THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-07-04

Immix Garbage Collection: Fast Collection, Space Efficiency, and Mutator Locality

Stephen M. Blackburn, Kathryn S. McKinley

August 2007

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

Immix Garbage Collection: Fast Collection, Space Efficiency, and Mutator Locality

Stephen M. Blackburn and Kathryn S. McKinley

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

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

Abstract

No current garbage collector combines space efficiency, low collector times, and mutator (application) locality for contemporaneously allocated objects. For example, free-list allocation coupled with mark-sweep collection achieves space efficiency, but poor mutator locality. On the other hand, contiguous (bump-pointer) allocation with copying collection provides mutator locality but poor space efficiency since it requires reserving half the heap for copying in case all objects survive.

This paper presents an algorithm that obtains space efficiency, low collection times, and mutator locality in a single regime. We call this garbage collector *immix*, which means to intimately commingle. The immix heap is organized hierarchically into *blocks* of contiguous memory, each consisting of *N* fixed size *lines*. Immix tracks memory use at a line granularity. Immix bump-allocates objects into one or more contiguous free lines to attain mutator locality. To attain space efficiency, immix uses mark-sweep collection to identify free lines. When needed to eliminate fragmentation, immix mixes copying and mark-sweep collection in a unique way to implement opportunistic *single pass* compaction.

We demonstrate immix as a whole heap collector, as a component of a generational collector, and as a mostly non-moving collector. Our experimental results demonstrate that immix comprehensively and uniformly outperforms state-of-the-art whole heap mark-sweep, mark-compact, and semi-space collectors on a range of heap sizes on multiple architectures. Immix is the first collector to attain simultaneously space efficiency, low garbage collection times, and mutator locality.

1. Introduction

Modern garbage collection systems reveal the often observed tension between time and space. Designs for free-list heap organizations have focused on space efficiency [15, 21] and form the underpinnings of *mark-sweep* garbage collectors, whereas contiguous allocation with *semi-space* copying collection achieve locality for contemporaneously allocated objects.

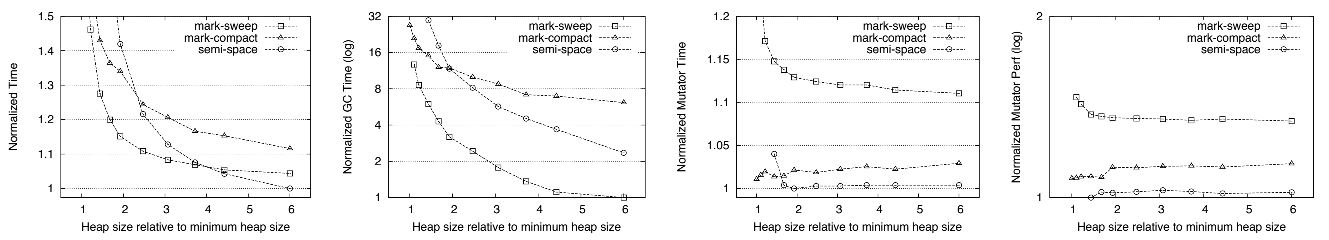
For instance, *mark-sweep* allocators find free memory on a free-list and assign it to new objects. The collector traces from the roots (stack pointers, globals, etc.) to all reachable objects, marking the live ones. It then *sweeps*, populating a free-list with unmarked, free memory. Optimizations such as *lazy* sweeping [18, 8] populate the free-list on demand, instead of visiting the entire heap. These collectors are space efficient and perform well since their work is proportional to the live objects. However, because they do not move objects and do not colocate contemporaneously allocated objects, they have less control over *mutator* (application) locality.

On the other hand, a *semi-space* allocator bumps a pointer into a large region of memory. The collector traces the live object graph. Instead of marking live objects, it copies each one the first time it encounters it into a separate space, and leaves a forwarding pointer to its new location. When the collector encounters subsequent references to the object, it updates them using the forwarding pointer. The collector then reclaims the free memory en-masse. This organization requires the collector to reserve half the heap, since all objects may survive; thus the name, semi-space collection. Semi-space collectors provide locality to the mutator at allocation time, and after copying they also provide some locality because they compress the heap, reducing the working set size. Unfortunately, they have poor space efficiency which results in more collections compared with mark-sweep in a fixed size heap.

Attempts to resolve this tension within contiguous heap organizations use *mark-compact* and other techniques [1, 14, 29, 24, 31]. These approaches however all require multiple passes over the live objects. Two recent schemes, JRockit and Vam, try to resolve this tension using a free-list by performing block reclamation and bump-pointers into free blocks [17, 19]. Vam is designed for non-moving explicit memory management; thus if a block becomes fragmented, Vam reverts to size-segregated free-lists which make no guarantees about locality. JRockit uses a range of block sizes, and only recycles completely free blocks. To limit fragmentation, it performs mark-compact on a fraction of the fragmented blocks on every garbage collection, incurring multi-pass overheads.

Because free-lists and contiguous heap organizations are the building blocks of all modern collectors, including the best performing generational collectors, we focus on full-heap collectors which are more fully exposed to performance tradeoffs due to program variation. Figure 1 shows the dichotomy of benefits offered by full-heap semi-space, mark-sweep, and mark-compact collectors implemented in MMTk [8] as a function of minimum heap size. The mark-sweep collector uses size segregated free-lists [21], and the mark-compact collector uses a classic three pass algorithm [31]. The figures plot the geometric mean of total time, collection time (log scale), and mutator time normalized to the best for the 18 DaCapo and SPECjvm98 benchmarks on an Intel Core 2 Duo. (See Section 4).

Consider garbage collection time in Figure 1(b). Collection time decreases for all three collectors as the heap grows, since collection is triggered when heap space is exhausted. The space efficiency of mark-sweep offers a factor of 3 to 8 reduction in collection time compared to semi-space. Mark-sweep collects less often and each collection is slightly cheaper (not shown in the graph). Mark-compact also collects less frequently than semi-space, but it only amortizes the additional passes it takes over the live objects in the



(a) Total Time (b) Garbage Collection Time (c) Mutator Time (d) Mutator L1 Misses

Figure 1. Classic Algorithm Performance Tradeoffs: Geometric Mean for 18 Benchmarks

smallest of heap sizes. Once the heap size grows to 2.5 times the minimum, semi-space performs much less work per collection and is thus faster.

Total time in Figure 1(a) shows that in smaller heap sizes (between 1 and 3), memory efficiency is key since the overheads of garbage collection overwhelm mutator performance. Once the heap size is large enough (greater than 4), garbage collection time reduces, and the locality of the mutator starts to dominate total time. Mutator L1 cache misses in Figure 1(d) show that the primary mutator advantage for semi-space, seen in Figure 1(c) is due to better locality rather than the relative simplicity of the bump allocator.

The design goals of the *immix* collector are therefore to get all of these benefits without their costs in a single collector: (1) space efficiency, (2) mutator locality for contemporaneously allocated objects, and (3) a high performance single pass collector. The design innovations we use to attain these goals and contributions of this paper are:

- The immix heap organization that tracks free memory at the granularity of *blocks* and *lines*, where each fixed size block is a contiguous piece of memory consisting *N* fixed size lines.
- Eager reclamation of completely free blocks and lazy reclamation of partially populated blocks.
- Bump-pointer allocation in free and partially populated blocks.
- An opportunistic, demand driven compaction algorithm that mixes copying and marking to attain single pass compaction on fragmented blocks.
- A comprehensive performance analysis that shows immix substantially outperforms previous full heap collectors by providing space efficiency, mutator locality, and efficient collection.

The basic immix allocator and collector are very simple. The allocator initially bump-allocates into free blocks until memory is exhausted. The collector uses a line mark map for the mark pass, marking any line with a live object. It then sweeps the line mark map for completely free blocks. Subsequent allocation bump-allocates first into partially populated blocks and then into free blocks until the heap is exhausted, and so on. A few optimizations enhance this basic design: *conservative marking* for mark efficiency, *demand driven overflow allocation* for objects larger than a line that do not fit in the current allocation hole, and *opportunistic* copying compaction for fragmented blocks.

Opportunistic copying gracefully handles pinning and eliminates the need for multi-pass compaction. Since immix competes for space with other consumers such as the large object space, a collection may be triggered before immix has exhausted the reusable space in its heap. Any space unused by the mutator is available to the compactor. At the start of each collection, immix decides whether compaction is required. If so, it selects some set of blocks as compaction candidates, taking into account fragmentation and available space. Then during collection, whenever an object on a candidate block is encountered, the object is opportunistically copied. Opportunism means that objects are left in place if they cannot be moved because they are pinned by the application or because the compactor has exhausted its available space. The copying mechanism is simple because it reuses exactly the same allocator as the application, picking up where the application left off until it is exhausted. Of course the compactor does not allocate into any blocks marked as compaction candidates.

We demonstrate that these immix policies dynamically and naturally adjust to the available heap size, achieving locality and space efficiency. For example, immix performs little or no compaction in large heaps and relatively more in small heaps. The immix heap organization cherry picks the best performance aspects of the previous collectors without their drawbacks, combining for the first time, space efficiency, mutator locality, and high performance collection in a simple collection regime.

2. Background and Related Work

This section presents previous garbage collection algorithms and their performance tradeoffs, and compares them with immix. For more background, see Jones and Lins [20] and Wilson et al. [33] for algorithms; and see Blackburn et al. [8] for performance analysis.

Following the literature, execution time consists of the *mutator* (the application itself) and periodic *garbage collection*. Some memory management activities, such as object allocation and the *write barrier*, mix in with the mutator. Collection may execute concurrently with mutation, but this paper assumes an independent collection phase, known *stop-the-world* collection.

We use *heap organization* to indicate the coupling of an allocation mechanism with a collection mechanism. *Whole heap* collectors use one heap organization. *Tracing* collectors, such as copying and mark-sweep, identify live objects by computing a transitive closure over the object reference graph starting with the *root* references (stacks, registers, and class variables/statics).

Generational collectors divide the heap into age cohorts [4, 23, 32] and use one or more heap organizations. Generational and incremental algorithms, such as reference counting, require a *write barrier* to remember pointers between independently collected regions. For every pointer store, the compiler inserts write-barrier code. At execution time, this code records only pointers that cross regions in remembered sets. At collection time, the collector policy determines the region and the collector assumes that pointers in the remembered set into the region are live, treating them like roots.

As far as we are aware, all modern collectors build on the following two heap organizations:

Contiguous: Contiguous Allocation with Copying Collection. Allocation appends new objects in a contiguous space by incrementing a *bump pointer* by the size of the new object. A copying collector evacuates or compresses live objects. A *semi-space* (SS) copying collector allocates into half the heap, the *to-space*. A collection flips the sense of the spaces and copies survivors from the from-space into the new to-space [13, 7]. A compacting copying collector (known in the literature as *mark-compact* (MC)) eliminates the copy reserve but requires multiple passes

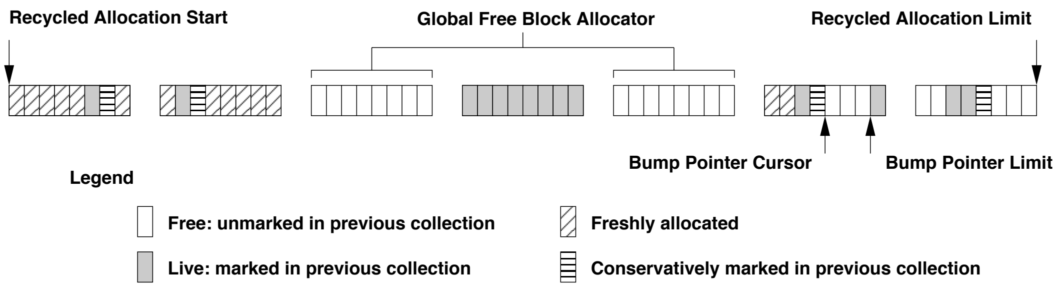

Recycled Allocation Start
Global Free Block Allocator
Recycled Allocation Limit
Bump Pointer Cursor
Bump Pointer Limit
Legend
Free: unmarked in previous collection
Freshly allocated
Live: marked in previous collection
Conservatively marked in previous collection

Figure 2. Immix Heap Organization

to compact the live objects into the same contiguous region in which they currently reside [31, 14].

Free-list: Free-list Allocation with Mark-Sweep Collection. The heap is organized into *free-lists* [15]. The allocator hands out memory either using *k* size-segregated lists to match object sizes [21, 8], or divides up larger chunks on demand [17, 19, 30]. A mark-sweep collector identifies and *marks* live objects and then populates the free-lists with *unmarked* memory.

Free-lists also accommodate reference counting collection. Instead of tracing, a reference counting collector tracks the number of references to an object, and when it falls to zero, puts the free memory on the appropriate free-list [16, 22, 12]. Reference counting alone is incomplete [6, 16], because it cannot collect cycles, and thus we do not consider it further here.

The historical development of these two heap organizations in the literature shows they were designed to solve different problems: when the collector has the option to move objects, contiguous heaps provide efficient en-masse freeing, matching object demographics in garbage collected languages. On the other hand when objects cannot move, free-lists provide space efficiency, matching object demographics in explicitly managed languages.

However, semi-space collectors have high collection times in small heaps because they reserve half the heap, causing them to collect more frequently than more space efficient collectors. More frequent collection also gives objects less time to die, leading to more copying of objects. These costs increase collection time, as shown in Figure 1(b). Semi-space is optimized and performs well for moderate to large heaps because (1) it provides locality to contemporaneously allocated objects, as shown in Figure 1(c) and (d), and (2) its collection cost is proportional the live objects rather than heap size. With enough space, many objects die between collections giving it less work to do. Approaches to improve the space efficiency of contiguous heap organization, such as mark-compact, MC^2, and copy size reduction [1, 14, 29, 24], all may substantially increase garbage collection times. One example of such overheads is shown in Figure 1(b) with a classic three pass algorithm that was available in MMTk and was introduced by Styger [31]. Although Abuaiadh et al. have introduced a more efficient two pass algorithm [1], the immix heap organizations enables elegant compaction in a single pass.

Researchers first designed free-lists for managing operating system virtual memory. They were optimized for relatively few allocations and incremental freeing [15], and to minimize fragmentation with policies such as best, first, and worst fit allocation [27, 28]. Free-lists were later refined to solve the problem of explicit memory management in the heap [26, 21]; the main innovation was size segregation into separate free-lists (*size segregated free-lists*). These organizations perform well for explicitly managed programs that require objects not to move, tend to have many and popular small object sizes, and incrementally free objects [21]. Their space efficiency keeps collection times down when heap space is scarce, as shown in Figure 1(b). However, they sacrifice mutator performance, as shown in Figure 1(c) and (d) due to poor locality [8].

The prior regimes closest to immix are Vam [17] and the JRockit collector [19] which have refined the free-list to provide more mutator locality. Both of these regimes recycle completely free blocks efficiently to provide bump-pointer allocation within a block. Vam is an explicit memory manager and cannot move objects. It handles block fragmentation by reverting to size segregated free-lists for each fragmented block, thus cannot guarantee mutator locality. JRockit's scheme is a garbage collector and may move objects. JRockit uses a range of block sizes and never allocates into partially free blocks smaller than some threshold (currently 2K). It handles fragmentation in these blocks by compacting a fraction of the heap on every collection, using a standard multi-pass algorithm. JRockit thus guarantees locality but sacrifices efficient collection times by using mark-compact every collection on a fraction of the heap.

3. The Immix Regime: Mixing Space Efficiency and Locality

Immix heap organizations differ substantially from the above contiguous and free-lists organizations. The immix heap divides memory into *blocks* of contiguous memory consisting of *lines*. Lines accommodate multiple small objects, and objects may span lines. Immix tracks in-use and free-memory at a fine-grain line and coarse-grain block granularity, and never builds an explicit free-list. With this organization (1) a bump-pointer allocates into free blocks and groups of lines to attain mutator locality, (2) reclamation is at a fine and coarse-grain level to attain space efficiency, and (3) copying can eliminate fragmentation without requiring multiple passes.

3.1 Overview

Immix's blocks and lines roughly mirror the granularity of operating system and hardware-level locality concerns respectively. Figure 2 illustrates this organization. To achieve scalable parallel allocation and collection, throughout immix we follow a design pattern which maximizes fast, unsynchronized thread-local activities and minimizes synchronized global activities. Thus the synchronized global allocator gives blocks to lightweight thread-local allocators, which are internally unsynchronized.

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

Collection. The collector traces through the live objects by performing a transitive closure starting with the application roots. As immix traces the object graph, it records in a map which lines contain live objects. When it completes the trace, immix performs a *coarse-grained sweep*, linearly scanning the line map and identifying entirely free blocks and partially free blocks. It returns entirely free blocks to the global pool, and it *recycles* the 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 unused lines). The thread-local allocator then bump allocates into that hole. Once the hole is exhausted, the thread continues its linear scan for another hole until the recyclable block is exhausted. Once the threads have exhausted all recyclable blocks in the heap, they resume allocating into empty blocks until the entire heap is exhausted.

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

The global allocator gives out recyclable blocks first and then free blocks because it competes for space with multiple consumers of memory and free blocks offer more flexibility. For example, one consumer is a separate non-copying large object space (LOS) for objects greater than a threshold. The LOS can only consume completely free blocks. Competing consumers may trigger a collection before immix has the opportunity to exhaust its recyclable space. To ensure that the entire heap is effectively recycled, immix resumes allocation into the recycled blocks at the same point where it left off in the previous mutator phase.

Opportunistic Compaction. The above algorithm does not move objects, and is thus subject to fragmentation. To eliminate fragmentation, immix implements opportunistic compaction. At the start of each collection, immix detects whether a compaction is desirable, e.g., if there are one or more unused recyclable blocks in which to compact. If so, immix uses statistics on the number of holes in a block (i.e., fragmentation), the amount of live lines in a block, and unused memory available to select a set of compaction candidates. When the collector encounters a live object that resides on a compaction candidate block, it will *opportunistically* copy the object. An object is *movable* only if the application has not pinned it and the compactor has not exhausted its available space. If an object is unmovable when the collector encounters it, the collector marks the object as live and leaves it in place.

To copy objects, the collector uses exactly the same allocator as the mutator. It begins allocation into the recyclable blocks at the point where the mutator left off. Once it exhausts unused recyclable blocks, the collector uses any available completely free blocks. The compactor of course does not allocate into compaction 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 copying. This *headroom* eases compaction and is counted against immix's overall heap budget. Section 3.2 describes compaction in more detail.

These mechanisms: contiguous blocks consisting of lines, line and block grain reclamation, and opportunistic compaction are the key elements of immix.

3.2 Algorithmic Features

This section describes the algorithmic details of opportunistic compaction, pinning, demand driven overflow allocation, and support for multiple, parallel application and collector threads. Section 3.3 then explores the consequences of immix design parameters and implementation details.

Opportunistic Compaction Details. A variety of heuristics may select compaction candidates. A simple policy would target fractions of the heap in a round-robin manner like JRockit. Our implementation instead performs compaction 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 a compaction at the beginning of the current collection.

If immix chooses to compact, it selects candidate blocks with the greatest number of holes since holes indicate fragmentation. It selects as many blocks as the available space allows. To compute these estimates efficiently, immix uses two histograms; 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 populates the mark histogram indicating the distribution of marked lines in the heap as a function of the number of holes in the associated block. If a block with thirty marked lines has 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 compaction. Immix creates the available histogram lazily, only once it decides to compact. Each bin in the available histogram reflects the number of available lines within the blocks with the given number of holes.

To identify compaction candidates, immix walks the histograms, starting with the highest (most fragmented) bin, increments the *required* space by the volume in the mark histogram bin, and decrements the *available* space by the volume in the available histogram bin. We decrement 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 compaction candidates all blocks in the previous bin and higher.

The collector mixes copying and marking, combining a mark bit and orthodox forwarding bits and forwarding pointer in the object header. Even without compaction, 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 immix only triggers compaction if there is available memory and tries to ensure it can accommodate compaction candidates. However since candidate selection is based entirely on space *estimates*, the compacting allocator may consume all the unused memory before copying 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 processes its children. If the object is unmarked and movable, it copies the object, installs a forwarding pointer, sets the forwarded bits, and then enques 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 uses classic copying and mark-sweep algorithms on a per-object basis. It also uses the same allocator for the mutator and collector. These innovations achieve opportunistic, simple, single-pass copying compaction.

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 known to be 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. The compacting phase subsequently never moves the object.

Demand Driven Overflow Allocation. We found that the greedy recyclable block allocator described above occasionally skips over and wastes many lines when trying to allocate medium objects. We solved this problem by pairing each allocator with an *overflow allocator*. The overflow allocator is also a bump-pointer, but it always uses blocks that are initially completely free. Whenever allocation *fails* and the failing object size is larger than a line, immix allocates it with the overflow allocator. Thus, medium objects are allocated in the overflow blocks only on demand. Allocation fails whenever the request exceeds the amount of memory between the bump pointer and bump pointer limit. This design is extremely effective at combating only on demand the pathological affect of occasional large object allocations into a recycled blocks dominated by small holes. Section 5.2 shows that this mechanism is important for memory constrained heaps that tend to fragment more.

Parallelism. Immix supports multiple collector threads and multiple mutator threads. Our initial implementation does not support concurrent collection in which the mutator and collector execute in parallel. To achieve scalable multi-threaded allocation, thread-local allocators do not synchronize except when they request blocks from the global pool. Non-compacting collection 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. Coarse-grained sweeping is performed in parallel by dividing the heap among the available collector threads.

During compacting collection, as with any copying garbage collection, some form of synchronization is essential to avoid copying 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 compacting and non-compacting scenarios. This way we achieve all the benefits of compaction but only pay for the associated tracing overhead when it is actually required.

3.3 Implementation Details

During development, we explored many design variations and tested a variety of thresholds, finding repeatedly that the simplest approach was usually the most effective (Occam's Razor). This section discusses immix parameter choices and some performance-critical 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 block boundaries. 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 can not share any space smaller than a block. Smaller block sizes would thus lead to better space sharing, but higher synchronization overheads and a higher worst case fragmentation bound. Larger blocks would lead to 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 at small heap sizes.

We roughly size lines to match the architecture's cache line and to accommodate more than one object. The 128 byte line size reflects a tension between metadata overheads for small lines (since we require 1 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, and this sensitivity seems to be tied mostly to the space efficiency of line, since on average, a line size of 128B accommodates four objects for our benchmarks. See Blackburn et al. for more on allocation and live object size demographics [10]. Tuning this parameter seems therefore most closely tied to the programming language, rather than the architecture.

Recycling Policies. We explored numerous polices for block recycling. Our initial design prioritized blocks for recycling according to the amount of space available on each and avoided blocks with few available lines. We implemented an allocation table which contained a sorted list of recyclable blocks which thread-local allocators used in order. We also experimented with thresholds which avoided allocating into blocks which had very few available lines. To our surprise, the most effective approach was the much simpler, exhaustive round-robin approach described above, which recycles every partially used block by walking the heap in address order. We show in Section 5.3 that the overhead of recycling blocks with few available lines is only incurred with any regularity when space is very scarce, which is precisely when such diligence is most rewarded.

Conservative Marking. Immix may allocate objects across line boundaries. This choice impacts both 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 calculates the start and end addresses of each object and marks all lines in the interval. Unfortunately, these operations are expensive when compared to mark-sweep which simply sets a bit in the object header. To set all the line marks, the collector must interrogate the object's type information to establish its size, and then iterate through the address range marking the relevant lines. We found experimentally that both of these operations were quite expensive.

We use *conservative line marking* to avoid this overhead by leveraging object demographics which show that the overwhelming majority of objects are small (less than 128 bytes) [10]. We perform an optimization that requires the collector to differentiate small (sub 128B line-size) and medium (between line size and 8K) objects at *allocation time*, instead of collection time. We identify medium objects at allocation time by setting a bit in their header. This operation is cheap because the JIT compiler usually constant-folds the size test away at the allocation point, since in the common case, object size is known. At mark time, we omit the line mark for the last line of medium objects and we omit the extra line mark for a small object that spills into another line. Before allocating into a hole, immix corrects the line marks by conservatively marking the first line of every hole. At mark time, the collector examines the size bit and for the fairly rare medium objects, the collector looks up the object size, sets all but the last line mark for the object. Thus medium objects achieve an exact mark. For the commonly occurring small objects, immix simply sets the line mark for the line corresponding the object's start address, which is cheap since the object's start address is at hand. In the worst case, conservative marking wastes a line for every hole. Figure 2 shows example line marks due to conservative marking. Conservative marking 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 tracing in mark-sweep. For example, we examined the compiled IR for the allocation sequence to determine that setting a bit in the header of medium sized objects (Section 3.3) was compiled away in the common case with zero performance impact.

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. We believe that because the scanning code is longer, since it loops over and examines any child references, that it provides better latency tolerance for hiding the cost of the line mark operation.

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

4. Methodology

Benchmarks. We use all 18 benchmarks from SPECjvm98 and DaCapo (v. 2006-10-MR2) suites. The DaCapo suite [10] is a recently developed suite of real-world open source Java applications which are substantially more complex than the SPECjvm98 workloads. The characteristics of both are described in detail elsewhere [11].

Hardware and OS. We use the 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. (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. All CPUs operate in 32-bit mode, and use 32-bit kernels. We use the perfctr [25] library to gather hardware performance counters on the Core 2 Duo.

Jikes RVM and MMTk. We implement our algorithm in the memory management toolkit (MMTk) [9, 8] in version 2.9.1 of Jikes RVM [3, 2]. 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 compiler produces machine code when the VM first encounters each Java method. The adaptive compilation system then judiciously optimizes the most frequently executed methods [5]. 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.

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 Table 3). To limit experimental noise, machines are stand-alone with all unnecessary daemons stopped and the network interface down.

We use Jikes RVM's *replay* feature to factor out non-deterministic heap allocation into the heap by the compiler due to timer-based adaptive optimization decisions [10]. 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. Without replay the system initially baseline compiles each method and then later recompiles hot methods at higher optimization based on sampling. 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. In each experiment, we time the second iteration of the benchmark. The first iteration is dominated by compilation and startup costs. 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.

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.

5. Results

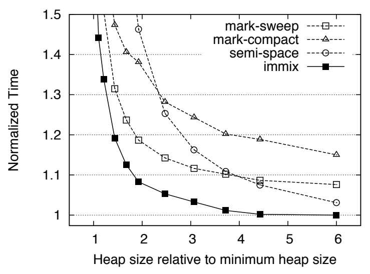
This section first shows that immix substantially outperforms mark-sweep, semi-space and mark-compact, across three architectures. We break down performance by mutator and collector. We use hardware performance counters to reveal the role of locality and minimum heap sizes to show space efficiency. Section 5.2 teases apart the relative importance of block and line size, block tracking, line tracking, conservative marking, compaction, head room, and demand driven overflow allocation. We show that immix is not terribly sensitive to line and block size and that the majority of allocation goes to recycled free blocks and when needed, demand driven compaction is able to eliminate fragmentation efficiently. Section 5.5 shows how immix gracefully handles pinning with very little performance loss. Section 6 discusses incorporating immix into a generational setting.

5.1 Overall Performance

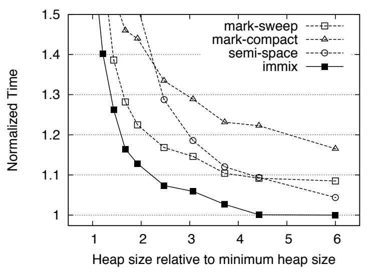
Figure 3 shows the geometric mean for all 18 benchmarks for three orthodox collectors and immix. Each graph normalizes performance to the best result on the graph and covers heap sizes from $1\times$ to $6\times$ the minimum heap in which mark-sweep will run each benchmark. Figures 3(a)-(c) show total performance for the three architectures, and Figures 3(d)-(f) show for the Core 2 Duo mutator time, mutator L1 cache misses and garbage collection time.

Figures 3(a)-(c) shows that immix outperforms all collectors on all heap sizes on all three architectures, typically by around 8-10%. Immix achieves this result by cherry picking the best features of the orthodox collectors: (1) Figures 3(d) and (e) show that immix matches the superior mutator times of semi-space, offering locality in the mutator due to similar numbers of L1 misses on the Core 2 Duo. L2 misses are similar across collectors due to the large L2 on the Core 2 Duo (measured, but not shown). (2) Figures 3(f) shows that immix matches the superior garbage collection times of mark-sweep.

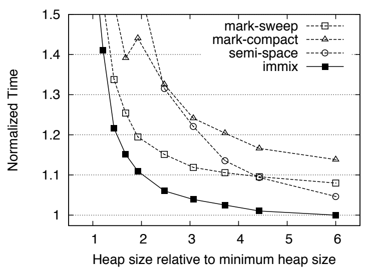
Figures 4 and Table 2 show that immix uniformly achieves its improvements; always improving over or matching the best of the other collectors. Figure 4(a)-(c) show total time on a log scale for three representitive benchmarks (worst, most substantial, best). These graphs include GenMS [8], Jikes RVM's production generational collector which has a copying nursery and mark-sweep older space, also a popular choice in product VMs. Figure 4(a) shows mtrt, where immix offers its smallest improvements compared to mark-sweep, semi-space, and mark-compact. It falls far behind GenMS due to the generational nature of this workload. The second benchmark, eclipse, is the most substantial in the suite and shows that for some workloads, immix not only performs much better than any of the other full heap collectors, but it approaches the performance of the fastest generational collec-



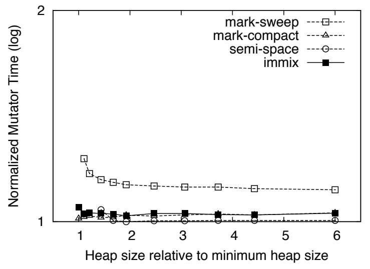
(a) Total Time Intel Core 2 Duo



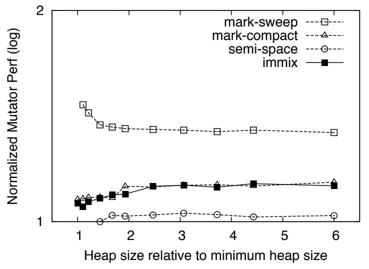
(b) Total Time AMD Athlon



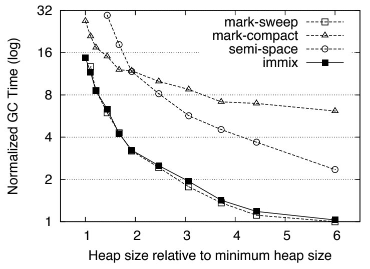
(c) Total Time PPC 970



(d) Mutator Time Core 2 Duo

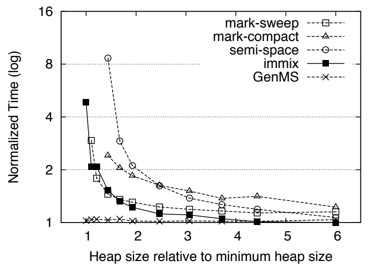


(e) Mutator L1 Misses Core 2 Duo

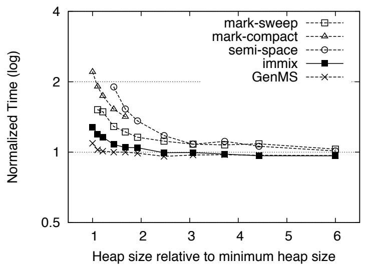


(f) Collector Time Core 2 Duo

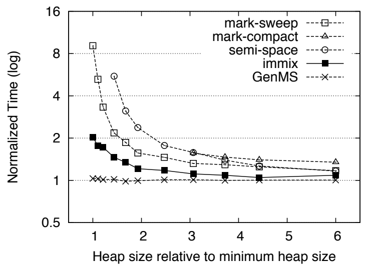
Figure 3. Geometric Mean Full Heap Performance Comparisons for Total, Mutator, and Collector Time



(a) mtrt Core 2 Duo



(b) eclipse Core 2 Duo



(c) jython Core 2 Duo

Figure 4. Representative Benchmark Performance Comparisons

tors. For jython however, GenMS is best. On jython, immix has the biggest performance advantage, outperforming mark-sweep by a factor of four in total time, in a very tight heap. Table 2 shows that immix (IX) improvements are uniform and robust across all 18 benchmarks against mark-sweep (MS), semi-space (SS) and mark-compact (MC). It uses representative results at two times the heap minimum normalizes total time to mark-sweep. Except mpegaudio which performs no GC at this heap size and degrades by only 0.3%, immix improves every program.

5.2 Breakdown of Algorithmic Features

This section explains how much and which features of immix contribute to its performance. Tables 1 shows how immix achieves locality and space efficiency. Table 3 shows how immix's space efficiency enables it to execute in very small heap sizes.

The first seven columns in Table 1 show marked objects in the allocation blocks for each benchmark at two times the minimum heap. We divide the blocks up into five categories: completely empty; $<$ 25% marked, between 25 and 50% marked, between 50 and 75% marked, and greater than 75% marked. Notice that at this 2$\times$ heap size, immix allocates in completely or mostly free blocks 79% of the time, which it needs to achieve locality, and only very occassionally (always less than 5%) allocates in mostly full blocks. On average, more allocation goes in less full blocks. 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$\times$ heap size, immix allocates 76% at a 6$\times$ 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.

The *overflow* column in Table 1 shows the percent of memory allocation for medium objects that immix gives to its overflow allocator. The two benchmarks jython and xalan substantially benefit from this mechanism, whereas the remaining benchmarks barely use this feature. The first two *Marking Wastage* columns show the relative benefits of line and block marking compared with our baseline, which includes compaction. We build a collector that only uses block or line marking, no compaction. We then determine the wasted amount at the beginning of each collection in the modified system based on blocks and lines to which it did not allocate. We express wastage as an overhead relative to the volume marked by immix. 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 compaction, wastage would inflate the amount marked by 23% on average. The column *consv* shows the memory wasted during allocation due to conservative marking. A few

	Allocation						Marking Wastage			Pinning		Compaction (1.5× Min Heap)					
												Compactions		Candidate Blocks		%	net
	clean blocks	< 25%	< 50%	< 75%	>= 75%	over-flow	block	line	consv	calls	pinned 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 1. Allocating, Marking, Pinning, and Compaction Statistics. Compaction at 1.5× Minimum Heap, All Others at 2× Minimum Heap

benchmark	time (ms)	MS	SS	MC	IX
compress	3708.36	1.000	0.997	1.011	0.994
jess	1586.43	1.000	1.278	1.272	0.853
raytrace	1076.38	1.000	1.396	1.381	0.935
db	6470.45	1.000	1.136	1.095	0.943
javac	2836.78	1.000	1.083	1.025	0.921
mpegaudio	2658.25	1.000	1.001	1.000	1.003
mtrt	1208.93	1.000	1.619	1.415	0.935
jack	2551.73	1.000	1.167	1.286	0.891
antlr	2325.60	1.000	1.346	1.263	0.948
bloat	9118.57	1.000	1.575	1.394	0.875
chart	7654.39	1.000	1.150	1.194	0.914
eclipse	37834.07	1.000	1.174	-	0.903
fop	1883.34	1.000	1.028	1.066	0.965
hsqldb	1536.75	1.000	1.253	0.851	0.858
jython	9515.32	1.000	1.516	-	0.772
luindex	10591.94	1.000	1.192	1.161	0.941
lusearch	12310.44	1.000	1.326	1.420	0.898
pmd	5620.18	1.000	1.420	1.406	0.939
xalan	11993.79	1.000	1.034	1.047	0.876
min		*1.000*	*1.001*	*0.851*	*0.772*
max		*1.000*	*1.619*	*1.420*	*1.003*
geomean		*1.000*	*1.248*	*1.192*	*0.908*

Table 2. Performance Details at 2× Minimum Heap

	Size					Immix		
	MB	MS	SS	MC	Immix	ZH	Line	Block
compress	19	1.00	1.37	0.95	1.00	1.00	1.05	1.21
jess	19	1.00	0.95		0.79	1.32	1.89	3.05
raytrace	16	1.00	1.38		1.06	1.00	1.19	1.63
db	19	1.00	1.58	1.00	1.00	1.00	1.00	1.16
javac	33	1.00	1.00	0.64	0.88	1.39	1.58	3.33
mpegaudio	13	1.00	1.15	0.85	0.92	1.00	1.15	1.46
mtrt	20	1.00	1.45		0.95	1.05	1.35	1.70
jack	17	1.00	1.00	0.71	1.12	1.35	1.41	2.06
antlr	24	1.00	1.33	0.83	0.83	0.92	1.46	2.29
bloat	33	1.00	1.48		0.97	2.24	3.61	5.79
chart	49	1.00	1.20	0.78	1.10	1.55	1.78	2.31
eclipse	84	1.00	1.42		0.76			
fop	40	1.00	1.28	0.80	0.83	1.00	1.40	2.53
hsqldb	127	1.00	1.39	0.83	0.92	1.01	1.06	1.57
jython	40	1.00	1.28		0.78	1.18	1.83	3.08
luindex	22	1.00	1.32	0.86	0.82	1.05	1.50	1.95
lusearch	34	1.00	1.24	0.97	0.88	1.21	1.29	1.68
pmd	49	1.00	1.31	0.94	0.86	1.33	1.51	3.53
xalan	54	1.00	1.07	0.87	0.78	0.81	1.37	1.85
min		*1.00*	*0.95*	*0.64*	*0.76*	*0.81*	*1.00*	*1.16*
max		*1.00*	*1.58*	*1.00*	*1.12*	*2.24*	*3.61*	*5.79*
geomean		*1.00*	*1.26*	*0.84*	*0.90*	*1.16*	*1.45*	*2.15*

Table 3. Minimum Heap Sizes

benchmarks, such as javac and fop waste 25 to 20%, but most benchmarks waste less than 6%.

Table 3 shows the consequences of memory efficiency on minimum heap size, comparing immix with the orthodox collectors. The first four columns normalize heap size to the minimum in which mark-sweep executes. The missing mark-compact entries are due to failed runs. The results show that for jack and chart, immix requires a slightly larger heap than mark-sweep, but on average, immix reduces the minimum heap size in which a collector can execute to 90% of what mark-sweep achieves, while still achieving markedly better performance. The last three columns show how the minimum heap size is influenced by eliminating (1) the default 2.5% compaction headroom, (2) compaction, and (3) both compaction and line-grain reclamation. Eliminating compaction and line-grain reclamation *Block*, thus only performing block-grain recycling, would require more than doubling the average minimum heap size. Performing line-grain marking, but no compaction *Line* still increases the minimum heap size significantly, on average 45% and up to 361%. Headroom also significantly benefits immix. With zero headroom *ZH*, 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.

5.3 Opportunistic Compaction

The last six columns of Table 1 show statistics for opportunistic compaction at a heap size of 1.5 times the mark-sweep minimum. As the heap grows, immix performs less and less compaction. The two compaction columns show the number of compactions and the percentage of collections that trigger compaction. Even at this relatively small heap size, 8 of the 18 benchmarks do require compaction and only compress triggers compaction on every collection. The columns labeled *Candidate Blocks* show the volume of blocks marked as compaction candidates (in KB, divide by 32KB for the number of blocks) 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

	Immix	Algorithmic Features				Block Size		Line Size	
benchmark	(ms)	Block	Line	No HR	No Ov	16KB	64KB	64B	256B
compress	3666	1.001	0.997	1.001	1.012	1.004	0.998	1.004	1.005
jess	1301			0.981	1.001	1.040	0.992	1.020	0.994
raytrace	971	1.071	1.003	0.996	1.011	0.976	0.956	0.971	0.959
db	6184	1.035	1.037	1.031	1.001	1.004	0.995	1.005	1.002
javac	2596		0.987	0.996	1.010	1.033	0.984	0.992	1.019
mpegaudio	2660	0.991	0.996	0.997	1.002	1.001	0.999	1.010	1.001
mtrt	1079	1.260	1.017	0.995	0.982	0.987	0.976	1.004	0.997
jack	2233		1.010	1.016	0.974	0.989	0.982	0.983	0.974
antlr	2080		1.061	1.047	1.036	0.995	1.066	0.976	1.009
bloat	7739				0.988	1.008	0.988	0.995	1.014
chart	6891		1.032	1.000	1.006	1.015	0.996	1.007	1.003
eclipse	32783				1.023	1.028	1.020	1.049	1.020
fop	1776		1.024	1.009	1.014	1.007	0.996	1.017	1.001
hsqldb	1373	1.090	1.078	1.284	1.002	0.997	0.923	0.985	0.913
jython	7076		3.146	1.811	1.081	1.025	0.989	1.057	1.005
luindex	9922	1.244	1.047	1.037	0.995	1.006	1.010	0.994	0.997
lusearch	10347	1.571	1.133	1.048	1.054	1.038	0.995	1.047	1.026
pmd	4998		1.035	1.053	1.000	1.002	0.995	1.001	0.998
xalan	10747	1.599	1.201	1.981	0.911	0.956	0.967	0.929	1.030
min		*0.991*	*0.987*	*0.981*	*0.911*	*0.956*	*0.923*	*0.929*	*0.913*
max		*1.599*	*3.146*	*1.981*	*1.081*	*1.040*	*1.066*	*1.057*	*1.030*
geomean		*1.188*	*1.117*	*1.107*	*1.005*	*1.006*	*0.991*	*1.002*	*0.998*

Table 4. Performance Sensitivity Studies Normalized to Immix at 2× Minimum Heap

on candidate blocks. The second to last column shows % reuse of recyclable blocks; the vast majority of compacted live objects are tucked into recyclable blocks rather than being allocated to completely free blocks. The last column shows the net yield of the compaction as a percentage of candidate blocks. Opportunistic compacation has limited success on compress and lusearch, but neither of these programs stress the collector much. On the remaining programs, compaction successfully converts 95% or more of the candidate space to completely free blocks. These statistics show that opportunistic compaction is only occassionally triggered and yet effectively compresses the heap on demand.

5.4 Performance Sensitivity

This section establishes the performance sensitivity and benefits of immix parameters and optimizations. Table 4 normalizes the total time performance results to base immix results at two times the minumum heap size. The last four columns show the sensitivity of Immix to the choice of block and line size. The base block size is 32KB and the base line size is 128B; we measure half and twice these sizes. Variation based on block size ranges from 7% better to 7% worse for large blocks. Large blocks show a slight improvement at this heap size, but they perform worse in smaller heaps. Smaller blocks have slightly less variation, but neither on average change immix much. The benchmarks eclipse and xalan are most sensitive to line size, and sensitivity grows in smaller heap sizes. Most benchmarks are not that senstive to line size, but 128B is more consistant and better across many heap sizes.

Columns three through seven in Table 4 show the performance benefits of the various elements of immix. Without anything but block recycling, the collector sometimes fails even with two times the minimum heap size. Adding line reclamation, but no compaction, still causes three benchmarks to fail. Similarly, compaction without 2.5% headroom of free blocks still fails on two benchmarks. Overflow allocation is less important for performance in this heap size, but helps robustness in small heap sizes (measured, but not shown). All these features contribute to immix robustness and performance benefits.

5.5 Pinning

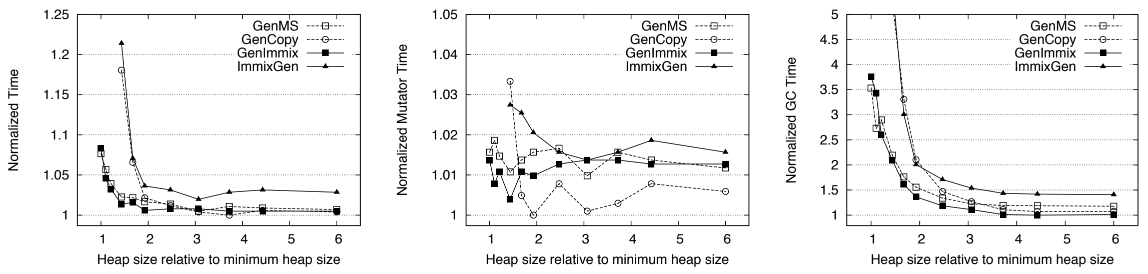
Opportunistic copying allows immix to elegantly support object pinning. Although Java does not support object pinning, it is an important feature of C# and internally, Jikes RVM's class libraries optimize for pinning in the classes `gnu.java.nio.VMChannel` and `org.jikesrvm.jni.VM_JNIFunctions`. In each case, assurance that an object will not move allows the code to avoid an additional level of indirection and buffering. Table 1 shows 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 already pinned by prior calls do not need to be pinned again. 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. Pinning is enabled in immix in all of the results reported in this paper. We performed detailed performance analysis (which we omit, for space) and found that pinning has no affect on performance for most benchmarks; has a very small advantage for three benchmarks which use pinning heavily; and very slightly degrades performance for a fourth benchmark.

6. Generational Collection

Generational collectors preferentially collect young objects and are generally considered the best performing algorithms. We demonstrate two generational variants of immix and compare them with with GenMS and GenCopy, two MMTk generational collectors. GenMS and GenCopy combine a variable-size copying nursery with mark-sweep and semi-space old spaces, respectively. We add GenImmix which uses an immix old space. These three collectors share the same address-segregated copying nursery and write barrier. Their old space implementations are the same as their full heap counterparts. None of them support pinning due to their copying nursery and each reserves a conservative copy space for nursery survivors.

We designed another generational collector, ImmixGen, to support pinning. It allocates new objects directly into the immix heap. ImmixGen identifies mature objects via their mark state, rather than their address, when performing write barriers or nursery collections. This design eliminates the need for a copy reserve. During a nursery collection, ImmixGen opportunistically copies nursery survivors which improves performance significantly over leaving the survivors in place.

Figure 5 plots the geometric mean of total, mutator, and collector times. GenImmix and GenMS essentially achieve the same average performance in the same heap sizes. Both GenImmix and GenMS improve over GenCopy in a small heap because of their space efficiency, i.e., GenCopy collection time increases more in



(a) Total Time (b) Mutator Time (c) Collector Time

Figure 5. Generational Collector Performance Core 2 Duo

small heaps. These three collectors achieve similar mutator locality because they share a contigous nursery which provides almost all the mutator locality. ImmixGen performs similarly except at small heap sizes, however it handles pinning. We did not spend much time building or experimenting with these versions of immix and leave to future work tuning immix for use in a generational setting. However, we believe ImmixGen is particularly promising because it only copies opportunistically.

7. Conclusion

This paper introduces a simple, novel garbage collector that organizes the heap at a coarse-grain block granularity and within each block at a fine-grain line granularity. Our comprehensive performance results show how this regime achieves collector efficiency, heap space efficiency, and mutator locality all at once, and therefore significantly out performs the orthodox collectors. Our design includes enhancements such as support for pinning, demand driven overflow allocation, conservative marking, and opportunistic compaction. We use information from the previous collection to drive compaction and compact on a per-object basis, marking or copying objects opportunistically to implement an elegant single pass algorithm. With these innovations, immix breaks the space, time dilemma forced by previous heap organizations.

References

[1] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtien. An efficient parallel heap compaction algorithm. In *ACM Conference on Programming Language Design and Implementation*, pages 224–236, Ottawa, CA, 2006.

[2] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), Feb. 2000.

[3] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, Denver, CO, Nov. 1999.

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

[5] 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.

[6] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In J. L. Knudsen, editor, *European Conference on Object-Oriented Programming, ECOOP 2001, Budapest, Hungary, June 18-22*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235. Springer-Verlag, 2001.

[7] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[8] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the ACM Conference on Measurement & Modeling Computer Systems*, pages 25–36, NY, NY, June 2004.

[9] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Scotland, UK, May 2004.

[10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, Oct. 2006.

[11] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Technical Report TR-CS-06-01, Dept. of Computer Science, Australian National University, 2006. http://www.dacapobench.org.

[12] 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.

[13] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.

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

[15] G. O. Collins. Experience in automatic storage allocation. *Communications of the ACM*, 4(10):436–440, Oct. 1961.

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

[17] Q. Feng and E. Berger. A locality improving dynmaic memory allocator. In *The ACM Workshop Memory Systems Performance*, pages 68–77, Chicago, IL, June 2005.

[18] R. J. M. Hughes. A semi-incremental garbage collection algorithm. *Software—Practice and Experience*, 12(11):1081–1084, Nov. 1982.

[19] B. S. Inc. Using the JRockit runtime analyzer. Technical report, 2007. http://-edocs.bea.com/wljrockit/docs142/usingJRA/looking.html.

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

[21] D. Lea. A memory allocator. Technical report, 1996. http://gee.cs.oswego.edu-/dl/html/malloc.html.

[22] 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. ACM.

[23] 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.

[24] P. McGachey and A. L. Hosking. Reducing generational copy reserve overhead with fallback compaction. In *The ACM International Symposium on Memory Management*, pages 17–28, Ottawa, CA, June 2006.

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

[26] P. W. Purdom, S. M. Stigler, and T. Cheam. Statistical investigation of three storage allocation algorithms. *BIT*, 11:187–195, 1971.

[27] B. Randell. A note on storage fragmentation. *Communications of the ACM*, 12(7):365–372, July 1969.

[28] J. M. Robson. Worst case fragmetation of first fit and best fit storage allcation strategies. *The Computer Journal*, 20(3):242–244, 1977.

[29] N. Sachindran, J. E. B. Moss, and E. D. Berger. MC^2: High-performance garbage collectionn for memory-constrained environments. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 81–98, Vancouver, Canada, Oct. 2004.

[30] K. Sagornas and J. Wilhelmsson. Mark and split. In *The ACM International Symposium on Memory Management*, pages 29–39, Ottawa, CA, June 2006.

[31] P. Styger. LISP 2 garbage collector specifications. Technical Report TM-3417/500/00 1, System Development Cooperation, Santa Monica, CA, Apr. 1967.

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

[33] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, number 986 in Lecture Notes in Computer Science, pages 1–116, Scotland, UK, Sept. 1995. Springer-Verlag.