Real-time Concurrent Collection on Stock Multiprocessors

Andrew W. Appel[1]
John R. Ellis[2]
Kai Li[1]

Abstract

We have designed and implemented a copying garbage-collection algorithm that is efficient, real-time, concurrent, runs on commerial uniprocessors and shared-memory multiprocessors, and requires no change to compilers. The algorithm uses standard virtual-memory hardware to detect references to "from space" objects and to synchronize the collector and mutator threads. We have implemented and measured a prototype running on SRC's 5-processor Firefly. It will be straightforward to merge our techniques with generational collection. An incremental, non-concurrent version could be implemented easily on many versions of Unix.

Introduction

This paper presents the first copying garbage-collection algorithm that is efficient, real-time, concurrent, runs on stock commercial uniprocessors and multiprocessors, and requires no change to compilers.

A collection algorithm is *efficient* if the amortized cost to allocate, access, and collect an object is small compared to the cost of initializing the object.

An algorithm is *real-time* if the mutator (the program) is never interrupted for longer than a very small constant time. A collector has small *latency* if the interruptions are short. An interactive workstation requires latencies of less than 0.1 second if collections are not to affect communications, mouse tracking, or animation on the screen.

An algorithm is *concurrent* if the collector can do its work in parallel with the mutator. A concurrent collector allows for multiple mutator threads (lightweight processes) and multiple processors. Concurrency pays even on a uniprocessor, since the collector can run while the mutator is waiting for external events such as user input, page faults, and i/o.

[1]Department of Computer Science, Princeton University, Princeton, New Jersey 08544
[2]Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, California 94301

An algorithm runs on *stock hardware* if it can run on commercial architectures such as the VAX and the 68000. We assume that the multiprocessors have an efficient shared memory.

Shared-memory multiprocessors are becoming widespread, so it's important to find efficient concurrent collection algorithms. With today's technology, the marginal cost of adding extra processors and caches to a machine is small. Most new large mainframes are multiprocessors, and the System Research Center's Firefly [Thacker 87] has shown that it is economical to build multiprocessor workstations.

Fine-grained synchronization between the collector and the mutator is a problem for concurrent collectors. Our algorithm uses the virtual-memory hardware of traditional processors to implement a medium-grained synchronization that is coarse enough to be efficient, yet fine enough to keep latency low. The full algorithm needs only slight modification to most Unix-like operating systems, and a real-time, sequential version can be built on many standard versions of Unix. Further, our algorithm will work with any compiler already geared for a copying collector.

Previous Work

In recent years, researchers have built collectors that are far less obtrusive to programs than the original "stop-the-world" collectors.

Reference-counting collectors first demonstrated that collection could be unobtrusive even to systems programs [Rovner 85a]. The current Modula-2+ collector [Rovner 85b] is real-time, concurrent, and runs on the Firefly, a shared-memory multiprocessor built from 5 MicroVAX II CPUs (1 MIP each) [Thacker 87]. But compared to copying collectors, it isn't particularly efficient, and it cannot reclaim circular structures.

No one has yet built a practical concurrent mark-and-sweep collector, and it may be difficult to make them efficient [Hickey 84].

Asymptotically, copying-based collectors have the potential to be much more efficient than reference-counting or mark-and-sweep collectors [Baker 78, Appel 87]. Copying collectors (including our algorithm) do work proportional only to the amount of reachable objects; thus, as the total available memory is increased relative to the number of reachable objects, the time to reclaim an object approaches zero in the limit. But reference-counting and mark-and-sweep collectors must do at least a small, constant amount of work to reclaim each object.

However, in practice other factors dominate the asymptotic behavior: the cost of allocation, the cost of counted assignments, the load placed on virtual memory. Copying collectors may increase locality of reference by compacting reachable objects and clustering related ones, but concurrent copying collectors may also have larger working sets due to the copying and the interleaved mutator and collector page references. Unfortunately, we still don't have enough experience with scaled-up implementations to make final comparisons.

Baker's algorithm [Baker 78] was the first real-time, copying-based collector. But it isn't concurrent, it requires special hardware, and its first implementation on Lisp machines wasn't efficient (users turned it off). Brooks [Brooks 84] showed how to implement Baker's algorithm on stock hardware at the cost of an extra word per object and extra instructions to initialize and reference objects.

Later generational collectors of various sorts [Moon 84, Ungar 86, Shaw 87] were much more efficient and provided much better average latency by reducing the load on virtual memory. But none of them are concurrent. Only Moon's is real-time, but it requires special hardware.

The Concert Multilisp collector [Halstead 84] generalized Baker's algorithm to a shared-memory multiprocessor. The algorithm is concurrent, in the sense that multiple threads can be executing at the same time, but as with Baker's algorithm, the execution of each thread is interleaved incrementally with calls to the collector. The collector requires special hardware support for acceptable efficiency.

The Pegasus collector [North 87] starts with Brooks's basic technique to make a concurrent collector that runs on stock uniprocessors. But it relies on basic assumptions that would prevent it from running on a multiprocessor. Like Brooks's algorithm, objects require an extra word and an extra indirection on every reference. Also, assigning a pointer into an already initialized object is very expensive, making the algorithm suitable only for programming styles that are mostly applicative.

Stop-and-Copy Collection

In the simplest stop-and-copy collector, memory is divided into two contiguous regions, *from-space* and *to-space*. At the beginning of a collection, all objects are in from-space, and to-space is empty. Starting with the registers and other global roots, the collector traces out the graph of objects reachable from the roots. As each object is visited, it is copied into to-space. When there are no more objects to visit, all the reachable objects have been copied into to-space, and all the objects remaining in from-space are garbage. At that point, the roles of to-space and from-space are reversed (a *flip*), and the mutator resumes allocating from the new to-space.

When an object is copied from from-space, a distinguished *forwarding pointer* is left in its place, pointing to that object's new location in to-space. If the collector encounters another pointer to that from-space object, it updates the pointer to refer to the object's new location in to-space. Examining a pointer into from-space, copying the referenced object if necessary, and updating the pointer is called

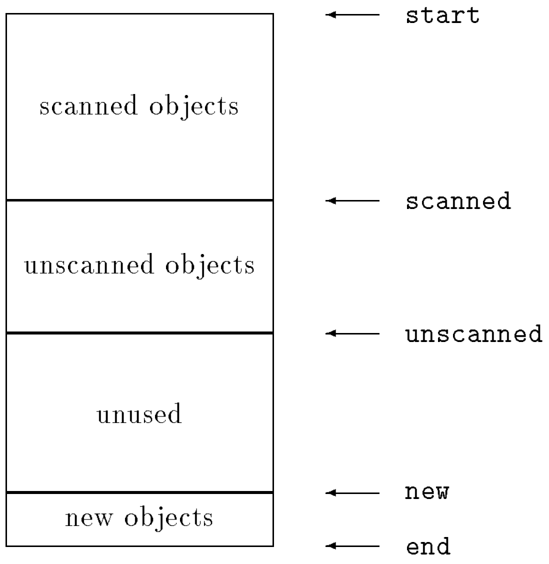

Figure 1: To-space

forwarding.

To-space is partitioned by three pointers (figure 1). During a collection, objects are copied from from-space to the end of the unscanned area (from `scanned` to `unscanned`), which grows down. Starting at the `scanned` pointer, the collector scans the objects in the unscanned area, looking for pointers to from-space objects. When it finds such a pointer, it copies the object to to-space (if it hasn't already been copied), and updates the pointer to point at the object's new location.

When `scanned` meets up with `unscanned`, there are no more reachable objects to be copied from from-space, and the collection is finished. Everything remaining in from-space is garbage. The mutator resumes and starts allocating objects in the new area (from `end` to `new`), which grows up. When to-space fills up, the mutator stops and initiates a new collection.

Stop-and-copy collectors can be very efficient, especially when the total available memory is much larger than the amount of live objects [Baker 78, Appel 87]. But they are unsuitable for real-time or interactive applications because of the long delay while the mutator is suspended.

Baker's Algorithm

In Baker's real-time algorithm [Baker 78], when to-space fills up, the collector stops the mutator, flips, but then copies only the root objects (for example, those referenced from the registers). It then resumes the mutator immediately. Reachable objects are copied incrementally from from-space while the mutator executes. Every time the mutator allocates a new object, it invokes the collector to copy a few more objects from from-space.

The mutator sees only to-space pointers in its registers, and thus can reference to-space objects only. To satisfy this invariant, every pointer fetched from an object must be checked to see if it points to from-space. If it does, the from-space object is copied to to-space and the pointer updated; only then is the pointer returned to the mutator. This checking requires hardware support to be implemented efficiently.

Baker's algorithm stops the mutator at a flip only long enough to copy the root objects. Every fetch and allocation is slowed down by a small, bounded amount of time. Thus Baker's algorithm is more suitable for real-time applications.

In the absence of hardware support, Baker's algorithm is not very efficient, since a few extra instructions must be performed on every fetch. Brooks's variant [Brooks 84] is intended to be efficient on stock hardware; but it requires an extra word per object, it does an extra memory reference on each fetch, and it requires a few additional instructions to implement the basic operations of comparing pointers, replacing the contents of a cell, and initializing a new object.

Neither Baker's algorithm nor Brooks's variant is concurrent—the mutator stops whenever the collector does a bit of work. Implementing a concurrent version on a multiprocessor would require fine-grain locking on each object, adding more overhead.

Our Algorithm

Instead of using microcode to check every pointer fetched from memory, our algorithm uses virtual-memory page protections to detect from-space memory references by the mutator.

We take advantage of (and maintain) the following invariants maintained by the copying algorithm:

> The mutator sees only to-space pointers in its registers.
>
> Objects in the new area contain to-space pointers only (because new objects are initialized from the registers).
>
> Objects in the scanned area contain to-space pointers only.

Objects in the unscanned area contain both from-space and to-space pointers.

Thus, only the unscanned area can contain from-space pointers.

The collector sets the virtual-memory protection of the unscanned area's pages to be "no access." Whenever the mutator tries to access an unscanned object, it will get a page-access trap. The collector fields the trap and scans the objects on that page, copying from-space objects and forwarding pointers as necessary. Then it unprotects the page and resumes the mutator at the faulting instruction. To the mutator, that page appears to have contained only to-space pointers all along, and thus the mutator will fetch only to-space pointers to its registers.

The collector also executes concurrently with the mutator, scanning pages in the unscanned area and unprotecting them as each is scanned. The more pages scanned concurrently, the fewer page-access traps taken by the mutator.

The virtual-memory hardware provides an efficient, medium-grained synchronization between the collector and the mutator. Because the mutator doesn't do anything extra to synchronize with the collector, compilers needn't be reworked. Multiple processors and mutator threads are accommodated with almost no extra effort, since the operating system must support concurrent operations on virtual memory anyway.

Initially, we'll assume that objects are no larger than a page and that the collector never copies an object so that it crosses a page boundary. Later we'll relax these constraints.

Page Traps and Concurrent Scanning

The collector uses two threads, one to handle access traps and one to scan the unscanned area concurrently with the mutator. A single lock protects the use of the `scanned`, `unscanned`, and `new` pointers by the two threads and the allocator.

The trap thread executes:

```
LOOP
    thread, pageAddress := WaitForTrappedThread()
    LOCK lock DO
        ScanPage( pageAddress )
    ResumeThread( thread )
```

It waits for a page-access trap from a mutator thread, grabs the lock, scans and unprotects the page, and then resumes the thread.

The scanner thread scans unscanned pages continually:

```
LOOP
    LOCK lock DO
        WHILE NOT (scanned < unscanned) DO
            WAIT( lock, unscannedPages )
        ScanPage( scanned )
        scanned := MIN( scanned + PageSize, unscanned )
```

Once `scanned` catches up with `unscanned`, there will be no more objects to be copied from from-space until after the next flip. In that case, the scanner thread blocks until the flip signals the condition variable `unscannedPages`, indicating that there are more pages to be scanned.

The `ScanPage` procedure is called by both the trap and scanner threads with the address of a page of objects to be scanned:

```
ScanPage( objectAdr ) =
    page = Page( objectAdr )
    IF Unprotected( page ) THEN RETURN

    WHILE Page( objectAdr ) = page AND
          objectAdr < unscanned
    DO
        ScanObject( objectAdr )
        objectAdr := objectAdr + ObjectSize( objectAdr )

    Unprotect( page )
```

If the page is unprotected, that means its objects have already been scanned as the result of a previous page trap, so `ScanPage` returns immediately.

Otherwise, `ScanPage` scans successive objects, forwarding from-space pointers by copying objects from from-space and advancing `unscanned`. As new pages are added to the unscanned area, they are immediately protected before any objects are copied into them. The scanning stops when either a page boundary is crossed or when `unscanned` is reached (meaning there are no more unscanned objects in to-space).

Finally, `ScanPage` unprotects the page so the mutator can reference it.

You might wonder how `ScanPage` could copy objects into a protected page or scan its contents; if the mutator can't read the page without trapping, how can the trap and scanner threads? `ScanPage` can't unprotect the page before scanning it because then the mutator, running concurrently, would be able to reference unscanned objects. Most stock architectures with virtual memory also provide two modes, kernel and user; the operating system runs in kernel mode, and ordinary programs run in user mode. Pages have two protections, one for kernel mode and one for user mode. By running the trap and scanner threads in kernel mode and the mutator in user mode, and by changing only the user-mode protections of pages, the collector threads can read and write pages not accessible to the mutator.

It wasn't hard to add two new kernel calls to the Firefly's operating system that implemented the trap and scanning loops. At program startup, the collector forks two threads which then make the kernel calls and which never return until the program halts.

Allocation

The Allocate procedure, called from the mutator, allocates a new object:

```
Allocate( size ) =
    LOCK lock DO
        unused := new - unscanned
        IF unused < size OR unused < FlipThreshold THEN
            Flip()
        new := new - size
        RETURN new
```

If the size of the unused area (between scanned and new) is too small or is less than a given threshold, the collector initiates a flip. Then it allocates the object and returns. (The `FlipThreshold` must be set large enough so that there is room for the collector to finish scanning, copying any remaining reachable objects into to-space [Baker 78].)

In our experiments, we quickly discovered that the collector and the mutator (via the allocator) were contending for the global `lock`. This contention would get worse with many mutator threads. So we modified `Allocate` to use a two-stage allocation, where each mutator thread would grab off a large chunk of storage and then allocate from the chunk without holding the lock. Only when `Allocate` needed another chunk would it get the lock and check for a flip:

```
Allocate( size ) =
    chunkLeft := chunkLeft - size
    IF chunkLeft < 0 THEN
        AllocateChunk()
    chunkNew := chunkNew - size
    RETURN chunkNew

AllocateChunk() =
    LOCK lock DO
        unused := new - unscanned
        IF unused < ChunkSize OR unused < FlipThreshold THEN
            Flip()
        chunkNew  := new
        chunkLeft := ChunkSize
        new       := new - ChunkSize
```

The variables `chunkNew` and `chunkLeft` are specific to each mutator thread; `chunkNew` points at the last allocated object in the chunk, and `chunkLeft` is the space remaining in the chunk.

Besides reducing contention for the global lock, this version of `Allocate` is small enough to be compiled inline. The thread-specific `chunkNew` and `chunkLeft` can be put in dedicated registers or in a thread-data area pointed to by one dedicated register. `Allocate` then compiles to just a few instructions, such as these VAX instructions:

```
subl2   size,chunkLeft
blss    CallAllocateChunk
subl2   size,chunkNew
```

Depending on the number of mutator threads and the actual value of `ChunkSize` relative to the cost of a page trap, it may be profitable to forego an explicit `chunkLeft`, using instead an inaccessible guard page at the end of the chunk. When the allocator tries to initialize an object in the guard page, it will trap; the trap handler can then grab a new chunk and resume the mutator.

But `ChunkSize` must be fairly big for the use of guard pages to be more efficient than explicit tests. And if there are a hundred or more threads (as there are in many Modula-2+ programs), it may not be practical to have chunks that are large enough. The chunks could be processor-specific instead of thread-specific (assuming there are far fewer processors than threads), but that introduces further complexity—allocations must be atomic relative to rescheduling, and implementing that on stock hardware would probably be at least as expensive as using an explicit test on the chunk size.

Flipping

The procedure `Flip` performs these steps:

> Stop all the mutator threads.
>
> Scan any remaining unscanned objects.
>
> Flip the roles of the two spaces, and initialize `start`, `end`, `new`, `scanned`, `unscanned`.
>
> Copy the root reachable objects from from-space.
>
> Resume the mutator threads.
>
> Signal the `unscannedPages` condition variable, resuming the scanner thread.

"Root reachable objects" are those objects referenced in registers, in global variables not stored in the heap, and on stacks (if stacks aren't stored in the heap).

A flip could have rather high latency if there are a large number of root objects, such as with large stacks or with many threads (each with its own stack and registers). But we can reduce the number of root objects copied using two tricks.

First, if stacks are not stored in the heap, then instead of scanning them during a flip, the collector just sets their pages to be inaccessible. The pages of a stack can then be scanned like pages of the unscanned area, both concurrently and incrementally as they are referenced by the mutator.

Second, even the registers of threads needn't be scanned at flip time. Instead, `Flip` stops the threads and changes their program counters to the address of a special routine, saving away the old PCs. It then resumes all the threads. When the thread is next scheduled and actually runs, it resumes at the special routine, which *then* scans the registers and jumps back to the original PC.

This technique is especially important when there are many more threads than processors, and many of those threads are blocked on locks or condition variables or waiting for a remote procedure call to finish. (A large Modula-2+ program could have 150 threads or more.) Each thread's registers are scanned concurrently with the execution of other threads. We haven't implemented this technique yet, but it takes about 80 msecs to stop and resume 150 threads on the 5-processor Firefly.

Before a flip, when there are no more objects in the unscanned area, all of from-space is known to be garbage. The collector discards those pages, reinitializing them to be demand-zero-on-write or undefined; this discards any backing store and physical memory attached to those pages. Backing store and physical memory will be reallocated on demand as the pages are referenced.

Objects that cross page boundaries

So far, we've assumed that the trap thread scans just the single page referenced by the mutator. Each page in the unscanned area would begin with an object and no object would cross a page boundary (thus implying that no object could be bigger than a page).

It's not difficult to remove this restriction. We present two schemes: the first is more appropriate to languages like Lisp or ML, which tag every pointer; the second to languages like Modula or Simula, which tag every record.

Languages like Lisp typically tag every word in the heap as "pointer" or "non-pointer." This is necessary even on conventional hardware so that the garbage collector can know what references to trace. On a byte-addressable 32-bit machine, for example, only the high-order 30 bits of each pointer are relevant, so the low-order 2 bits can be used for a pointer/non-pointer tag; of course, this limits the size of "unboxed" integers (non-pointers) to 30 or 31 bits as well.

Some implementations use several bits of tag within each pointer to indicate the type of the object pointed to, while other implementations put the type information at the beginning of the record. A descriptor word at the beginning of a record should be tagged appropriately: assuming that it is not a pointer, it should be tagged as a non-pointer.

Finally, there may be some kinds of objects (like character strings) that contain untagged data, but no pointers. These objects should be kept in a separate area of the heap. Objects from these string areas will be copied to to-space string areas, but the string areas won't need to be scanned or page-protected because pointers are never fetched from them.

Now, the behavior of the page-trap scanner is very simple: it just ignores object boundaries entirely! It simply scans each word on the page; if the beginning of the page is really the second half of some object that crossed a page boundary, no harm will be done because of the uniformity of the object representation. In particular, it is not necessary to examine the record descriptor before scanning a record, because each pointer is individually tagged.

In Modula, on the other hand, pointers and non-pointers are indistinguishable to the garbage collector; the descriptor at the beginning of a record indicates which fields are pointers. Therefore, it is necessary for the scanner to examine the descriptor of a record before scanning it; and thus the scanner must determine where records begin. The record descriptor (at the beginning of the record) will tell the size of the record, and if each page begins with the beginning of an object, then the scanner can find the first and subsequent objects on each page.

For objects smaller than a page, the collector could skip to the next page if an object being copied doesn't fit on the current one. If most objects were small relative to the page size, this wouldn't waste much space. But what if objects are larger and the wastage becomes significant? And what about objects larger than a page?

One solution is to maintain a map, `crossing`, such that `crossing`$[p]$ is true iff an object crosses the boundary between page $p-1$ and p. When the collector gets a page trap for page p, it scans backward for the first page $o < p$ such that `crossing`$[o]$ is false; that is, such that o begins with a fresh object. The collector then scans all the pages from o on, until it encounters a page $q \geq p$ such that `crossing`$[q]$ is false. This map must be maintained for objects in the unscanned area only, since that is the only part of to-space scanned as the result of page traps.

The crossing map allows the collector to trade off latency for wastage. If it wants to waste absolutely no space at all, it can simply append copied objects directly to the end of the unscanned area, setting entries in the map appropriately. Of course, this could result in large page runs being scanned as the result of a single page trap, thus increasing the latency of a page reference.

A more reasonable policy uses a small amount of wastage to limit the number of consecutive page crossings and thus reduce latency. When copying an object that doesn't fit entirely on the current page, the collector will skip to a new page if the wasted fragment is less than a small percentage w of the page size. The collector also skips to a new page if `crossing`$[p]$ is true for the previous c pages. In a typical Lisp system with smaller objects, for example, setting w to 10% and c to 1 would probably be adequate to keep the latency to little more than one page scanned per page trap and to keep the wastage around 10%.

Large arrays pose another problem: Copying or scanning a large array takes time proportional to the size of the array, and thus the time for a page trap wouldn't be bounded by a small constant. To solve this problem, we can use a technique similar to that suggested by Baker for incrementally copying and scanning arrays.

Each large array in to-space has an extra header word for storing a pointer to the from-space copy of the array. When the collector copies an array to the unscanned area, it just copies the array header and reserves space for the elements without actually copying them. It also sets the from-space link to point at the from-space copy. On a page trap, the collector uses the crossing map to scan backwards to find the array header and the pointer to the from-space copy. The collector then copies and scans only those elements on the referenced page. Similarly, the scanner thread of the collector copies and scans elements one page at a time.

Unlike Baker's scheme, this imposes no additional burden on the mutator's array indexing. But it does require that array elements must be padded so that the element size divides the page size evenly.

Scanning Stacks

Thread stacks are large objects requiring special treatment. In many language implementations, stacks are usually small; for example, the average size of a stack in a Modula-2+ program is about 300 bytes, less than a page. But any one stack could be much larger, especially in highly recursive algorithms, so we'll need to scan them incrementally by protecting their pages after a flip and fielding the page traps.

If the language implementation tags pointers and non-pointers as described in the previous section, then scanning individual stack pages is straightforward. But again, our algorithm does not require tags.

Individual stack frames are small and bounded, with large objects always allocated in the heap (this is true, or could easily be made true, for most Lisp and Algol-family language implementations without affecting performance). Given a frame, the collector can easily discover its size and the location of pointers within the frame. (For example, there may be a map maintained by the compiler from program-counter locations to frame descriptors.)

In many languages, including Lisp, a mutator can reference only the top frame of a stack, and thus page traps can happen for the top frame only. But languages like Modula-2+ provide by-reference parameters in which a called procedure is passed a pointer to a stack-allocated object in the caller's frame; for these languages, the trap handler must be prepared for trapped page references anywhere in a stack.

When the mutator references a protected page in the middle of a stack, the trap handler must scan its pointers, and to do that, it must first find the frames on the page. It can start at the top or bottom of the stack (whichever is closer) and skip over frames until it gets to the referenced page. If a frame overlaps the page boundary, the trap handler scans only the fragment on that page.

Of course, skipping over frames to get to a page in the middle of a stack takes time proportional to the stack size, and if stacks have unbounded size, then the algorithm isn't real-time. But in practice, skipping over frames is very cheap compared to the cost of scanning a page and can be considered "constant". For example, supposing each frame has a dynamic link to the previous frame, the necessary inner loop would take about 2.5 msecs per 1000 frames on MicroVAX II; whereas it takes anywhere from 15 to 40 msecs to field a page trap and scan a page. (Of course, implementations that don't have by-reference parameters won't have to worry about middle-of-stack traps.)

Scanning a stack concurrently while the mutator executes is, after a little thought, not hard. If the mutator attempts to change a write-protected stack page by, say, pushing and writing a new stack frame, the algorithm will guarantee the page will be scanned first.

But the mutator can pop frames, unsynchronized with the collector, simply by adjusting the stack pointer. What happens if the collector starts scanning a page of stack frames while the mutator pops some of those frames from the stack? The collector could suspend a thread every time it scanned one of its stack pages, but that wouldn't be satisfactory for threads with large stacks; the scanning would be real-time but not concurrent.

Instead the collector can simply ignore the fact that some of the frames it is scanning may already have been popped, since the mutator can't change the page until it is fully scanned and unprotected. At worst, a few extra garbage objects recently popped from the stack may get copied and survive this collection. To minimize this possibility, the collector should examine the current stack pointer before scanning each stack page.

Scanning-order heuristics

It is not necessary to scan the to-space or the stack in a purely sequential order (as the stop-and-copy algorithm does). The pages of the to-space and the stack may be scanned by the scanner thread in any order, except for page traps which force an immediate scan of the trapping page.

Different scanning orders will not affect the total number of pages scanned—this is determined only by the amount of reachable data at the time of the flip. However, the number of page traps might vary with different scanning orders. We can suggest some scanning heuristics that might take advantage of the locality of reference to minimize the number of page traps.

For example, after a page trap, the mutator is likely to trace references reachable from the trapping page. Therefore, the scanner should work on those references first. These references are largely to be found in the newest pages added to the to-space as a result of scanning the trapping page. A good heuristic might be: *After a page trap, scan the newest pages of to-space first.*

Clark [Clark 77] found that even in a pointer language like Lisp, references to an address were often followed by references to nearby addresses. This leads to the heuristic: *After a page trap, scan the pages adjacent to the trapping page.*

Brooks [Brooks 84] and others have suggested that it is better to scan stacks from the bottom up, the bottom-most objects are more likely to survive the current collection (pointers to objects on the top of the stack will soon get popped), and work won't be wasted. But the principle of locality, as discussed in the previous paragraph, suggests just the opposite. The top of the stack will be referenced soon; so to avoid page traps, *Scan the stack from the top down.*

All of these heuristics must be considered speculation until detailed experiments are done.

Derived Pointers

A *derived pointer* is a pointer into the middle of an object that may arise during the address calculation of an array or record access. Architectures like the VAX, 68000, or 370 provide efficient index-mode addressing, so derived pointers aren't necessary. But reduced-instruction-set machines may not have index-mode addressing and thus require derived pointers for array indexing.

Derived pointers cause problems for a concurrent collector. The collector may suspend the mutator threads at any time and initiate a collection. At the flip, the collector must know which registers contain derived pointers and the objects corresponding to those pointers.

The simplest scheme would reserve one or more register pairs to hold a pointer and its derivative, making it easy for the collector to identify derived pointers and their base objects. But this would make good code generation harder on a machine with relatively few registers (though newer RISC machines tend to have more registers).

A less-constraining scheme would mix pointers and derived pointers freely in registers, using the crossing map to find the base object of a pointer. (The new area doesn't have an explicit crossing map, but the collector can assume objects don't cross the boundaries of allocation chunks.) To find a base object of a pointer, the collector would skip backwards to the first page starting with an object and then skip forwards over objects until getting to the base object.

For small objects this would take time proportional to the page size, while for large objects this would take time proportional to the object size. In either case, the cost is much smaller than the cost of copying and scanning an object. For example, on a MicroVAX II it would take about 5 msecs to skip over 1000 pages in the crossing map. Since each thread has a small, constant number of registers and those registers are scanned concurrently after a flip is finished, neither latency nor concurrency would be affected.

A Sequential, Real-time Version

The general concurrent algorithm can be specialized for single-threaded programs in which the collector is real-time but not concurrent, resulting in a variant similar to Baker's original algorithm but more efficient than Brooks's variant (by avoiding the need for extra header words, indirection, and longer code-sequences for dereferencing pointers). The sequential, real-time verison would be suitable for languages implemented on traditional operating systems like Unix that allow access to virtual-memory facilities but don't provide multiple threads or cheap synchronization.

To handle page traps by the single mutator thread, the collector provides a "trap handler" routine to the operating system that fields page traps (these are called "signal handlers" on Unix). When the mutator references a protected page, the handler is invoked. The handler unprotects the page, scans it, and returns, automatically resuming the mutator.

To ensure that all reachable objects are copied before to-space fills up, the `Allocate` procedure scans a small number of unscanned objects every time it is called (as in Baker's algorithm).

Because the algorithm is synchronous and single-threaded, there is no need for any kind of locking or special kernel-mode threads. It is easily implemented on many different variants of Unix (for example, AT&T's System V, Apollo's Domain, DEC's Ultrix, and CMU's Mach all provide the necessary page-protection and trap-handling primitives).

But of course, this sequential version can't run the collector while the mutator is waiting for i/o, page faults, or interactions from the user. And even on uniprocessors, more and more operating systems are starting to offer multi-threaded capabilities as support is added for distributed computing based on remote procedure call.

A First Implementation and Experiment

We have implemented the collector in an early version of ML, a statically type-checked polymorphic language [Cardelli 84]. While this version of ML is not particularly efficient, it is both simple and tractable. The implementation runs on the Firefly using SRC's Taos operating system [Thacker 87], which extends DEC Ul-

	Stop-and-Copy	Sequential	Concurrent
Total elapsed time	252 sec	281 (1.11)	207 (.82)
Mutator time	180	180	180
Mutator overhead	29%	36%	13%
Total CPU time	247	257 (1.04)	266 (1.08)
Collector time	67	77 (1.15)	86 (1.28)

Table 1: Execution times

trix (Berkeley Unix 4.2) with multiple threads, cheap synchronization, and virtual-memory primitives.

We built three versions of the collector: simple stop-and-copy, sequential real-time, and concurrent. They all used the same object representations and the same copying and forwarding primitives. The object representations were somewhat complex and not well tuned for a copying collector, and allocation was implemented by a procedure call, not compiled inline.

The sequential real-time and the concurrent versions scanned all of the stacks (there were several) at a flip, instead of scanning them incrementally, as described above. When copying objects, the collectors used a more primitive heuristic for limiting the number of page crossings by unscanned objects; but that didn't matter in our benchmarks since there were very few crossings. We didn't implement the incremental scanning of large arrays (but the benchmarks didn't allocate any).

All three versions used a page size of 1K bytes and a heap with 3 megabytes per space. The sequential version scanned from 1 to 4K bytes for every 1K bytes allocated. The concurrent version used an Allocate chunk size of 65K.

We picked the Boyer benchmark from Gabriel [Gabriel 85] as our first experiment. It is a small rule rewriter designed to test the performance of Lisp systems executing theorem provers. It allocates a large amount of non-trivial, fine-grained data structures and then accesses those structures repeatedly. According to Gabriel, the Boyer program does 226,000 list-cell allocations and 1,250,000 fetches of pointers from the list cells. In addition to the allocations by the program itself, the ML implementation allocates procedure-argument records in the heap, adding to the load on the collector.

Table 1 shows the execution times of the three versions. The numbers in parentheses express table entries as ratios with the corresponding entries for the stop-and-copy version.

Total elapsed time shows that the sequential real-time version is 11% slower

	Stop-and-Copy	Sequential	Concurrent
Bytes allocated	12.2M	12.2M	12.2M
Number of flips	7	7	8
Time per flip	9.5 secs	.164	.120
Traps per second	—	1.7	3.3
Time per trap	—	43 msecs	38
Time per allocation	127 μsecs	166 (1.31)	83 (.65)
allocation	56	67 (1.20)	56 (1.00)
collection	71	99 (1.39)	27 (.38)

Table 2: Latency

than the more efficient stop-and-copy, while the concurrent version is 18% faster.

Mutator time is the time spent executing the mutator, including calls to the allocator but excluding all other collection-related time: page traps, all scanning, and flips.

Mutator overhead is the percent of total elapsed time that wasn't spent executing the mutator. For the stop-and-copy and sequential versions, overhead is about 1/3 of total time, typical for Lisp-like systems. The overhead for the concurrent version is less than half of that, only 13%.

Total CPU time is the total CPU time used by all the threads in the benchmark (as measured by the operating system). The sequential version takes 4% more than stop-and-copy, and the concurrent version takes 8% more.

Collector time is the total CPU time on all processors used by the collector, including page traps and scanning. The sequential and concurrent collectors take 15% and 28% more CPU time than stop-and-copy.

Table 2 shows the latency of the three versions, each of which allocated 12.2 megabytes during execution.

A stop-and-copy flip takes 9.5 seconds, while the sequential and concurrent versions take .164 and .120 seconds. A tenth of a second is almost good enough for an interactive workstation; using incremental stack and register scanning, the flip time should be well under that. The sequential collector flips more slowly than the concurrent one because it usually has to scan a few remaining pages before flipping.

Traps per second and **Time per trap** show that the disruption due to page traps is extremely small.

Time per allocation is the average elapsed time needed by the mutator to allocate and collect a two-pointer list cell. That time is broken into the time spent in the `Allocate` procedure (excluding page scanning) and the elapsed time the mutator is blocked waiting for the collector (including page scanning). The sequential version is about 1/3 slower than stop-and-copy, and the concurrent version is about 1/3 faster. The time spent in `Allocate` could easily drop to 10 μsecs, and the collection time should drop quite a bit with better object representations, hand-tuned collectors, and generational collection.

How concurrent is the concurrent collector? That is, how much of the stop-and-copy collection time could be successfully overlapped with the mutator? Look at the collection part of the **Time per allocation**—44 μsecs (62%) of the stop-and-copy collection time was overlapped with the mutator, leaving 27 μsecs (38%) not overlapped. That 27 μsecs breaks down as follows:

Page-trap scanning	7.8 μsecs
Operating-system trap overhead	2.0
Flips	.9
Allocator waiting for lock	.5
Page trap waiting for lock	5.3
Unexplained	10.4

Trap overhead is the cost of the context switch caused by a page trap. The waiting-for-lock time is the time the allocator and the trap handler must wait for the scanning thread to finish scanning a page and release the global lock.

That leaves 10.4 μsecs (15%) of the collection time unaccounted for. We have a hypothesis: The Firefly's MicroVAX II CPU has a tiny 8-entry VM translation buffer; interleaving the page traps and their scanning with the mutator will cause more translation-buffer misses. Also, the operating system must flush the translation buffer (on all 5 processors) whenever it changes the protection of pages. No pages were swapped in during the experiment, so we can rule out differences in swapping behavior.

Several improvements waiting in the wings should increase the concurrency by quite a bit, perhaps to as high as 85 or 90%. We should double the speed of scanning by using better object representations and hand-coded scanning. An experiment showed that scanning two or four consecutive pages per trap would reduce the number of traps and the overhead per trap, while still providing adequate latency. Page-trap handling in the Firefly operating system is untuned, costing 3 msecs compared to 1 msec on Ultrix. Newer CPUs have much larger translation buffers, reducing the effect of frequent traps. And generational collectors should reduce translation-buffer misses and contention for the global lock by reducing scanning and copying. Finally, incremental stack scanning will reduce flip time.

More complicated scanning heuristics might reduce the number of page traps by exploiting the mutator's presumed locality of reference. For example, after a page trap, the scanning thread might do better to scan the pages next to the trapped page or the pages added to the unscanned area as a result of the trap.

A Comparison with Reference Counting

Out of curiosity, we implemented the same Boyer benchmark in a small, compiled Lisp that uses the Modula-2+ allocator and concurrent reference-counting collector [Rovner 85b]. The Lisp version allocated 1.8 megabytes compared to the 12.2 megabytes allocated by the ML version; unlike the ML implementation, the Lisp doesn't allocate things like argument records in the heap.

Here is a comparison showing the ratio of the execution times of the reference-counting collector with the ML concurrent collector:

Elapsed time	1.3
Total GC CPU time	.81
Time per allocation	3.5

The **Elapsed time** ratio shows only that the Lisp and ML compilers produce code in the same ballpark.

The **Total GC CPU time** and **Time per allocation** ratios show that, though the reference-counting collector consumes 20% less CPU time than the concurrent collector, the ML mutator can allocate and collect cells 3.5 times faster than the Lisp/Modula-2+ mutator. (There's no discrepancy here—the ML concurrent collector can overlap more of its work with the mutator.)

Neither collector has been tuned much, so these results should be treated gingerly.

Generational Collection

Generational collection can drastically reduce the work of a copying collector by scanning and copying far fewer objects [Lieberman 83, Moon 84, Ungar 86]. Many fewer pages are touched by the collector, resulting in better virtual-memory performance [Shaw 87].

Generational collection is based on two observations: new objects have a higher death rate than old objects, and few old objects reference new objects. The collector allocates new objects in a small "new" area and somehow remembers all pointers to new-area objects that the mutator stores outside of the new area. During most collections, only the new area and the remembered pointers must be scanned, copying only the new-area objects that are still live. As objects survive collections, they are "aged" and copied outside of the new area; the entire heap is collected very infrequently.

Ungar [Ungar 86] and Shaw [Shaw 87] show how to implement generational collectors on stock hardware while reducing the demand on virtual memory. Shaw suggests using the dirty-page bits of virtual memory hardware to keep track of where the mutator stores pointers, and he compares that to schemes that use a few extra instructions per assignment. He presents a scheme for managing the various segments of the heap efficiently. But Shaw's proposal is neither real-time nor concurrent.

We expect to merge our techniques with Shaw's with little effort. Instead of the traditional stop-and-copy to extract live objects from the new area, the collector can use our algorithm to collect concurrently while the mutators execute. Though the arrangement of the heap areas is somewhat different from that of our algorithm, the basic invariants are still maintained.

Based on our experiments and the results of Ungar and Shaw, the reduced load on virtual memory should increase efficiency, reduce latency, and perhaps improve concurrency.

Multiple Mutator Threads

How many mutator threads can one collector support while remaining real-time and concurrent? In general, this depends on the total amount of memory, since copying collectors are arbitrarily efficient as the ratio of memory size to reachable objects increases [Baker 78, Appel 87].

How many threads could our initial implementation support with its 6-megabyte heap? Even though ML limited us to one mutator thread, we can make a rough estimate.

Assume that each thread has its own processor and allocates and references objects with the same frequency as the Boyer benchmark. Then the critical resource is the thread that handles page traps and scanning (it doesn't matter if there are separate threads for traps and scanning, since they are serialized by the global lock).

In the concurrent version, for every 207 seconds of elapsed time, the trap/scanner thread executes 86 seconds. Thus the processor running the trap/scanner thread should support $207/86 = 2.5$ mutator threads.

But our Boyer/ML benchmark does a huge amount of allocation compared to most programs written in Lisp or Modula-2+. In addition to the large number of allocations by the mutator, the ML implementation allocates things such as argument records in the heap that more efficient implementations would put on the stack.

Also, a generational scheme should reduce dramatically the amount of scanning, copying, and page traps. Based on Ungar's experience, the total amount of copying and scanning might be reduced by a factor of three or more.

Thus a single collector might very well support at least 10 continuously executing threads.

Multiple Collector Threads and Shared Virtual Memory

What if there are a very large number of threads and processors? Using multiple heaps like those of the Multilisp collector [Halstead 84], our general techniques should allow for multiple scanning and trap threads. (The "multiple heaps" can be viewed as partitions of a single heap.) Each heap would have its own trap and scanning threads. A lock bit per from-space page ensures that an object is copied at most once. Only at a flip must all the threads synchronize.

Kai Li [Li 86] introduced the notion of *shared virtual memory*, providing a single, shared virtual-memory address space for many processors having distinct physical memories. Pages are exchanged between processors using a high-bandwidth bus or local network, with only one processor at a time allowed write access to a page. In such a system, it's crucial to ensure that processors don't thrash on a shared writeable page.

Our algorithm, extended with generational collection and multiple heaps, is well suited for shared virtual memory. Because each thread has its own allocation chunk, there won't be thrashing of newly allocated pages. In addition, each processor would have its own collector thread. To scan a page, the collector must, if necessary, acquire the page for write access from another processor. It then "pins" the page while scanning it, preventing other processors from trying to access it. This imposes almost no extra overhead on the virtual memory implementation and naturally prevents thrashing by the collector threads.

Acknowledgements

Tom Rodeheffer improved the Firefly's VM implementation for us and answered our questions, and Roy Levin gave us many helpful hints about the Firefly's operating system. Cynthia Hibbard edited the paper.

References

[Appel 87] Andrew W. Appel.
Garbage collection can be faster than stack allocation.
Information Processing Letters, 25(4):275–279, 1987.

[Baker 78] H. G. Baker.
List processing in real time on a serial computer.
Communications of the ACM, 21(4):280–294, 1978.

[Brooks 84] Rodney A. Brooks.
Trading data space for reduced time and code space in real-time garbage collection on stock hardware.
In *SIGPLAN Notices (Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments)*, pages 256–262, 1984.

[Cardelli 84] Luca Cardelli.
Compiling a functional language.
In *1984 ACM Symposium on LISP and Functional Programming*, pages 208–217, 1984.

[Clark 77] Douglas W. Clark and C. Cordell Green.
An empirical study of list structure in Lisp.
Communications of the ACM, 20(2):78-87, 1977.

[Gabriel 85] Richard Gabriel.
Performance and Evaluation of Lisp Systems.
MIT Press, 1985.

[Halstead 84] Robert H. Halstead, Jr.
Implementation of Multilisp: Lisp on a Multiprocessor.
In *1984 ACM Symposium on LISP and Functional Programming*, pages 9–17, 1984.

[Hickey 84] Tim Hickey and Jacques Cohen.
Performance analysis of on-the-fly garbage collection.
Communications of the ACM, 27(11):1143–1154, 1984.

[Li 86] Kai Li.
Shared Virtual Memory on Loosely Coupled Multiprocessors.
PhD thesis, Yale University, 1986.

[Lieberman 83] Henry Lieberman and Carl Hewitt.
A real-time garbage collector based on the lifetimes of objects.
Communications of the ACM, 23(6):419–429, 1983.

[Moon 84] David A. Moon.
Garbage collection in a large LISP system.
In *ACM Symposium on LISP and Functional Programming*, pages 235–246, 1984.

[North 87] S. C. North and J. H. Reppy.
Concurrent garbage collection on stock hardware.
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 113–133. Springer–Verlag, 1987.

[Rovner 85a] Paul Rovner.
On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language.
Xerox PARC Report CSL-84-7, 1985.

[Rovner 85b] Paul Rovner, Roy Levin, and John Wick.
On Extending Modula-2 For Building Large, Integrated Systems.
Research Report 3, DEC Systems Research Center, 1985.

[Shaw 87] Robert A. Shaw.
Improving Garbage Collector Performance in Virtual Memory.
Technical Report CSL-TR-87-323, Computer Systems Laboratory, Stanford University, 1987.

[Thacker 87] Charles P. Thacker and Lawrence C. Stewart.
Firefly: A Multiprocessor Workstation.
In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172. ACM, 1987.

[Ungar 86] David Ungar.
The Design and Evaluation of a High Performance Smalltalk System.
MIT Press, 1987.