REAL-TIME CONCURRENT COLLECTION IN USER MODE

Kai Li

CS-TR-291-90

October 1990

Real-Time Concurrent Collection in User Mode

Kai Li

Department of Computer Science
Princeton University
35 Olden Street
Princeton, NJ 08544

Abstract

We previously presented a real-time, concurrent garbage-collection algorithm that uses the virtual memory page protection hardware to synchronize collector threads and mutator threads. The algorithm requires the mutator threads to access protected pages that prevent collector threads from accessing. This paper investigates three other alternatives to achieve such a goal: page-copying, multiple address mapping, and page sharing in different address spaces. We will present our experiment with the page-copying version and compare it with the kernel-mode, simple stop-and-copy, and sequential real-time versions.

Previous Algorithm

We previously proposed a real-time, concurrent garbage-collection algorithm for stock uniprocessors or shared-memory multiprocessors [2]. The essential idea in the algorithm is to use the hardware page fault mechanism in the virtual memory system to achieve medium-grain synchronization between collector and mutator threads [2]. The paging mechanism provides synchronization that is coarse enough to be efficient and yet fine enough to make the latency low. The algorithm is based on the Baker's sequential, real-time copying collector algorithm [3].

A basic stop-and-copy collector [5] divides its memory heap 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.

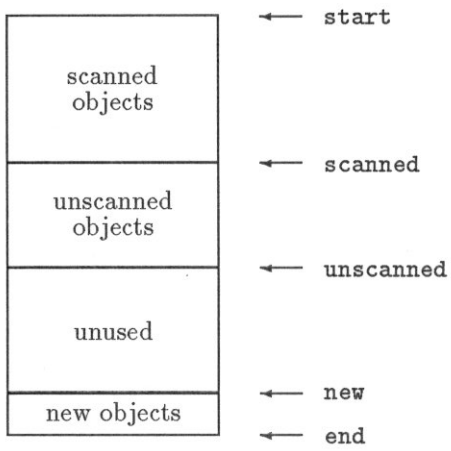

Figure 1: To-space

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 upward. When to-space fills up, the mutator stops and initiates a new collection.

Baker's sequential real-time collector [3] copies only the root objects (for example, those referenced from the registers) at a flip. It then resumes the mutator immedi-

ately. 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. Baker's algorithm maintains the following invariants:

- 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.

To satisfy the invariant that the mutator sees only to-space pointers in its registers, 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 [7], since otherwise a few extra instructions must be performed on every fetch.

Instead of checking every pointer fetched from memory, the concurrent collector uses virtual-memory page protections to detect from-space memory references and to synchronize the collector and mutator threads. To synchronize mutators and collectors, the algorithm 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.

Kernel-Mode vs. User-Mode

The implementation of the algorithm we previously presented was not as trivial as it sounds because the collector threads executes concurrently with the mutator threads, scanning pages in the unscanned area and unprotecting them as each is scanned. The difficulty lies in how the collector threads 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? The collector thread can't unprotect the page before scanning it because then the mutator, running concurrently, would be able to reference unscanned objects.

Our previous solution was to use the property that 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. We added 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.

The advantage of the kernel-mode method is that it simply uses the same page table entry for getting different protections on the same virtual page. Thus, there is no additional memory requirement for page mapping.

The kernel-mode method has a number of drawbacks. First, it requires a safe, and flexible system interface. Using special system calls for a particular garbage collector can be difficult because different langauges would have different object representations and thus use different collectors. If the collector has a bug, it may alter kernel memory data structure and probably crash the system. This approach is also lack of portability. Most existing operating systems do not have an interface that fits the kernel-mode requirement. Many language designers would like to port their compilers without creating new system calls.

User-mode approaches

In addition to running collector threads in kernel-mode, there are several ways to achieve user-mode access to protected pages and each requires certain kind of system services [1].

- **Page copying**

  A system call could be provided to copy memory to and from a protected area. The collector threads use this call to copy protected pages into a buffered, unprotected area for scanning. This method requires only a few pages for buffering, depending on the number of collector threads. The collector threads need to make two additional copies for each page: copying to the scan buffers before scanning and copying the scan buffers back to the protected area after scanning. In addition to copying, the collector threads need to calculate forwarding addresses by using the offset of the buffers from the protected pages.

The advantage of this approach is its simplicity. It requires operating systems to provide a very simple system call which is as secure as system calls for protecting memory pages. The main disadvantage is the overhead of data copying.

- **Multiple page mapping**

  The virtual memory provides a call that allows multiple mapping of the same page at different addresses with different protections in the same address space. The garbage collector has access to pages in to-space at a "nonstandard" address, while the mutators see to-space as protected. Like the page copying method, the collector threads need to do address calculation, but they do not need to use any buffer space.

  With a virtually-addressed cache, the multiple virtual address mapping approach has a potential for cache inconsistency since updates at one mapping may reside in the cache while the other mapping contains stale data. However, the problem can be solved for the garbage collection algorithm. While the collector threads are scanning the page, the mutators have no access to the page; and therefore at the mutators' address for that page, none of the cache lines will be filled. After the collector threads have scanned the page, they should flush their cache lines for that page (presumably using a cache-flush system call). Thereafter, the collector threads will never reference that page, so there is never any danger of inconsistency.

  The main disadvantage of this approach is that it requires additional memory space for page table entries. When mapping two virtual addresses onto one physical page, it requires two page table entries for the same page. In order to reduce the memory resource requirement, one can also provide mapping for only the scanned pages. This approach reserves a few pages in the virtual address space for scanning and maps them onto the scanned, protected pages. Similar to the page-copying method, the collector threads view these reserved pages as scanning buffers except the page contents are never copied.

  For some virtual memory systems that have memory object abstractions [8], such a mapping may require other additional resources and may decrease the performance of other virtual memory primitives such as protect and unprotect pages.

- **Page sharing**

  In an operating system that permits shared pages between processes, the collector can run in a different heavyweight process from the mutator, with a different page table.

  The main advantage of this approach is its simplicity. It does not require collector threads to do any address calculation since two address spaces can map to the same set of physical pages. It does not cause any cache inconsistency. In fact, many operating systems support shared memory mapping between two address spaces. This approach may not have any requirement to operating systems.

  The main disadvantage of this approach is its overhead. Similar to the multiple page mapping method in the same address space, there are two page table entries for each page. In addition, it requires two expensive heavyweight context switches on each garbage-collection page-trap. On shared-memory multiprocessors where there is enough processor resource, one can apply spinning or efficient user-level RPC mechanism to reduce the overhead of heavyweight context switches.

We advocate that for computer architectures with physically addressed caches, the multiple virtual address mapping in the same address space is the cleanest and most efficient solution. It does not require heavyweight context switches, data structure copies, nor running things in the kernel.

Experiments

For investigating alternatives of running collector threads in user mode, we have implemented the page copying method described in the previous section for an early version of ML statically type-checked polymorphic language [4], The implementation runs on the Firefly using SRC's Taos operating system [9], which extends DEC Ultrix (Berkeley Unix 4.2) with multiple threads, cheap synchronization, and virtual-memory primitives.

We compare our experiments with our previous results from three other versions of the collector: simple stop-and-copy, sequential real-time, and kernel-mode concurrent. All for collectors 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.

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 al-

	Stop-and-Copy	Sequential	Concurrent (kernel)	Concurrent (user)
Total elapsed time	252 sec	281 (1.11)	207 (.82)	220 (.87)
Mutator time	180	180	180	180
Mutator overhead	29%	36%	13%	18%
Total CPU time	247	257 (1.04)	266 (1.08)	299 (1.21)
Collector time	67	77 (1.15)	86 (1.28)	119 (1.78)

Table 1: Execution times

	Stop-and-Copy	Sequential	Concurrent (kernel)	Concurrent (user)
Bytes allocated	12.2M	12.2M	12.2M	12.2M
Number of flips	7	7	8	7
Time per flip	9.5 secs	.164	.120	.291
Traps per second	—	1.7	3.3	3.3
Time per trap	—	43 msecs	38	50
Time per allocation	127 μsecs	166 (1.31)	83 (.65)	96 (.76)
allocation	56	67 (1.20)	56 (1.00)	56 (1.02)
collection	71	99 (1.39)	27 (.38)	40 (.56)

Table 2: Latency

located. The concurrent version used an Allocate chunk size of 65K.

We picked the Boyer benchmark from Gabriel [6]. 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 Boyer benchmark of the four 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 than the more efficient stop-and-copy, while the kernel concurrent version is 18% faster and the page-copying concurrent version is 13% 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 kernel concurrent version is less than half of that, only 13% and for the page-copying concurrent version is 18%.

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, the kernel concurrent version takes 8% more, and the page-copying concurrent version takes 21% more.

Collector time is the total CPU time on all processors used by the collector, including page traps and scanning. The sequential takes 15% more CPU time than stop-and-copy while the kernel concurrent version and the page-copying take 28% and 78% more.

Table 2 shows the latency of the four 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, the kernel concurrent version is about 1/3 faster, and the page-copy concurrent version is about 1/4 faster. The time spent in `Allocate` could easily drop and the collection time should drop quite a bit with better object representations and tuning.

Conclusions

In this paper, we have discussed four approaches for the collector threads to access protected pages for the concurrent copying garbage collection method that requires using the page protection mechanism to synchronize collector threads and mutator threads: kernel mode, page copying, multiple address mapping in the same address space, and multiple address mapping in different address spaces. Each method has advantages and disadvantages.

The kernel-mode version is probably the most efficient method. It requires less system resources, but it requires a careful design for the interface to the virtual memory to avoid security leaks. The portability is poor.

The page-copying method is less efficient, but it requires less resource. Our implementation shows that it can obtain substantial efficiency over the stop-and-copy version although it is less efficient than the kernel-mode version.

The method of multiple address mapping in the same address space should be as efficient as the kernel-mode method, but it requires more system resources such as additional memories for page tables.

The method of multiple address mapping in different address spaces is expected to be less efficient than that in the same address space because of the heavyweight context switches. Its main advantage is its portability.

Although the user-mode approaches seem more expensive than the kernel-mode approach, our experiments indicate that even the page-copying approach provides quite reasonable performance for the Boyer benchmark on a small scale shared-memory multiprocessor.

Acknowledgements

This research was supported in part by National Science Foundation Grant CCR-8814265 and DEC External Research Program and DEC Systems Research Center.

References

[1] Andrew Appel and Kai Li. Virtual Memory Primitives for User Programs. Technical Report CS-TR-276-90, Princeton University, July 1990.

[2] A.W. Appel, J.R. Ellis, and K. Li. Real-time Concurrent Collection on Stock Multiprocessors. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

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

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

[5] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[6] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

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

[8] R.F. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[9] C.P. Thacker and L.C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.