# A Concurrent Copying Garbage Collector for Languages that Distinguish (Im)mutable Data

Lorenz Huelsbergen and James R. Larus*
University of Wisconsin–Madison†

## Abstract

This paper describes the design and implementation of a concurrent compacting garbage collector for languages that distinguish mutable data from immutable data (*e.g.*, ML) as well as for languages that manipulate only immutable data (*e.g.*, pure functional languages such as Haskell). The collector runs on shared-memory parallel computers and requires minimal mutator/collector synchronization. No special hardware or operating system support is required.

Our concurrent collector derives from sequential semi-space copying collectors. The collector requires that a heap object includes a *forwarding pointer* in addition to its data fields. An access to an immutable object can be satisfied either by the original or the forwarded copy of the object. A mutable object is always accessed through the forwarded copy, if one exists.

Measurements of this collector in a Standard ML compiler on a shared-memory computer indicate that it eliminates perceptible garbage-collection pauses by reclaiming storage in parallel with the computation proper. All observed pause times are less than 20 milliseconds. We describe extensions for the concurrent collection of multiple mutator threads and refinements to our design that can improve its efficiency.

*This work was supported in part by the National Science Foundation under grant CCR-9101035. L. Huelsbergen is supported by an ARPA fellowship in parallel processing.

†Authors's address: Department of Computer Sciences, 1210 West Dayton Street, Madison, Wisconsin 53706. Email: {lorenz,larus}@cs.wisc.edu

## 1 Introduction

The convenience of programming languages with implicit storage allocation (*e.g.*, Lisp[19], ML [20], Haskell [17]) is tempered by the run-time cost of reclaiming discarded storage. Automatic recycling of storage is known as garbage collection. A garbage collector retains a program's data that are in-use (*live*) and reclaims data that are unreachable (*garbage*) by the program. Garbage collection is necessary since even large (virtual) memories are finite. In addition to reclaiming storage, a garbage collector can also restore *locality* to fragmented data by dynamically compacting it, thereby improving the performance of caches and virtual memory systems and reducing storage requirements [10, 12]. Automatic storage reclamation, however, introduces disruptive pauses into interactive programming environments and slows program execution.

Parallel computers offer the potential to remove much of the garbage-collection overhead by delegating storage reclamation to processors running in parallel with a program [24, 6, 14]. We show how programming languages such as ML and Haskell can harness parallel machines to make their implementations more responsive. This in turn enables programmers to use these languages to develop interactive systems (operating systems, editors, programming environments, *etc.*).

Our concurrent garbage collector uses a single thread, the *collector*, to reclaim storage for a single program, the *mutator*. This concurrent approach potentially reduces the collection overhead for sequential programs since the collector continually recycles the garbage that a program running on other processors creates. Hence, existing sequential programs can benefit directly from this approach. Furthermore, in an interactive programming environment, sequential program execution will not be

disrupted by garbage collection if the concurrent processor reclaims garbage at least as quickly as the program creates it. To collect garbage in parallel systems, we describe extensions to our design that support multiple mutators.

Since a concurrent compacting collector relocates dynamic data and allows concurrent mutator access to these data, the cost of data access and allocation operations must be close to their cost with efficient sequential collection. To this end, we exploit the compile-time distinction between mutable and immutable data in some mostly-functional languages, such as ML. Pure functional languages (Pure Lisp, Haskell, *etc.*) can also be concurrently collected by this approach since they manipulate only immutable data.

Allocation in our design only requires initialization of an additional pointer in each object. Access to immutable data is as efficient as in a sequential implementation; since the data are immutable, they are constant, even if copied. The cost to access mutable data increases slightly in our design.[1] This is because a modification to a mutable datum requires that the modification always occur in the forwarded copy of the datum if one exists. If no copy exists, the mutator modifies the original. The modification of the original is followed by an additional check to ensure that the datum was not copied by the collector while the mutator performed the modification. If the collector's copy operation overlapped the mutator's modification, the modification is redone in the copy. Neither the mutator nor the collector explicitly lock the object since hardware and software locking is expensive; fine-grain synchronization based on a known order of memory operations is used instead.

We have implemented our design in the Standard ML of New Jersey (SML/NJ) optimizing ML compiler [7] on a shared-memory parallel computer. It successfully reduces mutator pause latency to a few milliseconds. Relative to simple sequential stop-and-copy collection [10, 12], our collector is very efficient. Its efficiency must, however, be further improved for it to outperform efficient sequential collectors based on generational scavenging [18, 26].

After stating the assumptions that underlie our design, we overview previous work on concurrent garbage collection (Section 2). Section 3 describes sequential copying collection before describing our concurrent copying collector. Section 4 describes an implementation of this collector in SML/NJ and Section 5 gives performance measurements of this implementation. Extensions to multiple mutators and further implementation improvements are in Section 6.

[1] Depending on a language's object-equality semantics, equality testing (pointer comparison) may also incur a slight additional overhead if an extra indirection is required to correctly determine equality.

### Assumptions

We assume that the mutator and collector share a common address space and that this common memory is *processor consistent* [13].[2] Reads and writes of machine-word quantities must be atomic. These assumptions allow ordered memory operations to implicitly perform fine-grain synchronization without explicit synchronization primitives (*cf.* lock-free synchronization [15]). The compiler must also distinguish mutable-data accesses from immutable-data accesses. This assumption allows the compiler to generate code for mutable accesses that implicitly synchronizes with the concurrent collector. Run-time separation of mutable and immutable data is not necessary. It is also assumed that the concurrent collector can inform the mutator that an event is pending via software interrupts or polling. This device is necessary since the mutator must periodically supply (upon collector request) its root set—the pointers that reach all live data. The mutator need not report its roots immediately, but may continue execution until gathering and reporting them is convenient. We assume that mutator allocation is heap based [1]. This simplifies the exposition of the collector's algorithms since it elides the details of stack management during collection. Finally, this design assumes that most mutator data accesses are to immutable data. This is often valid for programs written in a mostly-functional language and certainly valid for pure functional programs.

## 2 Previous Work

Various designs for concurrent garbage collectors have been proposed. Most of these require special hardware or operating system support; few have been implemented on parallel machines.

Our design is based on sequential copying collectors [10, 12]. Copying collectors are attractive since

[2] Processor consistency guarantees that writes from a processor $p$ appear to all other processors as occurring in the sequence in which $p$ executes them. For example, if processor $p_0$ executes the sequence (`x` $\leftarrow$ `0; x` $\leftarrow$ `1; x` $\leftarrow$ `2`) and $p_1$ reads the value `1` for `x` then $p_1$ cannot subsequently read the value `0` for `x`. Processor consistency is weaker than sequential consistency.

they operate in time proportional to the amount of data in use by the system. They remove fragmentation and restore locality by compacting data as it is copied. Copying collectors are more efficient than reference-counting and mark-and-sweep collectors in theory and practice [8, 1]. Zorn provides analyses and simulations of mark-and-sweep and stop-and-copy collectors [27].

Sequential implementations of mostly-functional languages provide fast object allocation and access [4, 5, 1]. To retain efficiency, concurrent collectors must not substantially increase the costs of object allocation and access. Central to this goal is Brooks's idea that all dynamic heap objects contain an extra field for a forwarding pointer [9]. His approach sets the forwarding pointer of an object $a$ to point to $a$ itself if the object has not been copied. When the collector copies the object to a new object $a'$, $a$'s forwarding pointer is set to point to $a'$. Accesses to $a$ simply indirect through $a$'s forwarding pointer.

Our collector implements Brooks's forwarding pointer, but indirects through it only on access to mutable data. This is possible due to the assumption that the language being collected separates mutable data from immutable data at compile-time. In mostly-functional languages, indirection is rarely necessary. The Pegasus system's garbage collector [23] also employs Brooks's technique, but does not provide a concurrent implementation and the outlined concurrent design relies on the explicit locking of objects.

Real-time response is highly desirable in garbage-collected systems. Baker's algorithm [8] addresses this issue by interleaving allocation with collection, but his algorithm requires special hardware to run efficiently. Halstead [14] extended Baker's algorithm to shared-memory multiprocessors by using fine-grain synchronization on individual objects. Efficient implementation of Halstead's algorithm relies heavily on hardware support. A concurrent lock-free version of Halstead's algorithm is given by Herlihy and Moss [16]. Their algorithm never performs global synchronization. This, however, is achieved only at the cost of expensive object access and allocation. Again no implementation is provided.

Appel, Ellis, and Li implemented the first concurrent copying collector on a stock shared-memory multiprocessor [6]. It directly addresses the efficiency and response-time issues. The Appel-Ellis-Li collector uses operating-system memory-protection devices to detect mutator references to heap objects that have not been collected. Their collector does not require compiler cooperation. Response latency is on the order of hundreds of milliseconds—but, as noted by the authors, this is still too slow for some interactive applications. Efficiency of their concurrent collector is measured only in terms of a simple sequential stop-and-copy collector, not in relation to common generational collectors [18, 26] that provide very efficient sequential collection.

Doligez and Leroy implemented a hybrid concurrent collector for ML [11]. A copying collector reclaims storage in a processor's local memory and a mark-and-sweep algorithm collects shared global memory. Since mark-and-sweep collection fragments storage, the shared-memory objects in this design must eventually be relocated—the authors do not provide a concurrent relocation algorithm. Worst-case pause latencies of their collector, not including shared-data relocation costs, are on the order of hundreds of milliseconds.

Our collector provides excellent response; average response latency is a few milliseconds. Furthermore, our collector requires no hardware or operating system support and only minimal compiler cooperation. We provide empirical evidence that this collector is more efficient than sequential stop-and-copy collectors and describe refinements that can improve its efficiency to that of fast generational collectors [3] used in sequential implementations.

# 3 The Collector

In this section we describe the basic structure of sequential copying collection (§3.1) and then our concurrent copying collector (§3.2).

## 3.1 Sequential Copying Collection

A basic sequential copying collector [10, 12] requires two memory spaces of equal size. Denote these spaces as *From-Space* and *To-Space*. Figure 1 illustrates these spaces and the pointers that manage them. Mutator allocation of heap objects always occurs in *From-Space* at the next unused location (*New*). If allocation crosses a predetermined *Threshold*, *From-Space* is deemed full and garbage collection occurs. The *From-Space* in Figure 1a is full and requires collection.

At the beginning of a collection, *Scanned* and *Unscanned* point to the bottom of (empty) *To-Space*. *Scanned*/*Unscanned* form a queue for a breadth-first traversal of the live data in *From-Space*. *Scanned* points to the head of the queue

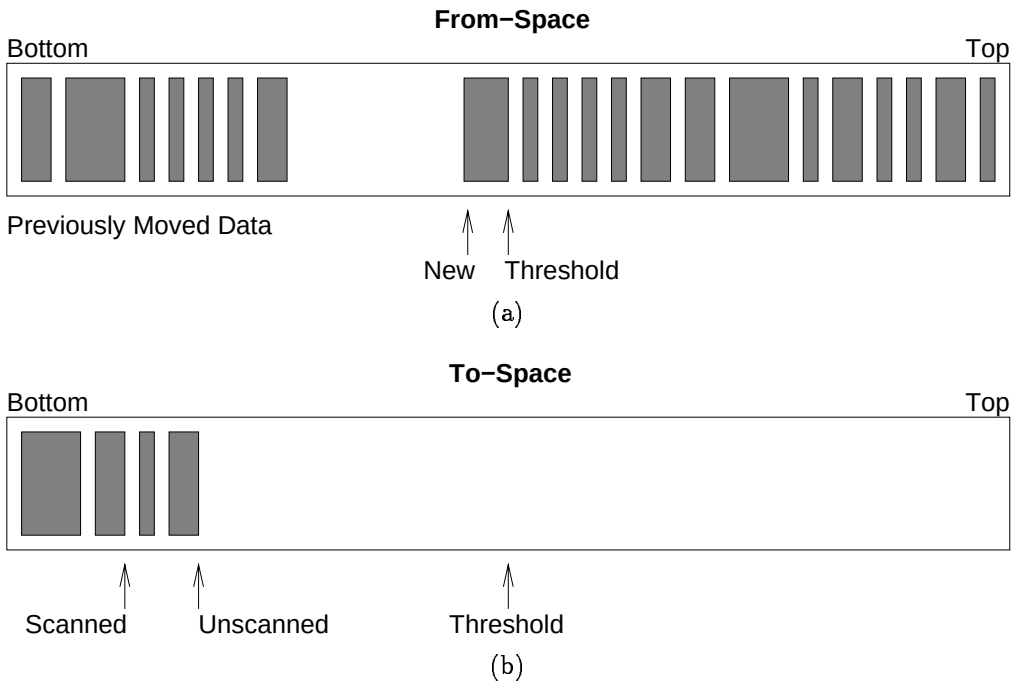



**Figure 1:** *From-Space* (a) and *To-Space* (b) and associated pointers for a sequential copying collector. After mutator allocation at *New* crosses *Threshold* in *From-Space*, the mutator halts and all live data are contiguously copied into *To-Space*. After copying, the spaces flip roles and mutator computation resumes.

and *Unscanned* to the tail. This queue is initialized with the objects pointed to by the *roots* of the mutator. Any pointer into the active heap (*From-Space*) that the mutator has access to (*e.g.*, registers) is a root and all such pointers constitute the mutator's root set. All live data available to the program is reachable via pointers from this root set. After an object's relocation in *To-Space*, the collector overwrites the original object in *From-Space* with the object's new location in *To-Space* (its *forwarding* pointer).

Objects in the initialized *Scanned*/*Unscanned* queue are scanned (from *Scanned* to *Unscanned*) for pointers into *From-Space*. A previously uncopied *From-Space* object is inserted into the queue and forwarded. A pointer to a previously copied object is translated to point to the copy using the forwarding pointer. All live data resides in *To-Space* when the queue of unscanned objects empties; that is, when *Scanned* = *Unscanned*. A *flip* operation then reverses the roles of the two spaces: *To-Space* becomes *From-Space* and *From-Space* becomes *To-Space*. The collection is now complete and the collector supplies an updated root set to the mutator. Mutator allocation proceeds anew from the top of the (empty) *From-Space*.

## 3.2 Concurrent Copying Collection

Our concurrent collector design is based on the sequential copying algorithm. It also copies live data between regions, but does not suspend mutator computation during the copying phase. Two modifications to the underlying data structures are required: objects contain an additional field, and the layout of *To-Space* is altered. First, every object $r$ created by either the mutator or collector must contain an additional field $f$ for a forwarding pointer. $f$ is distinct from $r$'s data fields. This is in contrast to implementations of the sequential algorithm that are free to overwrite $r$'s data fields with forwarding information. We require an additional field since the contents of $r$ and the contents of copies of $r$ are simultaneously accessible to the mutator. Let $A$ denote all address values for $f$. Distinguish four values in $A$: *Fresh*, *BeingTranslated*, *Translated*, *BeingCopied*. Partition $A$ into mutator *inaccessible* and *accessible* state sets:

$$inaccessible = \{BeingTranslated, BeingCopied\}$$

$$accessible = A - inaccessible$$

An object $r$ is always created with its forwarding pointer set to *Fresh*. Figure 2 shows the possible

$$Fresh \rightarrow \boxed{BeingTranslated} \rightarrow Translated \Rightarrow \boxed{BeingCopied} \rightarrow \text{(forwarding address)}$$

**Figure 2:** Possible transitions an object can undergo. The $\Rightarrow$ indicates that a flip occurs during this transition. A $\boxed{Boxed}$ state is one in which an object is inaccessible to the mutator.

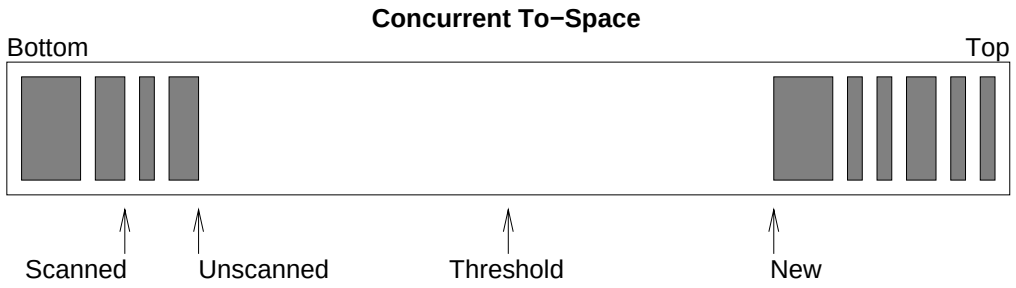



**Figure 3:** Concurrent *To-Space*. This *To-Space* is similar to the sequential *To-Space* (Figure 1b), but allows the mutator to allocate new data at *New* while the collector concurrently scans and inserts objects into the *Scanned/Unscanned* queue. The collector is also responsible for translating *From-Space* pointers in freshly-allocated data (from *New* to *Top*). Concurrent *From-Space* is identical to the sequential *From-Space* (Figure 1a).

transitions for $r$'s forwarding pointer. The mutator may access a mutable object only when it is in an accessible state. If the state of the object changes during the access, the mutator must redo the access. A change in state is easily detectable since the transitions can be implemented using a monotonic representation (*i.e.*, with reserved address values for the states). An access is performed in the forwarded copy if the state of the object is a true forwarding address. This synchronization is extremely fast—only two fetches from an immutable object are required in the expected case of no mutator/collector conflict. Figure 5 contains pseudo-code for mutator read and write operations to mutable data.

The other modification is to the heap's *To-Space*. Figure 3 illustrates the layout of the concurrent *To-Space*.[3] This change is also conceptual—mutator allocation occurs in *C.To-Space* and not in *C.From-Space* as per the sequential algorithm. This is necessary to properly identify and collect all live data while permitting concurrent mutator allocation.[4]

Figure 4 contains the concurrent copying collector.[5] As the **repeat** loop in *Collect* indicates, the collector runs continually. The algorithm divides into three logically-related parts: flipping (a)–(e), copying live data (f)–(m), and translating old *C.From-Space* pointers into their *C.To-Space* equivalents (n)–(p). We describe these parts in turn.

Lines (a)–(c) prepare the collector for interchanging *C.From-Space* and *C.To-Space*. Only during this part of the algorithm is the mutator inactive. It is, therefore, important that this section execute quickly. At line (a) it is known that all live data resides in *C.To-Space* (newly-allocated data is placed in *C.To-Space* at *C.New*). Pointers to *C.From-Space* may, however, still exist in the new allocation arena (from *C.New* to *C.To-Space.Top*). Since all live data has been copied out of *C.From-Space*, it is only necessary to translate these *C.From-Space* pointers in the allocation arena. This is accomplished at line (c). Since the new-data allocation arena may be large, it is not possible to translate it in its entirety without pausing the mutator for an exorbitant amount of time—overhead we are attempting to avoid in the first place. For this reason, most of the allocation arena is translated concurrently (lines (n)–(p)) with mutator allocation. We discuss this portion of the algorithm below. *TranslateAllocArena* at line (c) only translates a small piece of this arena, namely the data not translated after the last iteration of the loop at line (n). Line (d) performs the flip and repositions the pointers that govern the semi-spaces.

[3] We preface global identifiers used for sequential collection with *S.*<*identifier*> and global identifiers used for concurrent collection with *C.*<*identifier*>.

[4] If the mutator were to allocate in *C.From-Space*, it would be difficult to determine when all live data has been copied from *C.From-Space* since the mutator's root set continually changes.

[5] The predicate *Forwarded?*($p$) is true if the object at $p$ has been forwarded. *Forward*($p$,$q$) forwards the object at $p$ to location $q$ and *Translate*($p$) returns the forwarding address of the object at $p$ if one exists, $p$ otherwise. *SizeOf*($p$) returns the number of fields in the object at $p$.

```
Collect()
    volatile Roots: set of pointers
    repeat
        begin                                          (* copies of all live data in C.To-Space *)
(a)         suspend mutator
(b)         Roots ←mutator's root set
(c)         Roots ← TranslateAllocArena(Roots)         (* translate remainder of allocation arena *)
(d)         Flip(C.From-Space, C.To-Space)             (* all pointers are into C.To-Space before flip *)
(e)         release mutator                            (* mutator allocates in (empty) C.To-Space *)
(f)         C.Scanned ← C.To-Space.Bottom              (* initialize live-data queue *)
(g)         C.Unscanned ← C.Scanned
(h)         for all r ∈ Roots do                       (* Roots reach all live data in C.From-Space *)
(i)             r ← Copy&Forward(r)                    (* copy root objects into queue *)
(j)         while C.Scanned ≠ C.Unscanned do           (* perform breadth-first copy *)
                let object = object at C.Scanned
                    ptrs = set of pointers in object
                in
(k)                 Forward(object, BeingTranslated)   (* indicate impending modification to object *)
                    for all p ∈ ptrs do
                        p ← Copy&Forward(p)
                    C.Scanned ← C.Scanned + SizeOf(object)
(l)                 Forward(object, Translated)        (* modification done *)
(m)             end
(n)         while mutator still allocating do          (* translate pointers in allocation arena *)
(o)             Roots ←mutator's root set              (* until mutator runs out of space *)
(p)             TranslateAllocArena(Roots)
        end

Copy&Forward(p : object pointer) : object pointer
(q) if not Forwarded?(p) then
        begin
(r)         Forward(p, BeingCopied)                    (* indicate copy operation to mutator *)
(s)         copy object at p to C.Unscanned
(t)         Forward(C.Unscanned, Fresh)                (* new copy is not forwarded *)
(u)         Forward(p, C.Unscanned)                    (* inform mutator that copy is complete *)
(v)         C.Unscanned ← C.Unscanned + SizeOf(C.Unscanned)
(w)         return C.Unscanned − SizeOf(C.Unscanned)
        end
(x) else return Translate(p)
```

**Figure 4:** Concurrent copying collector. *Collect* runs continually. Lines (a)–(c) ensure that all pointers point into *C.To-Space*. Lines (f)–(m) perform the copying proper. Lines (k) and (l) indicate to the mutator that the object is undergoing modification. Lines (n)–(p) incrementally translate *C.From-Space* pointers in freshly-allocated data into their *C.To-Space* equivalents. *Copy&Forward* communicates to the mutator at line (r) that the object pointed to by $p$ is about to be copied and at line (u) that the copy is complete.

```
Read(r : object pointer,i : integer) : value
    if Inaccessible(r) ∨ Forwarded?(r) then
        Read(Translate(r),i)
    else
        let  s = State(r)
             v = field i of r
        in
             if s ≠ State(r) then
                 Read(r,i)
             else v
        end
```

```
Write(r : object pointer,i : integer,v : value)
    if Inaccessible(r) ∨ Forwarded?(r) then
        Write(Translate(r),i,v)
    else
        let  s = State(r)
        in
             set field i of r to v
             if s ≠ State(r) then
                 Write(r,i,v)
        end
```

**Figure 5:** Mutator mutable-data access operations to *Read* (*Write*) field $i$ of an object $r$. If the concurrent collector has forwarded $r$, the read (write) is performed in the copy. If the collector copies or updates $r$ while the mutable access is underway, the access is redone.

Lines (f)–(m) perform the actual copying and forwarding of live data from *C.From-Space* to *C.To-Space*. *Copy&Forward* provides implicit synchronization information to the mutator (*cf.* Figure 5) by setting the object's state to *BeingCopied* before the actual copy ensues (r) and making it accessible after the copy is complete (line (u)). Similar synchronization is also performed during the scan of an object (line (k) and (l)).

All live data resides in *C.To-Space* when control reaches line (n); that is, all live data from *C.From-Space* has been copied and forwarded into *C.To-Space* and all data concurrently allocated by the mutator since the last flip is allocated in *C.To-Space*. However, pointers into *C.From-Space* may still exist in the fresh allocation arena (*C.New* to *C.From-Space.Top*) and possibly in the copied data (*C.To-Space.Bottom* to *C.To-Space.Unscanned*). This latter case occurs if the mutator updates a mutable location in this region *after* it was scanned and translated by the collector. These *C.From-Space* pointers in the copied area are identified by requiring the mutator to maintain a list of mutable locations into which it stores pointers.[6]

*TranslateAllocArena* is an incremental depth-first marking algorithm.[7] *TranslateAllocArena* is given a set of pointers to translate. It recursively translates all *C.From-Space* pointers reachable from its parameter into their *To-Space* equivalent and returns a translated version of its parameter. For synchronization, *TranslateAllocArena* uses the *BeingTranslated* and *Translated* states in a manner similar to the scanning and copying portions of the collector. The algorithm is incremental in that successive invocations of *TranslateAllocArena* do not reexamine objects that were previously translated. This insures only short pauses when translating reachable, yet untranslated, objects immediately before a flip (line (d)).

[6] *Store lists* are used in generational collectors to detect pointers from older generations to younger ones [26, 3].

[7] The stack required for depth-first marking can be allocated explicitly or *threaded* through garbage objects in *C.From-Space*.

# 4 Implementation

We have implemented our concurrent collector in the SML/NJ 0.73 optimizing ML compiler [7]. ML has higher-order functions, parametric polymorphic types, statically-checked strong typing, and side-effecting operators that modify reference variables and perform IO [20, 21]. The SML/NJ implementation is garbage collected by an efficient generational stop-and-copy collector [3].

Our concurrent implementation runs on a Sequent Symmetry with 40MB of shared memory. Of the Sequent's 20 processors, we use only two—one to execute the program (the mutator) and one to collect the program's garbage (the collector). Each of the Sequent's 80386 processors is connected via a 64KB two-way set-associative cache to a bus to shared memory. The *sml2c* [25] back end to SML/NJ generates portable C code which is compiled with a native 80386 C compiler.

The compiler was modified to allocate an extra word as the forwarding pointer with all objects. Upon object allocation, this pointer is initialized to *Fresh*. The compiler's code generator was modified to include synchronization instructions on access to mutable data and to indirect through forwarding pointers when two object pointers are compared

| Knuth-Bendix (16MB) | | | |
|---|---|---|---|
| | Stop-&-Copy | Generational | Concurrent |
| Exec | 200.7 s | 200.4 s | 270.9 s |
| GC | 117.1 | 5.2 | 0.04 |
| Sys | 3.6 | 2.7 | 7.0 |
| Elapsed | 320.9 | 208.2 | 277.9 |
| Max Pause | 7380 ms | 420 ms | 10 ms |
| Avg Pause | 2251 | 199 | 1 |

| QuickSort (1MB) | | | |
|---|---|---|---|
| | Stop-&-Copy | Generational | Concurrent |
| Exec | 32.3 s | 32.4 s | 40.0 s |
| GC | 16.9 | 4.1 | 1.4 |
| Sys | 0.5 | 0.3 | 0.4 |
| Elapsed | 49.7 | 36.8 | 41.8 |
| Max Pause | 220 ms | 180 ms | 20 ms |
| Avg Pause | 98 | 31 | 13 |

**Table 1:** Timing results for sequential and concurrent garbage collection schemes. *Stop-&-Copy* is a simple sequential collector. *Generational* is an efficient two-generation collector. *Concurrent* is the concurrent copying collector described in this paper. Resolution of the clock is ten milliseconds.

for equivalence. Since the backend of the compiler emits C code, these additions were implemented as C functions. Some of the performance degradation due to this implementation can be alleviated by performing these operations in line.

The collector is written in C. The SML/NJ runtime system, the concurrent collector, and the program (produced by *sml2c*) are linked into an executable image. Upon initiation, this image creates two parallel processes that concurrently execute the collector and mutator threads. Communication among the mutator and the collector occurs through shared memory. The collector acquires the mutator's roots by setting a single location. The mutator polls this location when it performs a check for sufficient heap space. Upon receiving such a request, the mutator copies its roots into a shared vector and communicates to the collector that the roots are in place. The collector communicates new root sets to the mutator via the same vector.

# 5 Results

This section presents performance measurements of the concurrent collector for two benchmark programs.[8] We compare the absolute performance of the concurrent collector to that of two sequential collectors: a simple stop-and-copy collector and a fast copying collector [3] that maintains two generations of live data. The heap size was fixed[9] for the entire execution of a program and was set to the smallest size in which the concurrent collector always completes its copying phase before the mutator exhausts its allocation arena.

The first benchmark implements the Knuth-Bendix completion algorithm processing some axioms of geometry (*cf.* [5]). It performs side effects in the form of IO. IO buffers are dynamically allocated heap objects in SML/NJ and subject to collection and hence require mutator synchronization when they are read or written. Table 1 gives times for collecting the garbage using the concurrent and the two sequential collectors. The concurrent collector performs extremely well in reducing the amount of time the mutator spends in the garbage collector (less than 1% of the stop-and-copy or generational times). It only creates low latency disruptions—average pause times almost two orders of magnitude smaller than the pauses exhibited by the Appel-Ellis-Li collector [6].

Our implementation of the concurrent algorithm does, however, adversely affect the overall efficiency of the program. A comparison of the total elapsed times shows the concurrent design to improve on the simple collector by 13%. However, the sequential generational scheme is almost 34% faster than the concurrent scheme even though it uses only one processor. The difference in elapsed times stems entirely from the difference in execution times. Reasons for these differences and possible solutions are discussed below (§6). The Appel-Ellis-Li collector [6] attained an 18% improvement over a simple stop-and-copy collector on a single benchmark, but was not compared to a generational collector.

The second program we timed is a QuickSort applied to a list of 1024 random integers (Table 1). Here concurrent collection is more efficient than in the Knuth-Bendix benchmark, but is still 12% slower than the sequential generational collector. As with Knuth-Bendix, concurrent collection of QuickSort exhibits low pause latencies: worst-case pause latency is 9 times lower than generational

[8] All timings were performed on a Sequent Symmetry with no load. The C program files produced by *sml2c* and the SML/NJ runtime system were compiled at the "-O" optimization level. When compiling the concurrent version, the compiler was also instructed to treat all variables as *volatile* since we use the Sequent's processor-consistent memory to synchronize mutator/collector interaction. Volatile variables restrict many compiler optimizations.

[9] SML/NJ normally resizes the heap dynamically.

pause latency and average pause latency is less than half as long. Mutator time in the collector is again very low for the concurrent collector when compared to both sequential collectors.

# 6 Future Work

This section addresses the concurrent collector's inefficiencies and extends the design to multiple parallel mutators.

Inefficiencies in the concurrent collector are due to two factors: (1) overhead incurred by the mutator to correctly cooperate with the collector, and (2) decreased locality inherent in distributing work among parallel processors. An inefficiency in our algorithm is mutator overhead due to forwarding pointer creation and manipulation. The additional forwarding pointer on every heap object increases the amount of mutator computation on allocation and decreases heap space available for program data.[10] Depending on the language's implementation, the forwarding pointer can be integrated with the object's usual fields. For example, in SML/NJ all object's carry a word of tag information. It is possible to merge the forwarding pointer with this tag (*cf.* [22]), thereby reducing object allocation costs and improving memory utilization.

Performance degradation due to disruption of locality is a more subtle point. For example, the concurrent collector disrupts mutator locality when it installs a forwarding pointer in an object that resides in the mutator's data cache. Furthermore, all data constantly moves between the mutator's and collector's caches. We suspect that this effect accounts for a large portion of the inefficiencies in our design. Incorporating generational techniques into this concurrent collector is paramount to improving its efficiency (*cf.* [6, 11]).

Our concurrent collector can be extended to reclaim the discarded storage of $p$ parallel mutators as follows. On a flip, the collector partitions the new allocation arena (Figure 3) into $m$ blocks of size $k$ such that $m \gg p$. Each mutator is given a block in which to allocate. When a mutator exhausts the storage in its current block, it requests an additional block from the collector. In addition to dispensing heap blocks, the collector concurrently copies and translates live data (as in the case $p = 1$ described in this paper); the collector successively transmits requests for root sets to the $p$ mutators. After dispensing $m - p$ blocks, the collector halts each mutator $p_i$ in turn, completes the translation of $p_i$'s untranslated reachable data, hands a fresh block to $p_i$ (there are $p$ unused blocks in the allocation arena since the collector only dispenses $m - p$ blocks), and finally releases $p_i$. The flip occurs after the collector processes all $p$ mutators in this manner. No global synchronization is required under this scheme.

For a large number of parallel mutators, the collector must also be distributed among multiple processors.

[10] Objects in SML/NJ occupy approximately 2.9 words in the heap on average [2]. Adding an additional field to each object therefore reduces heap space by approximately one quarter.

# 7 Conclusion

We have designed and built a concurrent garbage collector. Our collector is derived from sequential copying collectors and exploits the compile-time separation of mutable and immutable data that some mostly-functional languages such as ML provide. The collector requires no special hardware or operating system support and is extensible to multiple parallel mutators. Experiments indicate that collector pause times are very short—less than 20 milliseconds. Our collector is more efficient than simple sequential collectors and we describe modifications to our design that potentially make it competitive with fast generational collectors.

# Acknowledgement

Thanks to the anonymous referees for detailed suggestions.

# References

[1] A. W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25:275–279, 1987.

[2] A. W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[3] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice & Experience*, 19(2):171–183, February 1989.

[4] A. W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3:343–380, 1990.

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

[6] 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.

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

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

[9] 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*, pages 256–262. Association for Computing Machinery, August 1984.

[10] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[11] 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.

[12] 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.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 15–26. Association for Computing Machinery, May 1990.

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

[15] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, January 1991.

[16] M. P. Herlihy and J. E. B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.

[17] P. Hudak, S. P. Jones, P. Wadler, et al. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

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

[23] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 113–133. Springer-Verlag, 1987.

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

[25] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

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

[27] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 87–98, Nice, France, June 1990.