A High-performance Garbage Collector for Standard ML

John H. Reppy

AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

We have designed and implemented a new garbage collector for the Standard ML of New Jersey System (SML/NJ). This collector has higher performance, lower latency and generally requires less physical memory than the existing SML/NJ collector. In addition, it is able to exploit the large secondary caches found on modern workstations. This paper describes the design of the collector, and presents comparative performance data that demonstrates the above performance claims.

1 Introduction

Standard ML of New Jersey (SML/NJ) is the most widely used implementation of Standard ML (SML) [MTH90, AM91]. SML/NJ is a high-quality implementation, with a sophisticated compiler [App92], and reasonable run-time environment [SML]. It has proven robust enough to serve as the platform for major systems consisting of tens of thousands of lines of code, such as HOL [GM93], eXene [GR93], and the SML/NJ compiler itself [AM91]. While SML/NJ has been remarkably successful, there have long been complaints about the performance of its garbage collector. Although its overall performance is considered good, it suffers from long pauses and large memory requirements.

This paper describes a new garbage collector for SML/NJ that, relative to the old collector, reduces garbage collection overhead by up to a factor of three or more, provides better performance, is less intrusive and requires less physical memory.[1] The new collector is a hybrid multi-generational collector. It has a fixed-size allocation arena (or nursery) that can be sized to take advantage of the large secondary caches found on modern workstations. It uses copying collection for most objects, but has a mark-sweep collector for managing large data objects, such as code. It also uses a new card-marking scheme for tracking intergenerational references. The collector is extremely portable; its only operating system requirement is a way to allocate memory.

In the next section, we give some background on the garbage collection issues related to ML, briefly describe the old collector, and discuss its problems. Then we given an overview of the new collector and describe its design in three sections: first we describe object representation and allocation, then the architecture of the heap, and lastly, the new collector itself. Following this is a comparison of the new and old collector's performance, in which we show that the new collector is faster, has lower latency, and uses less memory. We then discuss related work and conclude with a discussion of on going and future work.

[1]The collector is now part of the working version of SML/NJ, and will be included in the next public release of the SML/NJ system (sometime in early 1994).

2 Background

2.1 Copying and generational garbage collection

Both the old SML/NJ collector and the collector described in this paper are generational copying collectors. In a basic copying collector, the heap is divided into two spaces, called *to-space* and *from-space*. The user's program (called the *mutator*) allocates objects in to-space. When to-space is exhausted, the garbage collector is invoked. First it *flips* the two spaces, and then it *traces* the reachable objects in from-space, *forwarding* them into to-space. After all the reachable data has been copied, the from-space storage is reclaimed. Tracing is accomplished by starting with a set of *roots* (usually the registers and stack), and forwarding the objects that are directly referenced by them. The collector then *sweeps* to-space, forwarding from-space references as it goes. This sweeping effectively does a breadth-first traversal of the reachable data, with to-space acting as the work-queue.

Generational garbage collection is a technique that takes advantage of two properties of most garbage collected languages: most objects die young, and inter-object references tend to point from younger objects to older ones [Ung84]. In a generational collector, objects are segregated by age, with the younger generations being garbage collected more frequently. Since the younger generations have higher mortality rates, this allows the collector to work more efficiently. One difficulty with generational collection is that references from older objects to younger ones must be remembered. Since these references arise from mutation of existing objects, it is necessary to modify the mutator to track them.[2] For more information on basic garbage collection techniques, see Wilson's survey paper [Wil92].

2.2 The garbage collection requirements of ML

The heap usage profile of SML, and particularly SML/NJ, is different from that of Lisp or object-oriented languages such as Smalltalk. Thus, many of the empirical results of studying garbage collection for Lisp and Smalltalk may not apply to ML. As it turns out, ML has a number of static and dynamic properties, which make it a particularly good language to garbage collect. In the following, we describe the typical heap usage behavior of ML programs, as observed by us and others, as well as some of the requirements imposed by the semantics of SML.

Allocation rates SML/NJ has extremely high allocation rates; as much as an order of magnitude higher than traditional Lisp or Smalltalk systems [Ung86, Zor89]. Tarditi and Diwan have measured allocation rates of one word per 6–10 instructions executed [DTM94]. This is primarily because of the absence of a stack; the role of stack frames is played by heap allocated *continuation closures* [App92]. This high allocation rate has two important consequences: one, there is a guaranteed high mortality rate for newly allocated objects, and two, the mutator consumes address space at a vicious rate.

Object updates The semantics of SML distinguish between mutable and immutable objects, and the garbage collector is also able to make this distinction. SML programs tend to be mostly applicative. Typically, less than 1% of live data is mutable, and mutations are rare. For

[2] It is possible to do this in hardware, but not in a cost effective way.

example, the SML/NJ compiler does an update once for every 110 words it allocates, and only 60% of those updates are remembered. The small amount of mutable data, and the low rate of mutation means that SML is particularly suited to generational garbage collection.

Equality There are two aspects of equality in SML that affect garbage collection: equality is extensional (there is no general pointer equality), and equality is polymorphic (a function may test for equality on unknown types). The former property has been exploited by so-called replication-based collectors that permit multiple copies of objects to exist during incremental collection [NR87, HL93, NO93]. The latter property means that either the compiler must thread equality methods through the execution as implicit parameters (à la Haskell type classes [PJ93]), or there must be descriptors in objects that allow the polymorphic equality operation to parse them. In SML/NJ, we have taken the latter approach.

2.3 The old collector

The latest release of SML/NJ (version 0.93) uses a simple, semi-generational, collector developed by Appel [App89]. This collector stores the heap in a single contiguous chunk of memory, which is allocated using the UNIX `brk` system call. The top half of the heap is used to allocate new objects (*new-space*). When the new-space is full the garbage collector does a *minor* collection, evacuating the live objects from new-space to the bottom of the heap (*old-space*); the top half of the remaining free space is then made into new-space, and allocation begins again. At some point old-space will occupy half of the heap, at which point the collector does a *major* collection.[3] A major collection first forwards live objects from old-space to the free area at the top of the heap. Then the live data, which is now compact, is shifted back to the bottom of the heap. The heap is then resized, based on the amount of live data and various parameters.

Two dedicated registers are used by the mutator for managing the new-space: the `AllocPtr` points to the next word to allocate, and the `LimitPtr` marks the end of the allocation arena. The `LimitPtr` points some fixed number of bytes (currently 4096) below the true top of the allocation arena. Heap limit tests are done at the beginning of extended basic blocks; if the maximum allocation on any path through the block is less than the buffer size, a simple register-register comparison can be used for the test.

As with all generational collectors, the collector must treat references from old-space to new-space objects as roots for minor collections. The only way these references can be created is by assignment, so the SML/NJ compiler generates code to keep a list of all update operations, called the *store list*. The store list is allocated in the heap. The collector scans the store list at the beginning of each minor collection.

2.4 Problems with the old collector

Although the old collector is simple, portable and reasonably fast, it is a serious performance bottleneck, and suffers from several problems:

[3]Since old-space can actually occupy slightly more than half of the heap, the collector uses a clever pivot trick to avoid running out of memory (see [App89]).

- The lack of true generations means that major collections always copy every word of live data; for large applications this is unacceptable.
- Major collections occur frequently enough and take long enough that interactive responsiveness is destroyed.
- Programs with "hot-spots" of update activity cause the garbage collector to exhibit poor performance on minor collections.
- Applications with large amounts of code have poor garbage collection performance, since the code is heap allocated and copy collected.
- Since the collector uses the system call `brk` to allocate heap space, it is not possible to use the system supplied `malloc` in the run-time system. Unfortunately, there are a number of useful system services that use `malloc` in their implementation (e.g., `gethostbyname`).[4]

Since SML/NJ is now being used for increasingly substantial applications, these limitations have become a serious problem.

3 An overview of the collector

The new collector is designed to address the above problems, but before discussing design details, a quick overview of the collector architecture is in order. The new collector organizes the heap into a fixed-size allocation arena, and from one to fourteen older generations. The collector is actually two collectors: one for the allocation space (the *minor* collector), and one for the generations (the *major* collector). After each minor collection, a check is made to see if the first generation needs collecting. If so, a major collection is invoked. The major collector starts by determining the oldest generation to collect, which we call the *currently collected* generation. The collector collects all of the generations up to the currently collected generation. Objects in the older generations are grouped into different *arenas*, based on their kind. The collector uses a BIBOP (*BIg Bag Of Pages*) to map addresses to the appropriate collector data structures.

4 Object representation and allocation

The new collector uses the same basic allocation scheme and object representation as the old collector, with a few refinements that are described below.

An ML value is represented by a single word, and can either be a pointer to a heap object (*boxed*), or an immediate integer value (*unboxed*). The two low-order bits are used to tag an ML value as follows:

[4] In some recent operating systems (e.g., Solaris 2.x), the use of `malloc` is ubiquitous, which has caused porting headaches.

Low-order bits	Meaning
`00`	Pointer (boxed value)
`10`	Object descriptor
`01`	Integer (unboxed value)
`11`	Integer (unboxed value)

Descriptors are not ML values, but are distinguished to allow scanning the heap without starting at object boundaries.

There are two kinds of heap objects: those that contain pointers, and those that do not (called *atomic objects*). All elements of a pointer-containing object (e.g., a record) are tagged ML values, whereas the elements in an atomic object (e.g., a string) are untagged and may have sizes other than the machine's word size. Objects are allocated on word boundaries (some, such as floating-point numbers, may have stricter alignment requirements), and occupy an integer number of words. The first word is a descriptor, which consists of a 6-bit tag and a 26-bit length field.[5] As described above, the low-order two bits of a descriptor are `10`; the other four bits define the object's *representation class*.

Most objects are allocated by inline code; the cost is three or four instructions plus the cost of initialization, which leads to very fast allocation. Non-fixed length objects, such as arrays, are either allocated by assembly code (when they are less than 512 words), or by C code (when they are larger). Since large objects tend to be long-lived, they are allocated in the first generation, instead of in the allocation space.

Objects are usually referenced by a pointer to the word following the descriptor, but there are two cases in which pointers into the middle of an object can occur. Mutually recursive functions share closure records, which requires a pointer to a different slot for each function [App92].[6] Since descriptors are distinguished from ML values, the garbage collector can scan backwards to find the beginning of an object. The other case is pointers into code objects. The compiler generates a single code object for each compilation unit, which contains multiple entry points and embedded constants. In the old collector, this required special descriptors for embedded literals and backpointers in the code objects (this is described in [App90]); the new collector can use the regular descriptors for embedded literals, and does not require backpointers (see Section 6.3).

5 The heap organization

5.1 Allocation arena

The allocation arena is a fixed-size contiguous region in which the mutator allocates new objects. As with the old collector, the `AllocPtr` and `LimitPtr` registers are used by the mutator to allocate objects and to check if garbage collection is needed. Figure 1 gives the general layout of the arena. One difference with the old collector is the way that the store list is managed; this is described in Section 6.2.1.

[5] Obviously, this changes for 64-bit architectures.

[6] The new garbage collector supports other examples of pointers into non-atomic objects (e.g., for indexing an array), but the compiler does not exploit this yet.

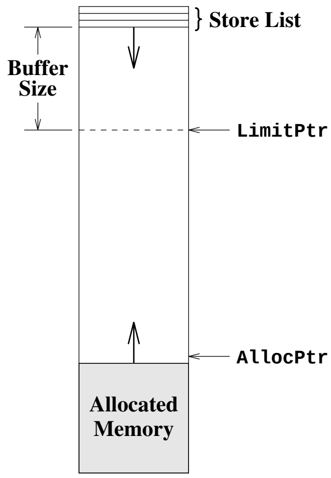

Figure 1: The allocation arena layout

5.2 The older generations

In addition to the allocation arena, there are up to 14 generations, each composed of arenas for the different classes of objects. Currently, there are five classes of objects supported: records, pairs, strings, arrays, and code objects. Records are the most common of ML objects. The reasons for segregating the other classes are as follows:

pairs These are two-element records, and typically make up 20-30% of the non-code objects surviving the allocation arena. For this reason, they are segregated and stored without a descriptor, which results in a 10-15% space savings.

strings These are the atomic objects (other than code), and do not have to be swept for pointers.

arrays These include all non-atomic mutable objects. They are segregated to give a higher density for the card marking scheme used to track intergenerational references (see Section 6.2.2).

code objects These tend to be large and long-lived, and often comprise 50% or more of the live data. They are managed using a mark-sweep collector (see Section 6.3).

Each generation consists of four arenas; one each for records, pairs, strings and arrays. The code objects of the generation are kept in a list (described in Section 6.3). Each arena is a contiguous piece of memory, and the arenas of a generation are allocated as a single chunk of memory. When a generation is collected, a second set of arenas is allocated for the generation's to-space; once the collection is over the storage for the generation's from-space is released. Since allocating and freeing these chunks of memory can involve significant system call overhead, the collector caches the unused chunks for the younger generations.

Objects in a generation are either *young* or *old*; when a generation is collected, live old objects are promoted, while live young objects stay in the same generation (and become old). There is a "high-water" mark associated with each arena. It is set to the top of the arena after the arena is collected. Objects below the high-water mark are old, while those above are young (i.e., the objects promoted since the last collection). This scheme is derived from Wilson's *Opportunistic Garbage Collector* [WM89b].

5.3 The BIBOP

The address space of the processor is logically divided into 64Kb *pages*.[7] Each page is dedicated to a specific use, and a BIBOP map is maintained that reflects this assignment. Each entry in the BIBOP table is 8-bits, split into a 4-bit generation number, and a 4-bit object class ID; the pair of these defines which arena the page is in. Values 0 and 15 are reserved for special purposes, so this scheme allows up to 14 generations of up to 14 different object classes (currently, there are six different class IDs). For a 32-bit address space, the BIBOP is a flat 2^{16} element array; larger address spaces will be supported by a multi-level table.

The BIBOP is used by the collector to detect from-space references; this is done by mapping a reference to a BIBOP entry, and then comparing the entry's generation with the currently collected generation. If the BIBOP generation is the same or younger, then the reference is from-space, otherwise it is to-space. The correctness of this test relies on the fact that a word in the to-space of a collected generation is only swept once.

5.4 The memory subsystem

The new collector is extremely portable; its only operating system dependency is the interface to the operating system's memory allocation mechanism. This is isolated as a library that supports the allocation and deallocation of BIBOP-page aligned chunks of memory. These operations are used to allocate and release the arenas for the generations. The memory subsystem is designed to coexist with `malloc`, which allows free use of C library routines in the runtime system. In fact, the collector uses `malloc` for managing its own data structures.

There are implementations of this library on top of MACH's VM primitives, System V's `mmap` mechanism, and the old UNIX `brk` system call. In the latter case, the library also provides an implementation of the `malloc` library. Porting the memory subsystem to new operating systems is straightforward; it should also be possible to take advantage of features, such as MACH's external pagers [CNS92], in some implementations of the subsystem.

6 The new collector

The new garbage collector actually consists of two collectors: a *minor* collector, and a *major* collector. The minor collector is responsible for collecting the allocation arena. It is designed to have minimal overhead and a small memory footprint; in some sense, it can be viewed as a specialization of the major collector (e.g., objects are always forwarded to the first generation).

[7] Of course, the page size is configurable.

6.1 The major collector

The major collector is a conventional generational copying collector. It starts by determining how many generations to collect; this is done by a checking a conservative estimate of the amount of data that might be forwarded into a generation against the amount of free space; if the free space is smaller, then the generation must be collected. The collector allocates memory for to-spaces for all of the collected generations. The collector then sweeps the roots, which consist of the registers plus the dirty cards of the generations older than the currently collected generation. We use a new technique for reducing the number of dirty cards swept (cf., Section 6.2.2). Once the roots have been swept, the collector sweeps to-space until all live objects have been forwarded. Then the from-space memory can be released. In the remainder of this section, we discuss the more novel aspects of the collector.

6.2 Mutator issues

Generational garbage collectors must deal with the intergenerational references created by mutation. The new collector uses two mechanisms for this: a *store list* for allocation space references, and a *card-marking* scheme for references between the older generations.

6.2.1 Store list

When the mutator updates a reference cell or array element, it records the updated address on a store list. This is much like the store list of the old collector, except that the list is represented as a stack that grows down in the allocation space (see Figure 1). The `LimitPtr` register is used as the top-of-stack pointer for the store list (offset by the allocation buffer size). Adding an element to the store list takes only two or three instructions, and only one memory reference. When a reference cell is updated with an unboxed value, it is not necessary to record the update in the store list. The compiler divides update operations into three kinds: *always-boxed*, *never-boxed*, and *maybe-boxed*. In the case of never-boxed updates, no store list code is generated. For maybe-boxed updates, it is possible to generate a straight-line code sequence (three additional instructions) that dynamically filters out unboxed updates. The linear representation of the store list improves on the old collector's linked representation in two ways: it has lower mutator overhead (both in time and space), and the minor collector has to do less work to scan the store list.

The minor collector uses the store list as a root set. While sweeping the store list, it records intergenerational references in the older generations using a card-marking scheme, which is described in the next section.

6.2.2 Card marking

A card-marking scheme [WM89a] is used for tracking intergenerational references among the older generations. The basic idea is that the array-arenas are logically divided into fixed-size *cards* of 2^k bytes (k is typically in the range $6 \ldots 10$). A card is marked as *dirty*, if there is a reference from the card to an object in a younger generation; a *card map* is kept to keep track of dirty cards. When the major collector is collecting generations 1 through n, the card maps for generations $n + 1$ and above are checked for dirty cards, and those dirty cards are scanned for possible roots.

There are several different possible representations of card maps; we have experimented with three of these. The most compact approach is to use a bit-vector; this also has the advantage that long runs of clean cards can be swept quickly (i.e., 32 cards at a time on a 32-bit machine). Chambers and Ungar have suggested using a byte per card, on the grounds that it makes marking cheaper (setting a bit requires read-modify-write, whereas a byte can be directly written[8]) [Cha92].

The third scheme that we tested is a new one. Like the Chambers-Ungar scheme, we use a byte per card, but instead of just storing a dirty mark, we store the index of the youngest generation referenced from the card. When sweeping a dirty card, we update this value to reflect promotions. This scheme has the advantage that more precise information about the intergenerational references is available, which eliminates many dirty cards from the potential root set.

For SML/NJ, the reduced cost of marking provided by the Chambers-Ungar scheme is outweighed by the increased cost in checking for dirty cards. This is not surprising, since SML programs tend to use very little mutable data and perform few updates, thus the cost of marking cards is dominated by the cost of checking the card maps. On the other-hand, our scheme of using the minimum referenced generation does pay off. For a heap of 5 generations and a card size of 256 bytes, this scheme reduces the number of cards swept by about 80% on a range of realistic SML benchmarks. Although this scheme shares the disadvantage of the Chambers-Ungar scheme, in that each entry in the card map must be examined individually, the cost of this is less than the cost of examining a word in a dirty card, and the overall cost decreases. As the number of generations increases, this scheme pays off even more.

6.3 Big objects

The new collector uses mark-sweep collection for large objects. Currently, this is only used for code objects, but supporting other large atomic objects, such as strings, only requires a trivial modification, and adding support for large arrays and vectors should also be easy. Mark-sweep copying of large objects reduces both garbage collection overhead (since they are not copied), and memory requirements (since space for only one copy is required).

Supporting mark-sweep collection does add some complexity to the heap data structures. Big objects are allocated in *big-object regions*, which consist of some integer number of BIBOP pages. Big-object regions are logically divided into smaller *big-object pages* (currently 4096 bytes each), and each big object occupies some number of contiguous pages. A big-object region has a header that includes a *big-object BIBOP* for the region; the big-object BIBOP entries are pointers to *big-object descriptors*, one per big object. Figure 2 gives the layout of these data structures. Each generation has a list of the big-object descriptors for its live objects. In addition, there is a doubly-linked list of free big objects.[9] When a big-object is encountered during tracing, it is marked (`FORWARD` if it is young, or `PROMOTE` if it is old). After tracing is over, the list of live big objects for each collected generation is scanned. Marked objects are either forwarded (i.e., kept in the list), or promoted to the next generation; unmarked objects are added to the free list.

During root tracing, the collector needs to map pointers in big objects to their descriptors. To facilitate this, we use two kinds of marks for big-object regions in the BIBOP: one for the first

[8]For some modern processors, e.g., the DEC AXP, setting bytes also requires a read-modify-write operation.

[9]A more sophisticated scheme for managing free objects will probably be necessary once other kinds of big objects are supported.

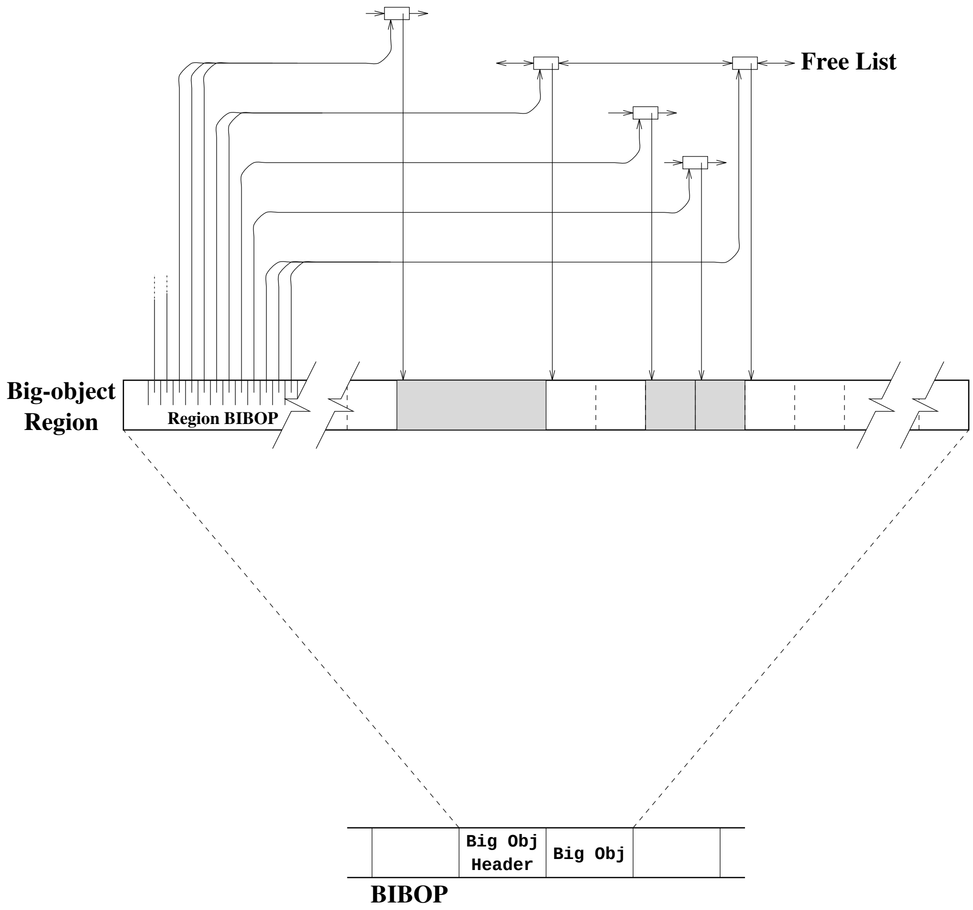

Figure 2: Big-object data structures

BIBOP page of the region (called the header), and another for the other pages. When a pointer into a big object is encountered, the collector maps it to a BIBOP page and then scans backward until the header is encountered. The BIBOP index of the header is then mapped to a pointer to the big-object region's header, and the object address is looked up in the region's big-object BIBOP, which gives the object's descriptor. This is a fairly expensive operation, so to optimize we tag the region's BIBOP pages with the generation of the youngest live big object in the region. Thus, if the youngest big-object in the region is older than the currently collected generation, then the object reference is a to-space reference, and the more complicated lookup is not required. In practice, this works for about 80% of the big-object references encountered by the collector.

6.4 Weak pointers and finalization

Weak pointers are pointers that are treated specially by the garbage collector. If at collection time, a weak pointer refers to an object that is only reachable via the weak pointer, then the collector nullifies the pointer and reclaims the object's storage. SML/NJ provides weak pointers as a language-level mechanism to detect when objects become unreachable. Weak pointers are used by the debugger [TA90] to cache checkpoints, and are used in the SML/NJ Library to implement a *finalization* mechanism. To implement weak pointers, the collector keeps a list of the encountered weak pointers; at the end of the collection it scans the list and nullifies those that point to dead objects. In a generational collector, using weak pointers to implement finalization is inefficient; we will probably support a mechanism like the one described in [DBE93] in the future.

6.5 Heap I/O

The new collector supports two forms of heap I/O: writing and reading entire heap images, and "*structure blasting*" [App90].

The reading and writing of heap images is used to save and restore the state of an ML session. One point of interest is that references to addresses outside the heap (e.g., to runtime system objects) are collected in a relocation table. This allows different versions of the runtime system to be used with the same heap image (for stand-alone programs, the runtime system can be stripped of unecessary features).

Structure blasting is use to read and write ML values in binary representation. As in the old system, the garbage collector is used to linearize an ML object. Because of the more complicated heap architecture, the implementation of structure blasting is more involved than in the old system.

7 Performance

To gauge the effectiveness of the new collector, we have run comparative benchmarks with the old collector on several different workstations using a number of realistic SML programs. Space does not permit a complete presentation of the data, but we include enough to justify the claim that the new collector provides significantly better performance than the old collector. In particular, we present overall performance data on two machines, a SPARCstation-2 (SPARC) and an SGI Indigo (SGI), and latency and memory usage data for the SGI Indigo. The SPARC has a unified 64Kb cache, and the Indigo has a split 8+8Kb primary cache and a 1Mb unified secondary cache. Both machines have 64Mb of RAM.

The benchmark we use is a run of the HOL-90 interactive theorem proving system. This involves a mix of compilation and application-code execution; it is fairly typical of large SML applications.

7.1 Overall performance

We have measured overall performance of the new collector with various different allocation arena sizes on both the SPARC and SGI. Figures 3 and 4 give the overall performance results on the HOL benchmark. The figures present the CPU time (broken into user, system, and collector parts) for the new collector under different allocation space sizes, as well as the total CPU time and collector time for the old collector.[10] Note that the user time includes the cost of minor collections. These measurements are consistent with the real-time numbers for the same runs.

[10] All runs of the old collector were with a 12Mb *softmax*, and a *ratio* of 50 [App89].

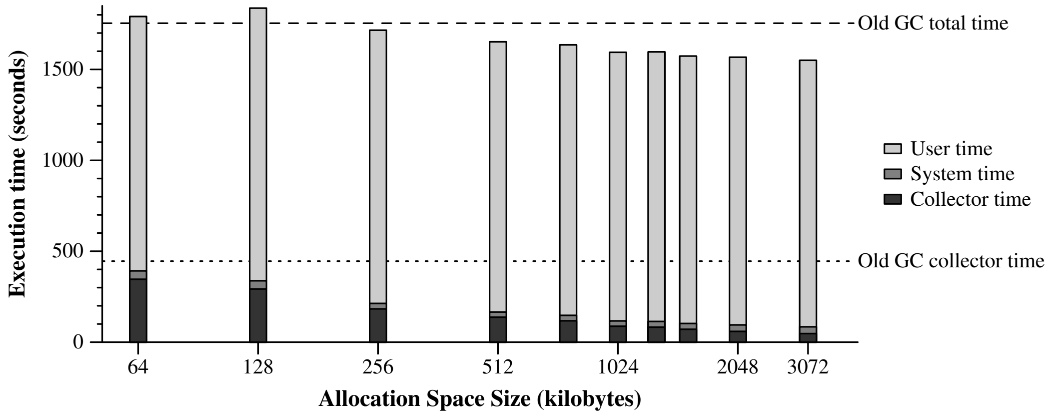

Figure 3: HOL benchmark results (SPARC)

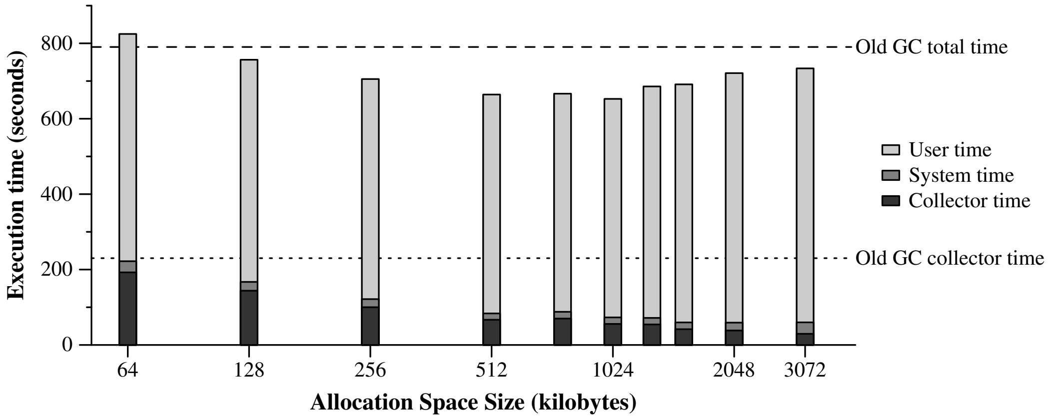

Figure 4: HOL benchmark results (SGI)

There are two aspects of these results that are worth noting. First, the new collector shows significant performance gains over the old collector. The total CPU time is 10% higher using the old collector on the SPARC and 20% higher on the SGI (assuming a 1Mb allocation arena). Second, the SGI data shows the effect of improved cahce locality when the allocation arena is sized to fit in the cache. The total CPU time used by new collector reaches a minimum when the allocation arena is the same size as the secondary cache.[11] This provides empirical evidence for the claim that sizing the allocation space to fit into cache can improve performance [Zor91, WLM90].

For the remaining comparisons, we measured the new collector with a 768Kb allocation arena.

[11] Note that there is a local minimum for the SPARC, but the overhead of frequent garbage collections exceeds the gains from cache locality.

7.2 Garbage collection latency

A significant disadvantage with garbage collection is the long pauses that can occur while garbage collecting. The new collector is designed, in part, to increase the responsiveness of SML/NJ, by reducing the number of longer garbage collection pauses. We have measured the distribution of garbage collection pauses for the new and old collector on the HOL benchmark. Figure 5 presents this data; the x-axis is the length of garbage collection pauses (measured to 10ms, with 20ms buckets), and the y-axis is the percent of total garbage collection time spent in collections of that length. The data demonstrates that the new collector is substantially less intrusive than the old collector. Although the new collector is more intrusive than incremental collectors, such as [HL93] and [NO93], those collectors achieve good latency at the cost of higher overall overhead.

7.3 Memory usage

The last thing that we measured was the relative memory usage of the new and old collectors. The data is given in Figure 6, with time (x-axis) being measured in bytes of allocation. The y-axis is the number of bytes of "*in-use*" data, which is defined to be words in the heap that may potentially contain live data plus the text segment. This data shows that the new collector usually requires significantly less physical memory for a given work load. It is the case, however, that the amount of virtual memory allocated from the operating system by the new collector is significantly larger than for the old collector, even though most of it is not used. This is because of the contiguous arenas and the virtual memory caching mechanism. In Section 9, we describe a scheme for non-contiguous arenas that should address this problem.

8 Related work

There have been a number of replacements for the SML/NJ collector, but most of these require special operating system support, compiler modifications, or both. Cooper, Nettles and Subramanian have described a collector for SML/NJ that exploits MACH's external pager mechanism [CNS92]; it should be possible to integrate this approach into our memory subsystem. Nettles, O'Toole, Pierce and Haines have developed a replication-based collector for SML/NJ that has good collector latency. This implementation, however, requires modification to the compiler and has worse overall performance than the old collector [NOPH92, NO93]. Huelsbergen and Larus have also implemented a concurrent collector for SML/NJ [HL93]. As with the Nettles et al. collector, this provides low garbage collection latency, but it also has worse overall performance than the old collector and required compiler modifications.

Moss et al. have developed a language independent garbage collector toolkit that has been used for SML/NJ [HMDW91, Ste91]. We have not seen any published figures as to the performance of this collector when used for Standard ML, so we cannot compare them in this respect. Our store list is much like their *sequential store buffer*; the main difference is that they use a guard page to detect overflow, which presumably requires operating system support, while we get the overflow check for free from our heap limit check. The other difference is that they use hash tables to remember intergenerational references between older generations, whereas we use card marking.

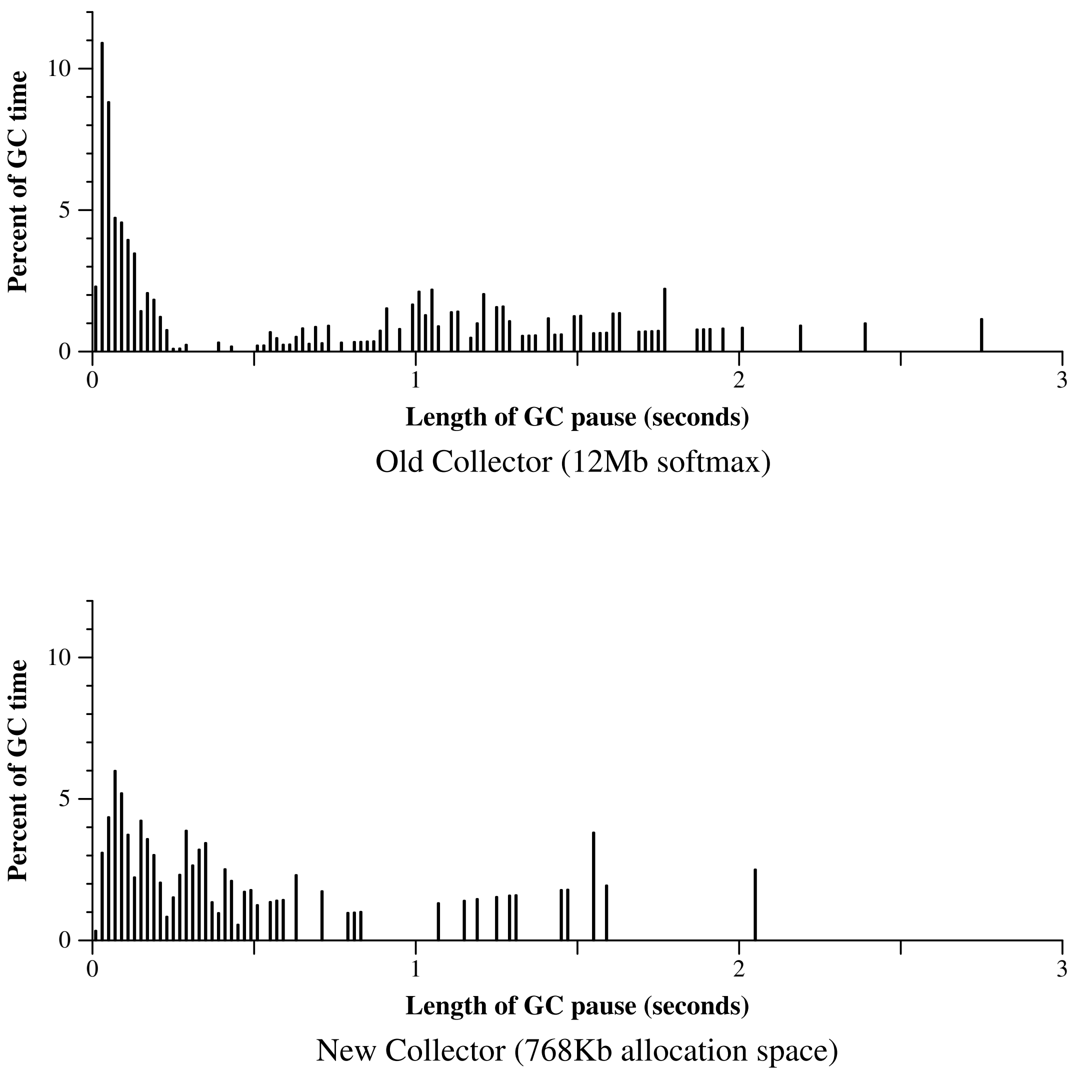

Figure 5: Garbage collection latency data for HOL benchmark

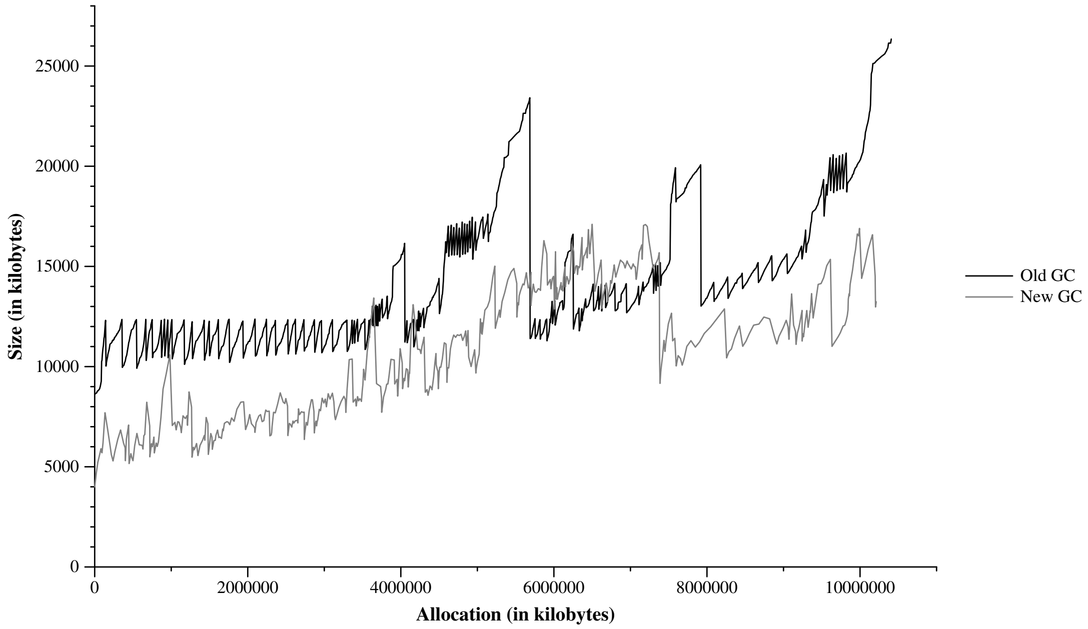

Figure 6: Memory usage for HOL benchmark (SGI)

9 Conclusions and future work

We have described a new collector for SML/NJ that provides better overall performance, reduced GC latency, and requires less physical memory than the old collector. There remain a number of areas for further work:

- We are currently working on a scheme to support non-contiguous arenas (much the way that the GC toolkit does [HMDW91]). This will allow us more flexibility in sizing generations, and determining when to collect. It should also result in lower virtual memory requirements. We hope to report on this in the final version of the paper.
- Older generations tend to yield little free space when collected. For this reason, mark-sweep collection may be more appropriate, since it has smaller memory requirements [Zor89]. We also plan to allow the user to turn off collection for the oldest generation, which will require the non-contiguous arenas.
- The big-object mechanism should be adopted to large arrays; this will require some modifications, since the arrays are pointer-containing, and must be swept.
- For large applications that may have live data the exceeds the size of the computer's physical memory, it is useful to reorganize data to improve locality [Daw82, WLM91]. Marcelo Goncalves', at Princeton University, has a modified version of the collector described in this paper, which uses a depth-first traversal of the live data. It remains to see how much this affects locality for large applications.

- Although the maximum and average pauses resulting from garbage collection are significantly less than in the old collector, the new collector is not unobtrusive. To address this problem, we plan to build a modified version of the collector that uses the Appel-Ellis-Li technique for incremental collection [AEL88].

Acknowledgments

The design of this collector has benefited from discussions with Andrew Appel. Lorenz Huelsbergen, Lal George and Anne Rogers provided useful comments on earlier drafts of this paper. The design and implementation of the first version of this collector was done while the author was a graduate student at Cornell University, and was supported, in part, by the NSF under NSF grant CCR-89-18233.

References

[AEL88] Appel, A. W., J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 11–20.

[AM91] Appel, A. W. and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, vol. 528 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1991, pp. 1–26.

[App89] Appel, A. W. Simple generational garbage collection and fast allocation. *Software – Practice and Experience*, **19**(2), February 1989, pp. 275–279.

[App90] Appel, A. W. A runtime system. *Lisp and Symbolic Computation*, **4**(3), November 1990, pp. 343–380.

[App92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, New York, N.Y., 1992.

[Cha92] Chambers, C. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-oriented Programming Languages*. Ph.D. dissertation, Department of Computer Science, Stanford University, March 1992.

[CNS92] Cooper, E., S. Nettles, and I. Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Conference record of the 1992 ACM Conference on Lisp and Functional Programming*, June 1992, pp. 43–52.

[Daw82] Dawson, J. L. Improved effectiveness from a real time Lisp garbage collector. In *Conference record of the 1982 ACM Conference on Lisp and Functional Programming*, August 1982, pp. 159–167.

[DBE93] Dybvig, R. K., C. Bruggeman, and D. Eby. Guardians in a generation-based garbage collector. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993, pp. 207–216.

[DTM94] Diwan, A., D. Tarditi, and E. Moss. Memory subsystem performance of programs using copying garbage collection. In *To appear in POPL'94*, January 1994.

[GM93] Gordon, M. J. C. and T. F. Melham. *Introduction to HOL*. Cambridge University Press, New York, N.Y., 1993.

[GR93] Gansner, E. R. and J. H. Reppy. *A Multi-threaded Higher-order User Interface Toolkit*, vol. 1 of *Software Trends*, pp. 61–80. John Wiley & Sons, 1993.

[HL93] Huelsbergen, L. and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Fourth SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993, pp. 73–82.

[HMDW91] Hudson, R. L., J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. *Technical Report 91-47*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, September 1991.

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Mass, 1990.

[NO93] Nettles, S. and J. O'Toole. Real-time replication garbage collection. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993, pp. 217–226.

[NOPH92] Nettles, S., J. O'Toole, D. Pierce, and N. Haines. Replication-based incremental copying collection. In *International Workshop on Memory Management*, vol. 637 of *Lecture Notes in Computer Science*, September 1992, pp. 357–364.

[NR87] North, S. C. and J. H. Reppy. Concurrent garbage collection on stock hardware. In *Functional Programming Languages and Computer Architecture*, vol. 274 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1987, pp. 113–133.

[PJ93] Peterson, J. and M. Jones. Implementing type classes. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993, pp. 227–236.

[SML] SML/NJ documentation. Included in SML/NJ distribution.

[Ste91] Stefanovic, D. The garbage collection toolkit as an experimentation tool. Presented at the OOPSLA'93 Garbage Collection Workshop, 1991.

[TA90] Tolmach, A. P. and A. W. Appel. Debugging Standard ML without reverse engineering. In *Conference record of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990, pp. 1–12.

[Ung84] Ungar, D. 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*, April 1984, pp. 157–167.

[Ung86] Ungar, D. *The Design and Evaluation of a High Performance Smalltalk System.* Ph.D. dissertation, Computer Science Division (EECS), University of California, Berkeley, 1986.

[Wil92] Wilson, P. R. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, vol. 637 of *Lecture Notes in Computer Science*, September 1992, pp. 1–42.

[WLM90] Wilson, P. R., M. S. Lam, and T. G. Moher. Caching considerations for generational garbage collection: a case for large and set-associative caches. *Technical Report UIC-EECS-90-5*, Software Systems Laboratory, University of Illinois at Chicago, December 1990.

[WLM91] Wilson, P. R., M. S. Lam, and T. M. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 177–191.

[WM89a] Wilson, P. R. and T. G. Moher. A card-marking scheme for controlling intergenerational references in generation-based GC on stock hardware. *SIGPLAN Notices*, **24**(5), May 1989, pp. 87–92.

[WM89b] Wilson, P. R. and T. M. Moher. Design of the opportunistic garbage collector. In *OOPSLA'89 Proceedings*, October 1989, pp. 23–36.

[Zor89] Zorn, B. G. *Comparative Performance Evaluation of Garbage Collection Algorithms*. Ph.D. dissertation, Computer Science Division (EECS), University of California, Berkeley, December 1989. Available as Technical Report UCB/CSD 89/544.

[Zor91] Zorn, B. The effect of garbage collection on cache performance. *Technical Report CU-CS-528-91*, Department of Computer Science, University of Colorado, Boulder, May 1991.

A High-performance Garbage Collector for Standard ML

John H. Reppy

AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

ABSTRACT

We have designed and implemented a new garbage collector for the Standard ML of New Jersey System (SML/NJ). This collector has higher performance, lower latency and generally requires less physical memory than the existing SML/NJ collector. In addition, it is able to exploit the large secondary caches found on modern workstations. This paper describes the design of the collector, and presents comparative performance data that demonstrates the above performance claims.