Appears in 1998 International Symposium on Memory Management

Very Concurrent Mark-&-Sweep Garbage Collection without Fine-Grain Synchronization

Lorenz Huelsbergen
lorenz@research.bell-labs.com

Phil Winterbottom
philw@plan9.bell-labs.com

Bell Labs
Lucent Technologies
Murray Hill, NJ, 07974, USA

Abstract

We describe a new incremental algorithm for the concurrent reclamation of a program's allocated, yet unreachable, data. Our algorithm is a variant of mark-&-sweep collection that—unlike prior designs—runs mutator, marker, and sweeper threads concurrently *without* explicit fine-grain synchronization on shared-memory multiprocessors. A global, but infrequent, synchronization coordinates the per-object coloring marks used by the three threads; fine-grain synchronization is achieved without locking via the basic memory consistency guarantees commonly provided by multiprocessor hardware. We have implemented two versions of this algorithm (called VCGC): in the Inferno operating system and in the SML/NJ ML compiler. Measurements, compared to a sequential generational collector, indicate that VCGC can substantially reduce worst-case pause latencies as well as reduce overall memory usage. We remark that the degrees of freedom on the rates of marking and sweeping enable exploration of a range of resource tradeoffs, but makes "optimal" tuning for even a small set of applications difficult.

1 Introduction

Garbage collection—the automatic reclamation of a program's spent and unreachable storage—is a valuable systems implementation technique. By automatically identifying accessible and hence potentially in-use data, garbage collection (GC) shoulders the error-prone task of memory allocation and deallocation for the programmer. In doing so, GC can improve code quality and programmer productivity.

In addition to the benefit of providing a high degree of safety at the language level, garbage collection becomes vital in systems where independent, perhaps untrusted, programs must efficiently coexist in a shared address space. In a distributed system, for example, code can migrate between compute nodes and may need to cross security membranes [10, 15]. An imported piece of code may need to share storage with other imported code; it is the operating system's responsibility to recover such storage only when no programs—local or remote—have live pointers into it. In a system with GC, programs cannot generate a pointer to data they do not own; this eliminates a large class of security problems without the imposition (by the hardware and OS) of memory protection.

Garbage collection, however, incurs costs that manifest themselves as combinations of increased memory usage, runtime overheads on data accesses, and long latencies that disrupt a program's execution. The requirements of large memory and fast processor render many current GC implementations unusable on small clients (*e.g.*, mobile communicators); long collection latencies impair deployment of GC in programs that require some degree of "real time" operation (*e.g.*, communications protocols such as Fox [5] or interactive window systems). Conventional stop-&-copy collectors [9, 22], for example, require substantial additional storage for making copies of live data. Even high-performance generational collectors [25, 27, 18] periodically siphon *all* data from memory through the processor and into another area in memory. Mark-&-sweep collectors (*e.g.*, [19, 7, 4]), on the other hand, have the disadvantage that they must continually step through the entire set of free and in-use program objects.

To address garbage collection's inherent costs, collector designers incorporate *concurrency* into their designs (*e.g.*, [4, 2, 23, 12, 11, 26, 7]) both to interleave computation with collection (thereby reducing GC latencies) and to delegate the task of GC to additional processors in a parallel machine (thereby standing to improve overall performance). Existing concurrent GC algorithms loosely fall into one of two classes: variations on mark-&-sweep collection (*e.g.*, [7, 26]) and incremental stop-&-copy collectors (*e.g.*, [11, 2, 23, 12]). Practical mark-&-sweep designs to date do not expose maximal concurrency: sweeping must strictly follow marking, they cannot overlap in time. (Lamport [17] and Queinnec, *et al.* [24] describe approaches that *do* overlap marking and sweeping. As we discuss (§3), their approaches are impractical due to quadratic-time marking and fine-grain mutator–marker synchronization.) Incremental stop-&-copy collectors, on the other hand, still require the large memories inherent in generational collection.

This paper's algorithm—called Very Concurrent Garbage Collection (VCGC)—also uses concurrency to offset GC's costs, but does so with a variation on the mark-&-sweep algorithm that, unlike its predecessors, has both of the following properties:

1. mutation, marking, and sweeping occur in parallel as three separate threads; and
2. no synchronization is required between the mutator, marker, and sweep threads.

By "no synchronization" we mean no locks, critical sections, fetch-and-add primitives, *etc.* We assume only that the computer's memory system performs machine-word writes atomically; that is, if location M holds x before y is written to M, reads of M at any point in the computation and from any processor will return either x or y, but never an "intermediate" value (such as the high bits of x and the low bits of y). Contemporary multiprocessor computers provide this form of memory atomicity.

Our algorithm does require infrequent global barrier synchronization events to demarcate the algorithm's phases—experiments indicate that such events are separated in time by many millions of instructions. Atomic machine-word writes, and infrequent barriers, suffice to synchronize the VCGC algorithm for multiprocessor operation. Extension of the algorithm to multiple mutator threads necessitates synchronization on allocation which may be avoided in part by allocating from multiple lists or in multiple arenas simultaneously (§2.4 and §4.2).

VCGC, as do other concurrent collectors [28, 13], requires a write (or read) barrier as a mechanism with which the mutator communicates data-graph changes to the collector. Our write barrier is asynchronous (non-blocking) and requires no inter-processor coordination aside from the aforementioned memory atomicity. We provide C source for this barrier in an appendix.

We have built two VCGC implementations to demonstrate that this algorithm is viable in real systems. The systems differ radically in the characteristics (size, lifetime) of the data they construct. This indicates that VCGC may be applicable to a wide range of systems requiring automatic storage management. Since marking and sweeping occur concurrently, their respective rates may be adjusted to "tune" memory usage, pause latency, and execution speed. Our first implementation is in Inferno [15]. Inferno is a distributed operating system that supports mobile code. Storage is collected by a reference-counting collection scheme augmented with VCGC; reference counting assures the instant reclamation of large objects (*e.g.*, bitmaps) and VCGC reclaims the cyclic structures that elude reference counting. Via VCGC, Inferno is able to overlap lengthy transactions (I/O and communication) with collection. For example, user input in Inferno occurs concurrently with GC.

Our second implementation is in the Standard ML of New Jersey compiler [3] where we replace SML/NJ's multi-generational stop-&-copy collector [25] with VCGC. This VCGC implementation uses an allocation arena (essentially a zero-th generation) as a buffer; most objects do not survive early generations [18] and hence their allocation from a storage list can be avoided. Empirical comparison, on a uniprocessor, of VCGC and SML/NJ's generational collector consistently finds large reductions (3X–7X) in the length of maximum GC pauses with VCGC. Maximum VCGC pauses are in the *tens* of milliseconds versus *hundreds* for stop-&-copy generational collection. We also find substantial reduction in overall memory usage—over 35% for one application. The dramatic improvements in pause and memory performance, however, come at the price of increased execution times. We argue that VCGC's performance in SML/NJ is comparable to that of traditional mark and sweep collectors [19, 7, 4] and that further speed can be gotten by execution on a multiprocessor.

The main contribution of this paper is the VCGC algorithm; it is described in the next section. We discuss related literature in Section 3. Section 4 describes the two VCGC implementations and presents results.

2 VCGC Algorithm

Here we describe the Very Concurrent Garbage Collection algorithm. We first review the conventional free-list allocation that underlies VCGC's allocator. Second, we give the VCGC algorithm for a purely functional mutator, that is, for programs without mutable state (references). In Section 2.3, we then extend the core algorithm to admit mutation via references. Finally, we describe how this algorithm can support multiple mutator threads (§2.4).

2.1 Free-list Allocation

VCGC requires a conventional free-list allocator that conceptually works as follows. (Implementations can avoid much of the overhead that general free-list allocators incur; Section 4 describes the allocator optimizations we implemented.) Upon initialization, a free-list allocator parcels the memory available for free-list allocation into one or more initially free *blocks.*[1] Free blocks are linked to form the *free list.* A request for storage (by the mutator) scans the free list for a block that fulfills the request. In the simplest schemes[2], upon finding the first such block, the allocator removes it from the free list and hands it to the requester. When a block is identified (by the sweeper) as no longer in use, it is again placed on the free list. We require that it be possible to find all blocks in the system and determine, for a given block, whether or not it is currently allocated.

2.2 Functional Algorithm

Figure 1 contains the functional VCGC algorithm. VCGC operates in *epochs.* The *current epoch* is the epoch denoted by the integer variable `epoch` (line a).[3] An epoch creates three concurrent threads (lines e,f,g) for the mutator, marker, and sweeper. We describe them separately below. An epoch i maps to one of three colors by the function:

$$\texttt{COLOR(i)} \equiv \texttt{i}\ \textbf{mod}\ \texttt{3}$$

All data (allocated objects) that are in use (*i.e.*, not on the free list) carry one of the three colors.

2.2.1 Mutator

The mutator is the application program. It is parameterized by the color of the current epoch, called the *mutator color* (line e in Figure 1). It allocates data by requesting space from the free-list allocator (§2.1). Data thus allocated are colored with the mutator color given by `COLOR(epoch)`. Only when no free block is available need the mutator wait for the sweeper to reclaim one. Objects with mutator color must be retained at least through the end of the current epoch.

There are no data races with the marker or sweeper since the mutator is "color blind." It only tags newly allocated data with the current epoch's color; it does not examine colors as it traverses data.

Mutator-sweeper synchronization on free-list accesses can be avoided by maintaining two free-list pointers—one for

[1] The terms "block" and "object" may be read interchangeably. We use the term "block" for storage with content anonymous to the operation being described.

[2] Extensions to this allocation scheme can select blocks based on best fit rather than first fit, coalesce adjacent free blocks, *etc.* See Knuth [14] for further details.

[3] The first epoch is numbered "2" to avoid "negative" epochs in color computations. The epoch variable is a "`big int`" that does not wrap. This declaration is solely expository since only the three most recent epochs (including the current one) need be distinguished; two bits of state suffice for this in practice.

```
(a)   big int epoch = 2;
(b)   root_set_t roots = {};
(c)   thread_t mutator, marker, sweeper;

(d)   forever {
(e)       mutator ← make_thread mutate(COLOR(epoch));
(f)       marker ← make_thread mark(roots, COLOR(epoch));
(g)       sweeper ← make_thread sweep(COLOR(epoch-2));
(h)       barrier_sync {marker, sweeper};
(i)       /* invariant:  all reachable data have COLOR(epoch) */
(j)       suspend_thread mutator;
(k)       roots ← get_roots(mutator);
(l)       delete_threads {mutator, marker, sweeper};
(m)       epoch++;
      }
```

Figure 1: The VCGC algorithm operates as a series of epochs. An epoch concurrently (1) runs the mutator, (2) marks, with the current epoch's color (`COLOR(epoch)`), all data that were reachable in the previous epoch, and (3) reclaims any data marked with `COLOR(epoch-2)`, the mutator color of two epochs ago. The function `COLOR(i)` is defined as `(i` **mod** `3)`.

mutator allocation and the other for sweeper reclamation. Thus, this producer-consumer handoff can occur without explicit synchronization. (Implementation details of such a mechanism are in Section 4.2.1.)

2.2.2 Marker

The marker thread is also parameterized by the current epoch's mutator color (line f in Figure 1). It is responsible for bringing reachable data from the last epoch "up to date"; these are data the mutator can reach and hence incorporate into new data structures. Once initialized by the mutator, *only the marker may alter an object's color.* This is central in allowing us to dispense with all fine-grain synchronization. The marker recursively traverses data reachable from the mutator's *root set*[4] of the previous epoch. This root set is copied from the mutator at the end of an epoch (line k) for use by the marker in the next epoch.

The marker can reach data that have either the previous epoch's color (`COLOR(epoch-1)`, called the *marker color*) or the current epoch's color (`COLOR(epoch)`, the mutator color). For each datum d encountered, the marker therefore does one of two things. If the color of d is the marker color, d's color is changed to the mutator color and data reachable from d are recursively marked in this fashion. Otherwise, if d's color is the mutator color, d is not changed or further examined. In this latter case, d was—during this epoch—previously examined by the mutator and d's descendants are known to have been marked.[5]

When the marker completes, the invariant that "all reachable data have the mutator color" (line i) holds. Note that in the current epoch the marker will never encounter data with `COLOR(epoch-2)` because at the end of the previous mark phase (and epoch) all reachable data must have had `COLOR(epoch-1)`.

There is no mutator-marker race since the marker's only visible side effect is to change colors on objects tagged with the previous epoch's color. As noted above, the mutator is oblivious to colors.

[4]All live data is reachable from the *root set*. This set usually consists of program registers and stack entries.

[5]With extension to references (§2.3), data with the mutator's color could also have been allocated in the current epoch.

2.2.3 Sweeper

The sweeper thread is parameterized by `COLOR(epoch-2)`, called the *sweeper color* (line g in Figure 1). It examines every block of storage in the system. If block b has sweeper color it may be deallocated and returned to the free list; reclamation of b is safe because neither the mutator nor marker can reach it. If b has `COLOR(epoch)` or `COLOR(epoch-1)`, it is skipped because it is still potentially in use. (Blocks with marker color (`COLOR(epoch-1)`) will either be marked with `COLOR(epoch)` during this epoch if reachable from the root set or their color remains unchanged, indicating that they have become garbage. Garbage thus identified will be detected by the sweeper in the next epoch.) It is simple matter for an implementation to make reclaimed free lists available for mutator allocation without explicit synchronization (§4).

There is no sweeper-mutator race and no sweeper-marker race since the sweeper reads colors but does not write them. The only color that may change from "underneath" the sweeper is a marker color changing to the mutator color (recall, only the marker may change an object's color). Marker and mutator colors are, however, ignored by the sweeper; the atomicity assumption (§1) constrains the values read to valid colors. The sweeper only modifies (reclaims) blocks with `COLOR(epoch-2)`, which are inaccessible to the mutator and marker.

Note that the interpretation of a concrete color changes as the epoch changes. During epoch i, for example, all data tagged with `COLOR(i)` are live because they were allocated by the mutator in this epoch or were marked by the marker with this color since they were reachable from the last root set. Data tagged with `COLOR(i-1)` may or may not be encountered before epoch end by the marker and marked with `COLOR(i)` depending on whether or not they are reachable from the root set. Data tagged with `COLOR(i-2)` will be reclaimed by the sweeper during epoch i; such data were not marked in epoch $i - 1$ since they were unreachable to the marker, and hence to the mutator.

When the sweeper and marker threads have completed their traversal of the system's blocks and of the program's live data respectively, they rendezvous at a barrier synchronization (line h). This barrier is the only synchronization in the algorithm and occurs very infrequently (only after complete marker/sweeper passes). All writes by all threads must complete before the barrier exits to ensure that all marker recolorings are visible to the next epoch's sweeper. At this point the mutator thread is suspended and the mutator's root set is copied and retained for marking in the next epoch (line k). Thread state is reset (line l) and the next epoch entered (line m).

2.3 Extension to References

State, in the form of mutable references, poses problems for concurrent[6] garbage collectors since a subgraph G' of the dynamic data graph G may become temporarily disconnected from G. Since a collector's marking phase traverses G, subgraph G' may now not be encountered and properly marked. Yet G' may later be reconnected to G in a place already visited by the marker. In this scenario G' has become inaccessible to the marker before being marked. This can result in G''s data being incorrectly reclaimed while still live.

For VCGC, the reference problem manifests itself as follows. During epoch i, the mutator fetches the content v of reference cell r. It retains v (perhaps in a register) before updating r with v'. That is, the mutator redirects r's link from pointing to v to pointing to v'. Data reachable from v is still live, but potentially inaccessible to the marker since the mutator may have altered the graph reachable from the roots. That is, r may have held the only reference to v and the marker may not have reached and marked v (via r) before the mutator replaced it with v'. The solution to this problem is to require the mutator to communicate the replaced content v as a root to the concurrently running marker thread.[7] In particular, the marker must process this new root before epoch $i+1$ can commence. In Wilson's classification [28], VCGC is a *snapshot-at-the-beginning* algorithm since it retains the data reachable from the roots at the beginning of an epoch into the next epoch.

Note that unlike conventional concurrent mark-&-sweep algorithms [7] and including designs that mark during sweep [17, 24], VCGC requires that the replaced object—and not the target of the redirection—be marked (*cf.* Dijkstra, *et al.*'s `shade` operation [7]).

Figure 2 is the functional VCGC algorithm extended to support reference update. Before an update to a reference, the mutator places the current content of the reference in a *store set*. The marker marks elements in the store set in the same manner as it marks roots in the root set (§2.2.2). The difference is that elements may appear in the store set *asynchronously* during an epoch. The `while` loop handles the (expectedly rare) case of marker completion followed by a mutator store-set insertion immediately before mutator suspension. Note that the sweeper is complete *before* the `while` loop and need not be restarted in the loop body because neither mutator nor marker can generate this epoch's sweeper color.

To circumvent all fine-grain mutator/collector synchronization in VCGC, the store set (write barrier) is implemented as a non-blocking *store list*. (A C implementation of such a store list is in the appendix.) During an epoch the mutator builds a linked list of store roots, with new store roots inserted at the head of the list. The marker traverses this list until it revisits a position in the list it has previously visited. The marker sees the store list as "empty" when it contains no elements or when the marker's store-list traversal ends at the head of the list. Of course the store set may become non-empty as the mutator inserts additional roots. When an epoch completes, the store list is reset to the physically empty list. The appendix provides further detail.

2.4 Multiple Mutators

VCGC straightforwardly admits multiple mutators. Each mutator thread allocates objects of mutator color and maintains a separate store set for its references. The marker must now examine multiple store sets instead of one. If allocation is from a single free list, synchronization amongst mutators is required on allocation. This synchronization can be avoided by providing multiple free lists or by buffering allocation in per-mutator allocation arenas.

3 Related Work

The VCGC algorithm of this paper is one in a long line of mark-&-sweep algorithms starting with McCarthy's [19]. Here we describe how VCGC differs from prior concurrent mark-&-sweep algorithms. In particular, we contrast VCGC with Dijkstra, *et al.*'s collector [7] and with Lamport's [17] and Queinnec, *et al.*'s [24] "mark-DURING-sweep" (MDS) extensions thereof. The MDS collectors partially attain our goal—concurrent marking and sweeping—but differ fundamentally in design, which impacts their practical implementation. We also briefly describe other relevant approaches to concurrent garbage collection. Jones and Lins' book [13] and Wilson's survey [28] provide further details and context.

3.1 Dijkstra, *et al.*'s On-the-Fly Collector

Dijkstra, *et al.*'s incremental collector [7] introduced the well known *tricolor abstraction* (white, gray, black) to reason about mutator–collector interaction. VCGC too uses three colors (mutator, marker, and sweeper), but does so in a significantly different manner. With the tricolor abstraction, objects in the root set are first grayed at the beginning of a collection phase. A gray object x is marked black by the marker, and x's children grayed. As they are generated, gray objects are queued for marking. Objects that retain the white mark are garbage upon marker completion. At this point, interpretation of white and black is reversed, and the next collection phase initiated. Note that the interpretation of gray does not change during a phase transition with the tricolor abstraction. Furthermore, in Dijkstra, *et al.*'s collector, sweeping must strictly *follow* marking since a white object's color may change (to gray and later to black) at any point in the mark phase.

VCGC, in contrast, allows interleaving of marking and sweeping. We can (unconventionally) cast VCGC using the tricolors as follows. During an epoch the VCGC sweeper reclaims white objects. The mutator allocates black objects. The marker blackens gray objects. Note that—unlike standard tricolor marking—the marker does not encounter white objects and hence never grays objects. When the mutator modifies the data graph, it communicates with the mutator to ensure that the overwritten object will be blackened in the current epoch. At the end of an epoch, colors are reinterpreted: black is remapped as gray, gray remapped

[6] References pose problems in conventional stop-&-copy collectors as well since pointers from older to younger generations (due to reference updates) may introduce additional roots.

[7] An optimization is possible when v's color is the mutator color: v need not be communicated as a root since it and the data reachable from it have been, or will be, encountered by the marker.

```
big int epoch = 2;
root_set_t roots = {};
root_set_t stores = {};
thread_t mutator, marker, sweeper;

forever {
    mutator ← make_thread mutate(stores, COLOR(epoch));
    marker ← make_thread mark(stores, roots, COLOR(epoch));
    sweeper ← make_thread sweep(COLOR(epoch-2));
    barrier_sync {marker, sweeper};
    suspend_thread mutator;
    while (stores ≠ ∅) {
        resume_threads {mutator, marker};
        barrier_sync {marker};
        suspend_thread mutator;
    }
    /* invariant:  all reachable data has COLOR(epoch) */
    roots ← get_roots(mutator);
    delete_threads {mutator, marker, sweeper};
    epoch++;
}
```

Figure 2: VCGC algorithm for mutable references.

as white, and white remapped as black. When the epochs change, gray objects—objects not reachable by the marker and hence not promoted to black—become white by virtue of this color remapping. It is not useful to state the conventional invariant that the mutator may not create a pointer from a black object to a white one because the mutator never sees white objects.

3.2 Mark-DURING-Sweep Collectors

VCGC pipelines mark and sweep phases in a manner similar to collector designs of Lamport [17] and Queinnec, *et al.* [24]. Both prior "mark-DURING-sweep" collectors were derived from Dijkstra, *et al.*'s algorithm [7]. Extra complexity (five colors) allows them to find the same concurrency as VCGC. However, both MDS collectors use Dijkstra's atomic `shade` operation in both mutator and marker. (Queinnec, *et al.* require it in their sweeper as well.) That is, existing MDS collectors require frequent inter-processor synchronization amongst marker and mutator during typical computation because either may concurrently modify a color visible to the other. VCGC requires no fine-grain synchronization beyond that provided by memory read/write atomicity (§1); this is possible because only the marker may change an object's color. Both previous MDS collectors are described with quadratic-time markers which make their implementation yet more impractical. To our knowledge, neither prior MDS approach has been implemented.

3.3 Other Concurrent Collectors

Steele's compactifying collector [26] performs a coloring mark-&-sweep collection, but does not interleave marking and sweeping. Steele's algorithm also requires fine-grain synchronization between mutator and collector. An orthogonal approach to concurrent mark and sweep collection is *incremental copying* which provides an avenue for future VCGC performance comparison. Baker designed an incremental copying collector [4] based on the tricolor white-gray-black abstraction. Halstead [11] extended Baker's core scheme to shared-memory multiprocessors by using fine-grain synchronization on individual objects. Appel, Ellis, and Li [2] propose using conventional memory-management hardware to serve as the read barrier in Baker's incremental copying collector [4]. Thereby they shift the granularity of the barrier from individual objects to pages of objects. They report pause latencies in an early ML compiler. Boehm uses memory management hardware to run mark-sweep collection in parallel with the mutator [6], but serializes marking and sweeping and requires mutator–collector synchronization on object allocation. Concurrent *replication-based* collectors [12, 23] designed for ML and implemented on parallel machines have—as has VCGC on a uniprocessor—reduced maximum pause latencies to tens of milliseconds. Unlike replicating schemes, VCGC does not rely on properties of the system being collected (*e.g.*, pointer-equality semantics). Doligez and Leroy [8] implemented a concurrent, hybrid, stop-&-copy mark-&-sweep collector for ML. Their copying collector reclaims a processor's local (*i.e.*, cache) memory and the mark-&-sweep collector reclaims storage in the shared global heap. Doligez and Leroy employ fine-grain synchronization and do not overlap the collector's mark and sweep phases.

4 Implementations

This section describes implementations of VCGC in two systems: Inferno [15] and SML/NJ [3]. The two descriptions are organized as an overview of the system in relation to GC, followed by discussions of the implementation of the free-list manager, colors, marker, sweeper, and mutator.

4.1 VCGC in Inferno

This section describes the hybrid reference-counting/VCGC collector of the Inferno operating system.

4.1.1 Inferno

Inferno [15] provides a distributed, network computing environment in which resources are made transparently accessible from anywhere in the network. Inferno's primary use is to provide a secure information dial-tone for network appliances and applications. The system treats application code as a resource that can be deployed throughout the network and run on a variety of clients or servers regardless of their processor architecture or underlying operating system. Code for the system is written in Limbo [16] and compiled for a virtual machine called Dis [15]. An Inferno instance contains either an interpreter, just-in-time compiler (JIT), or both.

Inferno was designed to run in small devices with as little as one megabyte of memory. Lazy algorithms like stop and copy work well when there is substantially more free memory than the working set to amortize the cost of collections [28]. Reference counting (RC) has two properties crucial to the Inferno environment. First, the cost of collection is constant and bounded, which permits audio and video encoders and decoders to run predictably. Second, (non cyclic) memory structures are reclaimed immediately after their last reference has been destroyed, making for a very small memory footprint. However, it is well known that RC does not detect cyclic garbage. By adding a concurrent collection algorithm (VCGC) that can coexist with RC we retain RC's advantages while incrementally reclaiming cyclic garbage. Reference counting manages about 98% of the objects while doing typical development (Limbo compiles, edits, runs, and debugs) in the Inferno window system (written in Limbo). The flexibility of this hybrid model has given Inferno real-time garbage collection while still allowing a substantial amount of control by the programmer over object lifetime.

A program running on the Dis virtual machine [15] consists of a set of threads. Each thread has a program counter and call stack but code and data may be shared with other threads running on the machine. Threads are scheduled to execute by the virtual machine rather than by the underlying operating system (if any). This allows Dis to multiplex many threads onto a single system process but also to control the interlocking of the heap. All threads occupy a single logical heap which is constructed from a linked list of large memory areas called regions. Memory within the regions is organized as a free list using an unbalanced ternary tree. Nodes in the tree are sorted by block size. As usual, adjacent free blocks are merged to control fragmentation. The thread call stacks and module pointers—global data for the currently executing modules—comprise the root set for Inferno GC.

During idle virtual-machine cycles, the GC thread runs to collect unreferenced data; it may be preempted at any time by a mutator thread becoming ready. This has proved effective in reducing the latency of garbage collection by overlapping it with I/O.

4.1.2 Inferno Marker & Sweeper

In Inferno, the VCGC marking and sweeping threads are merged. In a scheduler quantum, the GC thread visits a number of blocks in the heap by traversing the region table. Collection latency can be modified by increasing or decreasing the number of blocks visited in each quantum. The operation of the sweeper is trivial; the sweeper sets the RC of all blocks with sweeper color to zero, forcing their immediate deallocation. The Inferno marker uses a modified form of the algorithm described in §2.2.2. The marker implementation introduces a fourth static color into the algorithm, called the propagator, to avoid keeping both a stack for root set traversal and a store set of reference updates. (Techniques such as this for avoiding recursive marking are cataloged by Jones and Lins [13].) When an epoch begins each root is marked as a propagator. A count of propagators created during an epoch is maintained. When the marker reaches an object with the propagator color it is recolored with the mutator color and each pointer within the object is colored as a propagator and the count of propagators incremented. An epoch is complete when no new propagators are created during a pass through the region table. (Introduction of a propagator color requires a number of passes through the heap proportional to the depth of the deepest data structure; this has not been problematic in practice since RC reclaims most storage.)

4.1.3 Inferno Mutator

The Inferno mutator uses *type descriptors*, implemented as variable-length bitmaps, to describe which words in an object are pointers. Type descriptors allow the collector to traverse the heap without requiring and searching for tagged pointers. An object's type, along with its color, is stored adjacent to its data. The mutator copies references with the virtual machine's `movp` instruction which copies a reference from source to destination and performs the actions required by both garbage collectors—it increments and decrements the reference counts of the source and destination operands respectively. Since the source operand is a mutable reference as described in Section 2.3, the mutator colors the object the reference points to as a propagator and increments the propagator count—this is equivalent to inserting the overwritten reference in the store set. Since there are only a finite number of valid references in the heap, and all new objects are created in the mutator color, it follows that an epoch must terminate. While an epoch will terminate, many passes through the heap may be required for it to do so. We have not observed this potential problem in practice.

4.2 VCGC for SML/NJ

Here we describe a uniprocessor implementation of VCGC in version 109.31 of the SML/NJ ML compiler; a future port of this implementation to a multiprocessor stands to further improve performance.

SML/NJ [3] is a compiler for Standard ML [21, 20], a higher order and strongly typed language. ML programs are written in a mostly functional style; that is, reference updates are rare. The SML/NJ implementation is garbage collected by Reppy's multi-generational stop-&-copy collector [25] which is written in C. The SML/NJ system allocates a lot of small data at a rapid rate. On a set of common benchmark programs, we observed that over 90% of the allocated data is less than 96 bytes in length. For an earlier compiler, Appel reports that the system allocates a word of data for every 3–7 machine instructions executed [1]. Furthermore, most data ($\approx$ 93%) dies before the current allocation arena (typically a megabyte) is full. We use an allocation arena to buffer objects and reduce demand on the mark-&-sweep collector. Allocation in this buffer proceeds without coloring. When the allocation arena becomes full, live objects are assigned a color and copied into space obtained from VCGC free lists.

SML/NJ objects (*e.g.*, records, arrays, strings, *etc.*) are always tagged with a type and, when necessary, with a length field [25]. This tag is four bytes in size and is used by the runtime system and by the polymorphic-equality mechanism to identify objects dynamically. Two bits of SML/NJ's data-object descriptor tag are used to hold the three colors required for VCGC. The maximum size of variable-size ML data (*e.g.*, records, strings) is thus reduced by a factor of four (to 2^{23}).

4.2.1 SML/NJ Free Lists

We implemented two kinds of free-list manager in the runtime system. The first is a general-purpose free-list manager that can store objects of any size. General-purpose free-list managers perform buddy coalescing (see, *e.g.*, [14]) to avoid fragmentation. The second kind of manager is a *fixed-n* free-list manager that only handles data n words in length. In contrast to the general-purpose manager, a fixed-n manager is extremely fast since it never needs to do variable-size computations. Moreover, a fixed-n list cannot fragment. Sweeping is also fast in fixed-n managers since it involves only a fixed-size pointer increment. As a further optimization, we *run-length encode* contiguous free blocks—this speeds free-list creation, sweeping, and allocation.

We instantiate fixed-n list managers for all word sizes $2 \leq n \leq 512$ since most SML/NJ objects are small. Larger objects are handled by a single instance of a general-purpose manager. Code objects are handled by another instance of the general-purpose manager. (SML/NJ compiles to the heap.) A separate manager for code is necessary since pointers into code need not point at the head of the enclosing code object—during GC, it is necessary to map a code pointer to the head of the code object that holds it. Given a heap pointer p, we use SML/NJ's implementation [1, 25] of a Big Bag of Pages to "look up" which free-list manager, if any, is managing p.

The free-list managers construct free lists on demand; that is, when a free list l becomes empty, and the sweeper has not reclaimed objects for l, additional storage (in $C = 8K$ byte chunks) is requested from the operating system. We opted to extend the mutator's free lists instead of waiting for the sweeper to find appropriately sized garbage. Initially, no space is allocated to a free list—an initial allocation request creates the list. If objects of size m are never created, the fixed-m manager will not occupy storage.

For each free list, we maintain two pointers, one to the head of the list used for mutator allocation and one for lists reclaimed by the sweeper. When the mutator's list is exhausted, it copies the sweeper's pointer to its pointer and clears the sweeper's pointer. The sweeper, upon seeing a clear pointer, deposits a reclaimed list there. This producer-consumer handoff can be done without explicit synchronization. At an epoch boundary, the sweeper's list is appended to the mutator's.

4.2.2 SML/NJ Marker

When the allocation arena is full, SML/NJ enters the runtime system and runs the GC threads. This first empties the arena by copying its live data into the free lists and colors the thus copied data with the current epoch's mutator color. The marker is then permitted to mark M objects. Marking proceeds recursively via a stack. To date, we have found a fixed $16K$ mark stack to suffice; a future implementation will dynamically extend the stack when necessary, or thread the objects to be marked (*cf.* [13]).

A copy of the root set for the marker is obtained at the end of an epoch when the marker and sweeper have completed. This condition signals that at the next fill of the allocation arena, the root set is to be copied into the (now empty) marker stack.

4.2.3 SML/NJ Sweeper

The sweeper sweeps S free-list chunks every time an allocation-arena fill triggers GC. Recovered blocks are appended to the current free list under construction, retained in a local pointer. Contiguous free blocks are coalesced using run-length encoding (§4.2.1). The sweeper periodically checks if the mutator has acquired the last reclaimed list (clear sweeper-list pointer). If so, it updates the sweeper list with the list from the local pointer.

4.2.4 SML/NJ Mutator

Two minor compiler changes were necessary to use VCGC with code generated by SML/NJ. The first was to reduce the size of object length fields in descriptors to make space for a color. The second was to emit code to build a store list for VCGC (see §2.3). An element of the store list contains both the address of the updated reference (to track roots for copying into the allocation arena) and a pointer to the reference's prior contents (to track roots for VCGC marking).

4.2.5 Results

Timings for SML/NJ benchmarks using VCGC and conventional generational collection are in Table 1. We report the maximum pause latency, largest working set, total and GC times for compiling and running six ML benchmarks. Pause latencies are in milliseconds, memory sizes in megabytes, and run times in seconds using the notation *compile-time+execution-time*. The `knuth-bendix` program runs the Knuth-Bendix completion algorithm to produce a decision procedure for term equality in a theory; `life` implements Conway's game of Life, `mandelbrot` computes elements of its namesake's set[8], `ray` does some ray tracing, `simple` performs fluid-dynamics computations, and `tsp` optimizes traveling salesman tours. Timings were done on an unloaded 150Mhz R4400 SGI Challenge. Default compiler settings and an allocation arena of one megabyte were used.

Our initial goal was to reduce pause latencies to below 100 milliseconds. It was easy to find M and S values (that govern respectively the number of objects marked and the number of blocks swept per allocation-arena fill) that reduced the maximum pause to tens of milliseconds. It was however difficult to do so without increasing VCGC memory usage to beyond that of the generational collector. This is in part due to the conservative nature of VCGC which may retain *floating garbage* for a couple of epochs. The numbers in Table 1 are the result of a revised goal—to simultaneously minimize pause times *and* memory usage. We experimentally settled on the (albeit *ad hoc*) settings for M and S where $k = 4$ and E is the current epoch number. $M = kD/(E+1) + K_M$ where D is the number of objects marked in the previous epoch and $K_M = 1000$. $S = kC/(E+1) + K_S$ where C is the number of chunks currently appropriated by the free lists and $K_S = 10$. The tunable VCGC parameters require further study.

Memory usage was measured by tracking the maximum amount of OS memory held by the SML/NJ runtime at any

[8]The `mandelbrot` benchmark can be discounted because it requires essentially no GC.

	Max. Pause (ms)	Memory (MB)	Total Time (s)	GC Time (s)	
`knuth-bendix`	79	20.5	17.32+9.94	11.41+7.10	VCGC
	404	22.3	7.42+3.27	2.46+0.53	generational
`life`	80	16.6	7.14+13.82	5.20+7.06	VCGC
	314	21.9	2.36+6.58	0.68+0.09	generational
`mandelbrot`	80	14.0	0.75+2.35	0.47+0.00	VCGC
	315	21.9	0.29+2.35	0.04+0.00	generational
`ray`	79	16.4	8.86+27.82	5.50+16.86	VCGC
	314	21.9	3.84+11.69	0.91+0.04	generational
`simple`	80	24.4	48.63+34.54	33.73+23.92	VCGC
	475	31.7	26.06+9.88	13.01+0.43	generational
`tsp`	80	20.6	7.26+88.02	4.42+60.93	VCGC
	516	32.8	3.13+31.02	0.65+1.45	generational

Table 1: Uniprocessor timing results of ML programs using the SML/NJ compiler with VCGC collection, tuned to simultaneously reduce maximum pause latency and memory usage. Comparison values are timings of SML/NJ's generational collector. Run times are written *compile-time+execution-time*.

time. This amount contains the compiler since it must be resident to compile the application. Memory usage with VCGC is reduced by as much as 37% in the case of `tsp`. Other benchmarks show space reductions of over 30% (`life`, `mandelbrot`, `ray`, `simple`); `knuth-bendix` gains only 8%. We note that our VCGC implementation does not contain a valuable SML/NJ space optimization: stripping descriptors from pairs as they are promoted from the allocation arena. By moving colors out of descriptors, VCGC can perform similar optimizations and stands to improve locality and further reduce space usage.

Pause latencies were measured by enabling system timers, during GC, of the compiles and runs of the benchmarks. Maximum VCGC pauses range from threefold reductions (`life`, `mandelbrot`, `ray`) to sixfold reductions (`tsp`). The two other data intensive applications (`knuth-bendix` and `simple`) exhibit better than fivefold reduction in their maximum GC pause. As noted, one can trade an increase in memory usage for even better pause performance *and* better overall running times.

Overall VCGC performance suffers due to the goal of reducing both pause times and memory. Total time increased by as much as a factor of four and typically by a factor of two to three. These times can be reduced by lengthening epochs (less marking and sweeping per filled allocation arena) but at the cost of more memory in which to float garbage. We were able to approach to within about 30% of the generational times, but with larger memory sizes. (Pause times were excellent, <20ms, however.) However, the VCGC mutate, mark, and sweep threads do no more basic work than the respective phases of traditional mark-&-sweep collectors [19, 7, 4]. (To see this, simply delay the sweep thread until marking is complete.) Furthermore, VCGC's store set implementation is not overly expensive compared to standard barrier techniques (see Jones and Lins [13] or Wilson [28]). We therefore claim that the VCGC measurements are fairly indicative of mark-&-sweep collection (with an allocation buffer) of SML/NJ programs *in general* and hence are not due to the VCGC algorithm *per se*. We note that VCGC and other concurrent mark-&-sweep collectors (*e.g.*, [17, 24, 7]) float garbage and therefore require additional storage and its associated processing beyond that of sequential mark-&-sweep.

VCGC pause and runtime performance stands to greatly improve from implementation on a multiprocessor that can truly overlap marking, sweeping and mutation in time.

5 Summary

We have designed and implemented a new variant of mark-&-sweep storage reclamation called Very Concurrent Garbage Collection. In this algorithm, the mutator, marker, and sweeper threads operate concurrently within epochs; a novel coloring scheme identifies recyclable data. We implemented VCGC in the commercial Inferno operating system to detect and reclaim discarded cyclic data. Our other implementation is in the SML/NJ ML compiler, where VCGC can eliminate long GC pause latencies while reducing memory usage. Tuning VCGC for pause, memory, and execution performance—as well as characterizing its multiprocessor performance—are directions for further investigation.

Acknowledgements

Thanks to Eric Grosse, Brian Kernighan, Dave MacQueen, Rob Pike, and the anonymous reviewers for valuable comments.

References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Conference on Programming Language Design and Implementation*, pages 11–20. Association for Computing Machinery, June 1988.

[3] A. W. Appel and D. B. MacQueen. A Standard ML compiler. *Functional Programming Languages and Computer Architecture*, 274:301–324, 1987.

[4] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[5] E. Biagioni, R. Harper, P. Lee, and B. G. Milnes. Signatures for a protocol stack: A systems application of Standard ML. In *Proceedings of the Conference on Lisp and Functional Programming*, pages 55–64. Association for Computing Machinery, June 1994.

[6] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Conference on Programming Language Design and Implementation*, pages 157–164. Association for Computing Machinery, June 1991.

[7] E. W. Dijkstra, L. 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):966–975, November 1978.

[8] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Symposium on Principles of Programming Languages*, pages 113–123. Association for Computing Machinery, 1993.

[9] R. R. Fenichel and J. C. Yochelson. A Lisp garbage-collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[10] J. Gosling and H. McGilton. The Java language environment: a white paper. Sun Microsystems, Inc., 1995.

[11] R. H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the Conference on Lisp and Functional Programming*, pages 9–17. Association for Computing Machinery, August 1984.

[12] L. Huelsbergen and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Principles and Practice of Parallel Programming*, pages 73–82. Association for Computing Machinery, May 1993.

[13] R. Jones and R. Lins. *Garbage Collection*. John Wiley & Sons, 1996.

[14] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1969.

[15] Bell Laboratories. *Inferno Developers Guide*. Lucent Technologies, Murray Hill, NJ, 1996.

[16] Bell Laboratories. *The Limbo Language Definition*. Lucent Technologies, Murray Hill, NJ, 1996.

[17] L. Lamport. Garbage collection with multiple processes: An exercise in parallelism. In *Proceedings of the International Conference on Parallel Processing*, pages 50–54, August 1976.

[18] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[19] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, pages 184–195, April 1960.

[20] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

[22] M. Minsky. A LISP garbage collector algorithm using serial secondary storage. A.I. Memo 58, Massachusetts Institute of Technology, 1963.

[23] S. Nettles and J. O'Toole. Replication-based real-time garbage collection. In *Conference on Programming Language Design and Implementation*. Association for Computing Machinery, June 1993.

[24] C. Queinnec, B. Beaudoing, and J.-P. Queille. Mark DURING Sweep, rather than Mark THEN Sweep. In *Conference on Parallel Architectures and Languages Europe: PARLE*, pages 224–237, 1989.

[25] J. H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, January 1994.

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

[27] D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, 1987.

[28] P. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*. Springer Verlag, 1992.

```
typedef struct root_list {
  root_t root;
  struct root_list *next;
} root_set_t;

root_set_t *root_set = NULL;    /* set of elements */
root_set_t *last = NULL;        /* last root_set capture */
root_set_t *before_last = NULL; /* before_last root_set capture */
root_set_t *this = NULL;        /* next element to remove */
/* invariant: 'this' always lies between 'last' and 'before_last' inclusive. */

/* remove():
 *  Returns the next not yet removed element from 'root_set', NULL if there currently isn't an element in 'root_set'.
 *  Called only by the marker thread.
 */
root_set_t *remove()
{
  root_set_t *tmp;

  /* invariant: 'last' and 'before_last' elements, if non-NULL, have already been removed */
  if (this == before_last) {
    before_last = last;  this = last = root_set;
  }
  if (this == before_last) return NULL;
  tmp = this;  this = this->next;
  return tmp;
}

/* insert(x):
 *  Inserts list node 'x' into the root_set for subsequent marker removal.
 *  Assumes a root has been encapsulated in a fresh list node 'x' by caller.
 *  Called only by the mutator thread.
 */
void insert(root_set_t *x)
{
  x->next = root_set;
  MemoryBarrier();  /* force write of previous line to memory */
  root_set = x;
}

/* done():
 *  Returns true if every element in the root set has been removed by marker.
 *  Must be called with both mutator and marker threads halted, and their memory reads/writes complete.
 */
bool_t done()
{
  return before_last == root_set;
}
```

Appendix

The above C code provides non-blocking set functions for implementing an asynchronous write barrier for the VCGC algorithm. In particular, the code requires no explicit synchronization even though the mutator may add roots (with `insert`) to the set while the marker concurrently removes roots (with `remove`). This write-barrier algorithm assumes atomic machine-word memory writes, as defined in the introduction (§1). Note that there is a read/write race with the read of `root_set` in `remove` and the write to `root_set` in `insert`. This race is tolerable, however, because memory writes are atomic and subsequent calls to `remove` will retrieve the "missed" element(s). The `insert` function contains a memory barrier that forces all pending writes to memory on processors that may reorder writes; its operation however is local to the executing processor and does not involve inter-processor synchronization. Here it ensures that the insertion (`x->next = root_set`) is seen by other processors before `root_set` is reset to `x`. Note that the `root_set` list monotonically increases in length over time; it may immediately be reclaimed when the mutator and marker threads agree that the set is empty (*i.e.*, when `done` succeeds).