Nonintrusive Cloning Garbage Collection with Stock Operating System Support

Gustavo Rodriguez-Rivera
Vincent Russo
Purdue University
{grr, russo}@cs.purdue.edu

Abstract

It is well accepted that Garbage Collection simplifies programming, promotes modularity, and reduces development effort. However it is commonly believed that these advantages do not counteract the price that has to be paid: excesive overheads, possible long pause times, and the need of modifying existing code. Even though there exist publically available garbage collectors that can be used in existing programs, they do not guarantee short pauses, and some modification of the application using them is still required. In this paper we describe a snapshot-at-beginning concurrent garbage collector algorithm and its implementation. This algorithm is less intrusive than other garbage collecting algorithms, guarantees short pauses, and can be easily implemented in stock unix like operating systems. Our results show that our collector is faster than other implementations that use user-level dirty bits, and is very competitive with explicit deallocation. These advantages are especially true on multiprocessor machines where collector and mutator can run on different processors. Our collector has the added advantage of being nonitrusive. Using a dynamic linking technique, we have been able to run our garbage collector with success even in commercial programs where source code is not available. In this paper we describe the algorithm, the implementation, and provide a comparison with other incremental garbage collectors. [1]

1 Introduction

Garbage collection is the automatic deallocation of dynamically allocated memory that is no longer in use by a running program [Knu73]. The alternative, explicit deallocation, is error prone and requires the programmer to keep track of the liveness of the memory allocated. This is a frequent source of bugs that are difficult to track. If the programmer deallocates memory too soon, the allocator may return the same memory again later in the program. This could result in two different logical data items being stored in the same memory area causing unpredictable behavior in the program. On the other hand if memory is never deallocated, the computer could run out of memory swap space and crash in the case of a virtual memory system, or slow down the system for excesive paging.

There exist programs that help find memory errors such as the ones we have described [HJ92]. However, these programs only diagnose the problem not solve it. The programmer still has to find out when and where to correctly deallocate memory. Garbage collection solves both problems automatically.

Besides reducing development effort, garbage collection also promotes modularity [Wil95]. For example, assume that two modules use explicit deallocation. Module 1 allocates an object and passes its reference to module 2. The following questions arise: Which module is going to deallocate the object? How the module encharged for the deallocation will know that the other module not longer needs the object? The programmers of both modules will

[1] A expanded version of this draft has been submitted for publication to "Software Practice and Experience"

have to agree on the answers to these questions. This agreement adds extra dependencies between the modules. Garbage collection solves this problem by deallocating the object automatically when no module needs it.

There are several algorithms for automatic garbage collection: mark and sweep collection[McC60], mark-compact collection[CN83], copying garbage collection[FY69, Che70], and non-copying implicit collection [Wan89, Bak91]. We choose to use mark and sweep garbage collection in our implementation since it can be implemented conservatively.

Mark and sweep collection[McC60] consists of two phases: the *mark phase* and a *sweep phase*. The mark phase identifies the blocks that are garbage, and the sweep phase deallocates these blocks. As in any garbage collection, there are assumed to be a set of memory blocks that are never garbage called the *root set*. This root set typically consists of the global variables and the execution stack. For every block in memory there is a bit called *mark bit*, that is set when the block is found to be reachable from the root set. The goal of mark and sweep collection is to recursively mark all the blocks reachable from the root set. The mark phase starts by clearing all the mark bits. Then all the blocks in the root set are marked and pushed on to a *mark stack*. As long as the mark stack is not empty, one block is poped from the mark stack, and it is scaned for pointers to other unmarked blocks. The unmarked block is marked and pushed to the stack. When the mark stack is empty, all the blocks reachable from the root set are marked, and the unmarked blocks are garbage. Since the unmarked objects can not longer be reached by the program, they can be safely deallocated. This is done during the sweep phase.

Simple implementations of mark and sweep collection stop the execution of the program (mutator) completely when doing a collection. This prevents possible races due to changes in the heap during collection and, therefore, prevents memory from being erroneously garbage collected. These "stop-the-world" collectors are impractical in time critical and interactive applications where the implied pauses in execution are unacceptable.

There is a class of garbage collection algorithms called incremental collectors that reduce or eliminate pause time by marking live memory concurrently with the mutator's execution [DLM+78, Ste75, BDS91]. These techniques use read/write-barriers to keep track of pointer reads/writes and make sure that the mutator does not cause the collector to erroneously garbage collect accessible memory.

Read/Write-barriers are implemented in two basic ways: compiler generated [Bro84], and virtual memory supported [AEL88, AL91]. Compiler generated read/write-barriers insert extra code at every pointer read/write operation in the program. Besides the overhead implied by the execution of the extra code (proportional to the number of pointer reads/writes), compiler generated read/write-barriers are not possible in language implementations where no cooperation exists, e.g., standard C and C++ compilers[BW88, BDS91] [2]

On the other hand, read/write-barriers that use virtual memory support can be used more widely (as long as the OS provides the necessary support). Many systems allow read/write-barriers to be implemented directly by programs using virtual memory page protection mechanisms. In such implementations, pages are read-protected (or write-protected), and an application provided handler executes special code when the program tries to read (or write) any of the protected pages [AEL88, BDS91, AL91].

Virtual memory read/write-barriers implemented in user space tend to be inefficient and intrusive. The cost of a page trap, crossing the kernel-user boundary, and calling a signal handler for every page is expensive [HMS92]. Also when a page trap signal is produced inside a system call, the system call is unexpectedly interrupted. To prevent this, the trap has to be manually generated before the program makes a system call that potentially can cause a page trap. This limits their use since programmers need to write a stub for every system call that potentially causes page traps. Since only one handler exists in the program for this signal, a conflict will exist if another library is already using this handler.

Kernel virtual memory implementations of read/write barriers do not have these problems. However, not all modern operating systems explicitly support read/write barriers inside the kernel. In this paper we introduce a way to circumvent this problem and show a way to implement an efficient write-barrier for an incremental garbage collector using existing kernel primitives available on many stock operating systems.

The basic algorithm proceeds as follows: when it is time to do a garbage collection, the mutator creates a perfect copy of itself (clone) that has its own copy of the address space. This clone runs the mark phase in parallel with the mutator. Since mutator and clone have their own copy of the address space, modifications performed by the mutator, do not conflict with the mark phase performed by the clone. When the collection is

[2] We are not taking into consideration modifying an already compiled program and inserting new code to keep track of pointer reads/writes [HJ92]. This approach is not trivial, and it still may introduce a significant overhead.

done, the clone passes the mark bits to the mutator and exits. The mutator uses the unmarked blocks in future allocations.

This algorithm can be easily implemented in stock UNIX or any other POSIX compliant operating system using the fork() call. Fork() creates a exact copy of the caller process. Historically the fork() system call has been criticized as an ugly way to create a new process[3]. However, for the sake of our algorithm, fork() is the perfect system call to implement cloning garbage collection. Alternatively an implementation could use explicit copy-on-write memory if it is provided by the operating system.

Since UNIX uses fork() every time a new process is created, implementations of UNIX have optimized it extensively. The duplication of address spaces inside fork() is now commonly done by means of virtual memory copy-on-write techniques. Also since page traps caused by copy-on-write pages are handled efficiently inside the kernel, the write-barrier overhead is smaller than write-barriers implemented in user space.

We have implemented a conservative version of this algorithm for SPARC machines running Solaris 2.4 with excellent results (the implementation is not SPARC specific). The collector not only has insignificant pauses but it is also efficient and nonintrusive (it is not necessary to modify the programs that use the garbage collector). We have been able to run the garbage collector in commercial programs where source code is not available.

Even though our implementation of the algorithm is conservative, we believe that the algorithm can be applied also to languages that give the precise position of pointers.

The reminder of this paper gives some background of incremental garbage collection, a description of the basic algorithm, the implementation details, a comparison with other garbage collectors, the conclusion of our research, and future work.

2 Incremental Garbage Collection

If an unmodified mark and sweep garbage collector ran concurrently with the mutator, it would erroneously collect a reachable object in the following situation: during a collection, the mutator stores a pointer to an unmarked object into an already scanned object and then it removes all other paths that reach the unmarked object. Since the collector will not rescan already scanned objects, this unmarked object will never be marked and it will be collected, even though it is reachable.

It is easier to understand this problem using Dijkstra's three-color abstraction [DLM+78]. During a mark phase objects that are marked and that have been scanned completely are considered *black*. Objects that are marked but not yet scanned are considered *grey*. Objects that are unmarked are *white*.

The process of marking can also be described as the process of turning *white* objects into *black* objects. At the beginning of the mark phase the objects in the root set are *grey* and all the other objects are *white*. The mark phase transforms *white* objects that are reachable from *grey* objects into more *grey* objects. When a *grey* object has been scanned completely, it turns *black*. At the end of a mark phase all objects that are still *white* are considered unreachable (garbage).

The collector will miss a reachable object when both the following two conditions happen [Wil95]:

1. The mutator stores in a *black* object a pointer to some *white* object.

2. All the paths from *grey* objects to this *white* object are deleted.

Incremental collectors must provide a mechanism to prevent either of the previous conditions from happening in order to avoid collecting reachable objects.

[3]Most of the fork() calls are followed by an exec() call which destroys the newly created process' address space. One of the reasons why there are two different calls instead of a more efficient fork-exec() call is because it was easier to implement in this way. In early versions of UNIX, fork() was implemented in only 27 lines of assembly code by creating an extra copy of the current process in the swap area [Rit84]. An added advantage for having two calls instead of one is that the child process can setup input, output, and some other process context before calling exec() in a more elegant way [Bac86].

One way to prevent the first condition from happening is by using a read-barrier to make sure the mutator always sees only *black* or *grey* objects so no pointers to *white* objects will ever be written to *black* objects. When the mutator tries to load a pointer to a *white* object, the read-barrier turns the *white* object either to *grey* by pushing it to the mark stack, or to *black* by immediately scanning the block for pointers. An example of such a collector is given in [Bak78].

Incremental update garbage collection algorithms prevent the first condition from happening by using a write-barrier to detect when a pointer to a *white* object is being written into a *black* object. In this case the write-barrier either turns the *black* object to *grey* by pushing it back on to the mark stack, or turns the *white* object *grey* by marking it and pushing it to the stack. Optionally the *white* object could be turned to *black* by scanning it for pointers. Examples of garbage collectors that use incremental update are [DLM+78], [Ste75], and [BDS91].

Instead of preventing the first condition, snapshot-at-beginning garbage collection algorithms prevent the second condition from happening by making sure that there always remains a path from the root set to the objects that were reachable when the collection started. The approach is to use copy-on-write-barrier that takes a *snapshot* of the mutator's memory when the collection starts. The mark phase is then performed on this snapshot while the mutator continues in the original copy.

Our cloning garbage collector belongs to this class of algorithms. The snapshot is automatically done inside the kernel through virtual memory copy-on-write techniques when cloning the process. The snapshot can also be obtained by pushing old pointer values to a mark stack every time they are overwritten by the mutator [Yua90], by using special memory processors to create a copy-on-write of the mutator's memory [AP87], or by building a snapshot in user space by protecting the memory and catching page faults with a special signal handler. In the latter case the handler takes a snapshot of the page, unprotects the page, and then the collector uses the snapshot page for tracing[DWH+90].

Incremental collectors that prevent the first condition for happening are different in their real-time response than those that prevent the second condition. Collectors that prevent the first condition may have to iterate one or more times the mark phase until no *grey* objects exist [BDS91]. Also the collector may need to pause continuously for every read/write operation, or completely at the end of the mark phase to catch up with the mutator. All of the above cases make it difficult to predict one or more of the following: how long the garbage collection will take, how frequent the interruptions will happen, how long the interruptions will take, how long the final pause will last, or how much progress the mutator will be allowed to make.

Snapshot-at-beginning algorithms are more predictable. The amount of work done for every pointer write is bounded (in the worst case it can cause a kernel page fault in our implementation). Also there is no need to iterate the mark phase until the collector catches up. Therefore the time it takes to do a garbage collection is bounded and pause time at the end of the collection can be predetermined. Snapshot-at-beginning algorithms are thus more useful when (soft) real-time response is necessary.

3 Cloning Garbage Collection Algorithm

Snapshot-at-beginning algorithms work because any unused memory that is collected in the *snapshot* is still garbage in the mutator when the collection ends. Such a collection is conservative since not all the current garbage is collected (new garbage may be generated after the collection starts in the clone). The algorithm is formally described as follows:

1. Stop all threads but the one that triggered the collection.
2. Clone process (memory and threads)
3. If parent process, then restart all threads and proceed execution
4. If clone process, then perform a mark phase
5. When mark phase ends, the clone process passes mark bits to parent process and exits

6. The parent process uses unmarked blocks in future allocations (lazy sweep)

Note that the execution of all the threads but the current one have to be stopped before cloning. The reason is to have only the thread doing the mark phase running in the clone. Otherwise, other threads can potentially modify the reachability graph and interfere with the mark phase. Also in multiprocessor machines it is necessary to stop the other threads to take an atomic snapshot of all the processors' registers.

Also note that the pause in this algorithm will be equal to the time it takes to clone a process. This time is small in modern operating systems that duplicate address spaces using VM copy-on-write techniques. Since only kernel tables are modified in this call, this time slowly grows with the process memory size.

Another issue is how mark bits are passed from clone to mutator process. This can be done by using an inter-process communication mechanism such as pipes, files, or shared memory. Our implementation uses shared memory to avoid copying overhead.

When the mark phase is over, the mutator process runs a lazy sweep during allocation. This means that when the mutator runs out of blocks in some free list, it will get unmarked blocks from the next unsweeped page for this size. In this way, pages are swept on demand during allocation. This lazy sweep is similar to the one used in [BDS91].

4 Implementation

4.1 The Allocator

The memory allocator uses simple segregated free lists [Com64, WJNB95]. There are multiple free lists where each list contains blocks of a particular size. Each heap page contains objects of the same size and there is no splitting or coalescing of blocks. Once a page is committed to have blocks of an specific size, it will continue storing blocks of that size until the page is unmapped.

Also for each heap page there is a corresponding entry in a two level table of page-information entries that is indexed by the page address. A page-information entry contains: a pointer to the free list this page belongs to, a pointer to the next page used for the same size, a pointer to the start address of this block if this a multiple-page block, and some other information. Given a memory address (*addr*) it is possible to find the corresponding page-information entry (*pageInfo*) in constant time from the two level table of page-information entries (*pageTable*) by shifting and indexing.

$$pageInfo(addr) = pageTable[addr >> SHIFT1][(addr << SHIFT2) >> SHIFT3] = 0(1) \qquad (1)$$

Traditionally the heap is stored in a single memory mapping. When the allocator needs more memory from the system, it calls the *sbrk(increment)* system call to enlarge this mapping by *increment* number of bytes. Data and allocator data structures are intermixed in this single mapping. To make things worse, some user programs also call *sbrk()* directly creating *holes* in the heap that the collector has to treat in a special way during pointer finding.

In modern OS, there is a primitive called *mmap()*. *Mmap()* creates independent memory mappings in the virtual address space. Our allocator uses this call to create independent mappings for data and allocator data structures. This allows a better logical separation between them. Also a side effect of having a memory mapped heap is that the heap is allocated high in memory. This reduces the probability of false pointers (sequences of bits that look like pointers) since most of the false pointers are caused by low integer values [BW88].

Large blocks (blocks larger than a page) are also memory mapped. This allows the allocator to shrink the heap by unmapping large blocks that are not being used by the program. In this way no memory or swap space is committed to these blocks anymore. To prevent calling *mmap()* every time a large block is allocated, a large chunk of memory pages are premapped and used during allocation of several large blocks. If a large block is freed and stays in the free list for a prespecified number of garbage collections, then the allocator unmaps this block automatically. This feature allows the program to adapt its memory size to the current work-load.

Allocators that call *sbrk(increment)* can not shrink the heap in the same way. If the block we would like to remove is at the end of the heap we could use *sbrk(increment)* with a negative *increment* to reduce the end of the

heap by *increment* bytes. However, if the unused large block is not at the end of the heap, we will not be able to shrink the heap and swap space will still be committed to this unused large block. This problem can cause long leaved programs to use more swap space and physical memory than they really need, specially in servers that experience different kinds of memory requirements.

4.2 The Collector

A primary goal of our garbage collector is to use it with languages that do not have language cooperation such as C and C++. This is the reason our garbage collector is conservative. In our collector any sequence of bytes that is aligned to a word and that points to the heap is considered a pointer [BW88].

Our garbage collector also supports internal pointers. Any pointer that points to any address inside an object is considered to point to the object. The following algorithm is used to determine the starting address of an object: Given a memory address that points to the heap the page-information entry is obtained. If the size of the object is less than a page, an integer division by that size is used to determine the starting address. Otherwise for multi-page objects, the page-information entry has a pointer to the starting address of the object.

We also would like to use our garbage collector in any program, even when no information about the root set is given. This is a complicated problem because programs and libraries can arbitrarily create memory mappings and store pointers in them. For example, user-level thread packages create new memory mappings to allocate stacks. Also shared libraries use different memory mappings to store bss and data. Keeping track of all the different areas in memory that can be used to store pointers is difficult.

Since no complete information about the root set exists, we decided to choose a conservative root set. In our collector all memory mappings different than the heap and allocator data structures are considered part of the root set. A heuristic we used to reduce the size of the root set is to limit the root set to only the mappings that do not have the executable permission set. This excludes the text part of the program. In systems where dirty bits are available, it is possible also to reduce the root set to only the pages in the root set that have been dirty since the time the program started. If a page in the root set is found not to have pointers, then the page can be cleaned. This heuristic has not been implemented yet.

Process cloning is done in Solaris using the fork() primitive. fork() creates a new process that is a copy of the parent process and that has its own copy of the address space. Internally the copy of the address space is done using VM copy-on-write techniques. The child and parent share the same memory page as long it is not modified. Since only kernel tables are modified in this call, the cost of calling fork() grows only slowly with the process memory size.

There are two bitmaps in our allocator: mark-bits and allocation-bits. Every object has a corresponding mark-bit and allocation-bit in these bitmaps. The mark-bits are used during the mark-phase to mark reachable objects.

Allocation-bits allow running the mark and sweep phases in parallel. An allocation-bit that is cleared means that the corresponding block is available. An allocation-bit that is set means that the block is currently allocated and potentially it is garbage. When the allocator runs out of blocks in a free list, it sweeps one page (or one block if it is a multiple page block) and puts the blocks with the allocation-bit cleared back to the corresponding free list.

When a mark phase starts, the clone marks all blocks that have the allocation-bit cleared to avoid scanning of free objects, that is: $markBits = allocationBits'$. Since the bitmaps are stored as arrays of words, this step is done efficiently using bitwise word operations.

When a mark phase is over, the clone passes the mark-bits to the mutator. Mark-bits are allocated in memory that is shared by both clone and mutator. The clone simply sets a flag in shared memory to tell the mutator that the mark-phase has ended. The mutator checks this flag every time the allocator runs out of pages to sweep. When the mutator realizes that the mark-phase is over, for every block that has its mark-bit cleared the corresponding allocation-bit is cleared: that is $allocationBits = allocationBits \odot markBits$. This step is also done efficiently using bitwise word operations.

This form of lazy sweep is more less similar to the one used by [BDS91]. However the difference is that since they do not have allocation-bits, mark and sweep phases can not go in parallel. Just before starting a new collection and clearing the mark-bits, all the unswept pages have to be sweeped to put back in the free list all

the unmarked objects from the previous collection. This may cause long pauses if there are a lot of unsweeped objects.

4.3 Nonintrusive Garbage Collection

Another goal we had in mind when designing the garbage collector is to be able to run our garbage collector on any program even when the sources are not available. To be able to achieve this goal, we make use of the run-time linker. By default programs in Solaris are dynamically linked to the system shared libraries. One of the system libraries (libc) provides memory management routines that most of the programs use. The run-time linker automatically links these libraries when the program starts execution.

Our garbage collector is implemented as a shared library that provides its own memory allocation calls: malloc(), realloc(), new(), free(), and delete(). In order to make a program use these calls, we tell the run-time linker to link our library before any other system library[4]. In this way, we have been able to run garbage collection even in commercial programs where source code is not available. Even though this form of substitution is limited to dynamically linked programs, most of the programs in modern operating systems are dynamically linked.

Next we show the nonintrusiveness of our collector by running garbage collection in two commercial applications. The applications are *netscape-2.0* and *matlab-4.2a*. Netscape is a popular web browser that is currently used by thousands of people. Matlab is also a widely used numerical problem solver environment. In both cases no sources are available, and therefore no modification is possible. The GC library makes little assumptions about the programs that use the library.

The following shell session incorporates GC in netscape. During the netscape session we browsed several hyperlinks for a duration of 20 minutes. By setting the environment variable EXPLICIT_FREE_OFF we tell the GC library to ignore the calls to free(). Therefore the only way to deallocate objects is through garbage collection. The machine used for all the tests in this section was a multiprocessor SPARC-10.

```
csh> ( setenv LD_PRELOAD /usr/lib/libGC.so; setenv EXPLICIT_FREE_OFF; netscape &)
[1] 15416

csh> Heapinfo 15416
                     Heap   Alive  Root   Total  Max    Total
Collector            Size   Memory Size   Garbag Pause  Pause  #GCs
                     (Kb)   (Kb)   (Kb)   (Kb)   (ms)   (ms)
-------------------- ------ ------ ------ ------ ------ ------ ------
CloningGC              4896   1828    723  14168     31   1179     49
```

We see here that the heap size is about 2.5 times the alive memory. This gives us an estimate of the price in space that has to be paid for using our collector instead of explicit deallocation. One reason for having the heap this large is the memory fragmentation: internal and external. Remember that our allocator uses blocks of fixed sizes without splitting and coalescing. In the other side, once a page is used for blocks of an specific size, it will never be used for other sizes. Another reason is that sometimes the collector allows the heap to grow to run more efficiently. Also notice that the maximum pause is very acceptable: 31ms.

Next, we show the result of running the program matlab with the cloning garbage collector. The output shown was taken after running one of the demo programs. The demo program animates the natural bending modes of a two-dimensional truss. The program computes at real time all the calculations. The animation runs smoothly in the presence of garbage collection. We show next the result of running this program for 10 minutes.

```
                     Heap   Alive  Root   Total  Max    Total
Collector            Size   Memory Size   Garbag Pause  Pause  #GCs
                     (Kb)   (Kb)   (Kb)   (Kb)   (ms)   (ms)
-------------------- ------ ------ ------ ------ ------ ------ ------
CloningGC             11032   6713   2531 1000076     53  25416    606
```

[4]The LD_PRELOAD environment variable specifies the libraries that the run-time linker has to link first before the execution of a program

This application generates lots of garbage. During the ten minutes we ran the application it generated about 1 GByte of garbage. The maximum pause is still unnoticeable. The cost in space is less than twice the amount of live memory. The maximum pause is still acceptable.

However we have found some problems in applications like matlab that store arbitrary sequences of bytes that look like pointers (false pointers). False pointers are specially found in pictures and bitmaps stored in memory. In cases like this, applications that are programmed with garbage collection in mind could pass some information to the garbage collector to tell in these case which objects do not have pointers. In our GC library there is a special malloc call that allocates pointer free objects. We expect that this could reduce to almost none the problem of false pointers. A similar approach is used by [BW88].

5 Performance Comparison

In this section we compare our implementation of cloning garbage collector with the publically available Boehm-Demers-Weiser conservative garbage collector (*BDWGC*) [BDS91][5]. This garbage collector has been ported to a broad number of operating systems and architectures and currently is used by several language implementations and applications.

We use as a benchmark the ghostscript program. Ghostcript is a publically available version of a postscript previewer. The version of Ghostscript and the input files that are used in this comparison were obtained from the Zorn garbage collection test suite [Zor93][6]. We choose ghostscript because it is one of the most problematic programs in the test suite: It has a large heap and it makes extensive use of dynamic memory.

BDWGC can be configured to be either stop-the-world or incremental. The write-barrier of the incremental collector can also be configured to be implemented either by user level dirty-bits or by kernel dirty-bits. When implemented by user-level dirty bits, it write protects all memory pages and catches the faults. When the write-barrier is implemented by the kernel, a special system call gets the read/modified bits of all the memory pages of a specified process. This call is only available in certain operating systems [7].

BDWGC is preemptive. This means that the collection algorithm is executed inside the allocation calls themselves. This has the advantage of automatically slowing down allocations when the collector is falling behind by just spending more time in the collection algorithm. A positive side effect is that by spending more time in the collection algorithm it also prevents the mutator to dirty too many pages when the collection is taking place. Also there is no overhead in context switching between the mutator and the collector thread. However, the disadvantage is that in multiprocessor machines mutator and collector are not able to run in different processors.

The incremental mode of BDWGC works as follows. When it is time to do a collection, the collector clears the mark bits and the dirty bits of all memory pages. Then it runs the mark phase in parallel with the mutator. When the mark-phase is over, it stops the world, reads which pages were dirtied during the marking phase, and then it traces from marked objects in these dirty pages. This is because marked objects in dirty pages can potentially have pointers to reachable objects that have not been marked yet. Once all reachable objects have been marked, the mutator is restarted again.

This basic algorithm has the disadvantage that if the mutator dirties too many pages during the collection, the final pause can be very long. To prevent this problem, BDWGC restarts the world when it detects that the pause is getting too long and continues tracing in parallel with the mutator. Finally it stops the world again and traces from the current dirty pages. It iterates this step as many times as needed.

The complete algorithm used in BDWGC incremental mode is given next:

1. Clear mark bits and dirty bits

2. Mark objects reachable from roots

[5]The Boehm-Demers-Weiser Conservative Garbage Collector and other GC related material can be obtained from ftp://parcftp.xerox.com/pub/gc/gc.html.

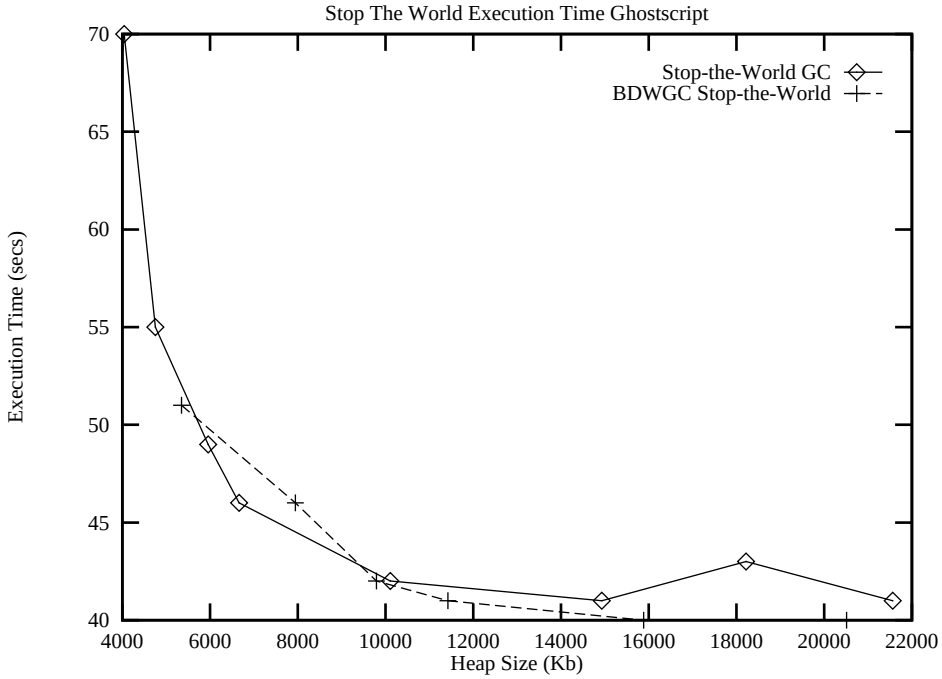
[6]Zorn's garbage collection test suite can be obtained from ftp://cs.colorado.edu/pub/cs/misc/malloc-benchmarks/.

[7]Solaris implements kernel dirty bits through special ioctl() calls to the procfs file system.

3. Stop the world
4. Read dirty bits
5. While there are objects to be marked
   (a) Mark a few objects reachable from marked objects in dirty pages from last read
   (b) If pause is getting too long and there are still objects to mark
      i. Start the world
      ii. End marking objects reachable from marked objects in dirty pages from last read
      iii. Stop the world
      iv. Read dirty bits
6. Start the world

The previous algorithm has two problems. First, it reduces the pause time but it increases the frequency of pauses. In fact when two or more pauses are produced, they are very likely to be almost back to back. Second, multiple iterations of the final marking phase will be needed if the mutator dirties lots of pages per unit of allocation. This will increase the total time the collection takes. If reading dirty bits is an expensive operation, this time will be even longer.

We next compare BDWGC and our collector. Notice that by comparing both collectors we are not only comparing the algorithms but also their implementation. Both collectors use different data structures and mechanisms. However we claim that the critical parts of the algorithm like conservative pointer detection are still equivalent. We show in the next graph that the execution times of running ghostscript and stop-the-world garbage collection are comparable[8]



Using as a baseline the fact that the execution time of stop-the-world garbage collection is comparable in both implementations, we compare next ghostscript using different collectors. The different collectors used are: our

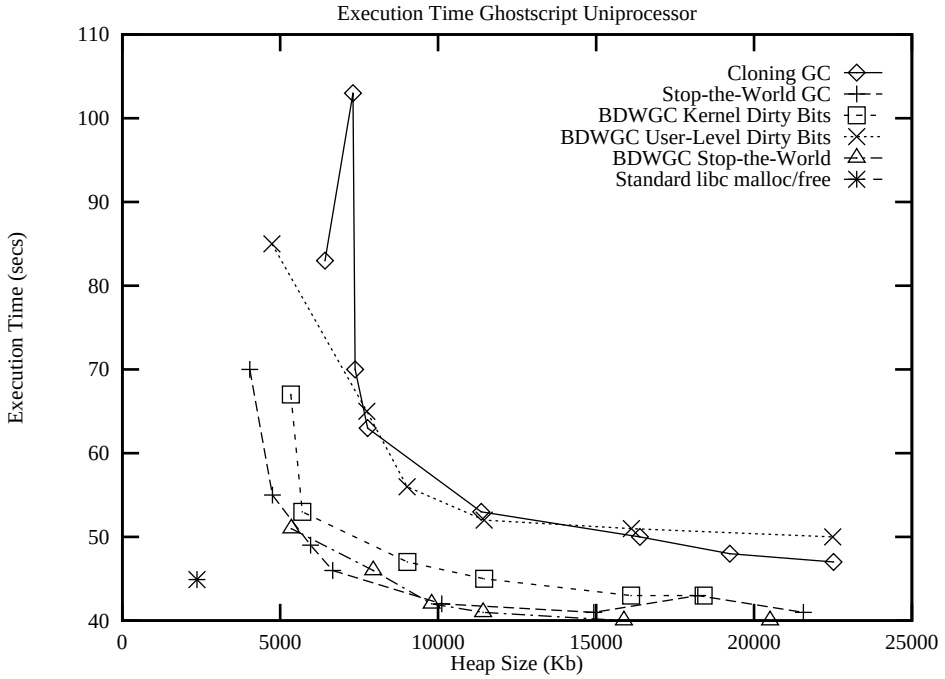
[8]The benchmarks shown in this section were obtained by running the experiment five times and taking the smallest real time. The time was obtained by using the *time* unix command. The CPU usage of the test program in every case was above 96%.

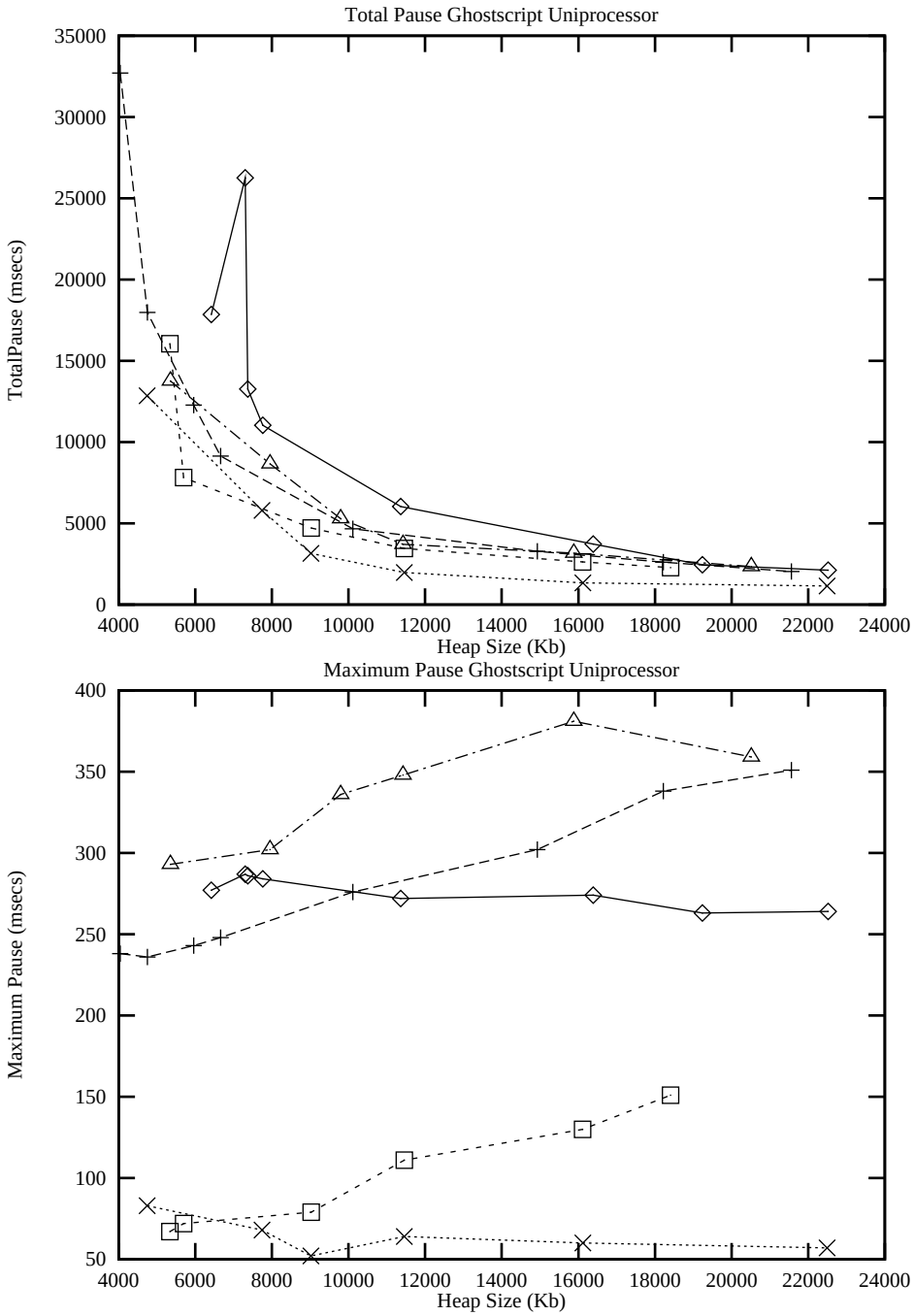
collector doing: 1. Cloning Collection, and 2. Simple Stop-The-World. BDWGC doing 3. Incremental collection using kernel dirty bits, 4. Incremental collection using page protection, and finally 5. Simple Stop-the-World collection.

There are two sets of graphs. The first set shows the execution of the garbage collector in a uniprocessor machine (1-cpu SPARC-10). The second set shows the execution of the collectors in a multiprocessor machine (4-cpu SPARC-10). For each set three graphs are shown: 1. Execution Time, 2. Maximum Pause, and 3. The Total Time Spent In Pauses. This was done for different heap sizes.

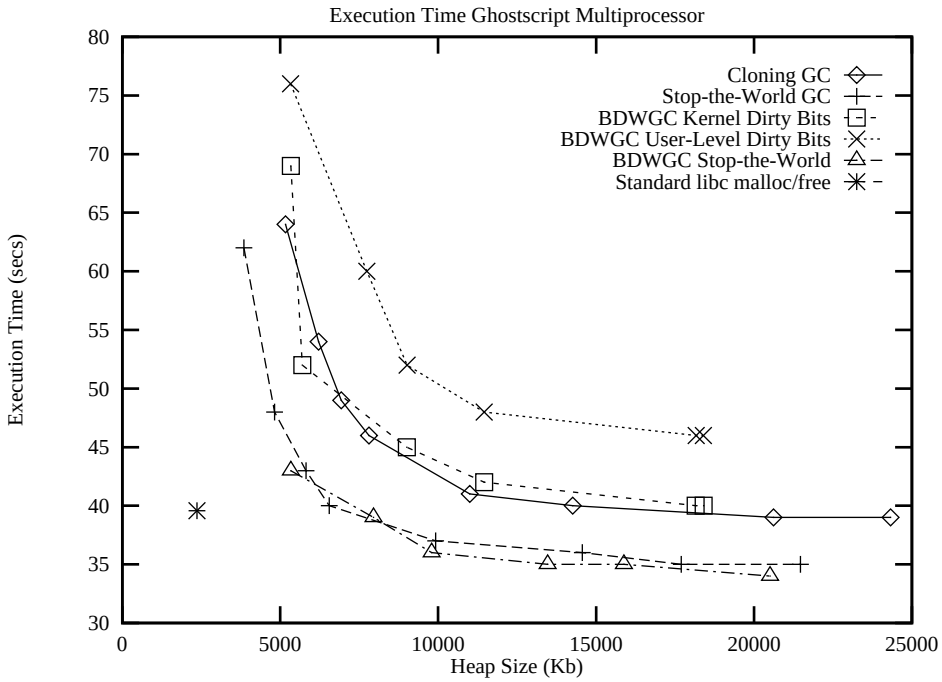
The results of running our cloning garbage collection in a uniprocessor machine are not as good as the results of a multiprocessor machine. However, we can see that the execution time of our cloning garbage collector is smaller for large heaps than BDWGC with user-level dirty bits. We account the difference to the overhead of catching the page faults in user space rather than catching them in the kernel. Also as we expected, the kernel dirty bits version of BDWGC is faster than both user-level BDWGC and our cloning garbage collector. The reason is that in BDWGC with kernel dirty-bits there is no overhead associated to catching page faults. Also in cloning garbage collection, both the garbage collector and ghostscript have to compete for cpu time.

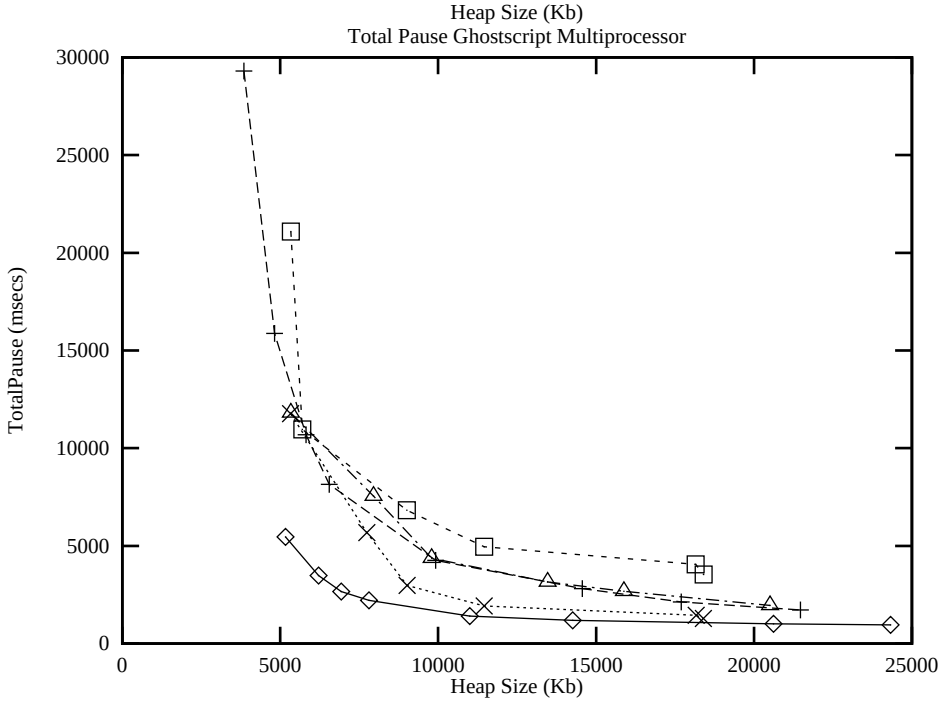
The minimum maximum pause unexpectedly does not belong to cloning garbage collection, but to BDWGC using user-level dirty bits. The reason is that in this case the time it takes to fork a process is equal to ≈250ms independently of the heap size. We show later that this pause time is much smaller in multiprocessor machines where most of the fork system call is executed by a second processor.

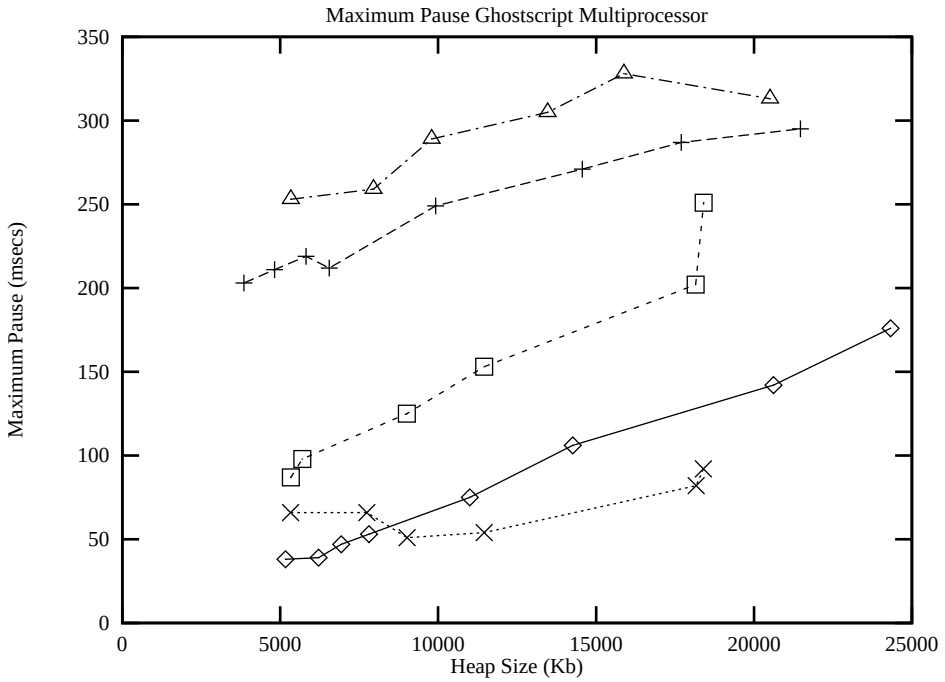




The results look much better for cloning garbage collection in a multiprocessor machine. In a multiprocessor machine the execution time of the cloning garbage collector is much smaller than BDWGC using user level dirty bits. It is also comparable to BDWGC using kernel dirty bits because this time cloning garbage collection uses two processors, one to run ghostscript and the other one to run garbage collection. In the same way, cloning garbage collection has the best total pause, and the smallest maximum pause for small heaps.


Execution Time Ghostscript Multiprocessor
Execution Time (secs)
Heap Size (Kb)
Cloning GC
Stop-the-World GC
BDWGC Kernel Dirty Bits
BDWGC User-Level Dirty Bits
BDWGC Stop-the-World
Standard libc malloc/free
80
75
70
65
60
55
50
45
40
35
30
0
5000
10000
15000
20000
25000


Total Pause Ghostscript Multiprocessor
TotalPause (msecs)
Heap Size (Kb)
30000
25000
20000
15000
10000
5000
0
0
5000
10000
15000
20000
25000



We can learn from these results that running cloning garbage collection in a multiprocessor machine gives us not only good execution times but also very small pause times. Even in uniprocessors machines the results are better than using user-level dirty bits. Even though using kernel dirty bits has smaller execution time than cloning garbage collection in uniprocessor machines, not all OS implementations have kernel dirty bits but most support some form of cloning.

Experience has shown us that it is worthwile running cloning garbage collection in programs that have large heaps and a large number of live objects. In programs with small heaps it may be enough to run stop-the-world garbage collection.

6 Conclusion

We have presented a snapshot-at-beginning garbage collection algorithm called cloning garbage collection and its implementation. This algorithm happens to have good real-time characteristics and a very simple and effective implementation. We showed how our implementation is nonintrusive by running our collector in two commercial applications.

We also compared our implementation with an existing incremental garbage collector. The results show that the execution times are comparable, and that the pause times of the cloning garbage collector are better in multiprocessor machines where collector and mutator run in different processors.

7 Future Work

We are now exploring other problems where cloning can be offered as a solution. Cloning could be used to checkpoint persistent heaps or in distributed garbage collection.

Even though some performance numbers were shown in this paper, we think that more work has to be done to find out the costs of cloning garbage collection.

The implementation of the cloning garbage collector shown in this paper uses conservative pointer finding. However there is nothing that can prevent using cloning garbage collection in environments were accurate information of pointers exists.

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, Georgia, June 1988. ACM Press.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, Santa Clara, California, April 1991.

[AP87] S. Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In E. Chiricozzi and A. D'Amato, editors, *International Conference on Parallel Processing and Applications*, pages 243–246, L'Aquila, Italy, September 1987. Elsevier North-Holland.

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bak91] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. Position paper. Also appears as *SIGPLAN Notices 27*(3):66–70, March 1992.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, Ontario, June 1991. ACM Press.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 108–113, Austin, Texas, August 1984. ACM Press.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[CN83] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, October 1983.

[Com64] W. T. Comfort. Multiword list items. *Communications of the ACM*, 7(6), June 1964.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 261–269, San Francisco, California, January 1990. ACM Press.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter 1992 Technical Conference*, pages 125–136, San Francisco, California, January 1992. USENIX Association.

[HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In Andreas Paepcke, editor, *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, pages 92–109, Vancouver, British Columbia, October 1992. ACM Press. Published as *SIGPLAN Notices 27*(10), October 1992.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1973. First edition published in 1968.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, April 1960.

[Rit84] D.M. Ritchie. The evolution of the unix time-sharing system. *Bell Laboratories Technical Journal*, 63(8):1577–1594, October 1984.

[Ste75] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[Wan89] Thomas Wang. MM garbage collector for C++. Master's thesis, California Polytechnic State University, San Luis Obispo, California, October 1989.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[Wil95] Paul R. Wilson. Garbage collection. *Computing Surveys*, 1995. Expanded version of [Wil92]. Draft available via anonymous internet FTP from `cs.utexas.edu` as `pub/garbage/bigsurv.ps`. In revision, to appear.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11:181–198, 1990.

[Zor93] Benjamin Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 23(7):733–756, July 1993.