Scalable BIBOP garbage collection for parallel functional programs on multi-core machines

Luca Saiu

Laboratoire d'Informatique de l'Université Paris Nord – UMR 7030
99, avenue Jean-Baptiste Clément - F-93430 Villetaneuse – France
saiu@lipn.univ-paris13.fr

Abstract

Exploiting modern multi-core processors requires task-parallel programs which are simply too hard to implement with traditional techniques; as high-level languages rely on automatic memory management subsystems, such subsystems must be made fast and scalable, tuned for today's complex memory architectures.

In this paper we describe our implementation of a new parallel non-moving garbage collector for shared-memory machines, based on a variant of the BIBOP organization. Building on the experience of Boehm's work and revisiting some older ideas in the light of current hardware performance trends, we propose a design leading to compact data representation and measurable speedups, particularly in the context of functional programs.

While discussing in detail the performance-critical sections of the implementation we provide an intuitive justification for our choices which we then corroborate with measurements.

This effort results in a particularly clean architecture based on just a few data structures, which lends itself to experimentation with alternative techniques.

Categories and Subject Descriptors D.3.4 [*Processors*]: Memory management (garbage collection); D.4.2 [*Storage Management*]: Garbage collection; D.1.1 [*Applicative (Functional) Programming*]

General Terms Design, Performance

1. Introduction

In recent years improvements in processor performance have been due more and more to increased parallelism, while the trend of rising processor clock frequency has dramatically slowed down. In contrast to what happens with instruction-level parallelism, the task parallelism offered by modern multi-cores must be explicitly exploited by the software, if *any* speedup is to be obtained [27].

Facing the inherent complexity of parallel programming, high-level languages and tools now become an absolute necessity. In this work we concentrate on memory management, the most complex part of a programming language runtime.

As multi-core architectures support a shared-memory model[1] the techniques presented here extend from the now ubiquitous desktop multi-core machines to the older multi-socket SMPs, and to most recent medium-size parallel machines containing several multi-core CPU dies.

The architecture we illustrate here is also suitable for sequential machine, but the need for such a software is particularly stringent in a parallel context. In a sense the rise of the number of CPUs *amplifies* the memory wall problem: the memory bandwidth is a limited hardware resource which all cores have to share, and raising the parallelism degree inevitably tightens up the bottleneck, even without any synchronization.

1.1 Motivations

While planning a new implementation of our functional programming language, GNU `epsilon`[2] , we looked for a free software parallel garbage collector; Boehm's conservative-pointer-finding collector [7] was the obvious choice, and essentially the only one: see Section 12 for a short discussion of the available alternatives.
Despite its ease of use and good speed we felt that it was possible to obtain better performance and scalability than offered by Boehm's collector with relatively little effort, particularly in the context of a functional language; some tests with `nanolisp`, a simple implementation of Lisp that we initially wrote as a prototype of `epsilon`'s runtime, confirmed this intuition.

1.2 Objectives

Boehm's garbage collector is the natural point of comparison for our work because of several design similarities, including the idea of (partially) conservative pointer finding, and the use of Unix

[1] The architecture shown here does not generalize so well to NUMA machines, more suitable as they are to a message-passing style where each task runs in its own addressing space; message-passing is also interesting, as the same interfaces could scale up to parallel computation *over the network*.
Moving away from thread parallelism to pure *process* parallelism (one heap per process) would essentially eliminate the problem of parallel non-distributed garbage collection, but such a revolution appears unlikely. Other organizations like NUMA machines composed by SMP nodes, or machines where the NUMA effect is pronounced only between "distant" nodes, look more realistic and are already being adopted by some current high-class machines [12]. For such a hybrid SMP-in-NUMA model the techniques shown here apply at the SMP level, just in the same way as they would apply to each single machine in a cluster of SMPs.

[2] `epsilon`[22] is a mostly functional language, officially part of the GNU Project [16]. It will be released as free software under the GNU General Public License. The new implementation, our fourth rewrite, will have essentially nothing in common with the previous ones and will be explicitly targeted to parallel machines.

signals to interrupt mutators[3]. For this reason it may be worth to quickly highlight the main objectives we have set forth for our implementation, in order to better explain the need for our effort and to illustrate key similarities and differences. Our objectives also more or less dictate several design and implementation choices which we prefer to make explicit from the beginning.

First of all, C is clearly the language providing the best control on performance for such a low level implementation where each memory access matters. A slightly less obvious choice is determined by the typical usage of *parallel* systems, tending to concentrate on bulk processing rather than interactive applications: for this reason we consider *bandwidth*, and not latency, to be a priority; this choice excludes most incremental schemes (but see Section 10), and favors a *stop-the-world* model where many threads can mutate in parallel or collect in parallel, but without any time overlap between the two phases — all which is similar to Boehm's solution. Since `epsilon` is a functional language and functional programs allocate a large number of small objects it is paramount to make a good use of the limited space in the primary and secondary caches (henceforth simply *L1* and *L2*), by tightly packing objects together: we want to avoid padding space between heap objects and not to force alignment constraints not specified by the user. Anyway, even if functional programs are our first concern, we would like our collector to be also useful for (human-written) C programs, which encourages us to adopt a non-moving strategy like *mark-sweep* and to avoid safe-points and use *conservative pointer finding* **for roots**; on the other hand there is no reason why other heap objects should not be traced exactly. The collector API should be usable by humans, but not necessarily similar to `malloc()` — an important difference with respect to Boehm's collector. In the same spirit of generality, our implementation should be usable from other systems, general and tunable at configuration time.

1.3 Contributions

In this work we describe an high-performance, versatile and relatively simple mark-sweep parallel garbage collector and memory system for SMPs with good scalability (see Section 9), making good use of the memory architecture (Section 6), particularly well-suited to the allocation patterns of functional programs but language-agnostic.

Most of our implementation ideas rely on a variant of the classic BIBOP strategy [25, 13] which, despite its simplicity, has been exploited surprisingly little: the only discussion of an actually implemented similar solution that we have found is in a 1993 paper by E. Ulrich Kriegel, [19]. In this paper we provide an intuitive and experimental justification of the validity of this particular fashion of dividing the heap into pages, now in the context of a modern multiprocessor architecture.

The enormous size of the garbage collection literature makes it hard to claim novelty for most ideas, some of which are variations of very old implementation techniques, possibly adapted to the changing trends of hardware performance. Nonetheless we feel relatively confident in declaring original the following points:

- Our user-level interface to the memory management system based on *kinds*, *sources* and *pumps* may have some aesthetic value, besides slightly boosting allocation performance.
- We define a set of core data structures and primitives which can be efficiently implemented, providing a clean and natural high-level architecture for parallel collectors.
- We show how the BIBOP scheme is appropriate for reducing memory pressure on machines with modern memory hierarchies, introducing the concept of *data density* which we show to be at least one reason for the good performance of our implementation.
- If further developments follow our implementation may evolve into a testbed for comparing the performance of different collection strategies, in a parallel setting; such a framework may show its utility in tailoring the best collection algorithm to a particular mutator program, on the machine at hand. As far as we know, no implementation is currently available to do this in a parallel setting and without being tied to a particular language.

2. The functional hypothesis

Functional programs tend to allocate many small objects, the great majority of which have one of only a few possible "shapes"; in practice, most heap objects will be conses, nodes of balanced binary trees, or more generally components of inductive data structures with fixed size and layout, often containing some constant attributes which must be frequently inspected at runtime such as types in Lisp, or constructors in ML or Haskell. Depending on the programming style closures might also be allocated in quantity; allocating other objects tends to be statistically much less frequent, hence less critical for performance.
We define the above set of assumptions as the *functional hypothesis*: our system is designed to run most efficiently when such hypothesis is verified, yet `epsilonGC` can and does work with any language, and may even be directly employed for user-written C programs. Anyway the functional hypothesis is the underlying idea dictating many implementation choices, and in particular the BIBOP strategy central to our design.

3. The user view: kinds, sources and pumps

At a very high level, any automatic memory management system serves to provide the illusion of an *infinite* stream of objects created on demand, each satisfying some specified requirements such as size and alignment.

Objects which are not useful any longer can be simply ignored: there is no need, in general, for a user interface to the recycling system itself as the whole point of garbage collection is to make object reusing *invisible* to the user, who just keeps creating more objects as if the memory were unlimited.

The user-level API is built upon three main data structures: the *kind*, each instance of which defines one particular set of requirements for a group of homogeneous objects, the *source*, which arranges for the creation of objects of one specified kind, and the *pump*, providing a single mutator thread with objects from a given source on demand, one object at a time.

Kinds: We define a *kind* as the specific representation of a group of homogeneous heap objects. Each kind is characterized by a given *object size*, *object alignment*, a *tracer* function specifying how to mark the pointers contained in an object given its address, and particular *metadata* values: metadata include[4] an integer *tag* and a *pointer*, sharing the same values for all the objects of the same kind. Given a pointer to a heap object, mutators are permitted to inspect, but not modify, its metadata.

[3] Notwithstanding the outdated information at `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gcdescr.html` Boehm's collector now also employs signals to stop mutators on all major platforms except Windows, where Unix signals are not supported but an analogous mechanism exists for suspending a thread from another thread. [6] mentions GNU/Linux, Solaris, Irix and Tru64. The Windows implementation in `win32_threads.c` uses signal-like primitives like `SuspendThread()`.

[4] Even if they currently comprise only tag and pointer, more metadata can be easily added in the future if the need arises.

In general a kind should not be confused with a *type*: rather than a type it identifies one *case* among the potentially many variants which, together, make up a type. For example a *cons* kind could be defined, but **not** a *list* kind, which would also comprise the empty list case, having of course a different representation — by the way, likely not on the heap.

The tag could be usefully employed in a dynamically-typed language such as Lisp, for example in order to test at runtime whether a given object is, effectively, a cons. In a statically-typed language like ML the tag can encode the constructor of tagged-sum objects. The pointer metadatum can be useful to refer any reflection-related data not fitting in a single integer.

All the needed kinds are typically defined at initialization time, as global structures shared by all mutator threads.

Sources: From the user's point of view a *source* can be seen as a global inexhaustible source of objects of a given kind. In the typical case the user will define exactly one source per kind at initialization time, as an object shared by all mutator threads; after initialization mutator threads will only refer sources to create their pumps.

Pumps: A *pump* is a *thread-local* data structure implementing but one user-level functionality, the creation of an object.

Each mutator thread will create its own pumps referring the shared, global sources, then use its pumps to obtain new objects. Pumps have to be explicitly destroyed at thread exit time.

Kindless *objects:* The strategy outlined above — creating objects of some kind which as been defined in advance — suffices the great majority of the objects ever created at runtime: for example in Lisp most heap-allocated objects will be conses, and Prolog heaps will mostly be made of terms. We call *kinded* all the objects created as shown above.

Some other heap-allocated objects do not fit so well in the picture as it is not possible to foresee in advance their exact size: arrays and character strings come to mind[5]. We provide more "traditional" allocation primitives for *kindless* objects.

Note how the kindless object API (see Figure 1) provides for less control: vector elements can be either *all* potential pointers, or they can be guaranteed by the user to include *no* pointers. There is not much control on metadata either: all objects share the same[6] tag and metadatum pointer; a user requiring more expressive metadata has to explicitly encode them in the payload.

Miscellaneous user functionalities: Other primitives are provided to initialize and finalize the collector, to register and unregister roots, to notify the memory system about new threads or exited threads, to explicitly force a collection, and to temporarily disable collections and re-enable them.

As all of this is canonical and not particularly interesting, we will not further pursue such details.

4. Architecture

Despite their visual intuitiveness, the data structures above were designed primarily for efficiency, and the actual role of each structure is not apparent to the user: in particular a central data structure, the *page*, is completely hidden.

[5] Other slightly less obvious cases are *procedure activation records*, which some runtimes of Scheme, Prolog and SML allocate on the heap; if the language supports dynamic code generation even *code blocks* (either machine language or bytecode) might be heap-allocated and garbage collected.

[6] The actual values can be specified at initialization time, but nonetheless they must be the same for all kindless objects; it is typically reasonable to choose some values not used for kinds, so that at least kindless objects can be distinguished from kinded ones.

Pointers are essential in the implementation of any language requiring dynamic memory allocation, and in order to make pointers easier to recognize at runtime and their dereference more efficient[7], we restrict the set of heap pointers considered valid to *word-aligned* pointers; one word is also the minimum size of a heap object representable without space overhead, and all the integers internally used in the implementation are of type `intptr_t`, so that the size of all memory structures remains a multiple of a word size.

The description below will proceed *from the bottom up*: since many data structures and operations are usable with different collection strategies requiring little or no modifications, we prefer to illustrate the various possible operations before our way of combining them, in the spirit of separating policy from mechanism.

4.1 Kinded objects

We represent each kinded object as a buffer of words, with *no header*; the rationale of this choice is discussed in more depth in Section 6, but the main idea is simply to have long packed arrays of objects in memory, without any padding unless absolutely necessary[8].

4.2 BIBOP pages

All kinded objects are allocated from data structures called *pages*[9], similar to Kriegel's "STSS cards" [19]: whenever a pump returns a pointer to a new object, the resulting address will refer a word contained in a page.

Each page can only contains objects of *one* kind. For each kind any number of pages, including zero, may exist at any given time.

All pages have the same size, which must be a power of two; the page size is also equal to its *alignment*: the rightmost log_2`epsilonGC_PAGE_SIZE_IN_BYTES` bits of a page pointer are always guaranteed to be zero.
A page is divided into *page header*, *mark array* and *object slot array*.

Page header: The page header contains a copy of the kind metadata, which of course are valid for all the objects in the page; the object referred by the metadatum pointer, if any, is shared by all the pages of the same kind: only the pointer is copied.

Other information contained in the header includes kind-dependant data such as the object size and effective size, the payload offset, and the number of object slots in the page. All of this is computed once and for all when a kind is created, and simply copied at page initialization time. The address of the first dead slot (see below) is also held in the header.

[7] On many RISC architectures pointers to misaligned objects may not be just a performance concern: some processor families such as the *Sparc* simply raise an exception in response to any attempt to dereference a non-word-aligned pointer. Others, such as the *x86* family, execute the misaligned dereference, but imposing a heavy execution time penalty.
We prefer to simply forbid such pointers for all architectures, which may improve performance and helps to avoid the misidentification of many false pointers.
We also assume convertibility from integer to pointer and vice versa without loss of information: even if not mandated by the C Standard (the type `intptr_t` itself is optional) such an assumption is in practice true on all architectures.

[8] Padding *must* to introduced sometimes in order to respect the alignment constraints provided by the user: for example the user might require a three-word structure to be aligned to two or four words; in such cases there is no way to avoid wasting some space for each object.

[9] There is no *a priori* relation between BIBOP pages and operating system pages, whose sizes may well be different: BIBOP pages will typically be at least a few times larger than operating system pages, but still smaller than the L2 cache. In the following we use the term *page* to mean "BIBOP page".

```
/* A tracer is a pointer to a function taking a pointer
   as its parameter and returning nothing: */
typedef void (*ourgc_tracer_t)(ourgc_word_t);

/* Create a kind: */
ourgc_kind_t
ourgc_make_kind(const size_t object_size_in_words,
                const ourgc_unsigned_integer_t
                  pointers_per_object_in_the_worst_case,
                const size_t object_alignment_in_words,
                const ourgc_metadatum_tag_t tag,
                const ourgc_metadatum_pointer_t pointer,
                const ourgc_tracer_t tracer);

/* Create a source from a kind: */
ourgc_source_t ourgc_make_source(ourgc_kind_t kind);

/* Initialize a (thread-local) pump from a source: */
void ourgc_initialize_pump(ourgc_pump_t pump,
                           ourgc_source_t source);

/* Finalize a pump before exiting the thread: */
void ourgc_finalize_pump(ourgc_pump_t pump);

/* Allocate a kinded object from a thread-local pump: */
ourgc_word_t
ourgc_allocate_from(ourgc_pump_t pump);

/* Lookup metadata: */
ourgc_tag_t
ourgc_object_to_tag(const ourgc_word_t object);

ourgc_metadatum_pointer_t
ourgc_object_to_metadatum_pointer(const ourgc_word_t
                                  object);

ourgc_integer_t
ourgc_object_to_size_in_words(const ourgc_word_t
                              object);

/* Allocate kindless objects: */
ourgc_word_t
ourgc_allocate_words_conservative(const ourgc_integer_t
                                  size_in_words);
ourgc_word_t
ourgc_allocate_words_leaf(const ourgc_integer_t
                          size_in_words);
ourgc_word_t
ourgc_allocate_bytes_conservative(const ourgc_integer_t
                                  size_in_bytes);
ourgc_word_t
ourgc_allocate_bytes_leaf(const ourgc_integer_t
                          size_in_bytes);
```

Figure 1. `epsilonGC`'s essential user-level API.
The source above is directly copied from header files, with only GCC function attributes (to force inlining and such) removed and comments eliminated. Despite looking unconventional the interface is not particularly complex, and in fact is conceived so that performance-critical operations such as `epsilonGC _allocate_from()` and metadata lookup functions can be easily re-implemented in assembly, to be generated by a compiler as intrinsics.

Since the header has offset zero within the page, given a pointer to any kinded object, even internal, the address of its page header can be trivially obtained by *bitwise and*ing the pointer and the *page mask*, defined as the *bitwise negation* of `epsilonGC_PAGE_SIZE_IN_BYTES`-1. This allows mutators to access metadata at runtime with an overhead of two to four assembly instructions, when needed; on the other hand the negligible space overhead of storing metadata once per page makes this solution completely acceptable even for languages which don't make use of them.

Mark array: The mark array is placed right after the header, with no padding: since the header size is a multiple of the word size, the mark array is guaranteed to always begin at a word boundary.

The mark array stores liveness information for each object[10] in the page: since we currently need only one bit per object, the array could conceptually always be implemented as a bit vector.

As marking is parallel, mark arrays are concurrently updated by several threads, which requires some atomic memory accesses (see Subsection 4.6). On many machines byte stores are always atomic, and even when suitable atomic instructions for bitwise operations are provided working with a *byte vector* may be more efficient[11]. On (hypothetical) architectures where the compiler did not support the required intrinsics, and where an atomic byte store were not provided, one could use a *word vector*. The implementation allows the user to choose at configuration time among bit, byte or word, bit being the default.

Alternatively, it is possible to enable *out-of-page mark arrays* at configuration time, so that mark arrays are stored as separate `malloc()`ed buffers; in this case the mark array area in a page degenerates to a single pointer, and accessing the mark array from a page requires one indirection. Our original rationale for implementing this strategy was to avoid some cache conflict misses due to the fact that mark arrays share the same alignment on all pages. As benchmarks in Section 9 show that this is not a problem in practice with modern multi-way set associative caches, this strategy has not been pursued further by separating headers from slot arrays.

[10] It is interesting to compare this with Boehm's collector, which stores one element per object *word*, thus making tracing simpler. We have chosen to slightly complicate the mapping from mark array elements to objects instead, to speed up the critical operation of page sweeping, and in general trading more computation for fewer memory accesses.

[11] [8], written in 2000, compares the solutions on several architectures, finding that the optimal solution depends on the machine. According to our recent tests, the best strategy between bit arrays and byte arrays remains machine-dependent as of 2008.

Object slot array: The *object slot array* begins after the end of the mark array, at the first word with the required alignment. Object slots contain the "payload" of each page. At any given time each object slot may be either *used* or *unused*: when used it contains an object payload; when unused, its first word contains a pointer to the next unused object in the same page, or `NULL` in the case of the last unused slot.

For each page unused slots make up an independent free-list where elements are always ordered by address.

In order to avoid mistaking free list pointers in unused objects for pointers in used objects during conservative pointer finding, free list pointers are stored in *concealed* form by default[12].

Concealing consists in applying some function $c : Pointer \rightarrow Pointer^C$ to a free list pointer; it is important for c to be bijective, as concealing and then *unconcealing* (i.e. applying c^{-1} to) a pointer must preserve information.

c is trivially implemented as a C macro computing the successor function in `unsigned` (wrap-around) arithmetic: since its domain consists of word-aligned pointers, the elements of its image are guaranteed to be misaligned[13], hence they cannot be mistaken for pointers. The cost of applying either c or c^{-1} is one assembly instruction requiring no memory accesses[14].

Depending on the kind, some unused space may be present between the end of the mark array and the beginning of the slot array, and at the end of the page; in either case these two padding spaces are strictly smaller than the object effective size.

The global page table: The global *page table* serves to recognize which part of the address space is being used for the garbage-collected heap; such information is important for avoiding dereferencing false pointers when doing conservative pointer finding. Moreover, the collector needs to be able to recognize whether a heap pointer refers a kinded object in a page slot array or a large object — no particular provision is needed for kindless small objects, but we defer the justification of this fact to Section 4.5. Since we support interior pointers for large objects, it must also be possible to efficiently map an arbitrary (word-aligned) interior pointer to an initial pointer.

[12] Free list pointer concealing can be disabled at configuration time.

[13] Here we are depending on byte-addressable memory; This is true for all contemporary general-purpose machines.

[14] Assuming instructions such as either `inc`/`dec` or `add`/`sub` with a small *immediate* parameter; again, all modern machines satisfy this condition.

We call *candidate pointer* a word which is suspected to be a (possibly interior) object pointer at marking time, and *candidate page* the address of the hypothetical page which would contain the object referred by a candidate pointer. Of course candidate pages have alignment log_2`epsilonGC_PAGE_SIZE_IN_BYTES`.

At an abstract level, the table implements a function f mapping a non-`NULL` candidate page p to an element s of the disjoint sum

$$Sort \triangleq \{kinded\} + \{nonheap\} + LargeObjects$$

If $f : p \mapsto kinded$ then the candidate page p is actually a page; if instead $f : p \mapsto nonheap$ then p is a pointer referring some object out of the garbage-collected heap, or a false pointer. Otherwise $f : p \mapsto l$, where l is the address of the beginning of the large object containing the word pointed by p.

Given a value for p stored as a key, a simple encoding allows us to represent any element of $Sort$ in a single word: `NULL` represents $nonheap$, $s = p$ stands for $kinded$, and any other value of s is interpreted as a large object pointer.

The table is implemented as a simple resizable chained hash where the first element of each bucket is stored within the bucket pointer array itself[15], as first described in [29]; the hash function is modulo.

One essential optimization at mark time consists in *not* consulting the page at all, which would be comparatively expensive, for `NULL` or misaligned candidate pointers.

It is interesting to notice how all *updates* to the global page table occur at mutation time, when creating or destroying[16] pages and large objects; unfortunately such updates require critical sections which, short as they are, may nonetheless limit scalability. By contrast at collection time the table is only *read*, which allows us to completely avoid critical sections for table access during that stage.

4.2.1 Page creation

Creating a page involves allocating space from the C heap, filling the header fields, initializing the mark and object slot arrays and registering the page in global structures.

Because of the alignment requirements we currently allocate pages with `posix_memalign()`[17]; as this may involve a kernel call and/or synchronization in the C library, such operation tends to be both expensive and hard to parallelize.

Filling the header involves little more than copying some fields from the kind data structure, which is directly referred by the source, and making the free-list head point to the payload beginning. Nothing of this is performance-critical.

The mark array has to be zeroed at creation, with a `memset()` call. This should be relatively efficient, just involving some evictions from L1 — however having the mark array in L1 at page creation time does not buy us anything, as mark arrays are only touched during collection. If out-of-page mark arrays are enabled then we should add a `malloc()` call to the cost.

Building the free list involves some memory traffic, as all objects need to be touched. Unless objects have effective size larger than a cache line the complete object slot array has to be brought into cache. Even if this phase by itself is expensive, it may work like a sort of prefetching: if the page is used soon, all of it will already be loaded at least in the L2 cache.
We define *backward free list building*[18] the strategy of building the free list starting from the *last* slot which will be used for allocation. This solution has locality advantages in case of large page size, under the assumption that a just-created page will be used soon for allocating: if the page size is larger than the L1 data cache, building the free list backwards makes it very likely that the memory touched first while allocating will be already in L1; the rest of the page will be still in L2. It is possible to choose between forward and backward free list building at configuration time.

The final step is registering the page in the page table, which requires a critical section on the global mutex, plus a `malloc()` call within the critical section in case of hash collision.
All of this makes page creation a relatively expensive and non-scalable operation.

4.2.2 Page sweeping

Sweeping can be performed on an individual page without need for synchronization or kernel calls. It simply involves scanning the mark array and, for each i-th element, either clearing the corresponding element if $array[i]$ is one, or making the i-th object slot *unused* by re-adding it to the free-list if $array[i]$ is zero. Since the mark array is examined in order (either forward or backward, as per the free-list building direction), free list elements are kept ordered by address in the list. All the words of dead objects other than the first one are overwritten[19], to prevent future false pointers referring the slot to keep alive the objects which were referred by the now dead slot.

Memory access patterns in sweeping are similar to the ones in mark array initialization and free-list construction; in particular a just-swept page will likely remain cached at least in L2 — and the next lines to be used will be in L1, if backward free list building is enabled.

---

[15] This optimization is the reason why we don't include `NULL` in the domain of f: we use the value `NULL` as a key in a hash table element out of the bucket to mean that the element is currently unused.

[16] See Subsection 4.6 for the reason why pages must be destroyed at *mutation* rather than collection time.

[17] An interesting alternative to explore would involve using `mmap()` to allocate a group of pages; for some (non-GNU) implementations of `posix_memalign()`, the `mmap()` solution might incur a significantly lower space overhead, at the cost of always involving the kernel in page creation. Using `mmap()` could in fact make deallocation more portable, as `free()`ing buffers allocated with `posix_memalign()` is only permitted on GNU systems, as far as we know ([20], "Allocating Aligned Memory Blocks", currently at subsection 3.2.2.7).
However the `mmap()` solution has some issues of its own: `mmap()` only guarantees `sysconf(_SC_PAGESIZE)` alignment, hence pages could only be reasonably `mmap`ped in large groups, with some space overhead at the beginning and the end. Making `epsilonGC _PAGE_SIZE_IN_BYTES` equal to `sysconf(_SC_PAGESIZE)` would solve the space overhead problem, but at the price of forcing pages to be unacceptably small. `unmmap`ping space from the middle of a `mmap`ped buffer is supported, but deallocation of single pages would still be a problem unless `epsilonGC _PAGE_SIZE_IN_BYTES` were chosen to be a multiple of `sysconf(_SC_PAGESIZE)`. Re-`mmap`ping a previously `unmmap`ped part of a buffer is *typically* supported, even if such behavior is not mandated by POSIX. In addition we would need some data structure to keep track of which pages in a large buffer are `mmap`ped at any given time.
Anyway, despite all the complexity, such an idea seems worthy of some exploration.

[18] The actual direction of free list building, from higher addresses down to lower ones or from lower addresses up to higher ones, has no effect on performance as long as it is *the opposite* of the allocation direction: note in particular how automatic hardware prefetching works in either direction on modern processors ([12], section 3.3.2, "Single Threaded Sequential Access").

[19] Each word is overwritten with a configuration-dependent value impossible to mistake for a pointer: either the `0xdead` constant (which is easy to recognize for humans) if the collector is configured in debug mode, or otherwise simply `0` (which might lead to a slightly more efficient implementation on some architectures, possibly saving a *load immediate* instruction). Overwriting dead slots can also be completely disabled at configuration time.

4.2.3 Page refurbishing

It is possible to re-use an empty page of some kind for objects of another kind: such operation is called *refurbishing*, and involves reconstructing the header, mark array and free list.

Refurbishing has essentially the same overhead as sweeping, and the cache effects of the two operations are also comparable: allocations from a just-refurbished page on the same thread which performed the refurbishing is efficient as all the page cache lines will still be in L1 and L2.

4.2.4 Page destruction

Destroying a page involves its deallocation and removal from the global page table: such operations are expensive and non-scalable, involving synchronization and possibly kernel calls.

4.3 Sources

From the implementation point of view a source is quite a trivial structure, serving as repository of pages. Each source simply contains two lists of pages, the *full pages list* and the *non-full pages list*, plus a mutex for synchronizing access to such lists.

4.4 Pumps

Pumps are performance-critical structures whose purpose at the implementation level consists in caching frequently accessed data about the objects to allocate. Such criticality is evident from the API in Figure 1, showing how existing pump data structures are *initialized* rather than dynamically allocated, in an effort to save a pointer indirection at runtime: pumps are conceived to be declared in programs as `__thread` variables of type `struct epsilonGC_pump`, rather than as pointers.

At any given moment a pump may conceptually "contain" a page reserved to the allocating thread, or no page; of course at the implementation level such an inclusion is represented with a page pointer field. Its other relevant field is the current head of the page free list, again kept in the pump rather than in the contained page in order to avoid a pointer indirection at allocation time: in fact the free-list head field of the page is, counter-intuitively, *not* updated at each allocation. The free-list head field of the *pump* is set to `NULL` when the pump contains no page.

4.4.1 The allocation function

Despite the allocation being the only user-level operation on a pump, such a functionality is very performance-critical. Allocating from a given pump involves unconcealing the free-list field into a temporary variable, if non-`NULL` dereferencing it, setting the free-list head to the just loaded value and finally returning the temporary. This shorter and far more common execution path is carefully optimized and costs about *ten assembly instructions*, with no taken[20] jumps; the other execution path is taken in case of *page change* time, when a page is filled and another one must be acquired from the relevant pool, or at the first allocation for a pump with no page: it involves synchronization with the pool mutex and access to its lists. If no non-full pages are available, a page is taken from a *global empty pages list* (at the cost of one further synchronization) and refurbished if needed. If no empty pages are available, an heuristic is employed to decide whether to create a new page, or to trigger a collection. Page change is also the taken as the occasion for destroying empty pages, if an heuristic says that there are more than enough: the rationale here is to avoid destroying pages too frequently, since they might be needed again and both creation and destruction are expensive.

Repeatedly allocating from a page which was recently swept by the same thread and which contains many unused slots should be cache-friendly: sweeping works like a prefetch phase to load the page payload into the L1 or L2 cache, and even without on-demand sweep the hardware automatic prefetch may be activated when there is much free space on the page, as consecutive addresses are generated. Using pumps automatically guarantees that a page is only used for allocation by one CPU at a time, which avoids cache ping-pong.

[20] It is worth to give GCC an optimization hint with `__builtin_expect()`.

4.5 Kindless and large objects

The data structures and primitives shown above provide no hints about the implementation of kindless objects, yet the idea is quite simple. A set of *implicit kinds*, *sources* and per-thread *pumps*[21], of user-definable sizes, are automatically defined: in this sense most kindless objects are just kinded objects "in disguise", only slightly less efficient because of the need for mapping an object size to a pump at runtime, and because of the possibility of internal fragmentation: not all possible sizes will be realistically provided, so the allocation of an object of a given size might be satisfied by using a larger buffer. For each size two kinds are provided, one with a fully conservative tracer, and another one with a leaf tracer (called "atomic" in the jargon of Boehm's collector).

It is easy to see how the solution above is not completely general, as it cannot satisfy allocation requests for objects larger than a page or even just larger than the maximum implicit kind size which has been fixed by the user. A different mechanism is provided for *large objects*, which are simply allocated one by one with `malloc()` and destroyed with `free()`. Their implementation is simple-minded and quite inefficient in both space and time, which given the functional hypothesis should hopefully not be serious. Of course the user-level API completely hides the difference between implicitly-kinded and large objects.

4.6 Garbage collection

A collection is initiated by one mutator, which stops all the other mutators with a signal. This choice has the advantage of allowing a simple user API, but significantly complicates the collector implementation: any function not reentrant with respect to signals, notably including `malloc()` and `free()`, can not be used at collection time: this is the reason why empty pages have to be destroyed at *mutation* rather than collection time.

The collection phase may internally proceed in two different orders according to a configuration option: if *on-demand sweeping* is enabled, as per the default, the three sub-phases are *non-deferred sweeping, root marking and marking*, otherwise they are *root marking, marking and sweeping*. In any case it is central to maintain the invariant according to which a complete heap marking is followed by a complete sweeping, before the next marking can begin.

On-demand sweeping consists in sweeping a page during mutation at page change time, *just before allocation from it begins*: such a choice is more cache-friendly than the traditional *stop-the-world sweep*, but it may leave some pages still to be swept when a collection begins: the non-deferred sweeping sub-phase, typically very short, serves to sweep such remaining pages. Non-deferred sweeping and stop-the-world sweeping share the exact same implementation.

After collection all mutators are restarted with a second signal.

Root marking: Root marking is very simple, and currently *sequential* (this should be changed in the future: see Section 9). Just like Boehm's collector in most of its configurations, it uses `setjmp()` for finding register roots in a portable way.

[21] Implicit pumps are created at thread registration and destroyed at thread un-registration time.

Marking: Given the atomicity of mark array stores parallel marking can easily proceed in parallel without synchronization, if we accept the possibility of some (statistically unlikely) duplicate work; our implementation is quite canonical and closely follows Boehm's one [9], with load balancing in the style of Taura and Yonezawa [14][22]. It should be noted that the BIBOP organization does not affect marking in any significant way.

Sweeping: Parallel sweeping is even simpler, with pages dictating the natural granularity for the operation of each thread: pages are simply taken from a list, swept and put back into another list.

4.7 Synchronization

One interesting and possibly original detail involves our locking style: in order to prevent a collection from starting during a critical section at mutation time, a global *read-write lock* is locked for reading at mutation, before acquiring the relevant mutex: the collection triggering function, before sending the signal, locks the same read-write lock *for writing*.

5. Implementation

`epsilonGC`'s implementation totals around 5000 lines of heavily commented C code, indeed quite easy to understand for being such a low-level concurrent piece of code not sparing C macros, `#ifdefs`, GCC function attributes and intrinsics in order to be support *Autoconf* options and be as general and efficient as possible.

The code will be soon publicly released.

6. Data density

The system internally measures object size and alignment in *machine words*, and one word is the minimum size of a kinded object which can be represented without padding, in absence of alignment constraints specified by the user; with an alignment greater than one word, it becomes necessary in some cases to add some padding space right after the object payload; we call the ***effective size*** of an object the sum of its size and its alignment padding.

Given a kind k of objects with alignment a_k and size s_k, we define the effective size e_k needed to store each object, and the corresponding ***data density*** d_k, the number of objects representable per word, as:

$$e_k \triangleq a_k \cdot \left\lceil \frac{s_k}{a_k} \right\rceil \qquad\qquad d_k \triangleq \frac{1}{e_k}$$

The definitions above intentionally disregard all the sources of memory overhead out of object slot arrays, including mark arrays and all garbage collector data structures, the rationale being that density is not meant as a measure of memory occupation, but rather as an index of *the number of objects fitting in a cache line*: as mark arrays and other collector data structures are mostly accessed at different times from the objects *per se* and reside in different cache lines, optimizing data density maximizes the amount of useful information stored in the physically limited cache space at mutation time.

Data density may be reasonably defined in the same way independently from the garbage collecting strategy, and indeed it is of some interest to compare the values of d_k in different memory management systems for two kinds which are widely employed in functional programs, the *cons* (two words: *car* and *cdr*) and the non-empty *node* of an Red-Black binary tree of some given color[23] (three words: *left*, *datum* and *right*). Neither kind has alignment requirements, hence $a_{cons} = a_{node} = 1$.

Several systems such as the *GNU libc* `malloc()` facility [20], all the other allocators derived from Doug Lea's `malloc()` and — even more interestingly — Boehm's collector [7], allocate all buffers at double-word-aligned addresses and may also add some internal status information *to each buffer*; metadata, when needed, must be represented as part of *each* object, adding to s_k. Instead many other systems, including just for example OCaml, do not force any alignment but always add one header word per object[24], sufficient to include a short tag, which again we consider part of s_k.

If metadata are accessed at runtime, as it is the case with dynamically-typed languages, with Boehm's collector we have $d_{cons} = d_{node} = \frac{1}{4}$. When metadata are not needed Boehm has optimal density in the cons case with $d_{cons} = \frac{1}{2}$, but again $d_{node} = \frac{1}{4}$. In OCaml, with or without metadata, $d_{cons} = \frac{1}{3}$ and $d_{node} = \frac{1}{4}$.

Independently of the need for metadata at runtime our model allows us to reach optimal density for both kinds, with $d_{cons} = \frac{1}{2}$ and $d_{node} = \frac{1}{3}$.

The data density of a particular representation seems likely to play a role in the *overall* efficiency of the system, even ignoring the cost of allocation and collection and considering only object accesses; anyway further empirical evidence will be needed to confirm this supposition for real world programs.

Subsection 9.1 includes a comparison of the performance of `epsilonGC` on `nanolisp` in its default configuration and when using a different data representation with intentionally lower data density.

7. Closures

Functional programs written in certain styles[25] create a considerable number of short-lived anonymous functions at runtime, implemented as closures. At a first look such a scenario does not seem to respect the functional hypothesis, as in principle closures can have many different shapes, depending on the number of non-locals captured in the environment, and on the fact that each non-local can be a pointer or a non-pointer.

Even if allocating all closures (or just their environments, when they need to be heap-allocated) as kindless objects would work, the overhead of such a simple-minded solution is in fact easy to avoid.

First of all it should be observed that the great majority of functions need either zero or one variable in their non-local environment; it may be worth to add specific kinds for such common cases, and possibly also for the most performance-critical functions with larger non-local environments, when it is possible to recognize them with compile-time heuristics or after profiling.

The number of needed kinds can be reduced by establishing a convention for ordering non-locals in their environment arrays, according to whether they are pointers or not: either first all pointers then all non-pointers, or vice-versa.

The idea of ***normalizing the representation*** is a sort of pattern in the BIBOP scheme, generalizable to many other cases when using statically typed languages: there is no reason why two cases

[22] Benchmarks in section 9 show that this simple solution can be satisfying, at least up to 8 processors. ***Room synchronization*** [10] or ***work-stealing*** [5] are two more complex, and presumably more scalable alternatives.

[23] The example trivially generalizes to AVL trees, the idea being simply that the balance-related information can usefully be represented as *meta*-data rather than data.

[24] Some systems add even more than one header word per object. Sun's JDK, MMTk [4] and Microsoft's CLR, for example, use two words.

[25] And in particular when using simple compilers or interpreters: higher-order code can be simplified with flow analysis.

of different concrete types, possibly completely unconnected at a semantic label but with the same effective size and number of potential pointer fields, cannot be represented in such a way to share the same kind.

8. Lazy and object-oriented languages

Lazy languages require a slightly more sophisticated data representation than call-by-value languages, as in a realistic implementation it must be possible to destructively update a still-unevaluated thunk, and replace it with the result at the end of its computation.
Unsurprisingly, `epsilonGC` does not provide any support for changing the kind of an existing object while maintaining its identity; that could be possible *at collection time* in a moving scheme, but not with mark-sweep[26].

Any standard solution already employed by the collectors for lazy languages such as Haskell can be adopted: unfortunately some of the cleanness of the BIBOP model is lost in this case, as data (*not* metadata, for obvious reasons) needs to be tagged with at least a boolean (two in a concurrent environment: objects may be *thunks*, *in flux* or *ready*) recording the evaluation state of an object; any unused bit sequence in the payload or even the mark array entry of the object can do the job.

Accessing possibly still-to-be-evaluated objects will often require a conditional at runtime, just like in conventional implementations of lazy languages; after an object is known to be ready, BIBOP metadata can be accessed just as for eager languages.

Such a solution also necessarily requires some form of synchronization if the mutator threads are more than one: of course it is always possible to add a synchronization word in the payload, if needed.
From this point of view the situation is not different for "managed" languages such as Java, where *each* object contains a header word reserved for that purpose; yet we believe that not forcing such an expensive representation for *all* objects is preferable in the general case; the user can always implement some additional logic where needed, out of the memory management system *per se*.

For most runtimes there is no reason for keeping *one mutex per object*, and even lazy languages such as Haskell normally employ *strictness analysis* to statically recognize many cases in which laziness is not needed, and more efficient traditional representations can be safely used.

The work about *Prolific Types* cited in Section 12 is relevant for object-oriented languages.

9. Benchmark

As this benchmark is intended to stress the memory system and the garbage collector, the programs look quite different from typical parallel applications, featuring essentially no synchronization: with one exception all programs are embarrassingly parallel and perform the *same* operations on the *same* data, in parallel from different threads. In every case the computation is performed eight times: depending on how many parallel threads are employed, some of these computations proceed at the same time; for example, a run using two processors will consist in using two threads for computing the same function in parallel, all of this four times, one after another.
The one exception is also the most realistic program, a Prolog interpreter written in functional style, which is *also* run with only one mutator, in order to test the scalability of the collector itself in the worst possible situation of unbalanced load, of course in a situation where an opportunity for parallel collection exists.

Benchmark programs are allocation-intensive but they are thought to be relatively realistic use cases of functional programming, as far as synthetic benchmarks can be; in other words, they are *not* limited to allocating at the highest possible rate.

Programs are written in `nanolisp`, a simple but realistic parallel Lisp compiler which supports either Boehm's collector or `epsilonGC`. Tests are also run on `nanolisp` configured to use `epsilonGC` with kindless s-expressions, in order to intentionally lower the data density of conses and functions from $\frac{1}{2}$ to $\frac{1}{4}$. The default `epsilonGC` configuration was tested with fixed heap sizes of 512Mb and 1Gb; the kindless configuration was tested with heap sized of 1Gb and 2Gb, in order to compensate for the less efficient space utilization. Boehm's collector was left in one case in its default configuration, and in the other one both the initial and the maximum heap size were set to 5Gb.

The test machine is an Intel Xeon double quad-core SMP with 8Gb RAM.

More details are available in the Appendix.

9.1 Evaluation

The system is quite scalable, particularly when configured with a larger heap: the total completion time of parallel programs with `epsilonGC` averages a scalability of 5.56 with 8 processors in the 1Gb configuration.

The worst case are program `sort` and `clone`, the first of which is confined in a heap too small for its temporary data, while the second was explicitly designed for growing a large alive heap, producing no garbage. At least `clone` would surely have benefited form a generational collector.

Mutation time scales noticeably better than collection time, which shows the unsurprising fact that parallel collection itself (as opposed to parallel allocation) is not profitable in the presence of only linear structures or too few alive objects.

The relatively high number of collection and the consequent overhead from the signals interrupting the mutators is probably one reason for the relatively disappointing scalability with the 512Mb heap; implementing safe points would solve the problem, at the cost of some flexibility.

The sequential Prolog program shows a case where the collection itself, and marking in particular, approaches linear scalability. Of course such a performance would not be possible if the alive heap were composed of long linear structures, instead of tree-like objects.

Using n processors multiplies by n the size of the alive heap, hence it is normal that collections become more frequent when the number of processors grow.

The kindless configuration performs consistently worse than the kinded one, even if the number of collections is similar.

OProfile reports a much larger number of L2 cache misses with Boehm's collector, which may partially explain the difference in scalability and completion time. The same phenomenon is observable with the kindless configuration of `epsilonGC`, despite in a less dramatic way.

Profiling also confirms that most cache faults with `epsilonGC` happen during the mark phase rather than during mutation. Our simpler implementation of the BIBOP table as a hash table might

[26] In a moving BIBOP collector otherwise similar to `epsilonGC` it might be reasonable to split each kind into a *evaluated* kind, plus a *thunk-or-evaluated* one: all the alive evaluated objects of a thunk-or-evaluated kind would be *re-kinded* at collection time. This idea does not look particularly hard to implement, but keeping the collector both efficient and language-agnostic might be challenging. Moving-time hooks definable by the user would solve the problem, at some cost; the overhead could be reduced by allowing to re-compile the hooks *as part of* the collector, to be called as inline functions, like described for example in [28].

play an important role, compared to Boehm's more complex multi-layered tree, requiring more memory accesses.

Our load balancing strategy seems to work fine at least on eight processors, despite being quite simple to implement; however the experience of [14] shows that we may be next to hit a scalability limit.

It is interesting to note that `nanolisp` was originally a simple interpreter without tail-recursion support. Early benchmarks with that first implementation showed quite different results, as the system was not able to exploit the available memory bandwidth of the machine due to the interpretive overhead. Scalability was way better than with the compiler, but the absolute performance of the interpreter was more than an order of magnitude lower. `epsilonGC` performed slightly better than Boehm's collector, but the difference was much less marked than with the interpreter; moreover the lack of tail-recursion optimization forced us to keep the parameters small, and unrealistically stressed the root marking part of the collector.

We can at the very least conclude from the benchmark that metadata lookup using BIBOP page headers is *not slower* than using object headers.

10. Further developments

As promising as it may already look `epsilonGC` needs several enhancements, the most urgent of which are support for *finalization* and *weak pointers*; root marking can be quite easily parallelized, and the heuristic which triggers collection and resizes the heap can definitely be tuned better, and optionally left for the user to provide.

Generational collection would be of benefit in most workloads; the fact that mark arrays can already be configured as vectors of bytes or words makes them attractive to couple generation numbers to mark bits, at the granularity of single objects. A write barrier could be either implemented with `mprotect()` (for human-written programs) as with Boehm's collector in generational mode, or by providing an explicit macro or function (for compiler-generated code). On the other hand, most current *incremental* strategies tend to make pauses shorter at the price of reduced bandwidth (see Chapter 8 of [18]): while for obvious reasons non-concurrent incremental schemes do not offer any advantage with respect to bandwidth, a multi-thread strategy where mutators and collectors threads actually worked in parallel could be attractive; unfortunately the invariably required collector/mutator synchronization incurs a cost which looks difficult to offset with parallelism.

An even more radical future change might involve changing from mark-sweep to a *copying* strategy: again it could be of some use to adapt mark (word) arrays to store forwarding pointers[27], and employ atomic *compare-exchange* intrinsics for parallel lockless moving. Pages would continue to dictate the natural granularity of parallel operations.
A copying strategy typically entails accurate pointer identification for best performance at allocation time, despite *mostly copying* collection also being a possibility [3]. In order to enable this it would not be hard to (conditionally) support safe points; rather it is not clear whether the time needed to stop mutators would adversely impact scalability, or instead improve it: the answer might depend on the number of processors.

Several such strategies could be usefully combined, for example by making a two-generation system with a non-moving old generation and a copying nursery; all the possibilities would remain available at configuration time or even at run time, in the spirit of what [2] does for C++ and [11] for managed runtimes.

The possibility of explicit destroying objects could be extended to *kinded* arenas, easy to implement with pages, allowing compilers to control the creation and destruction of kinded objects with statically known lifetime — the stack cannot be used for that if metadata have to be accessed.

Even if it is definitely possible to port the implementation to other operating systems, portability is one of the several areas where we cannot hope to match Boehm's work, another one being the ability to automatically find global roots. `epsilonGC` does not do blacklisting [7], either; even if using conservative pointer finding only for tracing roots, blacklisting would likely be useful on 32-bit processors.

In order to accurately evaluate its performance on standard benchmarks it would be very interesting to adapt some mature language runtime to use `epsilonGC`.

11. Conclusions

While describing our implementation of `epsilonGC`, a new BIBOP parallel garbage collector, we have outlined a possible general architecture on which other high-performance memory management systems for shared memory machines may be based. The user-level API based on *kinds*, *sources* and *pumps* provides an alternative to current allocation interfaces, making allocation particularly efficient and scalable.

The architecture is conceived for the allocation patterns of functional programs, but fairly tunable and in fact largely independent from the language.

Benchmarks show how the BIBOP model allows for efficient implementations; in particular notwithstanding its lack of fine-tuning our implementation compares favorably to Boehm's collector in performance.

We show how the efficiency of `epsilonGC` may be due at least in part to the optimal *data density* of its data representation, despite the need for stronger experimental evidence of this correlation.

12. Related work

Many free software programming language implementations include garbage collectors, some of which with excellent performance on uniprocessors; unfortunately such implementations tend to make assumptions on the language, the compiler or other parts of the runtime, and as a consequence they are hard to disentangle, reuse and even benchmark in isolation. We don't know of any stable[28] version of such collectors making use of SMPs.

Being portable and very easy to interface to nearly any C program, the Boehm-Demers-Weiser garbage collector [7], now maintained by Hans Boehm at HP, is employed by a large number of free software projects.

Toshio Endo, Kenjiro Taura and Akinori Yonezawa of the University of Tokyo maintained SGC [14], an interesting fork of Bohem's collector exploiting large shared-memory parallel machines, from 1998 to 2003. Unfortunately their project now looks abandoned, the code does not even compile on modern systems and we have been unable to easily fix it.

[27] Forwarding pointers could also be stored in the first word of the payload in a non-concurrent setting, but only under the assumption that forwarding pointers are recognizable at runtime; this is inconvenient if the data representation is otherwise untagged.

[28] A future version of the Haskell compiler GHC will support a parallel collector [21]. As of the last released version (6.8.3 at the time of writing), GHC supports SMPs but its garbage collector is still single-threaded. See `http://hackage.haskell.org/trac/ghc/wiki/GarbageCollectorNotes`. A promising experimental branch of MLton available on Subversion at `svn://mlton.org/mlton/branches/on-20080218-parallel-runtime-branch` supports parallel collection.

Around 2001 Boehm updated his collector to optionally mark in parallel on SMPs [8, 9]. Despite being less ambitious than the Japanese project, the parallel version of his collector yields a good speedup and — even more importantly — allows to allocate in parallel from several threads without too much locking; nonetheless opportunities of further improvement remain, some of which we have explored in this work.

Qish by Basile Starynkevitch [24] is another language-agnostic garbage collector. As a copying exact collector it would be interesting for a comparison, but unfortunately it is currently sequential and does not support multi-thread mutators.

Some more recent publications [1, 15] present parallel garbage collectors for Java. [1] is particularly interesting in highlighting the relatively high performance of a simple non-generational mark-sweep schema, which is competitive with much more complex solutions. All such collectors represent metadata in object headers, which is reasonable for an object-oriented language requiring thousands of different kinds, but does not fit the functional programming use case as well. Despite its good performance and multiprocessor support Sun's OpenJDK memory subsystem shares the same problem, as it represents metadata in structures named "klasses", pointed by one of the *two* words of the header in each object [26].

Object-Oriented programs also tend to respect a variant of the functional hypothesis (called the "Prolific Hypothesis" in [23]), the difference being in the fact that the number of potential kinds may be intrinsically much higher. [23] describes (among the rest) an interesting and more complex collector organization for the JVM, also based on the idea of reducing the object header size whenever possible, but unfortunately not directly suitable for functional languages. A variant of the hybrid BIBOP/type-based collector hinted at the end of Section 4 in [23] could be implemented over `epsilonGC`.

References

[1] C. Richard Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith. A comparative evaluation of parallel garbage collector implementations. In Henry G. Dietz, editor, *Languages and Compilers for Parallel Computing, (14th LCPC'01)*, volume 2624 of *Lecture Notes in Computer Science (LNCS)*, pages 177–192. Springer-Verlag (New York), Cumberland Falls, KY, USA, August 2001. Revised Papers 2003.

[2] Giuseppe Attardi and Tito Flagella. A customisable memory management framework. Technical Report TR-94-010, International Computer Science Institute, Berkeley, 1994. Also Proceedings of the USENIX C++ Conference, Cambridge, MA, 1994.

[3] Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation, Western Research Labs, 1988.

[4] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS'04, Performance Evaluation Review (PER)*, pages 25–36, New York, NY, USA, June 2004. ACM SIGMETRICS/IFIP.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[6] Hans Boehm. Re: Unregistering the main thread. Thread on Boehm's GC public mailing list, September 2008. `http://www.hpl.hp.com/hosted/linux/mail-archives/gc/2008-September/002334.html`.

[7] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, September 1988.

[8] Hans-J. Boehm. Fast multiprocessor memory allocation and garbage collection. Technical Report HPL-2000-165, Hewlett Packard Laboratories, December 21 2000.

[9] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *PLDI*, pages 157–164, 1991.

[10] Cheng and Blelloch. A parallel, real-time garbage collector. *SPNOTICES: ACM SIGPLAN Notices*, 36, 2001.

[11] Michal Cierniak and Intel Corporation. The open runtime platform: A flexible high-performance managed runtime environment. *Intel Technology Journal*, 7:5–18, 2003.

[12] Ulrich Drepper. What every programmer should know about memory. Technical report, RedHat, November 2007.

[13] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible, and efficient storage management for dynamically-typed languages. Technical Report TR 400, Indiana University, Computer Science Department, March 1994.

[14] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *High Performance Computing and Networking (SC'97)*, 1997.

[15] Christine H. Flood, David Detlefs, Nir Shavit, and Xiolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[16] Free Software Foundation. The GNU Project. Official web page.

[17] R. Halstead. Implementation of multilisp: Lisp on a multiprocessor. In *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[18] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, pub-WILEY:adr, 1996. Reprinted in 1999 with improved index, and corrected errata.

[19] E. Ulrich Kriegel. A conservative garbage collector for an eulisp to ASM/C compiler. OOPSLA 1993 Workshop on Memory Management and Garbage Collection, September 1993.

[20] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. *The GNU C Library Reference Manual*. GNU Press, 2006.

[21] Simon Marlow, Tim Harris, Roshan James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In Steve Blackburn, editor, *ISMM'07 Proceedings of the Sixth International Symposium on Memory Management*, pages 11–20, Tucson, AZ, June 2008. ACM Press.

[22] Luca Saiu. The epsilon project — a functional language implementation *(MD thesis)*, February 14 2007.

[23] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. *ACM SIGPLAN Notices*, 37(1):295–306, January 2002.

[24] Basile Starynkevitch. Qish introduction, 2005. `http://starynkevitch.net/Basile/qishintro.html`.

[25] Guy Lewis Steele. Data representations in PDP-10 MACLISP. Report A. I. MEMO 420, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, 1977.

[26] Sun Microsystems, Inc. HotSpot Glossary of Terms, 2008. `http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html`.

[27] Herb Sutter. The free lunch is over: a fundamental turn toward toward concurrency. *Dr. Dobb's Journal*, March 2005.

[28] Derek White and Alex Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, December 1998.

[29] F. A. Williams. Handling identifiers as internal symbols in language processors. *Communications of the ACM*, 2(6), June 1959.

Appendix — Benchmark data

Benchmark programs are written in *nanolisp*, a simple and by itself quite uninteresting compiler of a non-standard Lisp dialect similar to Scheme with a `future` construct [17]. The runtime of `nanolisp` was developed in C, as a prototype for `epsilon`'s runtime.

`nanolisp` is properly tail recursive (when the generated code is compiled by GCC). Environment data structures are created on the stack when possible[29]. `nanolisp`'s S-expressions are efficient: they use in-word tags for unboxed objects, and when the interpreter is configured for using `epsilonGC` it makes use of kind metadata for boxed objects; however, relying on conditional compilation and macros, `nanolisp` also supports Boehm's collector; in such configuration it represents boxed objects with a one-word header containing type information. `nanolisp` provides an opportunity to compare the two systems.

In an attempt to test the data density hypothesis we also developed a modified configuration of `nanolisp` which runs with `epsilonGC`, but using *one-word object headers* for the type tag of all unboxed objects instead of exploiting `epsilonGC`'s metadata features: conses and closures become kindless in this configuration, and both their data densities drop from $\frac{1}{2}$ to $\frac{1}{4}$; the intention is to make a comparison with Boehm's collector more fair.

As the following benchmark programs are intended to stress the memory system rather than `nanolisp`, they look quite different from typical parallel computation applications, featuring essentially no synchronization: all programs are embarrassingly parallel and perform the *same* operations on the *same* data, in parallel from different threads. In every case the computation is performed eight times: depending on how many parallel threads are employed, some of these computations proceed at the same time; for example, a run using two processors will consist in using two threads for computing the same function in parallel, all of this four times, one after another.

Benchmark programs are allocation-intensive but they are thought to be relatively realistic use cases of functional programming, as far as synthetic benchmarks can be; in other words, they are not limited to allocating at the highest possible rate.

The `clone` benchmark program consists of a function which recursively creates a modified copy of a given S-expression in which all leaves are replaced with deep copies of the whole original structure; this function is applied to a complete binary tree of depth 11, 10 times.

`fact` computes 500 times the factorial of 9, representing natural numbers in radix 1 as lists containing the symbol `foo`. Arithmetic functions are defined recursively as functions on lists.

`fibo` computes Fibonacci's 33^{th} number using the exponential recursive definition. Again, natural numbers are implemented as lists.

`sort` uses quick-sort to sort a random list of 600000 integers. Several functions use tail-recursive helpers which create garbage lists.

`prolog` is a Prolog interpreter written in functional style which computes all the permutations of a list of length 9 for the sequential version, 8 for the parallel version.

Even the synthetic benchmark programs are intended to simulate the dynamic behavior of typical functional programs, generating a lot of garbage and some alive linear memory structures, of a size varying according to the program. The Prolog interpreter is a more realistic symbolic application.

In order to force `epsilonGC` to collect more often on the test machine. we have set the heap to have relatively small fixed sizes. Apart from this `epsilonGC` has been left with its default settings.

Boehm's collector resizes the heap in an incremental way, which is clearly visible by the number of collections. Setting the environment variables `GC_INITIAL_HEAP_SIZE` and `GC_MAXIMUM_HEAP_SIZE` to 5Gb tends to improves the performance.

The test machine is a Dell Precision T7400 with two quad-core Intel Xeon (EM64T) chips at 3GHz, 8Gb of RAM and 1.3Ghz FSB. L1I and L1D are 32K per core, 8-way set-associative. L2s are 6Mb, 24-way set-associative, one per *pair* of cores; 64-byte cache lines at both levels. The operating system is a modified debian "unstable" GNU/Linux distribution with Linux 2.6.26 and GNU libc 2.7; all the relevant software was compiled in 64-bit mode with GCC 4.3.2, using the options `-O3 -fomit-frame-pointer` for production.

Boehm's collector is version *7.1alpha3-080224*, configured with the options `--enable-threads=posix --disable-cplusplus --enable-parallel-mark --disable-gcj-support --disable-java-finalization`, and run in non-incremental mode.

Tests were run in single-user mode, with the machine completely unloaded. OProfile was *not* on during the benchmarking runs, in order to avoid its overhead in measurements. L2 misses were estimated with the attribute `LLC_MISSES` at resolution 6000.

The complete source code of `epsilonGC`, `nanolisp` and the benchmark programs is available (at the moment for reviewers only) at the address `http://194.254.173.145/repos/benchmarks`.

[29] `lambda` and `future` create closures with environments implemented as flat heap arrays containing only the free variables which are actually used in the body; `let` and `let*` generate local C variables. The non-local environment is referred via a *static link* and only contains bindings for the non-locals which are actually bound in the body. No identifier lookup is performed during execution, as the static scoping rule allows to resolve any lexical identifier reference at compile time. Calling a function does *not* allocate heap structures. Mutable cells (in the style of ML's `refs`) must be allocated on the heap.

		epsilonGC (512 Mb)							epsilonGC (1024 Mb)						
program	CPUs	mutation	roots	marking	sweeping	completion	collections#	misses	mutation	roots	marking	sweeping	completion	coll.#	misses
clone	1	19.30	0.01	2.00	0.15	22.92	11	37.28	19.44	0.00	0.91	0.00	21.32	5	37.13
clone	2	9.88	0.01	2.39	0.10	12.62	13	75.33	10.06	0.01	0.56	0.17	10.95	6	83.04
clone	4	5.12	0.02	2.42	0.06	7.76	15	192.08	5.07	0.01	1.03	0.00	6.27	6	224.08
clone	8	2.80	0.03	2.47	0.00	5.42	19	380.00	2.77	0.01	0.92	0.00	3.75	6	416.71
list-fact	1	175.42	0.09	13.27	0.20	196.81	128	47.47	175.04	0.04	6.18	0.23	184.96	61	48.55
list-fact	2	88.32	0.11	9.15	0.17	98.61	131	99.02	87.96	0.05	4.56	0.22	93.24	62	101.74
list-fact	4	46.06	0.12	8.22	0.10	54.78	139	260.44	46.13	0.06	3.79	0.16	50.28	64	274.94
list-fact	8	25.75	0.15	8.42	0.02	34.54	156	464.36	25.59	0.07	3.51	0.01	29.29	66	518.19
list-fibo	1	34.28	0.01	10.31	0.00	47.97	13	20.92	34.02	0.00	3.49	0.00	39.22	4	20.38
list-fibo	2	17.48	0.02	9.30	0.09	27.64	18	47.52	17.30	0.00	1.73	0.00	19.19	4	41.56
list-fibo	4	9.17	0.03	7.15	0.07	16.62	22	94.12	8.56	0.01	1.43	0.16	10.35	5	118.79
list-fibo	8	4.71	0.06	10.34	0.00	15.85	48	219.77	4.47	0.01	0.90	0.00	5.48	5	201.06
prolog-par	1	36.10	0.02	27.87	0.74	70.85	20	266.16	35.01	0.01	8.04	0.91	48.21	7	130.66
prolog-par	2	18.17	0.04	15.69	0.56	35.22	22	568.04	17.52	0.02	5.91	1.15	24.99	10	339.65
prolog-par	4	9.43	0.07	8.38	0.31	18.51	22	1128.34	8.92	0.03	2.95	0.66	12.90	9	719.63
prolog-par	8	4.88	0.14	5.48	0.43	11.12	24	2343.64	4.61	0.03	0.91	0.04	5.71	5	1061.62
prolog-seq	1	45.53	0.05	65.62	0.14	120.94	31	344.46	42.46	0.01	9.65	0.05	53.66	6	125.28
prolog-seq	2	45.35	0.05	32.97	0.07	79.04	31	496.20	42.53	0.01	4.86	0.04	47.55	6	140.03
prolog-seq	4	45.35	0.05	16.73	0.13	62.61	31	620.76	42.46	0.01	2.45	0.05	45.13	6	146.17
prolog-seq	8	45.46	0.05	9.13	0.13	54.95	31	715.85	42.50	0.01	1.34	0.04	44.06	6	152.92
sort	1	10.71	0.01	2.20	0.00	14.64	8	66.79	10.84	0.00	1.36	0.00	12.91	4	68.51
sort	2	6.06	0.01	2.02	0.00	8.48	10	136.15	5.62	0.00	0.40	0.00	6.17	3	142.06
sort	4	3.08	0.01	1.84	0.00	5.03	11	287.17	3.04	0.01	0.78	0.00	3.90	4	321.27
sort	8	1.71	0.02	1.31	0.00	3.08	12	610.00	1.88	0.01	0.57	0.00	2.54	4	681.85

		epsilonGC kindless (1024 Mb)							epsilonGC kindless (2048 Mb)						
program	CPUs	mutation	roots	marking	sweeping	completion	collections#	misses	mutation	roots	marking	sweeping	completion	collections#	misses
clone	1	21.09	0.01	3.31	0.00	26.49	11	54.83	21.09	0.00	1.59	0.00	23.79	5	62.55
clone	2	11.05	0.01	3.10	0.00	14.41	12	119.87	11.10	0.00	1.24	0.00	12.49	5	130.90
clone	4	5.94	0.01	2.09	0.00	8.20	13	271.21	6.23	0.01	0.97	0.00	7.37	5	342.78
clone	8	3.34	0.03	3.32	0.00	7.00	19	523.19	3.44	0.01	1.10	0.00	4.66	6	640.71
list-fact	1	215.63	0.09	14.50	0.00	239.25	126	86.92	215.45	0.04	6.87	0.00	226.73	60	89.15
list-fact	2	108.72	0.10	11.27	0.00	121.38	129	179.20	108.41	0.05	5.17	0.00	114.24	61	181.65
list-fact	4	61.80	0.11	8.92	0.01	71.00	134	410.70	61.29	0.06	4.04	0.00	65.58	62	425.93
list-fact	8	40.73	0.15	8.86	0.01	49.97	148	652.39	40.85	0.06	3.87	0.01	44.95	65	677.34
list-fibo	1	35.31	0.01	7.91	0.00	45.68	11	36.30	35.14	0.00	3.03	0.00	39.92	4	38.60
list-fibo	2	18.04	0.01	4.35	0.00	22.89	11	74.17	18.00	0.00	1.79	0.00	19.93	4	79.39
list-fibo	4	9.12	0.01	3.32	0.00	12.61	12	179.19	9.26	0.00	1.08	0.00	10.46	4	178.63
list-fibo	8	5.16	0.05	8.50	0.00	14.45	35	359.29	5.00	0.01	1.15	0.00	6.25	5	384.76
prolog-par	1	39.59	0.02	29.38	0.03	73.07	15	351.68	37.69	0.01	9.24	0.01	48.61	5	181.91
prolog-par	2	20.35	0.03	14.66	0.03	35.29	15	712.92	19.42	0.01	4.63	0.01	24.19	5	371.31
prolog-par	4	10.52	0.05	8.30	0.06	19.16	16	1419.77	9.96	0.02	2.40	0.01	12.58	5	745.56
prolog-par	8	5.57	0.09	5.28	0.35	11.56	18	2670.45	5.11	0.03	1.38	0.07	6.76	5	1431.73
prolog-seq	1	49.61	0.04	75.79	0.07	133.76	27	455.56	45.44	0.01	13.54	0.02	60.60	6	204.96
prolog-seq	2	49.33	0.04	37.95	0.04	88.44	27	657.81	45.59	0.01	6.77	0.01	52.52	6	223.17
prolog-seq	4	49.39	0.04	19.71	0.06	69.46	27	841.50	45.72	0.01	3.48	0.01	49.41	6	248.06
prolog-seq	8	49.40	0.04	11.18	0.06	61.26	27	975.03	45.44	0.01	1.95	0.02	47.65	6	255.32
sort	1	13.60	0.01	2.31	0.00	17.71	8	117.28	13.83	0.00	1.62	0.00	16.58	4	130.17
sort	2	6.79	0.01	1.67	0.00	8.68	8	258.26	7.00	0.00	0.58	0.00	7.80	3	251.92
sort	4	3.98	0.01	1.69	0.00	5.84	10	579.93	3.87	0.00	0.59	0.00	4.59	4	585.33
sort	8	2.63	0.02	2.92	0.00	5.70	16	1362.57	2.63	0.01	0.75	0.00	3.51	4	1162.82

Figure 2. Times in seconds, number of collections and cache misses for each benchmark program with epsilonGC. The "misses" columns is *an index* of the number of L2 cache faults in a unit of time. It is obtained by dividing the number of samples reported by OProfile by the program completion time. The sweep column indicates only the non-deferred part of the sweeping phase: on-demand sweep times are considered as part of mutation. All the reported times are *total times*, not relative to a single collection.

		Boehm (default)			Boehm (5Gb Mb)		
program	CPUs	completion	collections#	misses	completion	collections#	misses
clone	1	36.16	54	873.30	33.86	27	805.23
clone	2	23.21	35	1624.39	23.17	25	1594.23
clone	4	15.21	33	2951.18	14.33	16	2986.56
clone	8	20.40	24	3357.54	20.90	12	3352.52
list-fact	1	324.46	1907	592.69	295.44	190	601.72
list-fact	2	241.41	1316	1333.37	210.07	199	1229.96
list-fact	4	162.65	1027	2284.49	133.40	214	2137.15
list-fact	8	258.52	535	2771.93	238.86	251	2773.76
list-fibo	1	44.67	70	381.72	42.80	15	302.84
list-fibo	2	26.10	61	834.78	25.10	19	732.25
list-fibo	4	15.99	55	1543.68	15.14	17	1474.16
list-fibo	8	14.87	52	2674.62	13.99	16	2707.37
prolog-par	1	54.12	143	891.81	45.40	14	371.64
prolog-par	2	34.66	118	2033.85	27.86	16	1057.73
prolog-par	4	23.14	108	3955.62	19.74	19	2778.06
prolog-par	8	30.64	107	5451.13	27.02	21	4657.34
prolog-seq	1	84.59	111	1595.69	77.58	24	1396.84
prolog-seq	2	73.46	111	1971.13	69.45	24	1682.54
prolog-seq	4	67.75	111	2163.63	65.86	24	1798.50
prolog-seq	8	67.40	111	2250.92	65.17	24	1883.80
sort	1	23.38	85	906.37	19.26	14	685.55
sort	2	17.02	53	1705.43	13.90	14	1439.38
sort	4	12.04	39	3022.95	10.72	20	2637.71
sort	8	17.72	32	3393.33	16.97	17	3427.09

Figure 3. Times with Boehm's collector