A Compacting Incremental Collector and its Performance in a Production Quality Compiler

Martin Larose and Marc Feeley

{larosem,feeley}@iro.umontreal.ca
Département d'informatique et recherche opérationnelle
Université de Montréal

Abstract

We present a new near-real-time compacting collector and its implementation in a production quality Scheme compiler (Gambit-C). Our goal is to use this system as a base for an implementation of Erlang for writing soft real-time telecommunication applications. We start with a description of Gambit-C's memory organisation and its blocking collector. The design and integration of the incremental collector within Gambit-C are then explained. Finally we measure the performance of the incremental collector and compare it to the original blocking collector. We found that the overhead of the incremental collector is high (a factor of 1.3 to 8.1, with a median of 2.24) but nevertheless the collection pauses are compatible with typical soft real-time requirements (we get an average pause of 3.25 milliseconds and a maximum pause of 18 milliseconds on a 133Mhz DEC Alpha 21064).

1 Introduction

Garbage collection (GC) frees the programmer from the tedious and error-prone task of memory management, thus making the programming language higher-level. On uniprocessors the work of the collector is interleaved with that of the main program (the *mutator*). If each parcel of GC is too long, the collector may adversely interfere with the execution of the mutator which becomes unresponsive while the collector is working. This problem is especially important in real-time applications (e.g. animations, real-time simulations and reactive systems) where the mutator is expected to progress at a steady rate.

Incremental collectors aim to diminish the disruptiveness of GC by spreading out the GC work into more uniformly distributed parcels of smaller and bounded size. Because incremental GC requires extra coordination between mutator and collector and higher conservatism, it is more expensive than blocking GC (where all of the dead objects are reclaimed every time the collector is run). There is a wide space of tradeoffs between GC overhead and predictability in the design of an incremental collector but unfortunately it is hard to pick the best tradeoffs for a given application because there have been few experimental studies on which to base a decision. We have designed a near-real-time incremental compacting collector and implemented it in a production quality Scheme compiler. This paper reports on the various tradeoffs we made and the performance of the collector on a wide range of benchmarks.

This is a revised version of the paper published in: Proceedings of the 1998 International Symposium on Memory Management. The results have been updated after we fixed a bug in the measurement software which slightly undervalued the pause time.

2 Context

The work reported in this paper is part of a larger effort to implement a compiler for Erlang [AVWW96], a concurrent mostly functional programming language for real-time telecommunication applications developed at Ericsson. Our compiler, called Etos [FL98], first compiles Erlang to Scheme and then uses the Gambit-C Scheme compiler [FM90, FMRW97] to compile the result into C. This approach is reasonable and efficient because Scheme and Erlang share many similarities (e.g. use of functional style, dynamic typing, data types).

The telecommunication applications targeted here are not hard real-time applications; it is permissible for an application to be unresponsive for short periods of time (say 10-50 milliseconds) as long as this is infrequent. An application will not fail if it is unresponsive for longer than this, its quality of service will simply degrade (for example connecting a telephone call to a destination must seem to be close to instantaneous to the human caller but if it takes a few seconds all it will cause is a small amount of frustration). The average pause should be in the range 2-5 milliseconds and at least 50% of the run time should be spent executing the mutator (assuming the code is reasonably efficient, i.e. generated by an optimizing compiler). These constraints are sufficient to write in Erlang the control software of an ATM switch (such as the AXD301 [AXD98] which aims to process each transaction within 7 milliseconds).

To improve responsiveness we designed an incremental compacting collector for Gambit-C to replace the blocking collector in the standard distribution. Given the application domain and use of a functional programming style, we anticipated high allocation rates. Our incremental collector is a refinement of Dubé's collector [Dub96, DFS96], an incremental collector developed for a small footprint interpreter-based Scheme implementation for embedded 8 bit controllers.

3 Gambit-C's Blocking Collector

Gambit-C was designed to be a very portable compiler (the code generated sticks to ANSI-C and uses few OS specific features) and to allow Scheme and C code to be mixed in one application (Scheme objects can be accessed and allocated from C code and control can jump from C to Scheme arbitrarily). The compiler uses an RTL-style virtual machine code (GVM [FM90]) as an intermediate representation and then translates each virtual instruction into the corresponding C code (the concept of virtual machine registers is important here because they are roots of the collector). For portability, the Scheme heap (which contains all the reclaimable Scheme objects) is allocated from the C heap by using the `malloc` C library routine.

3.1 The Stack-Cache

In order to properly handle tail-calls (see [FMRW97] for details) and to provide efficient first-class continuations, Gambit-C allocates continuation frames in a 36 Kbyte stack-cache which is separate from the C stack. When a deep recursion causes an overflow of the stack-cache, the continuation frames it contains are transfered to the Scheme heap thus allowing the recursion to continue. A continuation frame is copied from the heap back to the stack-cache when the stack-cache is emptied after a function return. When a continuation is captured with a call to **`call-with-current-continuation`** the base of the stack-cache is temporarily moved up so that any return to one of the captured continuation frames will cause it to be copied to the top of the stack-cache. This technique is basically the same as [HDB90] but of a finer granularity.

3.2 Memory Partitioning

There are three allocation classes for Scheme objects:

1. **Movable**: the object may be moved by the collector
2. **Still**: the object is never moved
3. **Permanent**: the object is never moved or reclaimed

The allure of still objects is that C code can easily manipulate them without worrying about their address suddenly becoming invalid after the collector is run (a conservative GC approach such as [BW88] is not an acceptable solution because it is not portable). Still objects have a reference count field which indicates how many references **from the "C world"** exist to this object (to prevent the collector from reclaiming them if they are not reachable from the "Scheme world"). Permanent objects are useful for program constants (which are nonmutable) and symbols (including dynamically created ones). Most objects dynamically allocated by Scheme code are movable objects. Movable objects are allocated efficiently by incrementing a free pointer. Within a basic block, allocations of movable objects are coalesced and a single heap limit check is performed. Most constant-size allocation primitives (`cons`, `list` and `vector` but not `make-vector`) are inlined by the compiler. The compiler also keeps floating point numbers in an unboxed state within each basic block, which greatly reduces the need for allocating flonums (boxed floating point numbers) on some floating-point intensive programs [HFA$^+$96].

Gambit-C allows the Scheme heap to grow and shrink dynamically as the program's needs change. For this reason, we opted not to implement the Scheme heap as one large block because this would cause severe fragmentation given that C code also allocates objects in the C heap. Instead, a collection of fixed size (512 Kbytes) noncontiguous sections is used to hold movable objects. Each movable object section is divided into equal size from-space and to-space. Still objects are allocated directly off of the C heap using `malloc`. Large objects (> 16 Kbytes) are always allocated as still objects to avoid fragmenting the movable object sections. Permanent objects are allocated statically if they are program constants or on the C heap if they are symbols.

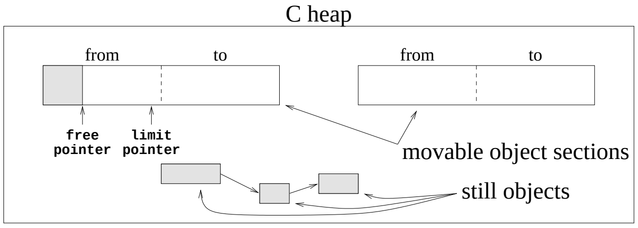

Figure 1: Allocation of movable and still objects in the C heap.

Figure 1 shows how the C heap is partitionned. When an allocation of a movable object pushes the free pointer past the limit pointer, the free pointer is advanced to the next movable object section if one is free otherwise the collector is run. Note that there is a "fudge" space (16 Kbytes) between the limit pointer and the end of the from-space. This is to accomodate the runtime library which sometimes needs to blindly allocate a bounded number of movable objects without checking the limit pointer until all objects are allocated. The stack-cache also has a fudge space (20 Kbytes).

3.3 Object Representation

Figure 2 shows how movable objects are represented. Objects are aligned on a 4 byte boundary except for flonums which are aligned on an 8 byte boundary. The two lower bits of a pointer are used to encode a primary type information: `00` for fixnums (small exact integers), `10` for other immediates (characters, booleans, empty-list, etc.), `11` for pairs, `01` for other memory allocated objects. The body of all memory allocated objects is prefixed with a single word header which contains the following fields: 24 bits for length of body in bytes, 5 bits for secondary type information (pair, vector, string, etc.), 3 bits for an allocation class tag (permanent, still, movable-but-not-forwarded, and movable-and-forwarded which counts for 2 tags because in this case the header contains the forwarding pointer). Still objects prefix the header with extra fields to accomodate the collector (a reference count and 2 links as explained below).

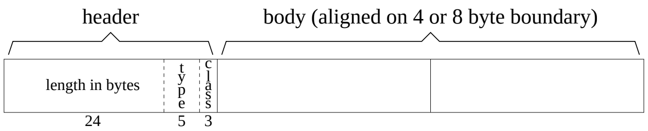

Figure 2: Movable object representation.

3.4 The Blocking Collector

Gambit-C's blocking collector combines the stop-and-copy technique (for movable objects) and the mark-and-sweep technique (for still objects). Permanent objects are not scanned by the collector because they do not need to be reclaimed and can only contain references to permanent objects.

Still objects are placed on a linked list when allocated. This list is used at the start of the collection to mark all the still objects which have a nonzero reference count, and at the end of the collection to reclaim all unmarked still objects (by a call to the `free` C library routine). A list of all marked but not yet scanned still objects is also maintained by the collector (this explains why still objects have 2 link fields).

The movable objects are handled by a Cheney-style copying algorithm [Che70] which overwrites the header with the forwarding pointer. Control alternates between the stop-and-copy and mark-and-sweep algorithms until the list of all marked but not yet scanned still objects is empty and there are no remaining movable objects that have been copied to to-space but have not yet been scanned.

The roots used by the collector are:

- the nonzero reference count still objects,
- the Scheme global variables,
- the virtual machine registers,
- the top part of the stack-cache (i.e. the continuation frames in the current continuation that have not been captured by a call to `call-with-current-continuation`)

At the end of the collection, the Scheme heap is resized by allocating or reclaiming some movable object sections. The default policy currently used is to make the heap twice the size of the space occupied by live objects (for movable objects the space occupied is multiplied by two because there is space needed for the actual object and its copy). The user can configure the resizing ratio, as well as the minimum and maximum heap size, when the program is launched.

4 Integrating the Collector Into Gambit-C

In replacing Gambit-C's blocking collector with Dubé's collector [DFS96] we had two goals: adapt the collector to a production quality compiler and measure the performance of the collector in a realistic setting. This section describes how Dubé's collector was modified.

Dubé's collector is a mark-and-compact collector which compacts by sliding objects (the ordering of objects in memory is preserved). Dubé's central idea is the use of a nonmovable handle which points to a movable part. Thus a reference to a Scheme object is encoded as a tagged pointer to a one word handle which contains a pointer to the body of the movable object. Because of this indirection it is possible to avoid an "update" pass to update all object references to the new location of the objects. This operation is a problem in a real-time setting because the number of references to update for a given object is not bounded and therefore can not be done atomically. However, an overhead is added to the mutator for every access to the object. The overhead we measured is reported in Section 5.

Because Gambit-C handles interrupts through polling [Fee93] and that polling points and heap limit checks can only occur at the end of basic-blocs, it is possible to maintain direct pointers to the movable part of objects temporarily (for the duration of a basic-bloc). This allows the indirection cost to be amortized over multiple accesses to the same object, even in a multi-threaded context. However, the Gambit-C compiler does not currently exploit this possibility.

4.1 Memory Partitioning

Dubé's collector assumes a fixed size heap and that all objects are movable. This simplifies the memory partitioning because each memory section can be preallocated. Gambit-C however allows the Scheme heap to grow and shrink on demand so a different approach is needed.

The memory partitioning is only slightly different from the blocking collector. The three allocation classes are maintained and the movable object sections are the same size. Because compaction is done by sliding objects, the complete size of each movable object section is used for allocation, not just half.

The allocation limit pointer is handled differently. Instead of pointing close to the end of the current movable object section, it initially points a constant amount (G words) further than the allocation pointer. The mutator passes control to the collector when the allocation pointer crosses this limit. When the collector is done, it sets the limit pointer to G plus the allocation pointer, unless there isn't enough space in the current movable object section in which case the next section is used. The value of G can be adjusted to control the granularity (and thus overhead) of the context switches between mutator and collector and also the collector pause time (which is roughly proportional to G as explained below). A setting of $G = 4096$ words offers a good compromise between pause time and overhead, and is used in our experiments.

Handles are nonmovable and are thus allocated outside the movable object sections in handle sections. Each handle section contains the (worst-case) number of handles needed for the objects in one movable object section, i.e. 1/3 the size of a movable object section. When a handle section is allocated off the C heap the handles in that section are linked together and added to the free handle list. This list shrinks and grows with the allocation and deallocation of the movable objects. Handle sections are never freed since they are not tied directly to a specific movable object section but individually to movable objects. So there will be N handle sections if the maximum number of movable sections in the past execution is N. The Scheme heap size accounts for the handle sections.

4.2 The Marked Object List

Dubé's collector uses a main heap (which is one contiguous section) for two purposes. Objects are allocated at one end and a marking stack is maintained at the other end. This stack holds pointers to all the objects that have been marked but not yet scanned. Marking an object adds it to the stack and scanning an object removes it from the top of the stack. The space for one pointer is reserved on the stack on every allocation (by incrementing the marking stack limit which separates the area reserved for objects from the area reserved for the marking stack).

We have implemented the marking stack by linking all objects that have been marked but not yet scanned into the "marking list". This required adding a field (the mark field) to movable objects which is also used to encode the color of the object. When the collector has not yet determined that an object is reachable its mark field is set to 0 (white). After being marked, the mark field contains the address of the next object in the marking list or a special end of list marker (gray). Finally, when the object is scanned, it is detached from the marking list and its field is set to -1 (black). Note that still objects already have a mark field, so an extra field is not needed for them. The mark field is also used for handling object mutation (details below).

4.3 New Object Representation

In order to access objects in the same way regardless of their allocation class, all objects are represented uniformly with a handle. For permanent objects a space for the handle is reserved before the header as in Figure 3. There is no need for a mark field.

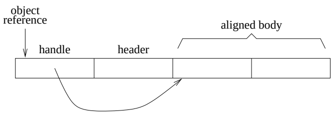

Figure 3: Permanent object representation

For still objects several fields come before the header as shown in Figure 4: the mark field which links still objects, the handle, a reference count, a link to the next still object, and a length (which is only needed for memory accounting purposes and because Gambit-C supports operations to shrink the size of an object which is useful for implementing bignums and string ports).

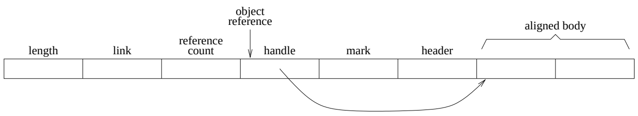

Figure 4: Still object representation

The allocation of a movable object requires an allocation of a nonmovable handle from the free handle list and an allocation of the movable part in the current movable object section. Note that there is always enough handles for all the movable sections, so it is not necessary to check exhaustion of the free handle list. As shown in Figure 5, the movable part has two more fields than for the blocking collector:

- **Back pointer**: points back to the corresponding handle. Needed in the compacting phase of the collector to update or free the handle.
- **Mark**: this links gray objects, as explained above.

The representation of movable objects may seem space inefficient but it compares advantageously to the blocking collector which has a hidden factor of two for the space reserved in to-space. For a n word body, the representation for the blocking collector is more space efficient for $n < 2$ (which is rare) and less space efficient for $n > 2$. For the frequent case of pairs ($n = 2$), the representations are equally space efficient.

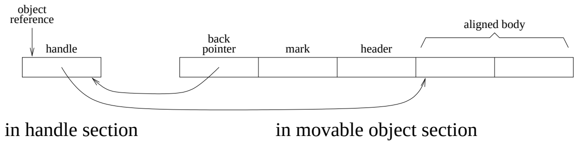

Figure 5: Movable object representation

Note that this object representation allows testing the color of any class of object by reading the field just before the header. In the case of a permanent object, the color will appear gray because the handle is neither 0 or -1.

4.4 The Collector

The collector is called on two types of events, when the allocation limit is reached and when the stack-cache overflows.

The collector can be in one of 4 states corresponding to each phase of the collection (mark roots, marking, pre-compaction, compaction). A collection cycle begins when the collector enters the mark roots phase. The time allotted to the collector for the next parcel of collection is kept in a global variable of the collector called the *word bank* (details below). When this time is up, control returns to the mutator and the next time the collector is called it will resume in the same phase.

1. **Mark roots** phase. This phase is performed atomically (even though it doesn't need to be). It initializes some global variables of the collector and marks the roots. The roots are the same as the blocking collector, except for the stack-cache. We observed that even for big applications the time needed for marking the roots is small enough not to exceed our real-time constraints. This is due to a limited number of global variables (the Scheme runtime library which is present in all applications contains 1500 global variables and the Gambit-C compiler, our largest Scheme benchmark at 20000 lines, adds another 1500 variables to that).
2. **Marking** phase. In this phase, the still object and movable object marking lists are scanned.
3. **Pre-compaction** phase. This phase is performed atomically. Each time it is entered the roots and the stack-cache are scanned again because the mutator might have stored references to white objects into them while the collector was in the marking phase. The use of a fixed size stack-cache bounds the amount of work to be done (on our test machine this phase takes up to 3 milliseconds for the `compiler` benchmark, and roughly 1 millisecond for the other benchmarks). If this marks new objects the collector goes back to the marking phase, otherwise the collector will:
   (a) free the unmarked still objects,
   (b) save a copy of the movable object allocation pointer such that all movable objects allocated between now and the end of the compaction phase will be considered black regardless of their mark field (movable objects are always allocated with 0 (white) in the mark field)

4. **Compaction** phase. The last phase compacts the heap. A copying pointer and a scanning pointer are set to the base of the first movable object section. Each object in the movable object sections is processed in turn using the scanning pointer. Unmarked objects are collected by transfering the corresponding handle to the free handle list. Marked objects are copied to the address indicated by the copying pointer and the corresponding handle is updated.

   When the compaction ends, the allocation pointer is set to the value of the copying pointer and the heap is resized (all the movable objects retained are considered live).

If the collector was called due to a stack-cache overflow, a stack collection routine is first called. Every frame in the stack-cache is copied to the Scheme heap, the word bank is updated according to the size of the frames, the stack-cache is emptied and the collector is called to continue normal processing as explained above.

4.5 Write Barrier

When a reference to object X is stored in object Y, the system must ensure that the collector will not neglect to mark X if Y ends up marked when the compaction phase is started (unless of course the reference to X in Y is overwritten). This will not happen automatically if X is white and Y is black. We have experimented with two write barriers to handle this case.

1. **Gray** X. Here the white object X is grayed by putting it in the marking list. This is the original barrier proposed by Dubé and is similar to Dijkstra's barrier [DLM+78].

2. **Gray** Y. Here the black object Y is grayed by putting it back in the marking list. This is similar to Steele's barrier [Ste75]. This is less conservative than graying X (i.e. X will possibly be reclaimed if the reference to X in Y is overwritten). We rejected a more precise barrier method that only grays the location of the mutation using a store list because we want to keep a strict bound on heap size. This is a reasonable compromise given that there are no mutation primitives in Erlang and Scheme programs are often mostly functional.

The write barrier is only used on heap allocated objects by the primitives: `vector-set!`, `set-car!`, `set-cdr!` and `cell-set!` (which is used for assignments to local variables). There is no barrier on the roots (the virtual registers, the stack-cache and the global variables) which are scanned in the pre-compaction phase of the collector. This eliminates the need for protecting Scheme's `set!` operation on global variables.

The pseudocode for the `vector-set!` primitive, including a "gray X" write barrier, is shown in Figure 6 (the other mutation primitives are similar). The procedure `gray(val)` adds object `val` to the head of the marking list.

Long objects are scanned incrementally to bound collector pauses. In the marking phase, the collector scans long objects in small segments and a pointer to the unscanned region is saved when control returns to the mutator. Consequently, when the "gray Y" barrier is used, mutation of a still vector object must check if the mutation is in the scanned region, in which case the collector must rescan it from the beginning in the next parcel of collection. This is not a perfect solution in general because the collector could get stalled on marking vector V if the mutator repeatedly mutates the beginning of V (this could lead to the heap overflowing). Fortunately, in the context of an Erlang system this is not a problem because we can write the runtime system in such a way that mutations are always performed on small vectors.

```
vector_set(vect, index, val):
  if memory_allocated(val) and gc_phase!=compaction
     and black(vect) and white(val) then gray(val)
  vect[index] = val
```

Figure 6: Pseudocode for the `vector-set!` primitive and write barrier.

4.6 Parceling Out Collection Work

The following analysis applies to the "gray X" write barrier and to the "gray Y" write barrier with no mutation to long objects. We will make use of the following definitions:

- H is the size of the heap (in words).
- R_i is the proportion of the heap occupied by objects retained by the collector at the end of collection cycle number i.
- W_{total} is the total amount of work for one collection cycle in number of words to mark and to compact.
- W is the amount of work in a parcel of collection.
- B is the value of the word bank.

The marking phase will touch at most HR_i words worth of objects and the compacting phase H words, so $W_{total} \leq H(1+R_i)$. This work is spread over the allocation of $H(1-R_{i-1})$ words by the mutator. So, if the collector touches C words per word allocated by the mutator, then the collection cycle will end before the mutator exhausts the free space as long as $C \geq \frac{W_{total}}{H(1-R_{i-1})} \leq \frac{1+R_i}{1-R_{i-1}}$.

We use the setting $C = \frac{5+3L}{2(1-L)}$, where L is chosen at program launch and is an upper bound on the proportion of the heap occupied by live objects. Figure 7 gives a plot of this function.

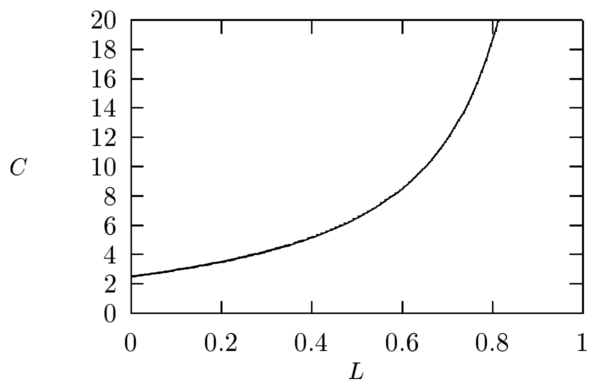

Figure 7: Value of C as a function of L.

This setting of C ensures that the collection cycle will end before the mutator exhausts the free space when $R_{i-1} \leq \frac{1+L}{2}$. Moreover, it guarantees that $R_i \leq \frac{1+L}{2}$. This is easy to prove by induction (see [DFS96] for a proof). An interesting corollary is that the collector can stay idle at the start of the collection cycle until the mutator has allocated enough objects to make the heap occupied to $\frac{1+L}{2}$. By staying idle in this way, the collector will be less conservative and thus more efficient at reclaiming garbage.

The word bank is used in parceling the collection work. At the start of collection cycle i, B is set to the negative value $-H(\frac{1+L}{2} - R_{i-1})$ so that the collector will stay idle at the start of the collection cycle. When the heap limit is crossed and when the stack-cache overflows, the number of words allocated (still and movable objects) is added to B. Thus, in the typical case (heap limit reached) B increases in steps of G.

If B is negative, the collector returns immediately to the mutator. Otherwise, the amount of collection work is calculated based on B and C (i.e. $W = BC$), the collector performs W words worth of collection, sets B to 0 and then returns to the mutator.

5 Results

To measure the performance of our incremental collector we used a set of 20 Scheme benchmarks. In all cases the programs were compiled with the Gambit-C 2.7 compiler using the declarations which gave the fastest execution (inlining of primitives, fixnum or flonum specific arithmetic, no runtime type checks). The short running programs were modified to repeat the computation several times so that the total execution time would be at least 5 seconds.

A first group of programs comes from the Gabriel benchmark suite [Gab85]. These programs are mostly kernels which stress specific features of the system (fixnum arithmetic, allocation, traversal, mutation, recursion, iteration). Some of these benchmarks don't perform any allocation so we ignored them (`tak`, `takl`, `triangle`, and the traversal phase of `traverse`). The second group consists of floating-point intensive programs: `fibfp`, `sumfp`, `mbrot`, `fft`, and `simplex`. The third group contains larger applications which mix various types of symbolic processing, including lots of allocations, object mutations and traversal of data-structures: `conform` (type checker, 700 lines), `peval` (partial evaluator, 800 lines), `earley` (parser, 800 lines), `maze` (construct a maze, 900 lines), and `compiler` (Gambit-C Scheme compiler, 20000 lines).

All benchmarks were run on an unloaded 160Mbyte 133Mhz DEC Alpha 21064 running Digital UNIX V4.0. CPU time statistics were measured with the C library routine `getrusage` which has a 1 millisecond resolution. We considered using the `gettimeofday` routine to measure time down to the microsecond but since it measures real-time some of the short duration statistics measured (such as the maximum collector pause) are too easily perturbed by OS context switches over which we have no control. Each benchmark was run once. All times are given in seconds.

To reduce the importance of differences in the memory partitioning of the different collectors, all programs were run with a fixed-size heap of 12Mbytes. This also avoided unexpectedly long collector pauses when resizing the heap (it seems that calls to `malloc/free` for large blocks sometimes takes over 10 milliseconds!).

5.1 Overhead of Incremental Collection

Our first goal is to measure the total overhead of using an incremental collector rather than a blocking collector in a production quality compiler such as Gambit-C. Also, we wish to find the overhead associated with performing the collection incrementally. For this purpose the programs were run with three different collectors:

1. **S&C**: this is the blocking collector in the standard Gambit-C distribution.
2. **M&C**: this is our collector when run as a blocking collector (i.e. the collection is done completely when the heap is full and there is no write barrier). A stack-cache overflow causes a full collection (which is what is done by S&C).
3. **M&C R-T**: this is the full incremental collector described in this paper, using a value of $L = 50\%$.

The results are reported in Figure 8. The first column gives the allocation rate of the program in Mbytes per second when run with S&C. The second column gives the execution time in seconds for S&C. The execution time for the other collectors is expressed relative to the time for S&C so that the overhead with respect to S&C stands out more clearly. The M&C R-T collector was run with each type of write barrier. Note that the results are ordered according to the overhead of M&C R-T with the "gray X" barrier.

	Alloc MB/sec	S&C	M&C	M&C R-T gray X	M&C R-T gray Y
`boyer`	3.14	16.46	.91	.92	.96
`puzzle`	.92	21.88	1.20	1.30	1.41
`compiler`	.91	49.99	1.19	1.31	2.52
`fft`	12.58	5.05	1.53	1.56	1.58
`traverse`	5.61	10.93	1.37	1.67	2.01
`browse`	3.95	33.36	1.25	1.73	2.46
`peval`	5.86	35.40	1.52	1.81	3.16
`conform`	2.88	25.52	1.29	1.85	2.60
`simplex`	15.34	10.91	1.70	2.10	2.32
`earley`	8.42	36.13	1.87	2.24	3.02
`cpstak`	46.94	13.95	2.41	2.44	2.46
`maze`	16.18	11.58	1.65	2.46	2.53
`destruc`	19.35	15.29	2.11	2.78	3.70
`deriv`	27.43	32.69	2.50	2.92	4.08
`fibfp`	47.05	11.80	2.43	2.94	2.96
`dderiv`	22.76	39.39	2.04	2.98	4.17
`mbrot`	60.20	13.03	2.87	3.62	3.64
`divrec`	54.92	16.66	3.15	3.83	3.77
`sumfp`	71.12	85.82	3.21	4.24	4.26
`diviter`	123.03	7.44	6.51	8.09	8.04

Figure 8: Execution time with each collector (S&C in seconds and others relative to S&C).

For the M&C collector the overhead includes: allocation of handles, indirection cost when accessing a memory allocated object and difference in collection algorithms. If we ignore `boyer`, the overheads range from 1.19 to 6.51, with a median of 1.7. We can see that the overhead is roughly correlated to the allocation rate. This is reasonable because object allocation is significantly more expensive than the simple pointer increment performed for S&C and all objects allocated including dead ones need to be processed in the compaction phase. The highest overhead is for `diviter` which spends most of its time in a tight loop performing 3 accesses to pairs and 1 allocation of a pair which is soon dead. An anomaly exists for `boyer` which is slowest of all when using S&C because the mutator and collector are in synch (the profile of live objects is like a sawtooth, going from 50Kbytes to 1400Kbytes, and with a 12Mbyte heap S&C always collects at moments of peak live objects whereas the other collectors do it at uniformly distributed levels, which is more efficient).

When the allocation rate is low the overhead depends more on the handle indirection cost and the difference in collection algorithms. It is interesting to see that a complex application like `compiler` has a low overhead of 1.19. We attribute this to the fact that its modular design causes a lot of time to be spent in procedure calls and returns between modules (which is unaffected by the collector but is rather slow in Gambit-C due to the tail-call support), that it was designed to minimize the creation of garbage and that it performs I/O.

The M&C R-T collector with "gray X" barrier has overheads in the range 1.3 to 8.09, with a median of 2.24. The overheads follow the same trend as the blocking M&C collector, and are a median factor of 1.21 higher with a maximum of 1.5 times higher for `maze`. So the transition from M&C to M&C R-T has a lower overhead than the transition from S&C to M&C (in other words most of the overhead of our incremental collector is not in the "incrementality" but rather in the use of a compacting collector with handles).

If we only consider the benchmarks which perform mutations on objects, the "gray X" barrier is always faster than the "gray Y" barrier, by a median factor of 1.35 and a maximum of 1.9 times faster for `compiler`. Our explanation is that the "gray Y" barrier may cause objects to be marked multiple times and this extra cost outweighs the benefit of lower conservatism.

5.2 Collection Pauses

Another important aspect to measure is the duration of the pauses of the incremental collector. The average pause is of course interesting but given the context of soft real-time applications, it is also important to know what is the maximum pause and also the percentage of total execution time spent in the collector (%GC). Figure 9 gives these measurements ordered according to %GC when using the "gray X" barrier.

The average pause is in the range 2.42 to 4.37 milliseconds, with a median of 3.25 milliseconds. The maximum pause is in the range 6 to 18 milliseconds, with a median of 8 milliseconds. The %GC is in the range 4% to 57%, with a median of 22%. These measurements are compatible with our real-time constraints.

The programs with the highest %GC are those which have high allocation rates and few live objects (the top 3 are floating point intensive programs which strictly allocate flonums). In this situation the collector will spend a large fraction of its time compacting black objects that are in fact garbage. So for most objects allocated there are two associated compactions needed (because during the compaction phase objects are allocated black).

Finally, Figure 10 shows the distribution of pause times. The X axis gives pause time in seconds and the extent of the X axis indicates the maximum collector pause. As can be seen, the distribution is compact around the average and there are no distant outliers.

	Avg	Max	%GC
`compiler`	.00437	.012	4
`puzzle`	.00242	.010	4
`conform`	.00325	.010	5
`peval`	.00299	.008	8
`browse`	.00319	.010	10
`boyer`	.00364	.009	12
`trav1`	.00397	.014	15
`simplex`	.00336	.008	20
`destruc`	.00319	.007	21
`earley`	.00394	.012	22
`dderiv`	.00321	.007	24
`fft`	.00370	.008	24
`deriv`	.00318	.007	29
`maze`	.00371	.018	35
`cpstak`	.00266	.006	40
`divrec`	.00323	.007	45
`diviter`	.00323	.007	48
`fibfp`	.00370	.008	52
`mbrot`	.00375	.007	55
`sumfp`	.00385	.007	57

Figure 9: Average and maximum collector pause in seconds and percent of time spent in collector.

5.3 Discussion

At first glance the overhead of the incremental collector seems too high for practical use. However this overhead must be put in perspective. Erlang programs compiled with Etos and Gambit-C 2.7 with the S&C collector are roughly 15 times faster than with the JAM 4.4.1 bytecode implementation of Erlang [FL98]. Even if a program using the incremental collector is slowed down by a factor of 2.24 compared to the S&C collector, the program is still over 6 times faster than when using JAM.

Of course the overhead and pause time that is tolerable depends on the application. However, it is reassuring that the measurements we have made on our collector fit very closely with the requirements of soft real-time telecommunication applications (2-5 millisecond average pause and 10-50 millisecond maximum pause). Note also that our test machine (133Mhz DEC Alpha 21064) is more than 5 years old at this writing and that much faster microprocessors are readily available. When executed on a now current 500Mhz DEC Alpha 21164A the `compiler` benchmark ran 6.7 times faster and the collector was about 3.5 times faster (collection pauses were 1 millisecond on average with a maximum pause of 3 milliseconds).

6 Future Work and Conclusion

Because of the way the word bank is handled, the collector only starts collecting after the mutator has allocated a fair amount (i.e. until the word bank becomes positive). It would be interesting to investigate if by starting the collector earlier we could reduce the collection overhead and length of pauses (it isn't clear that this is good because the collector will be more conservative, retaining some objects that

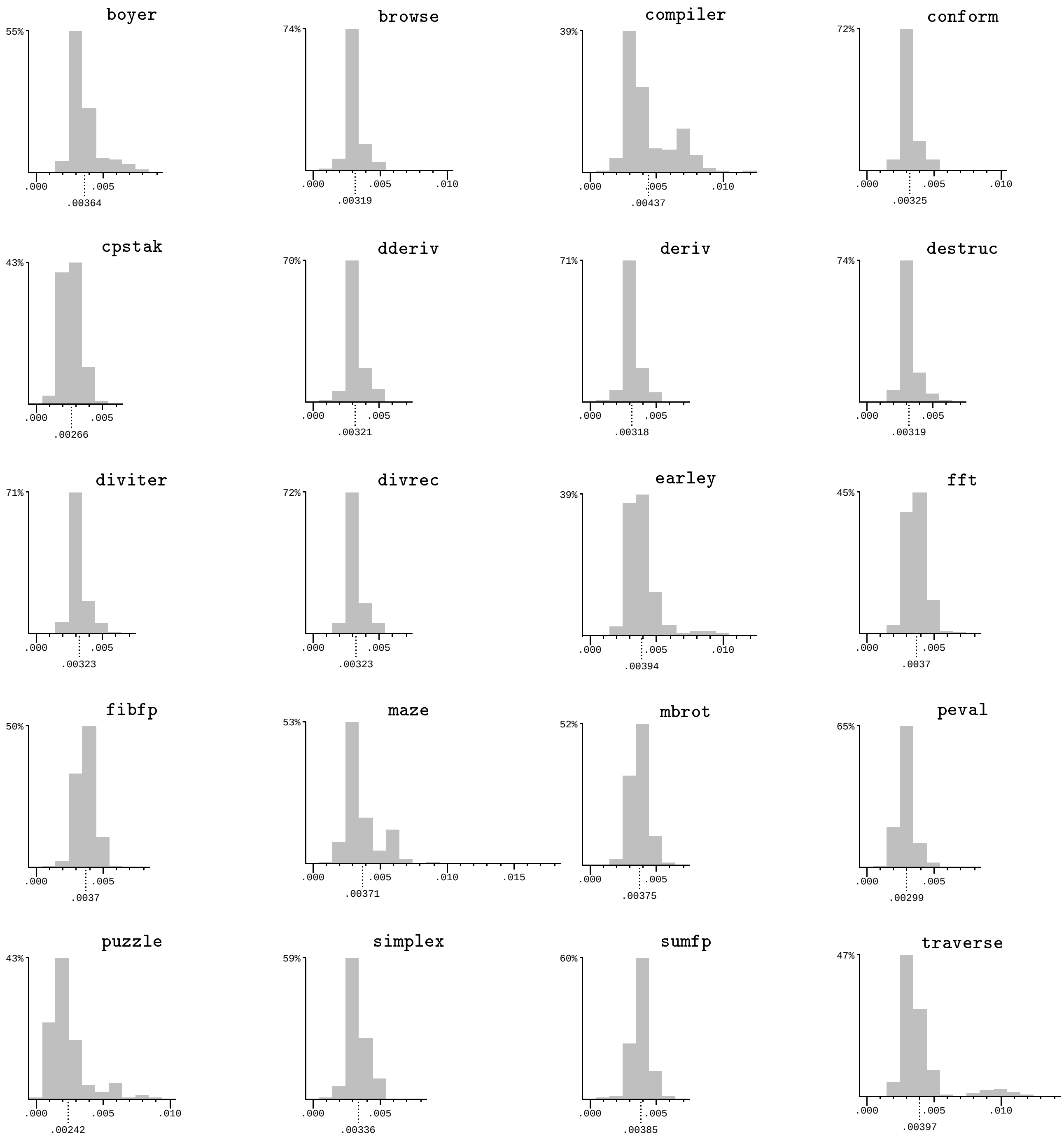

Figure 10: Distribution of collector pauses.

would have died). A dynamic calculation of L also seems necessary, since it would allow the collector to adapt to the behavior of the application.

There is also a need for testing the collector with soft real-time Erlang applications. This will have to wait until Etos is complete and robust.

Once Dubé's collector is fully integrated into Gambit-C, we plan to integrate other near real-time collectors (in particular [NO93] which seems well suited to our context) and compare their performance.

As our experimental results show, the incremental collector is able to meet the maximum and average pause time constraints needed by telecommunication applications. The overhead of the incremental collector with respect to a blocking collector is rather high (a factor of 1.3 to 8.1) but, given that we are working in the context of an optimizing compiler, the compute power left for the mutator compares favorably with a bytecode implementation of Erlang.

Acknowledgements

This work was supported in part by grants from Ericsson Telecom Ab, the Natural Sciences and Engineering Research Council of Canada and the Fonds pour la formation de chercheurs et l'aide à la recherche.

References

[AVWW96] J. L. Armstrong, S. R. Virding, C. Wikström, and M. C. Williams. *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996.

[AXD98] *AXD 301 High-performance ATM switching system*. Ericsson Telecom AB, 1998.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[Che70] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[DFS96] Danny Dubé, Marc Feeley, and Manuel Serrano. Un GC temps réel semi-compactant. In Guy Lapalme and Christian Queinnec, editors, *Journées Francophones des Langages Applicatifs*, volume 7, pages 165–181, Val-Morin, Québec, Janvier 1996. INRIA.

[DLM+78] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[Dub96] Danny Dubé. Un système de programmation Scheme pour micro-contrôleur. Master's thesis, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, April 1996.

[Fee93] Marc Feeley. Polling efficiently on stock hardware. In *Proceedings of the Functional Programming and Computer Architecture*, pages 179–187, Copenhagen, June 1993.

[FL98] Marc Feeley and Martin Larose. Compiling Erlang to Scheme. In *Proceedings of the 1998 Programming Languages, Implementations, Logics and Programs Conference*, September 1998.

[FM90] Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Conference Record of the 1990 ACM Symposium on Lisp and Functional Programming*, pages 119–130, Nice, France, June 1990. ACM Press.

[FMRW97] M. Feeley, J. Miller, G. Rozas, and J. Wilson. Compiling Higher-Order Languages into Fully Tail-Recursive Portable C. Technical Report 1078, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, Août 1997.

[Gab85] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press Series in Computer Science. MIT Press, Cambridge, MA, 1985.

[HDB90] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. *ACM SIGPLAN Notices*, 25(6):66–77, 1990.

[HFA+96] Pieter H. Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, Peter Baumann, Marcel Beemster, Emmanuel Chailloux, Christine H. Flood, Wolfgang Grieskamp, John H. G. van Groningen, Kevin Hammond, BogumiłHausman, Melody Y. Ivory, Richard Jones, Peter Lee, Xavier Leroy, Rafael Lins, Sandra Loosemore, Niklas Röjemo, Manuel Serrano, Jean-Pierre Talpin, Jon Thackray, Stephen Thomas, Pierre Weis, and Peter Wentworth. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4), 1996.

[NO93] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation. Published in SIGPLAN Notices*, volume 28, pages 217–226, Albuquerque, New Mexico, June 1993. ACM Press.

[Ste75] Guy L. Steele, Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.