow allocation helps provide space efficiency in tight heaps. Column 7 of Table 3 shows the percent of objects that immix sends to the overflow allocator. On average, it handles 4% of allocation, however jython and xalan are conspicuous outliers. Column 5 of Table 4 shows the performance effect of turning off the overflow allocator mechanism. Three benchmarks, antlr, jython and lusearch, benefit from this mechanism, and xalan is slowed down by overflow allocation. Note however, that the memory *savings* associated with a given use of the overflow allocator may vary widely, depending on the size of pending allocation and the level of fragmentation of the recycled blocks.

Importance of Blocks, Lines, and Defragmentation Columns 2 and 3 of Table 4 show performance for mark-region collectors that provide just block marking (block); block and line marking with overflow allocation, but without defragmentation (No DF); block, line, overflow, and defragmentation with no head room (No HR); everything but overflow allocation (No Ov). Some benchmarks perform remarkably well, while others are unable to run at all in a 2× heap. In columns 8, 9 and 10 of Table 3, we show the amount of 'dark matter' due to imprecise marking. Column 8 shows the imprecision overhead of marking *only* at a block grain, column 9 shows the overhead for line marking, and column 10 shows the overhead due to conservative line marks. We express overhead as a percentage of the actual bytes live at each collection, so a 100% overhead means marking was imprecise by a factor of two. If the collector only recycled blocks (*block*), block fragmentation would lead to it on average inflating the amount marked by 93% (nearly double). However, for some benchmarks such as db, compress and jython, a naive block-grain marking scheme is remarkably effective. If the system used line marking (*line*), but still did not perform defragmentation, waste would inflate the amount marked by 23% on average. We also measured the memory wasted due to conservative marking. A few benchmarks, such as javac and fop waste 25 to 20%, but most benchmarks waste less than 6%.

	Allocation						Marking Waste			Pinning		Compaction (1.5× Min Heap)					
	clean	<	<	<	>=	over-					pinned	Compactions		Candidate Blocks		%	net
	blocks	25%	50%	75%	75%	flow	block	line	consv	calls	objects	#	GCs	KB	live	reuse	yield
compress	0%	58%	38%	0%	2%	1%	18%	8%	2%	68	2	7	100%	5600	83%	89%	5%
jess	27%	61%	10%	1%	0%	0%	208%	22%	12%	127	3	1	5%	5280	13%	100%	100%
raytrace	60%	11%	2%	25%	1%	0%	100%	37%	20%	368	3	0					
db	87%	9%	3%	1%	0%	0%	15%	9%	3%	314	2	0					
javac	23%	38%	20%	12%	4%	3%	216%	75%	25%	5K	450	2	20%	5664	27%	100%	100%
mtrt	65%	12%	2%	18%	2%	0%	76%	44%	20%	488	4	0					
jack	26%	61%	7%	2%	2%	4%	188%	23%	10%	34	2	2	8%	12832	8%	100%	100%
antlr	51%	30%	7%	6%	1%	5%	48%	13%	5%	5K	4529	0					
bloat	30%	64%	3%	1%	1%	1%	101%	13%	6%	33	8	6	10%	33888	17%	100%	100%
chart	42%	48%	6%	1%	3%	1%	134%	15%	5%	29K	58	0					
eclipse	34%	54%	4%	1%	2%	6%	87%	13%	5%	112K	35K	13	24%	229696	21%	97%	95%
fop	36%	17%	26%	14%	4%	4%	134%	44%	20%	0	0	0					
hsqldb	99%	0%	0%	0%	0%	0%	26%	23%	1%	43	0	0					
jython	64%	11%	1%	1%	0%	23%	24%	7%	2%	14	0	2	5%	10816	50%	91%	96%
luindex	45%	39%	7%	3%	1%	4%	76%	12%	5%	43K	7K	0					
lusearch	25%	53%	9%	7%	3%	4%	52%	9%	3%	171K	9K	1	1%	8544	52%	3%	5%
pmd	43%	33%	11%	10%	1%	2%	110%	35%	15%	137	2	4	17%	43488	43%	90%	96%
xalan	11%	41%	18%	9%	3%	18%	59%	10%	4%	39K	27K	1	6%	7360	39%	100%	100%
min	*0%*	*0%*	*0%*	*1%*	*0%*	*0%*	*15%*	*7%*	*1%*				*1%*		*8%*	*3%*	*5%*
max	*99%*	*64%*	*38%*	*25%*	*4%*	*23%*	*216%*	*75%*	*25%*				*100%*		*83%*	*100%*	*100%*
mean	*43%*	*36%*	*10%*	*7%*	*2%*	*4%*	*93%*	*23%*	*9%*				*20%*		*35%*	*87%*	*80%*

Table 3. Allocating, Marking, Pinning, and Compaction Statistics. Compaction at 1.5× Minimum Heap, All Others at 2× Minimum Heap

Pinning Opportunistic evacuation allows immix to elegantly support object pinning. Although Java does not support object pinning, it is an important feature of C# and Jikes RVM's class libraries optimize for pinning in the classes `gnu.java.nio.VMChannel` and `org.jikesrvm.jni.VM_JNIFunctions`. In each case, the library avoids indirection and buffering when the VM assures an object will not move. Columns 11 and 12 of Table 3 show the number of times a call was made to pin an object during the second iteration of each benchmark, and the number of objects which were pinned as a result. Objects are pinned only once. The pinning interface always returns true for mark-sweep since it never moves objects, and always returns false for semi-space and mark-compact because neither support pinning. Immix pins the requested object and returns true. Immix performs pinning in all of the results reported in this paper. We performed detailed performance analysis and found that pinning has no effect on performance for most benchmarks; has a very small advantage for three benchmarks which use pinning heavily; and slightly degrades performance for a fourth benchmark.

Opportunistic Defragmentation Columns 13–18 of Table 3 show the behavior of opportunistic defragmentation in a 1.5× heap. Even at this modest heap size, only 8 benchmarks require defragmentation and only compress triggers defragmentation on every collection. We measured the volume of blocks marked as defragmentation candidates and the amount of *live* data on those blocks, expressed as a percentage. This percentage is an indirect measure of fragmentation since it does not quantify the number of holes. We see wide variations from 8% to 83% of live objects on candidate blocks. We see that the vast majority (87%) of evacuated live objects are tucked into recyclable blocks rather than being evacuated to completely free blocks. We measured defragmentation yield. Opportunistic defragmentation has limited success on compress and lusearch, but neither of these programs stress the collector much. On the remaining programs, defragmentation successfully converts 95% or more of the candidate space to completely free blocks. These statistics show that opportunistic defragmentation effectively compresses the heap on demand and is only occasionally triggered.

Defragmentation Headroom We also explored, but do not show in detail, how immix's minimum heap size is influenced by eliminating (1) the default 2.5% defragmentation headroom, (2) defragmentation, and (3) both defragmentation and line-grain reclamation. Eliminating defragmentation and line-grain reclamation, thus only performing block-grain recycling, would require more than doubling the average minimum heap size. Performing line-grain marking, but no defragmentation still increases the minimum heap size significantly, on average 45% and up to 361%. Headroom also significantly benefits immix. With zero headroom, immix would require an increase of between 16% on average, although for xalan and antlr, immix performed *better* with no headroom. We experimented with headroom of 1, 2, and 3%; all were sufficient to achieve small heap sizes in immix. Immix was not very performance sensitive to these choices; 2% performs slightly better than our current 2.5% threshold across a number of heap sizes, but all were sufficient for robust immix performance in small heaps.

Block and Line Sizes Columns 6–9 of Table 4 show the sensitivity of immix to line and block size, measuring half and twice the default sizes of 32KB and 128B for blocks and lines respectively. Per-benchmark variation based on block size ranges from 7% better to 7% worse for large blocks. Large blocks show a slight improvement at a 2× heap size, but they perform worse in smaller heaps. Smaller blocks have slightly less variation, but neither change immix much. The eclipse and xalan benchmarks are most sensitive to line size, and sensitivity grows in smaller heap sizes. Most benchmarks are not that sensitive to line size, but we found that 128B is more consistent and better across many heap sizes.

6. Conclusion

This paper describes *mark-region,* a new family of non-moving collectors that allocate and reclaim memory in contiguous regions. To combat fragmentation, we introduce lightweight *opportunistic defragmentation*, which mixes copying and marking in a single pass. We combine both ideas in *immix*, a novel high performance garbage collector which attains space efficiency, fast collection and mutator performance. Immix outperforms existing canonical collectors, each of which sacrifice one of these three performance objectives. As a mature space in a generational collector, immix matches or beats a highly tuned generational collector. By describing mark-region for the first time and presenting a detailed analysis of a high performance mark-region implementation, this paper opens up new directions for future collector design.

Acknowledgments

We thank our anonymous reviewers for helping us greatly improve the paper. We thank David Bacon, Daniel Frampton, Robin Garner,

	Algorithmic Features				Block Size		Line Size	
benchmark	Block	Line	No HR	No Ov	16KB	64KB	64B	256B
compress	1.00	1.00	1.00	1.01	1.00	1.00	1.00	1.01
jess			0.98	1.00	1.04	0.99	1.02	0.99
raytrace	1.07	1.00	1.00	1.01	0.98	0.96	0.97	0.96
db	1.04	1.04	1.03	1.00	1.00	1.00	1.01	1.00
javac		0.99	1.00	1.01	1.03	0.98	0.99	1.02
mpegaudio	0.99	1.00	1.00	1.00	1.00	1.00	1.01	1.00
mtrt	1.26	1.02	1.00	0.98	0.99	0.98	1.00	1.00
jack		1.01	1.02	0.97	0.99	0.98	0.98	0.97
antlr		1.06	1.05	1.04	1.00	1.07	0.98	1.01
bloat				0.99	1.01	0.99	1.00	1.01
chart		1.03	1.00	1.01	1.02	1.00	1.01	1.00
eclipse				1.02	1.03	1.02	1.05	1.02
fop		1.02	1.01	1.01	1.01	1.00	1.02	1.00
hsqldb	1.09	1.08	1.28	1.00	1.00	0.92	0.99	0.91
jython		3.15	1.81	1.08	1.03	0.99	1.06	1.01
luindex	1.24	1.05	1.04	1.00	1.01	1.01	0.99	1.00
lusearch	1.57	1.13	1.05	1.05	1.04	1.00	1.05	1.03
pmd		1.04	1.05	1.00	1.00	1.00	1.00	1.00
xalan	1.60	1.20	1.98	0.91	0.96	0.97	0.93	1.03
min	*0.99*	*0.99*	*0.98*	*0.91*	*0.96*	*0.92*	*0.93*	*0.91*
max	*1.60*	*3.15*	*1.98*	*1.08*	*1.04*	*1.07*	*1.06*	*1.03*
geomean	*1.19*	*1.12*	*1.11*	*1.01*	*1.01*	*0.99*	*1.00*	*1.00*

Table 4. Performance Sensitivity. Relative to Immix 2× Heap.

and David Grove for their insight, feedback and technical assistance. We thank all of the developers of Jikes RVM, without whom this research would not happen.