Proc. 20th Symp. Principles of Programming Languages, 1993, pages 113–123.

A concurrent, generational garbage collector for a multithreaded implementation of ML

Damien Doligez Xavier Leroy

École Normale Supérieure and INRIA Rocquencourt*

Abstract

This paper presents the design and implementation of a "quasi real-time" garbage collector for Concurrent Caml Light, an implementation of ML with threads. This two-generation system combines a fast, asynchronous copying collector on the young generation with a non-disruptive concurrent marking collector on the old generation. This design crucially relies on the ML compile-time distinction between mutable and immutable objects.

1 Introduction

This paper presents the design and implementation of a garbage collector for Concurrent Caml Light, an implementation of the ML language that provides multiple threads of control executing concurrently in a shared address space.

Garbage collection — the automatic reclamation of unused memory space — is one of the most problematic components of run-time systems for multi-threaded languages. The naive "stop-the-world" approach, where all threads synchronously stop executing the user's program to perform garbage collection, is clearly inadequate, since it introduces synchronization between otherwise independent threads. For instance, this can result in all threads being blocked for some time if one thread is in the middle of a lengthy, uninterruptible operation when garbage collection starts. This contravenes one of the main motivations for having multiple threads: to reduce the response time of interactive applications. To achieve this goal, a promising direction is to run the garbage collector concurrently with the threads that execute the user's program, with as little synchronization as possible between the collector and the mutators (the threads executing the user's program).

A number of concurrent collectors have been described in the literature, such as the concurrent mark-and-sweep algorithm [11, 15, 5], which requires no synchronization with the mutators, at the price of a moderate overhead on the mutators. However, these designs seem unable to meet the memory demands of typical ML programs. ML programs tend to have high allocation rates, but many allocated objects have a short life span. This is due in part to the ML language itself, which encourages a programming style where many intermediate structures are built; and in part to some compilation techniques [1, 10] that result in heap allocation for large amounts of environments and control structures.

The garbage collection technique most adapted to this allocation profile is generation scavenging [24], that concentrates reclamation effort on recently allocated objects. However, the efficient implementation of generation scavenging requires the ability to relocate objects by copying between the memory areas that hold the various "generations" of objects. Performing relocation while the mutators are running is problematic: we must ensure that the mutators are aware of the relocation, and do not try to access a relocated object at its old, invalid address. Some designs rely on tests when dereferencing a heap pointer [23, 4]; others, on an extra indirection word for each heap object [8, 20]; others, on virtual memory page protections [2]. All three approaches entail a significant run-time penalty on the mutators, unless special hardware or special system software is used.

The memory management system presented in this paper is an attempt to circumvent this weakness of concurrent copying collectors by relying on specific features of the ML language. This system has two generations, with a fast, asynchronous copying collector on the young generation, and a non-disruptive concurrent marking collector on the old generation. The aforementioned difficulties with copying are avoided by splitting the young generation into areas attached to the mutators, each area being accessed by one mutator only. The

*Authors' address: INRIA Rocquencourt, B. P. 105, 78153 Le Chesnay, France. E-mail: `Damien.Doligez@inria.fr`, `Xavier.Leroy@inria.fr`.

performance issue with concurrent mark-and-sweep is avoided by the fact that the allocation rate in the old generation is low, since most short-lived objects are reclaimed by the copying collectors. This combination results in quasi real-time performance for memory allocation, while keeping the overhead on the mutators low.

This design relies crucially on two features of ML. First, the ML type system distinguishes at compile-time between mutable objects (that can be physically modified) and immutable objects. Second, duplicating immutable objects is semantically transparent. The first point makes it possible to have different allocation policies for mutable and immutable objects. The second point allows copying the object residing in the private area of a mutator at arbitrary times.

The remainder of this paper is organized as follows. Section 2 briefly describes the Concurrent Caml Light system. Section 3 presents the memory organization; the concurrent aspects of the system (the mark-and-sweep major collector) are detailed in section 4. Section 5 comments on some experimental results. Section 6 discusses some directions for further work. Finally, section 7 compares our design with some other concurrent collectors.

2 Concurrent Caml Light

Concurrent Caml Light is an extension of Caml Light [12, 16], the authors' implementation of the ML language, with concurrency primitives. The concurrency model is lightweight processes (threads) with shared memory. The synchronization tools are locks and conditions. (Figure 1 shows the Caml Light interface to the module providing the concurrency primitives.) This is the model provided by the C Threads library under the Mach operating system [9]. On top of these concurrency primitives, we can to implement higher-level concurrency abstractions such as channels and events [22, 6].

The ML language is a conventional imperative language with functions as first-class values and strong static typing [21, 17]. From the standpoint of memory management, the ML language has two distinctive features that are crucial to the design described here. The first feature is that not all ML data structures can be modified in-place. That is, the updating primitives provided by the language operate only on specific data types, either built-in (such as references and arrays) or specially declared (such as the Caml record types with "mutable" fields). This fact, combined with strong static typing, ensures a clear separation at compile-time between mutable objects (that can be physically updated) and immutable objects (that can only be read once constructed). This permits different allocation policies for mutable and immutable objects; our design takes advantage of this fact.

```
type process;;
value fork : (unit -> 'a) -> process
  and exit : unit -> 'a
  and join : process -> unit
  and detach : process -> unit
  and yield : unit -> unit
  and self : unit -> process;;
type mutex;;
type condition;;
value new_mutex : unit -> mutex
  and new_condition : unit -> condition
  and lock : mutex -> unit
  and unlock : mutex -> unit
  and try_lock : mutex -> bool
  and signal : condition -> unit
  and broadcast : condition -> unit
  and wait : condition -> mutex -> unit;;
```

Figure 1: The interface to the module `thread` providing the concurrency primitives

Another important feature of ML is that it does not specify any generic physical equality primitive similar to `eq` in Lisp. The provided equality primitive implements structural equality on immutable objects, and physical equality on mutable objects. Consequently, there is no way to test two immutable objects for physical equality. Combined with the fact that immutable objects cannot be modified in-place, this means that it is always semantically correct to duplicate an immutable structure: the original structure and its copy cannot be distinguished by any program. Our collector does indeed duplicate immutable structures and keeps the two copies alive for some time — strange as it may sound for a system that is supposed to reclaim memory space.

3 Overview of the memory organization

The memory heap is organized as follows. (See figure 2.) First, there is a large, common heap shared between all threads. All threads can allocate, read, and update objects in the shared heap. Then, each thread possesses its own, small, private heap (typically 32K). On a modern shared-memory architecture with large, write-back caches, we expect the private heap to remain in one and only one cache most of the time, thereby causing very little bus traffic when it is accessed. This assumes that the system scheduler is clever enough to tie each thread to a single processor whenever possible.

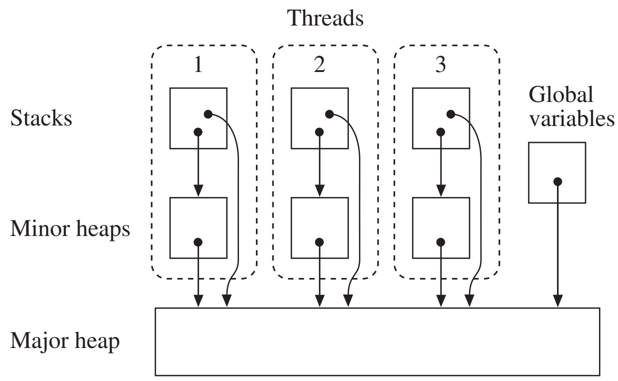

Figure 2: Memory organization

3.1 Two generations

Each thread treats the two heaps it can access (the shared heap and its own private heap) as two generations: the private heap contains the young generation; the shared heap contains the old generation. Each thread allocates immutable objects in its own private heap. Mutable objects are handled differently, as we shall see below. This allocation does not require any synchronization with the other threads.

When the private heap becomes full, the corresponding thread stops and performs a minor collection: it copies all live objects in the private heap to the shared heap. Live objects are those that are pointed to by the memory roots of the thread (the registers and the stacks of the machine), as well as their descendents. This copying makes the whole private heap available again for private allocation. Consequently, allocation in the private heap is performed linearly, and requires only one pointer comparison and one pointer increment. A minor collection can be performed at any time, regardless of the status of the other mutator threads. The only synchronization required is when allocating the copied objects in the shared heap.

Major collection on the shared heap is performed by a dedicated thread, which runs concurrently with the mutator (and minor collection) threads. It uses the concurrent mark and sweep algorithm described by Dijkstra *et al.* [11]. We postpone a complete discussion of the algorithm and the cooperation between the major collector thread and the other threads to the next section. Since the major collector does not move objects, no synchronization is required when accessing or modifying an object in the shared heap, either for the major collector thread or for the mutator threads. Race conditions can result in a dead object not being collected by the current major collection cycle; but they cannot result in a live object being reclaimed.

If the available space in the shared heap drops to zero before the major collection cycle is over, then the mutator threads attempt to enlarge the shared heap, by extending the process address space, instead of waiting for the major collection to finish. We want to avoid blocking the mutator threads as much as possible. Blocking is only required in the unlikely case where the virtual memory is exhausted.

3.2 Copy on update

The design outlined above assumes that there are no pointers from the shared heap to a private heap, nor from one private heap to another private heap. Otherwise, a private heap could contain objects that are live, but not directly reachable from the roots of the corresponding threads. Without special treatment, a pointer from the shared heap to a private heap can be created by updating an "old" mutable object, residing in the shared heap, with a pointer to a newly created structure, that still resides in a private heap; and a pointer between two private heaps can then be created by reading the mutable object from another thread.

This situation is avoided by copying the transmitted private object to the main shared heap and storing in the old mutable object a pointer to the copy, instead of a pointer to the original private object. The descendents of the transmitted object that reside in the private heap are recursively copied, too. This copying is very similar to a minor collection with only one root, the transmitted object. Indeed, it stores forwarding pointers from the copied objects to their copies, just as the minor collector does, so that the next minor collection will not copy these objects again, but reuse their copies.[1] Therefore, this "copy on update" strategy does not waste time: we just do some of the next minor collection right away. Also, it avoids the complexity of maintaining a remembered set of old objects that contain pointers to the young generation [24].

3.3 Allocation of mutable objects

Until the next minor collection, the thread that created the transmitted object can access both the original, private object and its copy in shared memory: the original object can still be reached through the memory roots of the thread, since we haven't updated the roots of the thread; the copy can be accessed by dereferencing the mutable object in which it was stored. Therefore, we must ensure that the two objects are semantically equivalent. This is the case if both objects contain only immutable structures; then, as pointed out in the previous section, no constructions in the ML language can distinguish one from the other. This is no longer true

[1] To implement this, the objects in the private heaps have one extra header word, to store a forwarding pointer without destroying the object. This extra word is stripped when the object is copied to the major heap.

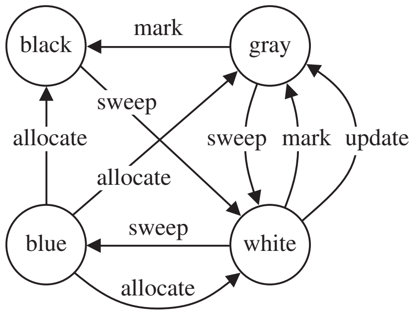

Figure 3: Color transitions

if the original object contains a mutable structure, because it would be duplicated during the copying process. This could lead to an update of the two mutable structures by two different objects, breaking the equivalence between the transmitted object and its copy.

To avoid this situation, it suffices to allocate mutable objects directly in the main, shared heap. Then, they will never be copied, since they already reside in the shared heap. This makes copying semantically transparent. Of course, a performance penalty is incurred: allocation in the shared heap is more expensive than allocation in the private heap, because of the required synchronization and free-list searching. However, most ML programs allocate relatively few mutable objects, and they tend to have a longer life span than average. This keeps the overhead reasonable.

4 The concurrent collector

The major collector implements the concurrent mark and sweep algorithm described by Dijkstra *et al.* [11]. In this section, we recall the basics of the algorithm, adapt it to our situation (Dijkstra *et al.* made some simplifying assumptions to keep correctness proofs manageable), and show how the mutator threads cooperate with the concurrent collector.

In this section, each thread along with its minor collector is considered a mutator thread by the major collector. The major collector will be called "the collector", and the mutator threads (and their minor collectors) will be called "the mutators".

The major collector does not essentially depend on the existence of the minor collectors. It only needs some way of asking a given mutator to mark the objects pointed to by its roots. In our design, this marking is performed by the minor collectors.

4.1 Four-color marking

Each object in the shared heap has one of four colors: white, gray, black, or blue. White denotes objects that have not yet been visited by the marking phase. Gray denotes objects that have been visited, but whose sons have not yet been visited. Black denotes objects that have been visited, and whose sons have been visited too. Blue is used for the free list objects: blue objects are always ignored by the collector.[2]

The color of a block evolves as summarized in figure 3. The marking phase sets to black all reachable objects. To do so, it sets the roots to gray and repeatedly finds a gray object and marks it. *Marking* an object means setting it to black, and shading its sons. *Shading* means setting the object to gray if it is white. The sweeping phase reclaims all white objects, setting them to blue and adding them to the free list. It also resets all black objects to white. Allocation in the heap turns blue objects back to white, gray or black, depending on the relative states of the collector and mutator, as detailed below.

4.2 The collection phases

The collector proceeds in three phases: root enumeration, end of marking, and sweeping. The root enumeration and end of marking together constitute the marking phase.

At the beginning of the root enumeration phase, the collector sets a global flag to signal the beginning of the marking phase. It then shades the global variables, and asks each mutator to shade its roots. During this phase, the collector also begins to find gray objects and mark them as described above.[3] The root enumeration ends when the collector has obtained the roots of the last mutator. The collector then completes the marking phase by repeatedly marking gray objects until no more remain.

When the marking phase is finished, the collector examines each heap object in turn. All black objects are set to white. All white objects are free; they are set to blue and inserted into the free list (or collapsed with the preceding free object, if adjacent). Some objects might have been set to gray by the mutators since the end of the marking phase. These objects are also set to white.

The marking phase assumes that no object is black when it starts, and it ensures that all reachable objects are black or gray when it stops. More precisely:

- all objects that are reachable from the roots of a mutator at the time the mutator shades its roots, or that become reachable after that time, are black at the end of the marking phase.

[2] In theory, the color blue is not needed: it suffices to consider the free-list head as a memory root, and the free-list blocks as regular reachable blocks. However, the blue color avoids the extra cost of tracing and coloring the free-list blocks.

[3] To quickly find the next gray object, a cache of recently shaded objects is maintained, avoiding the cost of an actual scanning of the heap in most cases.

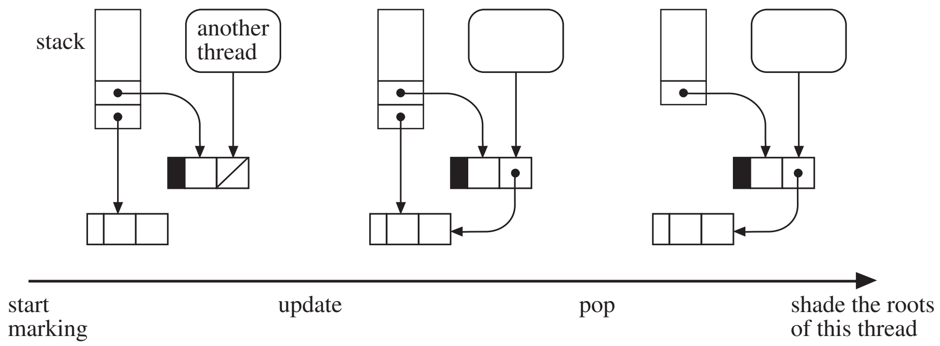

Figure 4: What happens if we do not shade the new value

Objects can become reachable by allocation and by in-place modification, which are performed by the mutators concurrently with the collection. These operations therefore require some cooperation with the collector, as described below. The sweeping phase assumes that all reachable objects are black or gray when it starts, and it ensures that only unreachable objects are inserted into the free list, and that no black objects remain when it stops. Again, allocation and in-place modification require some cooperation with the collector, in order to avoid setting objects to black.

These preconditions and postconditions ensure the correctness of the collector: only unreachable objects are ever inserted into the free list. The completeness (all unreachable objects are eventually inserted into the free list) stems from the following facts:

- no unreachable object ever becomes reachable again
- there are no blue objects outside of the free list
- all white objects unreachable at the start of the marking phase remain white
- all white objects are inserted into the free list by the sweeping phase
- gray objects that are unreachable at the beginning of the mark phase become black during marking, then white during sweeping, and are reclaimed by the next collection cycle.

4.3 Concurrent allocation and modification

As explained in [11], the mutators have to take the collector state into account when performing in-place modification on heap objects. Otherwise, updating an already black object could result in a reachable object that remains white at the end of marking. This problem is further complicated by the fact that the set of roots is not fixed during the collection: mutators can push and pop pointers on their local stacks without any cooperation with the collector.

To avoid this kind of situations, the modification operation must shade both the old and the new value of the modified field. Shading the new value ensures that it will be recognized as reachable by the collector, even if all other pointers to the new value disappear (e.g., by popping the last pointer from a stack). Shading the old value ensures that it will be recognized as reachable by the collector, in case some pointers to the old value are still kept on some stack.

In the simplified setting described in [11] (a fixed set of roots), shading either the old or the new value is sufficient. This is not true in our case. Assume we do not shade the new value. Since the collector starts marking objects before having obtained all roots, a mutator can modify a black object by storing a pointer to a white object which is only reachable from the local stack, then pop all pointers to this white object before shading its roots. This results in a reachable object that remains white. This kind of pointer smuggling is illustrated in figure 4. Now, assume we do not shade the old value. The mutator could give its roots, then push a white field of a white object onto its stack, then overwrite that field. This results in a white object that is reachable from the stack. (See figure 5.)

This coloring at modification time is only necessary during the marking phase. For the sake of efficiency, we do not perform it during the sweeping phase, avoiding the creation of gray objects that would survive a complete collection cycle before being reclaimed.

Concurrent allocation raises similar issues: the newly allocated objects must be assigned the right color, depending on the collector status. During the marking phase, objects are allocated black. This is justified by the fact that the allocated objects become reachable, and their sons were already reachable, hence will eventually be set to black. Setting the allocated objects to gray would also be correct, but the marking phase might not terminate.

During the sweeping phase, objects are allocated

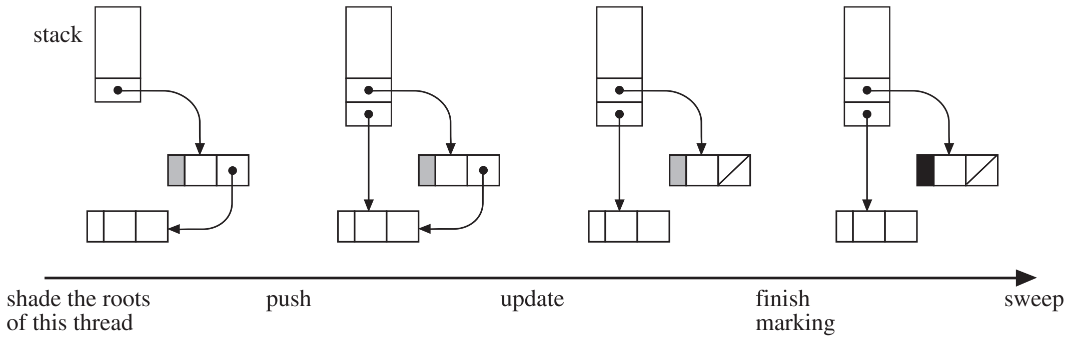

Figure 5: What happens if we do not shade the old value

white if they have already been swept, and gray otherwise, to avoid immediate deallocation.

4.4 Synchronization issues

The coloring scheme described above has one interesting property: it is always safe to set an object to gray. Of course, setting many objects to gray is inefficient, since an unreachable gray object will not be reclaimed at the end of the current collection cycle, but only at the end of the next cycle. However, this fact allows us to avoid synchronization whenever the resulting race condition can only end up in making an object gray instead of the intended color.

This trick is used in the modification and allocation procedures, to test the collector status without locking. For instance, the coloring of newly allocated blocks is implemented as follows:

```
1.  if phase = marking then
2.     set the object to black;
3.     if phase = sweeping then
4.        set the object to gray;
5.  else
6.     if address(object) < sweep_pointer then
7.        set the object to white;
8.     else
9.        set the object to gray;
```

There are two race conditions between this code and the collector. First of all, the collector may enter the sweeping phase between lines 1 and 2. Then, the object could incorrectly be set to black after being swept. In this case, lines 3 and 4 set the object back to gray, which meets the preconditions of the next marking phase. The collector must synchronize with all mutators before entering the marking phase, hence line 4 is guaranteed to complete before the next marking cycle. The other race condition is that the sweep pointer can change after the test in line 6. However, the sweep pointer is monotonically increasing, hence the race condition can only result in executing line 9 instead of line 7, i.e. in setting the object to gray instead of white, which is safe.

4.5 Interface with the minor collector

The shading of roots is performed by a variant of the minor collector that sets to gray all objects copied to the major heap, as well as all root objects that are already in the major heap. This requires little extra work compared with a normal minor collection.

Hence, the least disruptive technique for getting the roots is to set a flag telling the minor collectors to shade the roots, and wait for all the mutators to complete a minor collection. However, a mutator can execute a program that does not allocate; it can also be blocked on a lock, or waiting for input or output. In the former case (looping mutator), the major collector interrupts the mutator and forces a premature minor collection. In the latter case (blocked mutator), the major collector performs the copying and shading itself; this is similar to a minor collection, except that the minor heap is not emptied. The major collector also has to make sure the mutator does not resume execution before the copying and shading is complete. This is the most disruptive interaction between the collector and a mutator, but it is infrequent.

5 Experimental results

We have implemented the collector described above in a prototype ML system derived from Caml Light release 0.4. It runs on an Encore Multimax with fourteen NS32532 processors, under the Mach operating system. Each processor is rated at about 6 MIPS, and has a 256 K write-back cache. The Caml Light system is a fast bytecode interpreter; it runs 4 to 8 times slower than the SML of New Jersey native-code compiler. To put the timings below in perspective, an application of the identity function takes about 15 μs. The measurements used 32 K private heaps, that easily fit into the

Test program	Knuth-Bendix	Pipelined compiler	Parallel compiler	SIMPLE (30×30)
Number of threads	15	3	12	6.4 (avg)
Proportion of updates requiring copying	96 %	43 %	36 %	96 %
Major GC load	32 %	16 %	39 %	10 %
Minor GC, average	2.9 ms	2.3 ms	6.3 ms	2.1 ms
Minor GC, worst-case	64 ms	180 ms	110 ms	360 ms
Copy-on-update, average	260 μs	37 μs	55 μs	70 μs
Copy-on-update, worst-case	70 ms	6.9 ms	31 ms	20 ms
Free-list locking, average	60 μs	19 μs	1.6 ms	54 μs
Free-list locking, worst-case	17 ms	220 μs	110 ms	25 ms

Figure 6: Average performance

caches, along with the run-time system and the bytecode program.

In this section, we comment on some measurements performed on this implementation. We have used the following test programs:

- A parallel implementation of the Knuth-Bendix completion algorithm. The program comprises fifteen threads, and performs lots of interprocess communication via shared mutable data structures.
- A pipelined version of the Caml Light compiler, with one thread for the lexical analyzer, one for the parser, and one for the remainder of the compiler. This program is a typical example of the producer-consumer model. The amount of communication is respectable, though less important than in the parallel Knuth-Bendix program.
- A parallel version of the Caml Light compiler, that simultaneously compiles several files, each file being compiled sequentially by one thread. There is very little communication between the threads. Our test runs twelve compilers in parallel.
- The SIMPLE numerical benchmark from Appel's book [1], parallelized by Morriset and Tolmach [19]. This program is typical Fortran code translated to ML, and makes very heavy use of mutable arrays. The parallel version relies on "futures", that is, lazy structures with speculative evaluation.

The measurements have two goals: first, estimate the latency of memory operations such as allocation and in-place modification; second, determine whether the major collector keeps up with a high number of active mutators. For the first point, we have measured how long the mutators are interrupted by (1) minor collections, (2) copy on update operations, and (3) direct allocation in the major heap, which requires synchronization. For the second point, we take advantage of the fact that the major collector does not run continuously, but only when the amount of free space in the shared heap drops below a certain threshold (15% of the total heap size, in the experiments). Hence, the running time of the major collector compared with the execution time of the whole program gives an estimate of the load of the major collector.

The results are given in figure 6. The load of the major collector appears to be below 5% per mutator. This suggests that our design should scale to about 20 mutators. These results hold for the four realistic programs considered here. However, on artificial examples that do nothing but allocate mutable objects, the major collector cannot keep up with as few as four mutators. This is an experimental confirmation of the initial assumption that real ML programs do not allocate much mutable data.

The average latency times are remarkably low. Most minor collections complete in less than 10 ms. The copy-on-update strategy makes the cost of an assignment proportional to the size of the assigned value (with the size of the private heap as upper bound) in the worst case; in practice, assignment remains reasonably efficient, even in programs such as the Knuth-Bendix benchmark, that transmit large structures through mutable objects. Finally, the last case where a mutator can be delayed on a memory operation is when it accesses the free list to allocate objects directly in the shared heap: free list accesses must be mutually exclusive. To lower contention, each thread maintains its own small, private free list. The private free lists are replenished from the main free list when a request cannot be satisfied. Transfers from the main free list to a private free list are performed a large chunk at a time, to keep their frequency low. This strategy works well on three of our test programs, but does not avoid a certain amount of contention for the parallel compiler.

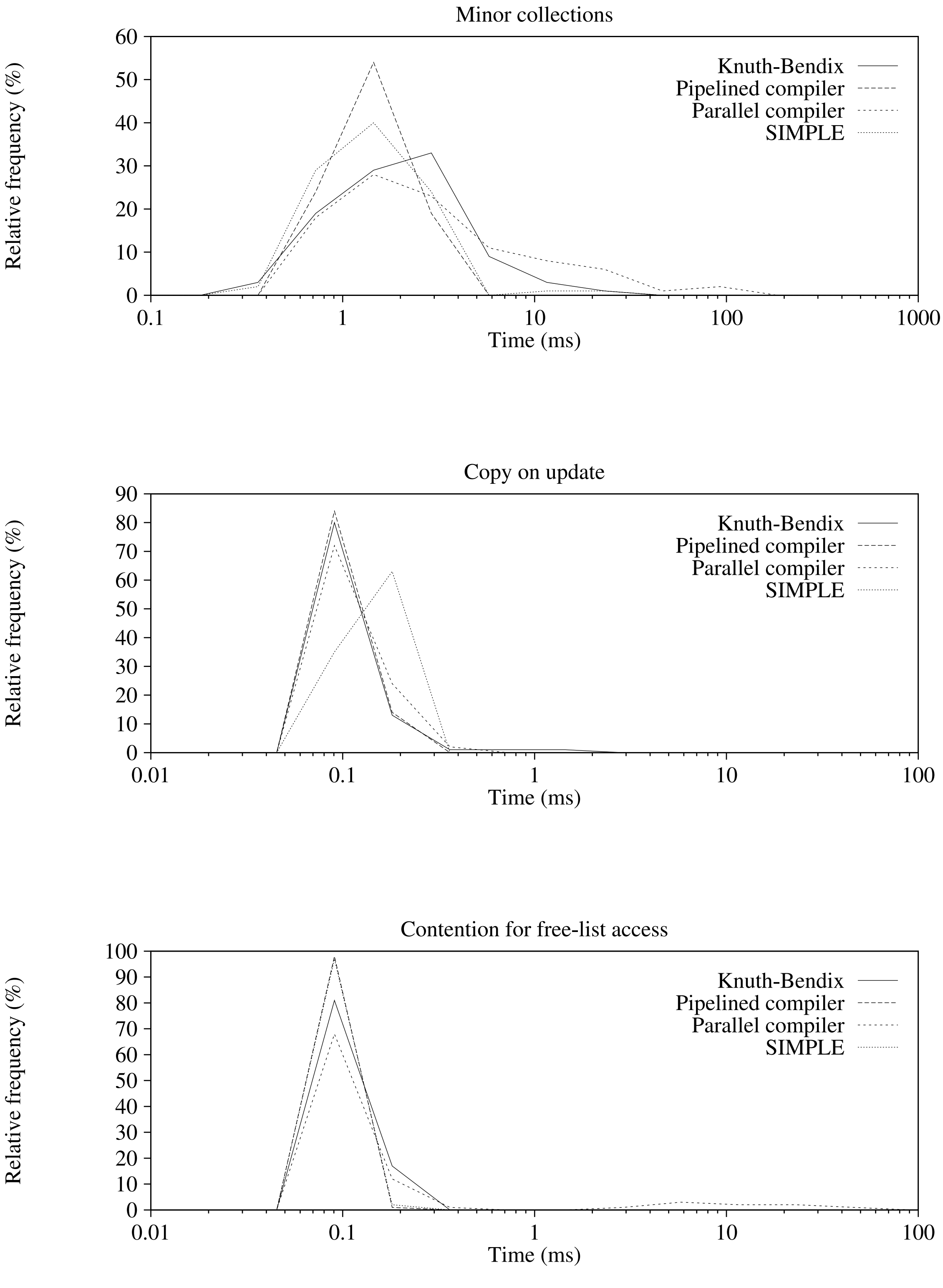

Figure 7: Latency distribution

From these results, we conclude that our design achieves good response time, and is adequate for interactive applications. However, it does not achieve true real-time performance: there is no guaranteed upper bound on the time taken by memory operations. A small number of these operations take much longer than the average time. This can be seen on figure 7, which plots the distribution of execution times for the three memory operations. For instance, a minor collection can take as much as 360 ms, in the worst-case where all objects in the minor heap are alive. Similarly, some copy-on-update operations may need to copy (almost) all objects from the minor heap. There is a trade-off between maximal latency and garbage collection overhead: the worst-case latency can be lowered by reducing the size of the private heaps, but this results in more time spent in minor collections, and an increased load on the major collector.

6 Extensions

The memory management system described above can be extended in several ways. The first direction is to parallelize the major collection, in order to keep up with more active mutators. The sweeping phase can straightforwardly be parallelized, since the heap is already divided in medium-sized chunks (256 K), which can be swept by independent threads. The marking phase can also be performed concurrently by several threads, though achieving good balance is more delicate.

Another area of improvement is the "weight" of threads. Since each thread has its own stack and its own private heap, thread creation is a relatively expensive operation: starting a Concurrent Caml Light thread takes about 3 ms, which is commensurate with the time it takes to start a Mach thread (about 1 ms), but still too important for applications that spawn a large number of short-lived threads. For these applications, a promising direction is to adopt the two-level scheme outlined in [19], where the user-level threads are multiplexed on top of a small number of kernel threads. Each kernel thread has its own private heap, and time-shares between a number of user-level threads. User-level threads can freely share a private heap, provided that the memory operations on the private heap are mutually exclusive, which the user-level scheduler can easily guarantee.

Finally, the concurrent collector described above can be simplified into an incremental, generational collector for uniprocessors. The idea is to perform a small part of the major collection at each minor collection. Since there is only one private heap, copy-on-update is no longer mandatory, and we can maintain a remembered set instead. We have integrated this incremental collector in the release 0.5 of the Caml Light system.

7 Related work

The system described in this paper is related to two trends in research on garbage collection. The first trend deals with concurrent variants of the classical mark-sweep algorithm, with as little synchronization as possible between the mutator and the collector [15, 11, 5]. The emphasis here is on proving the correctness of the proposed algorithms, rather than on practicality and efficiency. To our knowledge, none of these designs has made its way into an actual run-time system. There are good reasons to believe that collectors based on these designs would not be able to keep up with typical ML programs. Hickey and Cohen [14] provide some theoretical evidence of this problem. This problem is avoided in our system by the use of generation scavenging, that greatly reduces the allocation rate as viewed by the concurrent mark-and-sweep collector.

A different approach to the parallelization of the mark-sweep algorithm is described by Boehm *et al.* [7]. Their algorithm requires no cooperation from the mutators; instead they rely on virtual memory protections to keep track of modifications performed by the mutators. Their collector overlaps most of its work with the mutator activity but it has to stop the mutators to finish the marking phase. The resulting pauses are short (about 100 ms) but still one order of magnitude longer than in our system on average. Moreover, their technique must stop all mutators simultaneously, introducing a spurious global synchronization point between all threads. Avoiding this phenomenon was one of our main goals.

The second trend is the practical implementation of concurrent or incremental copying collectors. The first such collectors were described by Steele [23] and Baker [4], and later extended to generations [18] and to multiple mutators [13]. This algorithm requires a test on each heap pointer dereferencing, which imposes considerable overhead on the mutator, unless special hardware is used. A variant proposed by Brooks [8] replaces this test by a systematic indirection. On stock hardware, this technique slightly reduces the overhead, at the expense of one extra word per heap object. North and Reppy [20] have extended this technique with generations. Appel, Ellis and Li [2] propose to use virtual-memory protections to implement Baker's algorithm without tests on stock hardware. Their technique relies on sophisticated virtual memory primitives, which most widespread operating systems do not provide in an efficient way [3]. Thus, concurrent purely copying garbage collection has not yet been implemented on stock hardware and standard operating systems without major overhead on the mutators. Our mixed design avoids this difficulty by restricting the copying to unshared objects, which cannot be accessed concurrently.

8 Conclusions

We have described a memory management system for a multithreaded implementation of ML that achieves quasi real-time performance with low overhead on the mutators. This system relies crucially on the compile-time separation of mutable and immutable objects. In the case of ML-like languages, this separation is ensured by the type system, therefore demonstrating an unexpected spin-off of strong, static typing in the area of garbage collection. This technique can also be applied to dynamically-typed languages such as Scheme, as long as separate allocation primitives are provided for mutable cons cells and immutable cons cells, and similarly for other data types.

Acknowledgments

We would like to thank Ian Jacobs for his careful proofreading, and Greg Morriset for sending us the SIMPLE benchmark.

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. *SIGPLAN Notices*, 23(7):11–23, 1988.

[3] A. W. Appel and K. Li. Virtual memory primitives for user programs. Technical Report CS-TR-276-90, Princeton University, 1990.

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

[5] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Trans. Prog. Lang. Syst.*, 6(3):333–344, 1984.

[6] B. Berthomieu. Implementing CCS: the LCS experiment. Technical report 89425, LAAS, Dec. 1989.

[7] H. J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. *SIGPLAN Notices*, 26(6):157–164, 1991.

[8] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Lisp and Functional Programming 1984*, pages 256–262. ACM Press, 1984.

[9] E. C. Cooper and R. P. Draves. C threads. Technical report CMU-CS-88-154, Carnegie Mellon University, 1988.

[10] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[11] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Sholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercice in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[12] X. L. *et al.* The Caml Light system, release 0.6. Software and documentation distributed by anonymous FTP on `ftp.inria.fr`, 1993.

[13] R. H. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *Lisp and Functional Programming 1984*, pages 9–17. ACM Press, 1984.

[14] T. Hickey and J. Cohen. Performance analysis of on-the-fly garbage collection. *Commun. ACM*, 27(11):1143–1154, 1984.

[15] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *Foundations of Computer Science 1977*, pages 120–131. IEEE Computer Society Press, 1977.

[16] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

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

[18] D. A. Moon. Garbage collection in a large Lisp system. In *Lisp and Functional Programming 1984*, pages 235–246. ACM Press, 1984.

[19] J. G. Morriset and A. Tolmach. A portable multiprocessor interface for Standard ML of New Jersey. Technical report CMU-CS-92-155, Carnegie Mellon University, 1992.

[20] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In *Functional Programming Languages and Computer Architecture 1987*, volume 242 of *Lecture Notes in Computer Science*, pages 113–133. Springer-Verlag, 1987.

[21] L. C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[22] J. H. Reppy. CML: a higher-order concurrent language. *SIGPLAN Notices*, 26(6):294–305, 1991.

[23] G. L. Steele Jr. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.

[24] D. Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. ACM Press, 1984.