# Reducing the Latency of a Real-time Garbage Collector

Ralph E. Johnson

**Abstract** This paper shows how to make the latency of scanning a page in the Appel-Ellis-Li real-time garbage collector be proportional only to the number of object references on a page (the page size), instead of to the sum of the sizes of the objects referenced by the page. This makes the garbage collection algorithm much more suitable for real-time systems.

Garbage collection is especially useful in a long-lived system because the effects of a small program flaw that loses track of memory will accumulate until the system runs out of memory. Many long-lived systems are real-time, but most garbage collection algorithms are not suitable for real-time systems. A real-time system is one that must respond to events within a specified period of time, so the performance of real-time systems must be predictable, often to within a few milliseconds. Most garbage collection algorithms cause unexpected quarter-second or longer delays, and so are not suitable for such systems. In fact, real-time programmers often consider latencies caused by virtual memory to be unacceptable, and so real-time operating systems typically either do not support virtual memory or provide some way to ensure that all of a program is in physical memory.

A garbage collection algorithm's real-time bounds are improved by making it *lazy*, i.e., by breaking large units of work into smaller units of work and delaying the small units of work as long as possible. Mark-sweep garbage collectors and reference-counting garbage collectors can be made real-time without any extra hardware support[DB76, Yua90]. However, the most efficient garbage collection algorithms are based on copying, so that instead of finding garbage objects and deallocating their storage, the garbage collector copies all non-garbage objects to a new region of memory and then deallocates

Author's Address: Department of Computer Science, 1304 W. Springfield, Urbana, IL 61801 USA. *Email*: `johnson@cs.uiuc.edu, uunet!uiucdcs!johnson` *Phone*: (217) 244-0093

the old region[Che70, Ung84].

Cheney's original copying algorithm, from which all other copying algorithms are derived, divides memory into two spaces, and only uses one space at a time. It collects garbage by a depth-first traversal of all non-garbage objects, copying them from *from-space* to *to-space*. Once all non-garbage objects have been copied, it reclaims all the memory in from-space by reversing the roles of from-space and to-space. Real-time versions of this algorithm always perform lazy copying.

Baker's algorithm copies objects one at a time when they are first accessed by the mutator[1] [Bak78]. It checks pointers as the mutator traverses them, and translates a pointer to from-space to its new version in to-space. An object is copied the first time its pointer is traversed, and thereafter traversing a pointer to its version in from-space does not require copying. The real-time latency of this algorithm is the time to copy an object. Baker mentions a suggested improvement by Steele that copies large objects lazily so that the real-time latency would be reduced to the time to copy a fraction of an object. Thus, Baker makes Cheney's algorithm lazy, and Steele makes it even more lazy.

The main problem with Baker's algorithm is that checking a pointer each time it is dereferenced is slow unless there is special hardware or microcode support [Bak78, LH83, Nil88]. The Appel-Ellis-Li algorithm is another lazy version of Cheney's algorithm, but it can use the virtual memory address translation hardware to segregate references to copied objects from those to uncopied objects [AEL88], and so works well on conventional hardware. Instead of trapping uses of pointers to from-space, it traps reads that might obtain a pointer to from-space.

To-space is divided into scanned and unscanned pages, where scanned pages only contain pointers to to-space, but unscanned pages can contain pointers to from-space (see Figure 1). The mutator is only permitted to read pointers to to-space, *i.e.*, scanned pages. The address translation hardware protects the mutator from seeing pointers into from-space by trapping accesses to unscanned pages. The trap handler scans the page by translating each pointer from from-space to to-space, and then resumes the mutator. Translating a pointer to an uncopied object requires first copying the object

[1]The mutator is the program allocating memory and changing memory references, *i.e.*, the program whose garbage is being collected.

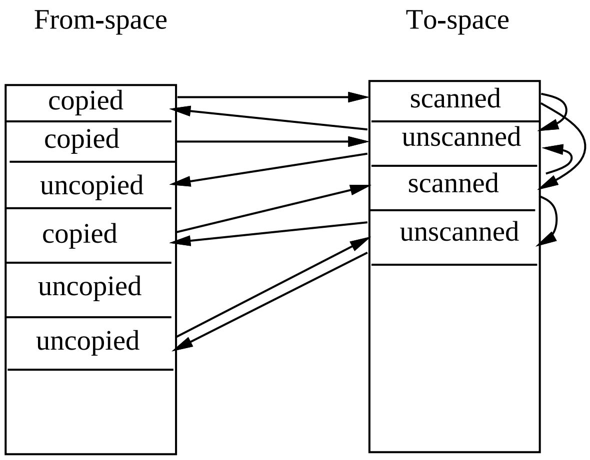


Figure 1: Memory Organization

to to-space, and then using the address of its copy. Thus, scanning a page can require copying at worst all the objects on the page.

The address translation hardware cannot protect each word of memory separately, but instead protects each page of memory. Thus, all the objects on a page must be scanned at once. Pages are not only scanned when the mutator needs to access them, but can also be scanned by a separate process that runs in parallel with the mutator.

One criticism of this algorithm is that the worst case real-time latency is large [Wil91]. The latency is the time to handle a page fault, which includes both the operating system overhead to signal and trap the fault and the time to scan a page. Unfortunately, the time to scan a page includes the time to copy all the objects referenced by any object on the page. Although large objects can be treated specially (as in the suggestion by Steele [Bak78]), even medium sized objects can make the latency large. Suppose that the size of a page is $P$ bytes and a pointer takes 4 bytes. A page can then reference $P/4$ objects. If the maximum size of an object is $N$ bytes then the maximum number of bytes copied is $NP/4$. $P$ depends on the machine, but common values range from 512 bytes to 4096 bytes. If $N$ is 512, this means the maximum delay is the time to copy from 64 to 500 kilobytes, which will take several milliseconds to hundreds of milliseconds on most machines. In

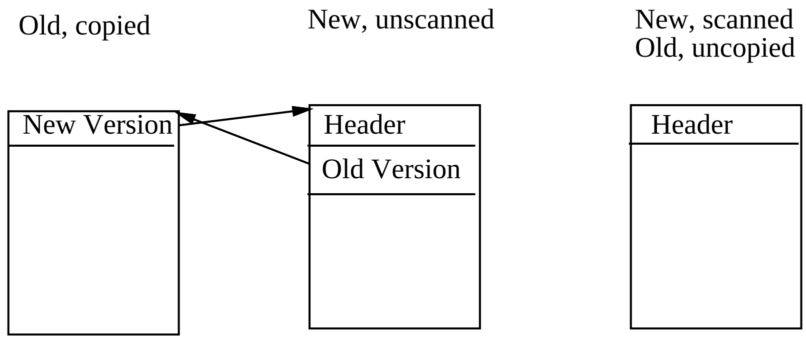


Figure 2: The Three Object Formats

practice, most objects are small, and most objects referenced by a page do not need to be copied, but the worst case latency is large.

This paper shows how to make the A-E-L algorithm more lazy, so it is to the A-E-L algorithm as Steele's suggestion is to Baker's algorithm. However, Steele's suggestion treats large objects specially and so only reduces the latency for copying large objects, while the technique described here treats all objects the same and so will reduce the maximum amount of work per page even if all objects are of medium size. The modifications described here give the A-E-L algorithm a latency that is proportional to the page size.

## An Improvement to the Algorithm

The A-E-L algorithm can be given a small worst-case latency by realizing that copying an object does *not* require copying its contents. Instead, an object can be copied by allocating space for it, initializing its header, and leaving a back-pointer to the data of the object. The object's data is not copied until the object is scanned. This enables an object to be copied in constant time. We will see that no more than one page is scanned at a time. A constant amount of work is done to scan each word of a page. Thus, the worst-case latency will be proportional only to the page size, not to the product of page size and object size.

The format of an object depends on whether it is scanned or unscanned and whether it has been copied. Figure 2 shows the three formats of objects:

copied objects in from-space, unscanned objects in to-space, and a single format for both scanned objects in to-space and uncopied objects in from-space. These formats assume that each object has a header that indicates its size. An object that has been copied will have its first word replaced by the address of the copy in to-space. An unscanned object in to-space includes the address of its original in from-space, in addition to its header. Except for these two words, an unscanned object is uninitialized.

Distinguishing between scanned and unscanned objects in to-space is easy, since the address translation hardware has a table that tells which pages are protected (are unscanned). Distinguishing between copied and uncopied objects in from-space is harder. The best way to differentiate between them is to use a bit in the header of each object. This can cause problems because the header of a copied object stores the address of the new version. There are many simple special-case solutions. For example, usually all objects start on word addresses, and the size of each object is a multiple of the word size. Thus, if the header stores one less than the size then the header of an uncopied object will be odd, but the header of a copied object (*i.e.* the address of its new version) will be even. We assume that some technique is used such that

1. size(obj) returns the size of obj (including its header) and
2. isCopied(obj) indicates whether obj has been copied.

Although Figure 2 implies that a header is a word, the lazy algorithm[2] does not require headers of any particular length, only that these two functions can be implemented.

## Detailed Description

Scanning a page requires scanning each object on the page. Finding the first object on a page is easy if objects never overlap page boundaries, but that implies that the maximum size of an object is a page. If objects can overlap page boundaries then scanning the first object on the page is different than scanning the other objects because the first object may not be entirely on the page. The function to scan a page in to-space (scanPage()) is thus defined in

The original A-E-L algorithm will be called the eager algorithm, while the new version is the lazy algorithm.

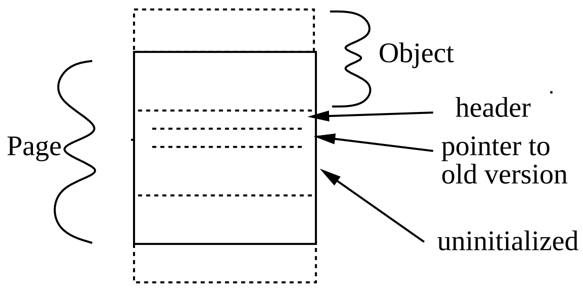


Figure 3: Unscanned objects overlapping page boundaries

terms of two functions scanNextObject() and scanFirstObject(), both of which return the address of the object to be scanned next. Both functions scan only data on the current page. Type object is a record (structure) with fields header and oldVersion.

```
scanPage(int *pageAddress)
{   object *nextToScan, *startFirstObject;
    int *nextPage;
    nextPage = pageAddress + PageSize;
    nextToScan = scanFirstObject(pageAddress, nextPage);
    while(nextToScan != nextPage))
        nextToScan = scanNextObject(nextToScan, nextPage);
}
```

Both scanNextObject() and scanFirstObject() are defined in terms of newVersion(). The function newVersion() translates an address of the old version of an object (the version in from-space) to an address of the new version of an object (the version in to-space). An object in from-space that has been copied contains the address of its new version, so its address is converted by returning the address of the new version. Otherwise, it is “copied” by allocating space for a new version on an unscanned page (assume that the address of the next new object is stored in a global variable nextObject), initializing the new version with its size (header) and the address of the old version, and marking the old version as copied by giving it the address of its new version.

Recall that the data for an unscanned object is stored in its old version. An object is scanned by copying its data from the old version and converting each pointer using newVersion(), which converts a reference to an object in from-space to its new version. Just as a page might start in the middle of an object, it might end in the middle of an object. Thus, scanNextObject() must scan only the part of the object on the page being scanned. In the version of scanNextObject() that follows, old is the address of the next word in the old version to be scanned, new is the address of the location in the new version to which the data will be copied, and end is the start of the next page, *i.e.,* the address of the word after the last word to be written.

```
object *scanNextObject(object * anObject, int *nextPage)
{    int *old, *new, *end;
     old = anObject->oldVersion+HeaderSize;
     new = anObject+HeaderSize;
     end = min(anObject+size(anObject), nextPage);
     while (new < end)
          *new++ = newVersion(*old++);
     return new;
}
```

The function scanFirstObject must find the starting address of the first object on the page. An easy way to find the first object is to have an array that gives the starting address of the object that contains the first word of each page. This costs a word per page, which seems particularly wasteful since most of the words on an unscanned page are unused. John Ellis suggested that the first word of each unscanned page could contain the address of the old version of the first object on the page. This word would be unused unless it were either the first or second word of the object, *i.e.*, either an object header or the address of the old version of the object. It is easy to distinguish object headers from pointers by using a tag. If the first word of the page is a header then the first object on the page starts on the beginning of the page. If the first word of the page is the second word of an unscanned object then it already is the address of the object's old version, so it already is the address of the old version of the first object on the page.

scanFirstObject() scans every word before the first object header on a page and returns the address of the header. It is given the address of the page to scan and the address of the next page. It first checks whether an object starts at the beginning of the page and, if so, returns its address. Otherwise, it calculates the part of the old version that contains data to be copied to the page and scans the part of the object that overlaps with the page.

```
object *scanFirstObject(object *pageAddress, object *nextPage)
{    int *old, *new, *end;
     object *oldVersion;
     if (isHeader(pageAddress->header)) return pageAddress;
     oldVersion = *pageAddress;
     old = oldVersion + (pageAddress - oldVersion->header);
     new = pageAddress;
     end = min(oldVersion->header+size(oldVersion), nextPage);
     while (new < end)
          *new++ = newVersion(*old++);
     return new;
}
```

An object contains both pointer data and nonpointer data, and only the pointers have to be transformed when the object's data are copied. There are many ways to determine whether a value stored in an object is a pointer. These can be divided into techniques that only use the value, such as using a tag bit, and those that depend on the object in which it is stored, such as marking some objects as storing only nonpointers or providing a table for each object that indicates which fields contain pointers. newVersion() adopts the first way: it uses a function isNotPointer that indicates whether a value is a pointer. newVersion() and scanNextObject() can be easily changed to handle the other ways of distinguishing pointers.

The definition of newVersion() uses the primitive function samePage(), which indicates that two addresses are on the same page, startOfPageContaining(), which gives the starting address of the page containing an address, and newAddress(), which marks an old object as copied and sets its forwarding pointer.

newVersion() checks if a value is a reference to an uncopied object. If it is, it "copies" the object by allocating space for it in to-space and copying its header and setting up the back-pointer, and then returns the address of the new version.

newVersion() has to initialize a page whenever an object overlaps a page boundary. This has the potential to be expensive, because large arrays can have many interior pages. However, newVersion() can avoid initializing any of the interior pages by keeping a list of large objects and a bit-array that indicates which pages have been initialized. The page-fault handler will call scanPage() for initialized pages. Uninitialized pages are always interior pages, so will have no object headers nor forwarding pointers from old-space to new-space. Thus, an uninitialized page can be scanned by changing the page table to move the page from old-space and new-space and replacing each word with its new version.[3]

The large object table is always sorted because nextObject increases monotonically. Thus, the object that includes a particular page can be found quickly by a binary search. The space overhead will be one word for each large object and one bit for each page. Even for 128 byte pages, this is unlikely to be more than 0.1% of the memory used.

```
object *newVersion(object *oldObject)
{    object *newObject;
     if (isNotPointer(oldObject)) return oldObject;
     if (isCopied(oldObject)) return oldObject->header;
     /* copy oldObject to newObject */
     newObject = nextObject;
     nextObject = nextObject + size(oldObject);
     if ( !samePage(newObject,nextObject)) {       /* Initialize new page */
          if (size(oldObject) > PageSize) largeObject++ = newObject;
          if (nextObject != startOfPageContaining(nextObject)
               *startOfPageContaining(nextObject) = oldObject;
     }
     newObject->header=oldObject->header;
     newObject->oldVersion=oldObject;
     setNewAddress(oldObject, newObject);
     return newObject;
}
```

Henry Baker suggested changing the page table instead of copying.

## Performance Analysis

Note that newVersion() takes constant time: it has no loops and calls only primitive functions that take constant time. Function scanPage() iterates once for each object on the page, and scanNextObject() iterates once for each word in an object, so scanning a page takes time proportional to the page size.

Determining the exact time to scan a page is complex because there are several paths through scanPage(). scanNextObject() is called once for each object header on the page, while the number of times that newVersion() is called is the number of words on a page minus the number of times that scanNextObject() is called. Thus, the time to scan a page depends on the number of object headers on the page. The total time spent by newVersion() depends on the number of non-pointers or references to copied objects, the number of references to objects that have to be copied, and the number of pages that are initialized during the process of copying objects.

Both the original eager A-E-L algorithm and the lazy version were timed on a SPARCStation 2[4]. The compiler was the Free Software Foundation's gcc version 1.36. Pointers were represented by positive integers and nonpointers were negative. The eager version used nonpointers to represent the headers, but the lazy version (which needs to distinguish headers from the contents of an object) used odd positive integers. For both programs, all functions called by scanPage() were in-lined, and some of the global variables in the lazy algorithm were allocated to registers. The appendix contains the code for the eager version of the A-E-L algorithm.

The time to scan a page was not measured with a normal mutator. Instead, a new version of the heap was generated for each test run and a single page was scanned. This made it possible to control the frequency of the objects on the page being scanned and so determine the cost of scanning object headers, copied objects, and uncopied objects of various lengths.

Figure 4 shows the time to scan several kinds of pages. A page can contain object headers (objects), references to copied objects, and references to uncopied objects of various lengths. For example, case 1 describes a page that has only references to copied objects, and so is one page of a multipage object, the page of case 2 contains objects of length two that refer only to

[2]The SPARCStation 2 was chosen because it provided repeatable times that were accurate to the microsecond.

| | Objects on page | 128 eager | 128 lazy | 256 eager | 256 lazy | 512 eager | 512 lazy | 1024 eager | 1024 lazy |
|---|---|---|---|---|---|---|---|---|---|
| 1 | all copied | 18 | 30 | 35 | 51 | 66 | 94 | 130 | 181 |
| 2 | 50% copied, 50% headers | 15 | 28 | 27 | 48 | 51 | 90 | 98 | 176 |
| 3 | uncopied, 2 words long | 57 | 66 | 110 | 121 | 218 | 232 | 435 | 463 |
| 4 | uncopied, 4 words long | 84 | 73 | 165 | 133 | 327 | 262 | 659 | 507 |
| 5 | uncopied, 10 words long | 165 | 89 | 331 | 161 | 666 | 310 | 1354 | 600 |
| 6 | uncopied, 20 words long | 302 | 96 | 611 | 171 | 1245 | 315 | 2479 | 610 |
| 7 | uncopied, 30 words long | 430 | 111 | 924 | 181 | 1829 | 325 | 3617 | 621 |
| 8 | uncopied, 40 words long | 570 | 110 | 1186 | 185 | 2364 | 335 | 4728 | 624 |
| 9 | uncopied, very long | — | 112 | — | 211 | — | 415 | — | 813 |

Figure 4: Time in microseconds to scan pages of different sizes

copied objects, and the page of case 3 contains references to uncopied objects of length two, but no object headers, and so is a part of a multipage object. Case 9 shows the maximum time to scan a page with the lazy version of the algorithm.

As expected, Figure 3 shows that the time to scan a page increases linearly with page size. It also increases linearly with object size for the eager version. The numbers are repeatable to 1%. 128 byte pages take about a tenth of a millisecond to scan, while 1024 byte pages take eight tenths of a millisecond. Although few commercial machines have page sizes as small as 128 bytes, there are some like the IBM 6000 that provide memory protection to that granularity[Cor90].

The major anomaly of Figure 4 is that it takes much less time for the lazy algorithm to copy objects of size two than would be predicted from the time to copy larger objects. This is because the cache line size on a SPARCStation 2 is four words, so half the memory accesses hit the cache when objects of size two are copied. All the memory accesses miss the cache when large objects are copied.

Cases 1 and 2 show that the eager version of the A-E-L algorithm takes less time to process an object header or a reference to a copied object than the lazy version. On the other hand, the lazy version is only a little slower than the eager algorithm at copying objects of size 2, and has a small advantage for objects as small as 4 words. Its advantage grows with the size of the

| | 128 eager | 128 lazy | difference | 1024 eager | 1024 lazy | difference |
|---|---|---|---|---|---|---|
| 3 pages, objects 2 words long | 87 | 126 | 45% slower | 631 | 825 | 30% slower |
| 11 pages, objects 10 words long | 345 | 389 | 13% slower | 2654 | 2410 | 9% faster |

Figure 5: Total time difference between lazy and eager collectors.

objects, so it is six or seven times faster if every object referenced by the page is 40 words long.

There are several reasons why the eager version of the A-E-L algorithm is faster for object headers and copied objects. First, it only has to increment one pointer, while the modified algorithm has to increment two pointers. Second, the eager version does not have to check for object boundaries because it can treat the object header as a normal nonpointer[App90]. This avoids having an iteration for the object inside the iteration for the page. The lazy version cannot use this technique because the data being scanned is scattered throughout from-space instead of being contiguous. Third, the code for the eager version is smaller and is optimized better by the compiler. Editing the output of the compiler by hand made the lazy version about 5% faster.

These inefficiencies make the total cost of the lazy version greater than that of the eager version. Laziness has the greatest performance benefits when scanning a page pointing to large objects, but the more large objects, the more references to copied objects will appear later. Thus, it is not obvious which algorithm performs the least amount of work over the entire garbage collection cycle.

One way to measure the time of a collection is to measure the time to scan a set of pages with no external references. For example, a page of references to objects of size 2 would be combined with two pages that are 50% headers and 50% copied. Figure 5 shows the results of adding cases 2 and 3 from Figure 4. A page of references to objects of size 10 would be combined with ten pages that are 10% headers and 90% copied. Assuming that the time to scan a page of references to copied objects is about the same as the time to scan a page with 10% object headers, Figure 5 shows the results of adding cases 1 and 5 from Figure 4. Thus, if there are many large objects and the page size is large then laziness has little cost, but with small objects and small pages it can have up to a 45% overhead.

The cost of handling a page fault includes not only the cost of scanning the page, but also the cost of the interrupt and updating the page table to allow the mutator to access the new objects. This paper has concentrated on the time to scan each object on the page. However, the overhead in responding to the interrupt and updating the page table can also be large. This overhead depends on the machine architecture and the way the operating system is implemented. A recent paper by Appel and Li shows that this time can range from less than two thousand instructions to over thirty thousand instructions [AL91]. A real-time system can use small pages to keep the latency of the garbage collector low only if the overhead to handle a page fault and update page tables is low. Most of the operating systems surveyed by Appel and Li had an overhead of nearly a millisecond, which would make them useless for real-time systems requiring millisecond response. However, these operating systems were mostly general purpose workstation operating systems. Operating systems used for hard real-time programming are usually specialized for that purpose and usually do not support virtual memory. Therefore, it is likely that they can be customized to support page faults for garbage collection at a lower cost than on a general purpose operating system.

Smaller pages increase the number of page faults, which increase the total time spent on the overhead of handling page faults. Thus, although smaller pages provide smaller latency, they increase the total time spent handling page faults.

# Remaining Problems

It is important to realize that this paper only provides part of a complete solution to the problem of using the A-E-L algorithm for real-time garbage collection. Delays caused by a garbage collector that uses the A-E-L algorithm are due to

1. the length of time taken by each page fault,
2. the number of page faults, and
3. the amount of time needed to start a new garbage collection.

This paper has shown how to ensure that each page fault takes a small constant time. However, it doesn't address the potential problem of a large number of page faults occurring closely together, nor the problem of ensuring that starting a new garbage collection is fast.

The best way to ensure that there are not many page faults clustered together is to ensure that critical algorithms do not access many pages. Most real-time applications have a small part that has hard real-time requirements. Interrupt routines are a good example. These parts usually access only a few objects. The number of page faults to execute these parts are bounded by the number of objects that they access. Moreover, the number of page faults can be further reduced by ensuring that the objects that will be accessed by a routine with hard real-time requirements will be clustered on a few pages. Of course, this requires some way for the programmer to specify that certain objects are on the same page, and most languages with garbage collection do not provide that capability.

The second part of the garbage collector that can cause large delays is the start of a garbage collection cycle, when to-space becomes from-space. This requires several tables to be reinitialized, including the virtual memory address translation tables and the bit-array that indicates which pages have been initialized. The bit-array is small, and can be initialized quickly. The virtual memory tables are another matter. Appel and Li found that the time to protect a sequence of pages varies considerably [AL91]. This might be a limiting factor on some architectures.

# Conclusion

By delaying the work in scanning a page as long as possible, the A-E-L garbage collection algorithm can have a small bound on the time to scan a page that is independent of the sizes of objects. This means that it should seriously be considered for real-time systems with latencies less than a few milliseconds. There are still problems to be solved, but the algorithm looks promising.

Although the overhead of handling page faults and updating page tables makes the A-E-L garbage collection algorithm less efficient than a simple copying collector, Appel *et. al.* describe how a multiprocessor can perform most of the collecting on a separate processor from the mutator, causing the mutator to take less time for the A-E-L algorithm [AEL88]. Thus, the algorithm can be efficient on a stock multiprocessor.

As Appel *et. al.* describe, this algorithm can be used with standard techniques for improving copying collectors, such as separating generations of objects [LH83, Ung84], separating objects that contain only non-pointers from normal objects [CWB86, UJ88, App90], and providing a separate space for allocating new objects[Ung84]. The changes described in this paper do not require objects to have any particular format. The algorithm is adaptable to nearly any environment, though its efficiency and real-time latency will depend on the machine architecture, the run-time system of the mutator, and the operating system. It shows promise on today's high-end machines for applications that can tolerate millisecond delays.


# Acknowledgement

The main idea of this paper was generated during a conversation with Harold Johnson.

Henry Baker, John Ellis, Jeong Lim, Brian Marick and Carl McConnell provided helpful criticism of earlier drafts of the paper and gave several suggestions that improved the paper considerably. I also thank Referee 1 for prodding me for the performance measurements.


# Appendix

This is the eager version of the A-E-L algorithm that was used to generate the times in Figure 3. Note that all the subroutines have been combined into scanPage.

```
void scanPage(int *pageAddress, object *nextObject)
{   int *oldObject, *newObject, *startOfNextPage;

    startOfNextPage = pageAddress + PageSize;
    while(pageAddress < startOfNextPage) {
        /* convert *pageAddress, possibly copying what it points to */
        oldObject = *pageAddress;
        if (isNotPointer(oldObject)) goto EndOfConvert;
```

```
        if (isCopied(oldObject)) {
            *pageAddress = newAddress(oldObject);
            goto EndOfConvert;
            }
        /* copy oldObject */
        newObject = nextObject;
        nextObject = ((char *) nextObject + size(oldObject));
        newObject = *oldObject;
        setNewAddress(oldObject, newObject);
        newObject++;
        oldObject++;
        while (newObject < nextObject) *newObject++ = *oldObject++;
        pageAddress = newObject;
    EndOfConvert:
        pageAddress++;
    }
}
```

# References


[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 11–20, 1988.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96—107, 1991.

[App90] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3(4):343–380, November 1990.

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

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

[Cor90] IBM Corporation. *POWER Processor Architecture, version 1.52.* 1990.

[CWB86] Patrick J. Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In *Proceedings of OOPSLA '86*, pages 119–130, November 1986. printed as SIGPLAN Notices, 21(11).

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Nil88] K. Nilsen. Garbage collection of strings and linked data structures in real time. *Software–Practice and Experience*, 18(7):613–640, July 1988.

[UJ88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *Proceedings of OOPSLA '88*, pages 1–17, November 1988. printed as SIGPLAN Notices, 23(11).

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[Wil91] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. *SIGPLAN Notices*, 26(3):45–52, March 1991.

[Yua90] Taiichi Yuasa. Real-time garbage collection on general purpose machines. *The Journal of Systems and Software*, 11(3):181–198, March 1990.