Garbage Collection for Embedded Systems

David F. Bacon Perry Cheng David Grove

IBM T.J. Watson Research Center

ABSTRACT

Security concerns on embedded devices like cellular phones make Java an extremely attractive technology for providing third-party and user-downloadable functionality. However, garbage collectors have typically required several times the maximum live data set size (which is the minimum possible heap size) in order to run well. In addition, the size of the virtual machine (ROM) image and the size of the collector's data structures (metadata) have not been a concern for server- or workstation-oriented collectors.

We have implemented two different collectors specifically designed to operate well on small embedded devices. We have also developed a number of algorithmic improvements and compression techniques that allow us to eliminate almost all of the per-object overhead that the virtual machine and the garbage collector require. We describe these optimizations and present measurements of the Java embedded benchmarks (EEMBC) of our implementations on both an IA32 laptop and an ARM-based PDA.

For applications with low to moderate allocation rates, our optimized collector running on the ARM is able to achieve 85% of peak performance with only 1.05 to 1.3 times the absolute minimum heap size. For applications with high allocation rates, the collector achieves 85% of peak performance with 1.75 to 2.5 times the minimum heap size. The collector code takes up 40 KB of ROM, and collector metadata overhead has been almost completely eliminated, consuming only 0.4% of the heap.

General Terms

Experimentation, Languages, Measurement, Performance

Categories and Subject Descriptors

E.2 [**Data**]: Data Storage Representations—*Object representation*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

1. INTRODUCTION

Embedded devices have long since become the most widely deployed computing platforms in the world, and the trend is continuing to accelerate. For such devices, including motes [17], smart cards, cellular phones, and handheld organizers [37], the flexibility to dynamically download new functionality is increasingly important. Often the downloaded software is not under the control of the maker of or service provider for the device.

Therefore, a premium is placed on reliability of the downloaded code — while we are regrettably accustomed to crashes on the part of personal computer operating systems, consumers are much less accepting when "necessary" devices like cellular phones cease to function. As a result, the Java computing platform is becoming steadily more attractive for embedded devices due to its safety properties, since arbitrary downloaded software can not compromise the operating system or other applications. A key contributor to these safety properties is garbage collection [16, 28, 26].

While some applications have real-time requirements, there are many which do not. Furthermore, the additional complexity, space, and time overheads necessarily associated with real-time collectors are in direct conflict with many of the other requirements imposed by memory-constrained embedded systems.

The requirements on garbage collectors in such embedded environments are:

Code Size. It is imperative that the virtual machine consume as little code space as possible. As a result, many typical methods for improving collector performance are inappropriate because of the resulting complexity and concommitant increase in code size.

Memory Overhead. The overhead due to collector meta-data and memory fragmentation should be kept to an absolute minimum. As a result, semi-space copying collectors [20, 12] are not an option.

Compaction. Since many embedded applications run continuously for extended periods of time, the collector *must* be able to perform memory compaction, both to avoid arbitrary space consumption due to fragmentation, and to enable the virtual machine to release memory resources to the operating system as needed.

Reliability. System failure is not acceptable. This places a premium on both simplicity and on strong enforcement of invariants within the collector.

Smooth Performance. The likelihood that the application will run in a very constricted memory space is much higher than in

PC- or server-based virtual machines. Therefore, the collector's performance should degrade gracefully as memory is reduced.

Speed. Within the limits of the preceding requirements, the collector should be as fast as possible. Trading a small amount of space for a large improvement in time is acceptable for some, but not all, applications.

In this paper we discuss our experience in the construction of (non-real-time) garbage collectors for IBM's Java 2 Micro Edition (J2ME) virtual machine for the IBM WebSphere Micro Environment [24]. We have implemented several variants of two "micro-collectors" and several different object models for J2ME in order to explore the design space of collectors that meet the above requirements

While some implementation details are specific to our virtual machine architecture, the techniques we present are applicable to most virtual machines. Some of the object model optimizations are specific to Java, but many are applicable to other garbage-collected languages.

1.1 Problems with Traditional Techniques

The tight memory requirements place a number of restrictions on the system which will be counter-intuitive to those working on desktop or server virtual machines. These issues arise both in the allocation strategy and in the garbage collection methodology.

First of all, inlining the allocation sequence is only done if the inlined code is (statically) shorter than the save/call/restore code sequence, since inlining would otherwise lead to significant code expansion.

Secondly, many popular free space organization schemes are unacceptable due to their high rate of fragmentation. Examples include binary buddy [18, 27] and systems that compose all objects out of a single small block size. The latter eliminate all external fragmentation at the expense of greatly increased internal fragmentation as well as increased access times, especially for array elements [30, 33].

In our J2ME environment, we do not make use of generational collection [35], since despite its advantages, it consumes space for (1) the nursery, (2) write barriers in compiled code, (3) the root set for the nursery (implemented as a remembered set, a sequential store buffer, or with card-marking), and (4) additional code in the JVM image required by the increased complexity of the collector.

Other unfamiliar issues that arise in the embedded domain are the use of physical rather than virtually addressed memory, which makes a number of implementation techniques impossible; various types of segmented memory architectures, either due to the small architected word size of the processor, or to blocked allocation of non-virtual memory by the operating system; differing levels of memory performance (SRAM, DRAM, flash, etc.); and the requirement to reduce power consumption.

1.2 Organization

We present new techniques developed for garbage collection in memory constrained environments, describe our implementations, and quantitatively evaluate and compare them along the dimensions of code size, memory overhead, and speed, as well as measuring the relevant quantities which determine their performance.

2. THE MARK-COMPACT COLLECTOR

Embedded systems have tight memory requirements and applications are often long-running. Therefore, it is absolutely essential to be able to place a tight bound on memory loss due to fragmentation. This can not be done without compaction (or some other technique which moves objects).

Our mark-compact collector is based on Saunders' original mark-compact algorithm [32, 26]. It allocates linearly until the heap is exhausted and then compacts by sliding objects "to the left" (towards low memory). It therefore tends to preserve (or even improve) locality, and fragmentation is eliminated completely on every collection.

As in a semi-space copying collector, allocation is very fast: a simple bump pointer and range check. This allocation sequence has the advantage that it is short enough to consider inlining, although in our JVM we use a hand-coded, but out-of-line, allocation sequence.

However, note that on platforms that present a segmented, non-virtual memory interface (such as PalmOS [37]), fragmentation at the end of segments becomes an issue that must be addressed.

2.1 Compaction and Its Optimizations

The sliding compaction algorithm requires 4 phases:

1. **Mark:** Traverse the object graph beginning at the roots, marking each object encountered as live.
2. **Sweep:** Scan memory sequentially, looking for dead objects and coalescing them into contiguous free chunks. Compute the new address for each object and store a forwarding pointer in the object (see Section 2.2).
3. **Forward:** Change all object pointers to point to the forwarded value as determined by the Sweep phase.
4. **Compact:** From left to right, move objects to their new locations.

Typically, the Sweep phase is the most expensive since it needs to scan all of memory, while the other phases are proportional to the live data. The Mark and Forward phases are typically similar in cost, since they both essentially traverse the live objects and examine each field. The Compact phase is the fastest since it does not look inside objects, but just copies them a word at a time. Although the Forward and Compact phases scan the heap linearly, their costs are proportional to only the live objects since the previous Sweep phase has coalesced adjacent dead objects into contiguous free chunks.

It is in fact fairly straightforward to reduce by one the number of collection phases, a result which so far we have not been able to find in the published literature. We therefore present it here.

We observe that during the sweep phase, when we encounter an object, the forwarded addresses of objects to its left have already been computed. Therefore, we examine each pointer in the object in turn, and if it points to the left, we replace the pointer with the forwarded version stored with the destination object. If it points to the right, we leave it unchanged.

We can then omit the Forward phase entirely. The Compact phase is extended so that before moving an object (by sliding it to the left) we examine each pointer in turn, and if it points to the right, we know that (a) it was not forwarded in the previous pass, and (b) that the forwarding pointer is still available. Therefore, we forward exactly those pointers.

The result is an algorithm that traverses the heap 3 times instead of 4, which would seem to result in a lower load on the memory subsystem. However, we have found in practice that it makes almost no difference, and in fact tends to slightly slow down some programs. The reason is that there are now two passes that examine pointers in each object for forwarding. To do this they must look

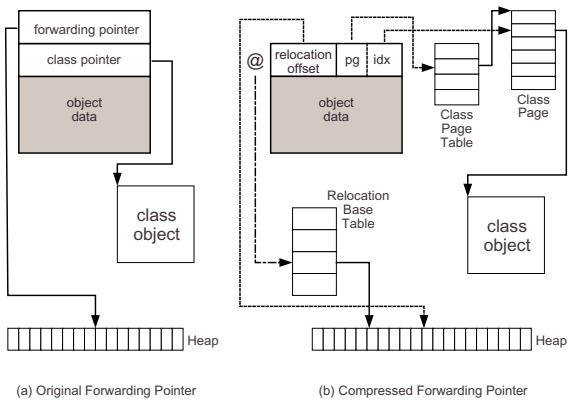

Figure 1: Implementation of compressed forwarding pointers using the relocation base table (RBT).

at the target of the pointer, which results in a random access pattern. Thus it appears that two sequential passes over live memory cost about the same as one pass with random access. The reason is that we are looking through the pointers in each object an extra time, and this is an expensive operation. By having a separate Forward phase, although we scan the live objects an additional time, we avoid scanning the pointers in the object a second time. Since there are fewer objects than pointers, the extra phase wins.

Therefore, we will not further consider this optimization, although it may give important performance benefits in systems with different languages, memory technologies, etc.

2.2 Forwarding Pointer Elimination

The compaction algorithm requires an extra forwarding pointer at the beginning of every object since, unlike a copying collector, the forwarding pointer cannot overlay any data or header fields. Due to the memory-constrained nature of embedded devices, we wish to avoid paying this space cost, without requiring any extra passes over the heap.

2.2.1 Encoded Class Indices

Every Java object contains a class pointer, which is used to find the table of virtual functions of the class, to perform class tests and cast operations, and to support various run-time system operations. The representation of a Java object with a fowarding pointer is shown in Figure 1(a).

We eliminate separate forwarding pointers by encoding the class pointer during the compaction phase, and then using the space made available in the class pointer word to store a compressed forwarding pointer. Instead of a class pointer we use a 14-bit class index, and obtain the class pointer by looking up the index in a table. To avoid the need for a large class index table (which would consume 64KB) we divide the class table into 1KB pages (each containing 256 class pointers). The *class page table* (CPT) contains pointers to the class pages. The CPT only requires 64 entries or 256 bytes. The loss of space due to internal fragmentation in the last class page is at most 1KB, and only 512 bytes on average.

The 14-bit class index is sub-divided into a 6-bit class page table index and an 8-bit class page offset. To reduce the overhead of CPT lookups, we use a single-element cache of the last lookup value.

Each class object must also contain its 14-bit class index (stored in a half-word). So the total overhead is 1.5 words per class plus 256 bytes plus 0-1020 bytes lost to internal fragmentation.

Class pointers are converted into class indices during the Forward phase, and are converted back into class pointers during the Compact phase.

2.2.2 Encoded Relocation Addresses

The reason that traditional compaction [32] requires an extra word per object is that a relocation address is computed for each object. Since we have freed space in the header word by encoding the class pointer with an index, we have a sub-word available to represent the relocation address. However, that relocation address must be encoded as well, since we do not have enough space for a full-width relocation pointer.

We observe that sliding compaction has the property that the relocation addresses of successive objects in memory increase monotonically. Therefore, for any region of memory of size s, as long as objects do not increase in size during relocation, the relocation address can be represented as the relocation address of the first object in the region plus an offset in the range $[0, s)$.

There are two potential sources of object expansion: one is the potential change in object size due to optimizations in object representation. These optimizations and the manner in which they avoid such expansion are described in Section 4. The other source of expansion is alignment requirements: an arbitrary number of objects may have been correctly aligned with no padding necessary at their original addresses, whereas their target addresses are misaligned. This can lead to a relocated region actually growing in size.

However, there is always a schedule of relocations that eliminates such mis-alignment. In particular, it is sufficient to align the first object in the page to the same alignment that it had in its original location. This is always possible, since if there is no space left to align it, we must be able to place it in exactly the same relative position. Preserving the alignment of the first object guarantees that there will be no subsequent growth within the memory region due to alignment changes.

Therefore, we divide memory into 128KB pages, and have a *relocation base table* (RBT) which contains the relocation address of the first live object in each 128KB page. We then use 15 bits in the header word (now available due to class pointer encoding) for the relocation offset (15 bits encode 32KW or 128KB of shift). The RBT is allocated at startup time based on the maximum heap size. For example, on a system with 16MB of memory, the RBT contains 128 entries, which consume 512 bytes. This is the only space overhead for relocation.

To determine the relocation address of an object, its (shifted) original address is used as an index into the RBT, from which the relocation base address is loaded. The relocation address is then the sum of the base plus the offset. The result is shown in Figure 1(b).

As with the CPT, we use a one-element cache of RBT translations to reduce the number of RBT lookups.

2.2.3 Our Implementation

In our systems, the class pointers are always in low memory. Therefore we do not actually compress the class pointers; we simply "steal" the high 8 bits for the relocation base table (RBT) offset. Therefore each RBT entry corresponds to 1 KB of memory (since it is used to represent a word offset), resulting in an RBT table space overhead of about 0.04%.

2.3 Mark-Coalesce-Compact

A variant of the mark-compact collector described so far is one that avoids compaction by skipping compaction entirely if it discovers enough contiguous free space. Compaction is only performed when a large allocation request can not be satisfied with contiguous memory, or if excessive fragmentation is discovered.

This technique has been used in a number of collectors. While it seems like it could provide large speedups, since it eliminates half of the collection phases, it eliminates the two fastest phases, so the performance impact is not as dramatic as might be expected. Nevertheless, it provides a potential improvement while introducing minimal additional complexity and code expansion, and may therefore be worthwhile.

2.3.1 Synchronization Issues

In fact, both the mark-compact and the mark-coalesce-compact collectors normally allocate into small thread-local chunks of memory. Otherwise, synchronization overhead would dominate the cost of allocation — causing roughly a 15% reduction in application throughput.

To eliminate this, the hand-coded assembly language allocation sequence attempts to allocate in the thread local area (typically 1-4 kilobytes, depending on object size demographics and the level of multiprogramming). If a large object is requested, or if the thread local area is full, a call is made to the synchronized allocator.

These synchronization issues have a significant impact on collector design. In particular, it means that the mark-coalesce-compact collector can not directly re-use all of the recovered space that it finds, but only contiguous free chunks sufficiently large to amortize the synchronization cost. We are currently experimenting with the tradeoffs in chunk size requirements versus collection frequency.

We are also investigating the creation of thread-local chains of small objects – but that inevitably leads to a collector architecture more like the one described in the next section. However, such an optimization may be particularly important in very tight heaps, since the smaller the heap the smaller the average contiguous free region.

3. PAGED MARK-SWEEP-DEFRAGMENT

The PMSD collector is a whole-heap, mark-sweep collector with optional defragmentation. The heap is divided into 1KB pages. Each page either holds meta-data that describes other pages or else holds application data. In our configuration, 1.5% of the heap is dedicated to meta-data. Pages that hold application data are categorized as holding small data (objects less than 512 bytes) or large data. Each small-data page has an associated size class (chosen from one of 25 sizes ranging from 16 bytes to 512 bytes). The page is sub-divided into blocks of the associated size [31]. A small object is allocated into the smallest free block that will accommodate it. Large objects consume multiple contiguous pages. The type and state of each page is stored in its corresponding address-indexed meta-data structure.

At the end of each garbage collection, contiguous free pages are coalesced into contiguous block ranges. There are two block range lists, one for holding singleton blocks and one for multi-block ranges. During allocation, page requests that result from (small) free block exhaustion are preferentially satisfied from the singleton block list. For multi-page requests and failed single page requests, a first-fit search from the multi-block list is used.

Whenever the free list of a size is exhausted, the dead (hence free) blocks of a small object page of the same size are linked together. The batching allows most small object allocation to be fast. If all small object pages of the request size are used, a completely fresh page is requested. To avoid expensive atomic operations on the free list, each thread has its own free lists, which are created dynamically in response to application demand.

Each garbage collection begins with a mark phase where traversal of all reachable objects from the roots causes the mark bits of live objects to be set. The sweep phase then clears the mark bits of live objects and designates blocks containing unmarked objects as dead blocks. In this phase, the overall fragmentation of the system is computed. If the fragmentation exceeds 25% or if the current allocation request is unsatisfiable due to fragmentation, defragmentation is triggered.

There are five sources of fragmentation in this scheme [1]. If a small object's size does not exactly match an existing size class, the next larger size class is chosen. This resulting per-object wastage is called *block-internal* fragmentation. Since the page size 1KB may not be a precise multiple of a size class, the end of each small-object page may be wasted. This is called *page-internal* fragmentation. Perhaps the most important source of fragmentation is *block-external* fragmentation which results from partially used pages of small objects.

Consider a program that allocates enough objects of the same size to fill 10 pages of memory. If every other objects dies and the program then ceases to allocate objects of that size class, then half of the blocks in those pages will be wasted.

Page-external fragmentation can result from the allocation of multi-page objects that leave multi-page holes. If there is a multi-page request is smaller than the sum of the holes but larger than a single hole, then the request will fail even though there are sufficient pages.

Finally, since using even a single block of a page forces the page to be dedicated to a particular size class, up to almost one page per size class can be wasted if that size class is only lightly used. In the worst case, the *size-external* fragmentation is the product of the page size and the number of size classes.

Page-internal fragmentation is eliminated by moving small objects from mostly empty pages to same-sized pages that are mostly full pages [7]. Since there is no overlap of live and dead data, the forwarding pointer can be written in the class pointer slot without any compression. In some cases, page-level defragmentation is necessary to combat page-external fragmentatoin. Currently, pages holding small objects can be relocated to empty pages by a block-copy but there is no multi-page defragmentation support. This temporary shortcoming puts PMSD at a disadvantage for applications that make heavy use of large arrays.

Because our size classes are statically chosen, the size-external fragmentation can be severe for very small heaps. One solution is to choose size classes dynamically. At runtime, neighboring size classes are coalesced if the smaller size class is not heavily utilized. In this way, the slight increase in page-internal fragmentation can be more than offset by the decrease in size-external fragmentation. Fewer size classes can also combat page-internal fragmentation. On the other hand, the same adaptive technique can create more size classes to densely cover size ranges where objects are prolific. In this case, more size classes will decrease block-internal fragmentation.

4. SINGLE WORD OBJECT HEADER

Typical Java run-time environments use 3-word object headers: one word for the class pointer, one word containing a thin lock [4], and one word containing a hash code and garbage collector information. Furthermore, mark-compact collectors previously required an additional word to hold the forwarding pointer, which is only used during garbage collection, as shown in Figure 2(a).

However, in an embedded environment, this profligate use of space is not acceptable.

In Section 2.2, we showed how to eliminate the extra word for the forwarding pointer, resulting in the object model of Figure 2(b), which is also that used by the paged mark-sweep collector.

Bacon, Fink, and Grove [3] showed how the object header (with-

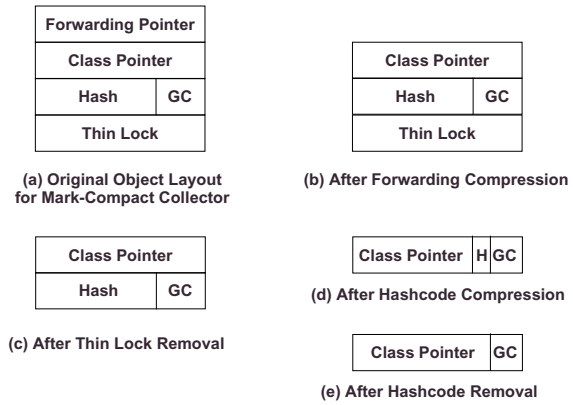

Figure 2: Object models, requiring from 4 down to 1 word per object.

out a forwarding pointer) can be compacted into a single word, at the cost of requiring a mask operation on the class pointer, or into two words at virtually no cost.

The optimizations can be briefly summarized as follows: the thin lock is removed from the object header and instead is treated as an optional field that is implicitly declared by the first synchronized method or synchronized(this) block that appears in the class hierarchy. The result is shown in Figure 2(c).

Since most objects are not synchronized, and virtually all objects that are synchronized have synchronized methods, this gives virtually the same performance as a dedicated thin lock in all objects, and yet only requires space in a very small number. A special case is instances of Object, which are provided with a thin lock since one of the only uses for such instances is to serve as a lock for synchronized blocks.

4.1 The Mash Table

In a collector that does not perform compaction, objects never move and the hash code can simply be implemented as a function of the object's address. However, compaction is a requirement for embedded systems.

Previous work [3] showed that the space for the hash code could be reduced to only two bits, by using the address of an unmoved object as its hash code. When an object whose hash code has been taken is moved, its original address is appended to the object and the hash function makes use of this value instead.

As a result, the extra state for each object is normally only 2 bits for the hash code and a few bits for the garbage collector state. If the collector state bits are sufficiently few, then class objects can be aligned on (for instance) 16-byte boundaries, providing 4 unused low bits in which to store object state. Then, to use a class pointer, the low bits must be masked out with an *and immediate* instruction. The result is shown in Figure 2(d).

However, this technique of hash code compression suffers from two signficant disadvantages: (1) it consumes bits in the header word of each object, even though hash codes are rarely used; even worse, those bits are modified during execution and during garbage collection, which tends to complicate the implementation. (2) It causes objects to change size during their lifetime, which significantly complicates garbage collection. In the mark-compact collector, the forwarding pointer compression technique relies on the property that live objects in a range of memory will be compacted into an equally sized or smaller range of memory. If objects can increase in size when they are moved, this is no longer true. In the paged mark-sweep collector, defragmentation is efficient because it is performed within a size class. If objects grow when moved, they may change size classes.

Benchmark	Classes	Function
Chess	92	Machine chess player
Crypto	75	Encrypt/decrypt (multiple algorithms)
kXML	75	XML parsing, DOM tree manipulation
Parallel	71	Parallel merge sort, matrix multiply
PNG	70	PNG image decoding
RegExp	80	Regular expression search

Table 1: Programs in the Embedded Microprocessor Benchmark Consortium (EEMBC) Java GrinderBench suite.

Therefore, rather than storing hash codes of moved objects at the end of the object, we store them in a structure we call the *mash table*. The mash table is a hash table of hash codes.

The mash table works as follows: when an object's Java hashCode() method is called, we compute a hash value based on its current address in storage. This is its hash index into the mash table. To avoid confusion, we call this the object's *mashcode*. If we find an entry whose key is the current address of the object, we return the corresponding value in the mashtable as its hash code. If we do not find an entry for the object in the mashtable, we insert a key/value pair where both the key and the value are the object's current address.

At garbage collection time, objects may move or die. Thus we must in essence perform garbage collection of the mashtable: references to dead objects are removed, and references to moved objects have their key field updated to the new address and are relocated in the mashtable based on the new mashcode. This is done after marking and forwarding have been performed, but before actual relocation of objects.

The mashtable therefore allows us to remove *all* extra state from the object header, leaving only the garbage collector bits, as shown in Figure 2(e).

The only complication with the mashtable is that we must prevent errors due to concurrent access to the mashtable by multiple threads. On a uniprocessor, if the virtual machine only switches between Java threads at "safe points", then this is achieved by not having safe points in the mashtable code.

On a multiprocessor or in the absence of safe points, we can take advantage of the fact that obtaining the hash code is an idempotent operation (the hashcode of an object never changes, and its mashcode only changes during garbage collection, which is synchronized already). Therefore we can have a small per-processor or per-thread cache of hashcode values, which allows us to reduce the frequency of synchronization with the global mashtable to acceptable levels.

In our current implementation we have implemented the mashtable in C++ as a separate structure. However, in order to be robust in the face of pathological cases it is necessary to be able to resize the mashtable and collect unused mashtable entries. Trying to do this in a separate region of memory is complex, error-prone, and inefficient. Therefore, in the next generation we plan to implement the mashtable in Java as a collection of private helper methods of java.lang.Object. This will include a helper method that can obtain the physical address of an object, and a helper method that is called by the system at the end of garbage collection to rehash the moved objects.

Config	Mfr.	Make	Model	CPU	Clock	L1	L2	RAM	OS	Version
IA32	IBM	Thinkpad	T23	Pentium III-M	1.2 GHz	32KB	512KB	1GB	Windows 2000	5.00.2195
ARM	Sharp	Zaurus	SL-6000	XScale PXA-255	400 MHz	32KB	—	64MB	Embedix (Linux)	2.4.18

Table 2: Measurement Platforms

	Allocation			Allocated (MB)				Maximum Live (KB)			
Benchmark	Objects	Obj/s	MB/s	4	3	2	1	4	3	2	1
Chess	5,736,819	99,418	2.08	163.7	141.8	119.9	98.0	39	37	35	32
Crypto	674,218	18,894	1.12	48.1	45.4	42.8	40.2	71	70	69	68
kXML	6,200,624	115,074	3.41	231.1	207.4	183.7	160.0	286	261	241	224
Parallel	1,375,140	37,139	1.20	54.8	49.6	44.4	39.2	263	248	228	211
PNG	942,306	26,993	2.28	86.7	83.1	79.5	75.9	106	101	97	96
RegExp	120,032	2,604	0.14	7.3	6.8	6.4	5.9	45	41	35	31

Table 3: Effect of header compression on heap consumption for header sizes from 4 to 1 words. Effect on both total allocated data as well as on maximum live heap size are shown.

4.2 Elimination of Header Masking

The single-word object models of Figures 2(d) and (e) only contain a class pointer and a few state bits. Thus when the system makes use of the class pointer (for virtual method dispatch or dynamic type tests), it must first mask off the low bits of the header word which are not part of the class pointer. This both slows down the code and increases the code size, due to the extra instruction.

However, after eliminating the hash code the only remaining object state bits are 1-3 bits for the garbage collector. As long as we perform stop-the-world garbage collection (as with both of our implemented collectors), the collection state bits are only used during the collection itself, and not during normal execution. Therefore, they are zero during normal execution, and the masking operation can be eliminated!

5. MEASUREMENTS

We have implemented two basic collectors with a number of variants in terms of per-object space overhead and the associated run-time support. In this section we present an evaluation of our collectors using the Embedded Microprocessor Benchmark Consortium (EEMBC) Java GrinderBench benchmarks [19]. These benchmarks are summarized in Table 1.

We have measured the collectors on two platforms: a Pentium-based IBM Thinkpad and an ARM-based Sharp Zaurus. We refer to these configurations throughout the text as the IA32 and ARM configurations, respectively. Details of the measurement configurations are shown in Table 2.

Both configurations use a bytecode interpreter with a light-weight JIT compiler that compiles frequently executed methods. The JIT compiler performs a moderate amount of optimization, since more optimization requires both more RAM at run-time and more ROM for the optimizing compiler. Infrequently executed methods are interpreted to conserve space and reduce JIT overhead.

5.1 Collector-Independent Measures

We begin by evaluating the effect of the space optimizations on collector-independent characteristics. These are shown in Table 3. The first three columns give a general picture of the allocation behavior of each of the benchmarks: the total number of objects allocated, and the allocation rate in both objects/second and megabytes/second (the latter two figures are for our most efficient collector — mark-compact with single word headers — run with a heap twice the size of maximum live memory on the ARM). For each per-object overhead, from the original four words down to a single word, we show the effect on both total allocated bytes and on the maximum live data size.

The first thing to notice is that reducing the header size can have a dramatic effect on the total bytes allocated (a 40% reduction for `Chess` and a 30% reduction for `kXML`).

However, the effect on maximum live memory is considerably less (18% and 21% respectively). This indicates that the average size of long-lived objects is larger than the average size of all allocated objects, which is not surprising.

Furthermore, there is enormous variation in the allocation rates: `kXML` allocates over 50 times as much data as `RegExp`. There is also considerable variation in the maximum live memory: `kXML` has a maximum live memory size almost 7 times larger than `Chess`.

However, there is no correlation between allocation rate and maximum live memory: `Chess` has the smallest maximum live memory but allocates only slightly slower than `kXML`. This issue is actually quite important when deciding how to evaluate the performance of collectors with respect to different benchmarks, since one can normalize to either maximum live memory or to allocation rate, and a compelling case can be made for both measures.

5.2 Overall Performance

We measured several variants of the two collectors described in this paper: the mark-compact (MC) collector and the paged mark-sweep-defragment (PMSD) collector. The initial per-object overhead in the MC collector was 4 words: a 3-word header plus an additional word for the forwarding pointer. We implemented forwarding pointer compression, thin lock removal, and the mash table, which allowed us to reduce the per-object overhead for each collector to only 1 word.

Due to details of the IBM J2ME implementation, there were extra read-only state bits kept in the object header which we did not have time to remove; therefore, in configurations using the one-word header, every access to the class pointer includes an additional `AND` instruction to mask the low bits. This slows down virtual function dispatch, class tests, and down-casting. However, this penalty is offset by the fact that the one-word header also has a simpler allocation sequence, so `new` operations are faster.

The software configurations are named MC.4 through MC.1 for the mark-compact collector with the corresponding header sizes,

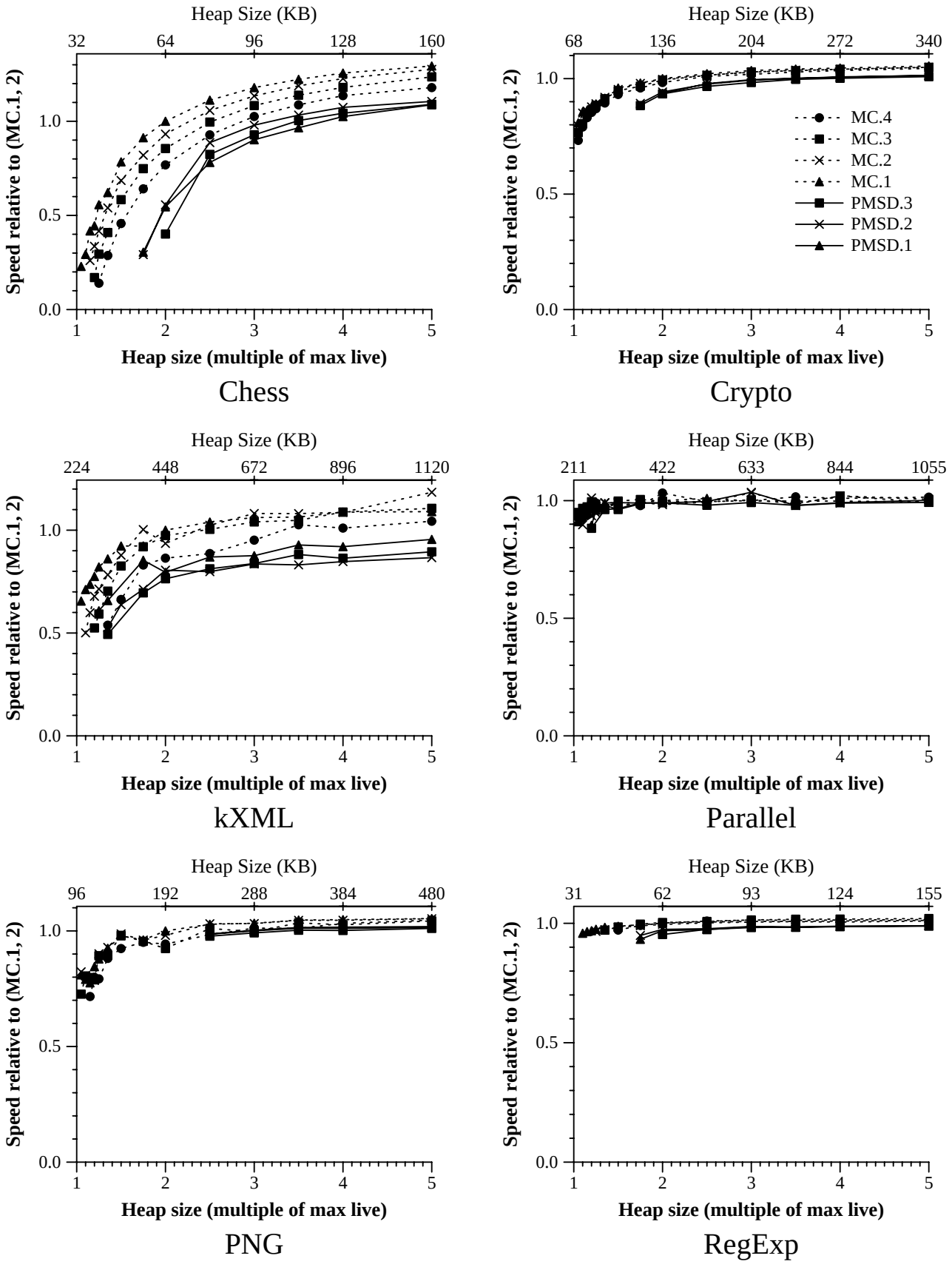

Figure 3: End-to-end execution time (ARM). Speed is relative to the best collector (mark-compact with 1-word headers, or MC.1) at a heap size of two times the maximum live data.

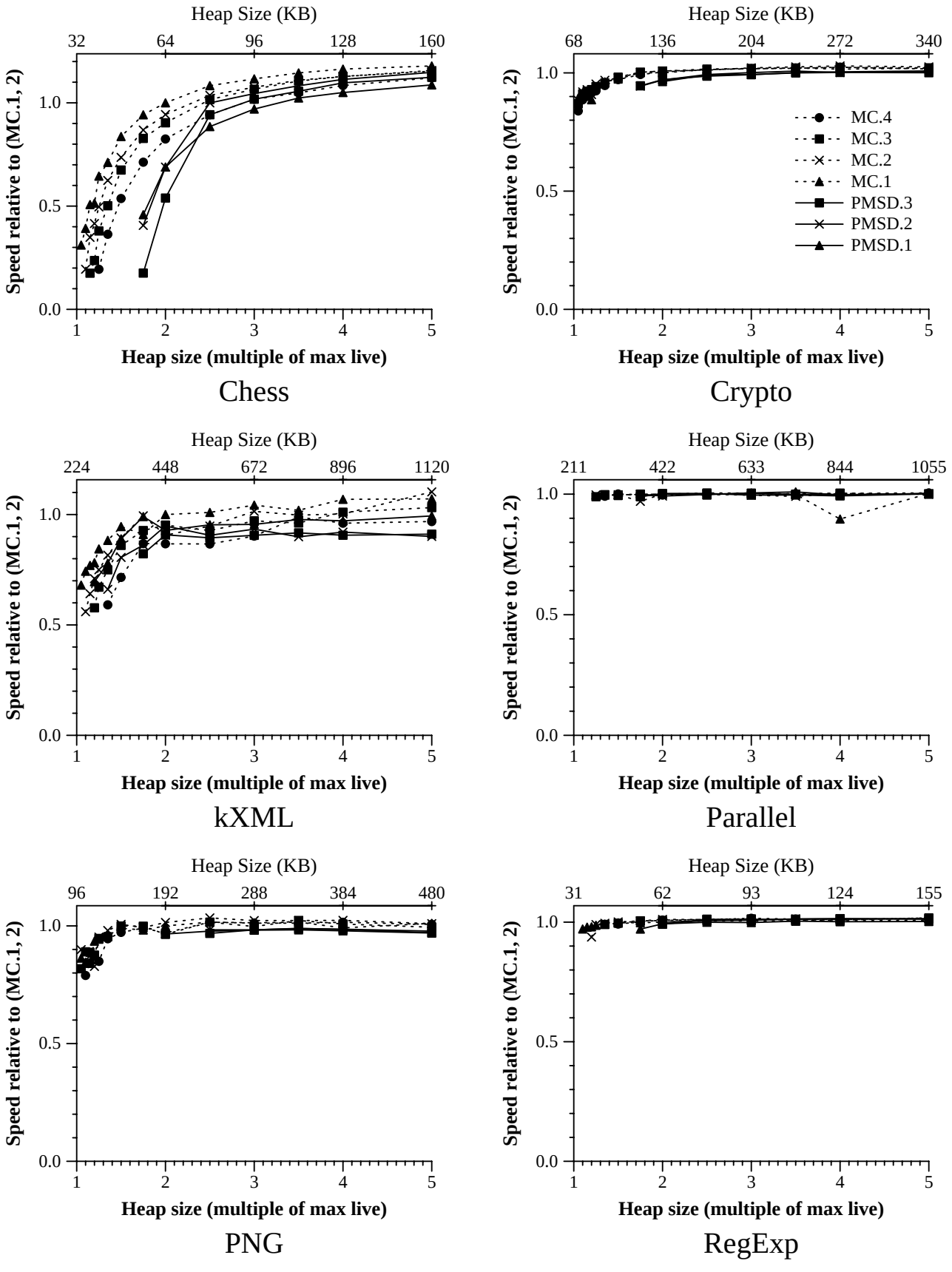

Figure 4: End-to-end execution time (IA32). Speed is relative to the best collector (mark-compact with 1-word headers, or MC.1) at a heap size of two times the maximum live data.

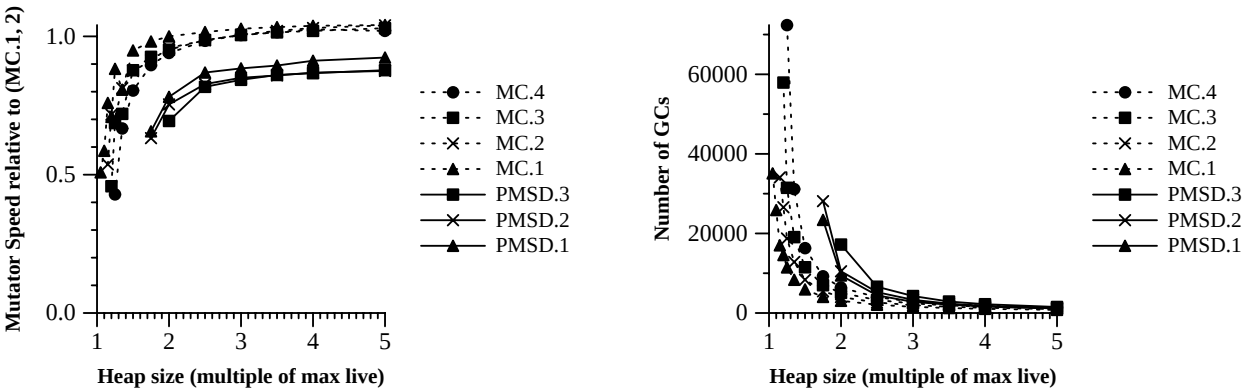

Figure 5: Chess benchmark: Performance details on the ARM. Mutator speed (left) and number of garbage collections (right).

and PMSD.3 through PMSD.1 for the paged mark-sweep-defragment collector. The collectors are measured at heap sizes from 1 to 5 times their maximum live memory (which is also the minimum heap size in which they can run). For brevity, when we say "heap size 2" we mean "heap size of 2 times the maximum live data size for this application".

We begin by examining the effect of garbage collection on absolute, end-to-end performance, as shown in Figures 3 (for ARM) and 4 (for IA32). Our discussion will focus on the ARM measurements, since they are much more relevant for embedded systems. On IA32, the entire heap often fits into the L2 cache.

The figures are normalized with respect to the performance of the mark-compact collector with a 1-word header (MC.1) run with heap size 2. We have chosen this configuration as our baseline because MC.1 generally gives the best performance, and heap size 2 is large enough to get past the "knee" in the performance curve. Furthermore, due to the constrained nature of embedded devices we do not generally want to use more than twice the necessary memory.

In general, the mark-compact (MC) collector consistently outperforms the paged mark-sweep-defragment (PMSD) collector, in both speed and in its ability to run in very small heaps (below heap size 2). The speed differential is even larger on the ARM than on the IA32 configuration. We will examine the reasons for these differences in detail below.

The benchmarks can be characterized in 3 groups: high, medium, and low collector loads.

The high-load benchmarks are `kXML` and `Chess`. Both allocate at a high rate, both in terms of MB/second and objects/second, and both allocate a large amount of total data. However, while `kXML` has the largest maximum live heap size (224 KB with 1-word headers), `Chess` has the second *smallest* maximum live heap size (only 32 KB).

This leads to an interesting question: should we base performance measurements of applications on their maximum live data size, or on their allocation rate? While `Chess` appears to be the worst-performing benchmark in Figure 3, it must be noted that at heap size 5, the absolute heap size is only 160 KB, which would correspond to heap size 0.7 for `kXML` – it could not even run!

On the one hand, it hardly seems fair to penalize an application for minimizing its maximum live heap size — to get a better "score", all a programmer would need to do would be to insert a large unused static array. On the other hand, it is important to know how well a program will run at its limits, since we expect this mode of operation to occur more frequently on embedded devices than otherwise. In the end, developers and evaluators must be aware of the distinction and carefully consider their metrics.

The medium-load benchmarks are `PNG` and `Crypto`: they each have similar (modest) allocation rates, maximum live memory, and total allocated bytes. They achieve excellent performance: with a heap size of only 1.3 (for `PNG`) and 1.25 (for `Crypto`), they achieve over 85% of the performance at heap size 5. Both medium-load programs perform about 1000 garbage collections at these heap sizes. Because the heaps are small and the collector is efficient, this does not impose a very large penalty on the application.

Finally, the low-load programs, `Parallel` and `RegExp`, have almost no collector overhead: even with a relative heap size of only 1.05, they achieve over 90% of peak performance. For `RegExp`, this is easy to understand because it allocates very slowly — almost ten times more slowly than the next slowest allocating benchmark (`Parallel`). However, `Parallel` allocates at about the same rate as `Crypto`, which is quite memory-sensitive below heap size 2.

Once again, the reason is the difference in absolute scale: the maximum live heap size of `Parallel` is almost exactly 3 times as large as that of `Crypto`. Therefore, the graph for `Crypto` is simply providing much higher resolution near the point of failure. In particular, for programs with the same allocation rate, a collection will be triggered each time they allocate the difference between their live heap size and the actual heap size. Assuming that the live heap size is proportional to the maximum live heap size, this means that the program with the larger live heap will appear to have better performance near the asymptote, although in fact there is no difference.

5.3 Mark-Compact vs. Paged Mark-Sweep

The mark-compact collector consistently out-performs the paged-mark-sweep collector and is able to run in much smaller heaps, as can be seen in Figures 3 and 4. It is important to understand the scope in which these results can be interpreted: only in our virtual machine, only in embedded applications, or more broadly?

MC is a simpler collector along many different axes. Fundamentally, it uses a much simpler heap organization, in which objects are simply allocated one after another in the heap. As a result, there is no fragmentation, almost no metadata, and the allocation operation is simpler.

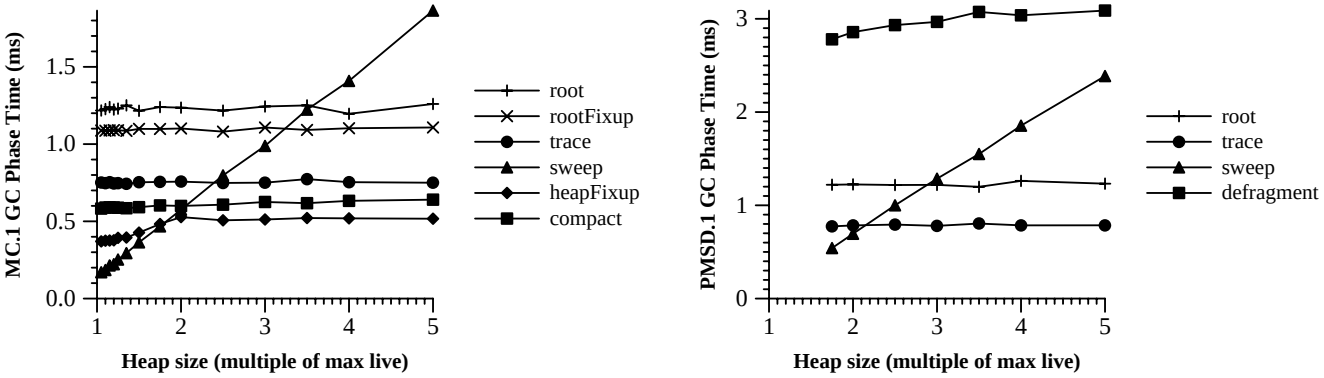

Figure 6: Chess benchmark: Garbage collector phase times on the ARM. Mark-compact (left) and paged mark-sweep (right).

Furthermore, since there is no need for the collector to manage multiple size classes and block metadata, the collector is simpler, resulting in simpler and more compact code, and fewer invariants. Finally, since objects are always allocated contiguously and compacted in the same order, locality is generally better.

Figure 5 shows how some of these issues manifest in the Chess benchmark. On the left, we have isolated the performance of the application ("mutator") by subtracting collector time. The graph shows that most of the performance gap at large heap sizes is due to the difference in mutator speed. This is due to a combination of factors: the slower allocation sequence, the periodic "formatting" operation when a new empty block is acquired for the free list of a particular size class, and the poorer locality of the resulting data.

While we believe it is possible to eliminate the formatting operation by combining it with work done by the collector which already traverses this portion of the heap, the other factors will remain. Thus PMSD will continue to suffer about a 4-8% performance penalty, even at large heap sizes.

At small heap size, fragmentation is a significant problem for PMSD. One important aspect of embedded programs that we observe is that the ratio between the size of the heap and the size of the largest object does not scale linearly. In a 1 GB heap, it is common for the largest object to be between 10 KB and 1 MB, for a ratio of 1000:1 up to 100000:1. However, for the EEMBC benchmarks, we see ratios as small as 10:1.

As a result, the application is vastly more sensitive to external fragmentation. The chance that there is a contiguous free chunk of memory that is $1/1000$ or $1/100000$ of the total heap size is quite high, even when the total heap size is not that much larger than the maximum live data size. However, the chance that there is a contiguous free chunk that is $1/10$ of the heap size is generally very low.

Our implementation of PMSD only moves large objects when it can find some other contiguous free range for them (small object defragmentation is used to maximize the availability of such free ranges). This is partially responsible for PMSD's inability to run in small heaps. To solve this problem, PMSD would have to perform sliding compaction for large objects (essentially turning it into a PMSD/MC hybrid) or to make large objects (namely big arrays) discontiguous in memory. This solution is called *arraylets* [1], and is quite effective at solving this problem — at the expense of slowing down array operations. In the context of a highly optimizing JIT compiler, this overhead can be greatly reduced, but with a simple JIT on an embedded device this is less attractive.

The more fundamental problem is that in small heaps, there is a tension between internal and external fragmentation. Since the heap is small, there are not that many pages. Therefore it is desirable to have a relatively small number of size classes — but this results in internal fragmentation (wasted space at the end of each object). Increasing the number of size classes reduces internal fragmentation, but leads to under-utilization of the size classes (external fragmentation).

The effect of this can be seen in the right-hand graph of Figure 5, which shows the number of garbage collections performed. PMSD approaches its vertical asymptote much more quickly due to space lost to fragmentation — in effect, it is able to use a significantly smaller proportion of the memory. The effect is particularly pronounced in Chess, since the absolute heap size is so small (32 KB).

Figure 6 shows the cost of the individual collector phases for both MC and PMSD. These graphs show the real-world impact of the difference in theoretical complexity between the sweep phase and the other phases of collection: sweep is $O(\mathrm{heap})$, while the other phases are $O(\mathrm{live})$, and therefore remain flat. As a result, the sweep phase dominates collection time at large relative heap sizes; at small relative heap sizes it is similar or even less than the time required by the other phases. PMSD has 4 phases instead of 6, but the phases are more expensive.

Some further tuning of PMSD, in particular reducing the defragmentation level, should further improve the performance at large heap sizes, allowing it to close the performance gap slightly.

6. RELATED WORK

Chen et al. [11] evaluate the power consumption properties of different parts of memory in an embedded JVM, and discuss collection strategies to minimize power consumption, particularly in a banked memory system where banks can be powered on and off individually.

In subsequent work, Chen et al. [10] use dynamic compression techniques to reduce the memory requirements of the application. They use two strategies: first, when heap space is exhausted, they perform compression on infrequently accessed objects. Second, they avoid allocating infrequently used fields of objects.

Such techniques are complementary to our approach, which emphasizes compaction of the object model and collector metadata, and maximizing performance of the general-purpose collector.

In the period from the early 1960's to the mid-1970's, Lisp systems ran with similar amounts of real memory as are available in today's smaller embedded environments, and there is therefore con-

siderable related work from this time period [31, 22, 14, 7, 8], well summarized by Cohen [15].

However, significant amounts of this pioneering work was driven by the desire to reduce paging. The semi-space copying collectors were a response to this pressure [20, 12].

Garbage collection in many early Lisp systems was considerably simplified by virtue of the fact that all memory consisted of CONS cells. This assumption was also implicit in the design of Baker's Treadmill real-time collector [6].

Siebert [33] has advocated a similar approach to eliminating external fragmentation in systems with variable object sizes: there is a single block size (32 or 64 bytes) and all objects larger than that size are made up of multiple blocks which are not necessarily contiguous. Arrays are represented as trees.

There are two major problems with this approach: the first is that it simply trades external for internal fragmentation, which can easily reach 50%. Second, access to large objects becomes expensive — in particular, array element access, normally an indexed load instruction, becomes a tree walk operation. Performance overheads can therefore be very large.

A number of variations of memory management based on regions [31] have been tried [9, 21, 34]. While automatically inferred regions can reduce the load on the garbage collector, they can not satisfactorily handle objects that have lifetimes that are not stack-like. Explicit regions significantly complicate the programming model, lead to brittle code, and expose more run-time errors. We have shown that garbage collection can run in constrained memory with good performance, obviating many of the reasons for using regions.

Johnstone [25] has claimed that fragmentation is not a problem in "real world" applications and that it is feasible to build non-compacting collectors. However, his measurements are for unrealistically short-lived programs. Furthermore, as our measurements have shown, fragmentation is a much bigger problem in the small heaps typical of embedded devices, since the ratio of the size of the largest object to the total size of the heap is much larger. In the embedded space, all collectors must perform compaction.

Much of the work on real-time garbage collection overlaps in its concerns with pure embedded collection, although the real-time concerns often lead to reduced throughput and increased complexity, both of which we have striven to avoid [2, 5, 13, 36, 23].

For real-time or embedded systems, it is very important to be able to know the memory requirements of a given application. One approach is to analyze the live memory requirements using a combination of programmer annotation of pointer types and recursion depths, and automatic analysis [29].

7. CONCLUSIONS

We have implemented and empirically assessed variants of two garbage collectors specifically designed for the unique requirements of small embedded devices. An important part of making these collectors efficient was developing algorithmic improvements and compression techniques that allow us to eliminate almost all of the per-object overhead that the virtual machine and garbage collector traditionally require.

For embedded applications, our mark-compact algorithm is uniformly superior to the paged mark-sweep algorithm: it is almost uniformly faster, runs in significantly smaller heaps, and consumes only half as much code space. While this is true for both the IA32-based laptop and the ARM-based PDA device, the speed differential is significantly higher on the PDA, suggesting that the greater inherent complexity of the paged mark-sweep algorithm is a poor match to an embedded CPU, with its simpler instruction set architecture, shallower pipelines, and reduced instruction-level parallelism.

Our techniques for object metadata compression require only a single word of metadata per Java object, even for the mark-compact algorithm which previously always required an extra word to allow for object relocation during garbage collection. Smaller objects mean a lower allocation rate, a smaller maximum live data set size, and a larger effective cache size. These properties result in higher performance and lower heap size requirements.

However, these benefits are not without cost: the more tightly packed the object metadata, the less redundancy and the more complex the invariants that must be maintained. As a result, there is a noticable cost in reliability, testing, and maintenance requirements. However, these costs are likely to be worth paying on virtual machines with a large installed base in memory-limited devices, such as cellular phones, smart cards, and sensors.

For applications with low to moderate allocation rates, our optimized mark-compact collector is able to achieve 85% of peak performance with only 1.05 to 1.3 times the absolute minimum heap size. For applications with high allocation rates, the collector achieves 85% of peak performance with 1.75 to 2.5 times the minimum heap size. The collector code requires only 40 KB of ROM, and collector metadata overhead has been almost completely eliminated, consuming only 0.4% of the heap.

Acknowledgements

We thank Martin Vechev and the anonymous referees for their helpful comments on the paper, Mark Wegman for stimulating discussions, and Trent Gray-Donald, Derek Inglis, Andrew Low, Ryan Sciampacone, and Dave Streeter for assistance with the IBM Java 2 Micro Edition (J2ME) virtual machine.

8. REFERENCES

[1] BACON, D. F., CHENG, P., AND RAJAN, V. T. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, California, June 2003). *SIGPLAN Notices, 38*, 7, 81–92.

[2] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices, 38*, 1, 285–298.

[3] BACON, D. F., FINK, S. J., AND GROVE, D. Space- and time-efficient implementation of the Java object model. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming* (Málaga, Spain, June 2002), B. Magnusson, Ed., vol. 2374 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 111–132.

[4] BACON, D. F., KONURU, R., MURTHY, C., AND SERRANO, M. Thin locks: Featherweight synchronization for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, Canada, June 1998). *SIGPLAN Notices, 33*, 6, 258–268.

[5] BAKER, H. G. List processing in real-time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.

[6] BAKER, H. G. The Treadmill, real-time garbage collection without motion sickness. *SIGPLAN Notices 27*, 3 (Mar. 1992), 66–70.

[7] BOBROW, D. G., AND MURPHY, D. L. Structure of a LISP system using two-level storage. *Commun. ACM 10*, 3 (1967), 155–159.

[8] BOBROW, D. G., AND MURPHY, D. L. A note on the efficiency of a LISP computation in a paged machine. *Commun. ACM 11*, 8 (1968), 558.

[9] BOLLELLA, G., GOSLING, J., BROSGOL, B. M., DIBBLE, P., FURR, S., HARDIN, D., AND TURNBULL, M. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, 2000.

[10] CHEN, G., KANDEMIR, M., VIJAYKRISHNAN, N., IRWIN, M. J., MATHISKE, B., AND WOLCZKO, M. Heap compression for memory-constrained Java environments. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications* (Anaheim, California, Oct. 2003). *SIGPLAN Notices, 38*, 11, 282–301.

[11] CHEN, G., SHETTY, R., KANDEMIR, M., VIJAYKRISHNAN, N., IRWIN, M. J., AND WOLCZKO, M. Tuning garbage collection for reducing memory system energy in an embedded Java environment. *ACM Transactions on Embedded Computing Systems 1*, 1 (Nov. 2002), 27–55.

[12] CHENEY, C. J. A nonrecursive list compacting algorithm. *Commun. ACM 13*, 11 (1970), 677–678.

[13] CHENG, P., AND BLELLOCH, G. A parallel, real-time garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices, 36*, 5 (May), 125–136.

[14] COHEN, J. Garbage collection of linked data structures. *ACM Comput. Surv. 13*, 3 (1981), 341–367.

[15] COHEN, J., AND NICOLAU, A. Comparison of compacting algorithms for garbage collection. *ACM Trans. Program. Lang. Syst. 5*, 4 (1983), 532–553.

[16] COLLINS, G. E. A method for overlapping and erasure of lists. *Commun. ACM 3*, 12 (Dec. 1960), 655–657.

[17] CULLER, D. E., HILL, J., BUONADONNA, P., SZEWCZYK, R., AND WOO, A. A network-centric approach to embedded software for tiny devices.

[18] DONAHUE, S. M., HAMPTON, M. P., DETERS, M., NYE, J. M., CYTRON, R. K., AND KAVI, K. M. Storage allocation for real-time, embedded systems. In *Proc. of the First International Workshop on Embedded Software* (Tahoe City, California, Oct. 2001), T. A. Henzinger and C. M. Kirsch, Eds., vol. 2211 of *Lecture Notes in Computer Science*, pp. 131–147.

[19] EMBEDDED MICROPROCESSOR BENCHMARK CONSORTIUM. Java GrinderBench version 1.0. URL www.eembc.org, 2004.

[20] FENICHEL, R. R., AND YOCHELSON, J. C. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM 12*, 11 (Nov. 1969), 611–612.

[21] GAY, D., AND AIKEN, A. Memory management with explicit regions. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, Quebec, June 1998). *SIGPLAN Notices, 33*, 6, 313–323.

[22] HANSEN, W. J. Compact list representation: definition, garbage collection, and system implementation. *Commun. ACM 12*, 9 (1969), 499–507.

[23] HENRIKSSON, R. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[24] IBM CORPORATION. Websphere micro environment. URL www.ibm.com/software/wireless/wme, 2004.

[25] JOHNSTONE, M. S. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, Dec. 1997.

[26] JONES, R., AND LINS, R. *Garbage Collection*. John Wiley and Sons, 1996.

[27] KNUTH, D. E. *Fundamental Algorithms*, second ed., vol. 1 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[28] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM 3*, 4 (1960), 184–195.

[29] PERSSON, P. Live memory analysis for garbage collection in embedded systems. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems* (Atlanta, Georgia, May 1999). *SIGPLAN Notices, 34*, 7, 45–54.

[30] RITZAU, T., AND FRITZSON, P. Decreasing memory overhead in hard real-time garbage collection. In *Proceedings of the Second International Conference on Embedded Software* (Grenoble, France, Oct. 2002), A. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 2491 of *Lecture Notes in Computer Science*, pp. 213–226.

[31] ROSS, D. T. The AED free storage package. *Commun. ACM 10*, 8 (Aug. 1967), 481–492.

[32] SAUNDERS, R. A. The LISP system for the Q-32 computer. In *The Programming Language LISP: Its Operation and Applications*, E. C. Berkeley and D. G. Bobrow, Eds. MIT Press, Cambridge, Massachussetts, 1964, pp. 220–238.

[33] SIEBERT, F. Eliminating external fragmentation in a non-moving garbage collector for Java. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (San Jose, California, Nov. 2000), pp. 9–17.

[34] TOFTE, M., AND TALPIN, J.-P. Region-based memory management. *Information and Computation* (Feb. 1997). An earlier version of this was presented at POPL94.

[35] UNGAR, D. M. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, 1984), P. Henderson, Ed. *SIGPLAN Notices, 19*, 5, 157–167.

[36] VIRDING, R. A garbage collector for the concurrent real-time language Erlang. In *Proc. of the International Workshop on Memory Management* (Sept. 1995), H. Baker, Ed., vol. 986 of *Lecture Notes in Computer Science*, pp. 343–354.

[37] WILSON, G., OSTREM, J., AND BEY, C. *Palm OS Programmer's Companion*, vol. 1. Palm Source, Inc., 2003. Document no. 3004-008.