# *Reliable Real-Time Garbage Collection of C++*

Kelvin Nilsen Iowa State University

ABSTRACT: Garbage collection of C++ offers the potential of improving programmer productivity, reducing the occurrence of dynamic memory management errors in both prototype and production software, and increasing the level of abstraction provided by many reusable software components. The hardware-assisted real-time garbage collection system described in this paper offers the additional benefits of providing more predictable memory utilization and response times than are available from traditional dynamic memory management techniques for C++. This paper describes a C++ dialect that allows real-time garbage collection of heap-allocated objects in a manner that is compatible with traditional real-time development methodologies. This paper also provides a brief overview of established methodologies for development of reliable real-time software, with emphasis on issues that are relevant to garbage collection, and summarizes the shortcomings of existing real-time garbage collection techniques.

## 1. Introduction

The term *garbage collection* describes the automated process of finding previously allocated memory that is no longer in use in order to make the memory available to satisfy subsequent allocation requests. Automatic garbage collection greatly simplifies the development effort required to manage dynamic memory. In systems that provide garbage collection, programmers need not concern themselves with explicit freeing of memory that is no longer in use. Besides reducing the programmer's intellectual burden, this eliminates two very common dynamic programming errors: the failure to free memory when it is no longer in use, and accidental freeing of memory before its useful lifetime has ended. These sorts of programming errors are especially difficult to find and correct because the consequence of the error is generally not detected at the time the error occurs. Rather, the error manifests itself much later in program execution, usually in one of the following ways:

- The system runs out of memory because of an accumulation of failures to free unused memory.
- The system's dynamic memory manager becomes confused because objects accidently placed onto its free list are still being used. The bookkeeping information that the dynamic memory manager stores within objects on its free list is likely to become corrupted by continued use of the object.
- The application becomes confused because the dynamic memory manager reallocates an object that was erroneously placed onto its free list. Thereafter, the memory serves two different purposes.

Rovner [44] estimates that the programming effort required to perform dynamic memory management is approximately 40 percent of the total cost of developing a large software system. This includes both the costs of developing and of debugging the memory management routines.

Besides reducing the complexity of dynamic memory management, some modern garbage collection algorithms offer storage throughputs that exceed the capabilities of traditional memory management techniques. For example, the time required by the system described in this paper to allocate and reclaim dynamic memory has been measured as approximately one fifth the time required to manage the same total amount of dynamic memory using typical implementations of `malloc` and `free`.

Traditional garbage collector implementations periodically suspend application processing in order to traverse all of memory in search of segments that are no longer in use. The processing delay associated with occasional garbage collections is inconvenient in interactive applications and unacceptable in real-time environments. Incremental garbage collectors allow application processing to continue while garbage collection is performed. But the frequent synchronization that is required between background garbage collection activities and ongoing application processing significantly reduces system throughput [8,36].

One other significant overhead associated with most garbage collection systems is the effort required to tag data stored within the garbage-collected heap so that the garbage collector can traverse live data structures.[1] In dynamically typed languages, like Smalltalk [13] and Icon [14], all data structures are generally tagged to support dynamic type checking, so the tags required for garbage collection are available without any additional run-time overhead. However, in statically typed languages such as C++, the burden of tagging data in order to support garbage collection has been measured to nearly double the execution time of many real programs [45]. For a combination of these reasons, many developers feel that garbage collection is a luxury they simply cannot afford.

## 2. *Real-time Computing*

Typical real-time computer systems must run uninterrupted for days or months at a time. High reliability and fault tolerance of both software and hardware are of utmost importance. In many cases, human lives and significant company revenues depend on reliable operation of real-time computer systems. In order to ensure reliability, software engineers have developed formal techniques and methodologies that allow pre-runtime analysis of real-time scheduling constraints. Properly engineered real-time software is guaranteed to meet all real-time deadlines, even under worst-case workloads.

In general, the real-time software engineer is more concerned with ensuring the system's reliability than optimizing its throughput. In fact, real-time characteristics of the physical environment with which the real-time computer system

1. Conservative garbage collectors avoid the cost of data tagging by assuming that every memory word and register that contains a value representing a legal address is, in fact, a pointer to an object residing in memory.

interacts generally provide both upper and lower bounds on the rates at which data is available for processing. Consider, for example, a real-time computer system that has been configured to handle its worst-case workload of 10,000 interrupts per second. In this environment, there may be nothing to be gained by increasing the system's throughput to support up to 20,000 interrupts each second. This is because the device that generates interrupts is presumably unable to generate any more than 10,000 interrupts per second.

Nevertheless, performance considerations are a concern to real-time engineers for the following reasons:

- The schedulability analysis of a real-time system depends on the worst-case execution times of each real-time task that participates in the system. Each task's workload is represented by the product of the task's worst-case execution time and its desired execution frequency. To guarantee that all real-time deadlines in the system will be met, the software engineer must demonstrate that the combination of task workloads is less than some fixed percentage (generally less than 100 percent) of the system's available computing resources. If performance is poor, the engineer cannot guarantee compliance with real-time deadlines.

- Because it is difficult to derive tight upper bounds on the worst-case times required to execute particular tasks, modern real-time systems are likely to be comprised of a combination of essential and optional real-time tasks [21,26]. Rigorous proofs guarantee that essential tasks never miss their deadlines. But optional tasks are executed only as time permits. These systems are designed so that the essential tasks provide the minimal functionality required to sustain robust operation of the system. But system functionality is greatly enhanced by completion of as many optional tasks as is possible. Under normal circumstances, essential tasks complete in much less time than has been set aside for their worst-case execution requirements, and a large percentage of the optional tasks also complete on schedule.

- Certain dynamic real-time systems are composed of tasks that are prioritized according to importance [29]. As the system load increases, the run-time task scheduler automatically sheds some of its workload by not dispatching low-priority tasks for which it cannot guarantee compliance with the relevant real-time deadlines.

- Some general-purpose computing environments support combinations of hard and soft real-time, traditional interactive, and batch processing tasks [28,40,47]. In these systems, real-time behavior is provided by giving highest priority to the real-time components of the system. General-purpose computing needs are best served when real-time components complete ahead of schedule.

In summary, the engineer of real-time software systems is concerned primarily with reliability and secondarily with optimization of system throughput. Whenever possible, it is desirable to engineer a system that provides both predictable compliance with real-time deadlines and high throughput. Tradeoffs between predictable performance and high throughput are permitted only insofar as they do not inhibit the engineer's ability to prove compliance with the system's real-time deadlines.

Numerous techniques have been developed for scheduling real-time tasks and for analyzing real-time systems of tasks to determine whether all scheduling constraints will be satisfied at run time [9,24–26,29,48,49,53,54,56]. Regardless of which analysis and scheduling techniques are used, it is essential that the real-time engineer be able to determine the worst-case execution times of all the real-time tasks that compose the system. A variety of techniques for analysis of task execution times is available [3,15,20,23,30,32,35,41–43,46].

## *2.1. Soft Real-Time Systems*

Real-time engineers often distinguish between hard and soft real-time systems. In hard real-time systems, a correctly computed result delivered after its deadline is as incorrect as an incorrectly computed result. In soft real-time systems, it is desirable to deliver all computed results prior to their deadlines, but a result delivered late is better than no result at all. Coordination with traffic lights is an example of a hard real-time process. A driver who enters an intersection after his light has turned red is just as wrong as the driver who ignores the traffic light entirely. Multimedia teleconferencing provides an example of soft real-time constraints. Users would generally prefer delayed responses over complete communication failure.

Existing real-time systems are often composed of combinations of hard and soft real-time components. When developing soft real-time systems that contain no hard real-time components, software engineers have the option of entirely ignoring traditional methodologies for design and analysis of real-time systems. However, even when an occasional missed deadline will not lead to catastrophic results, it is desirable to use established real-time methodologies in the design and analysis of the soft real-time system. There are several reasons for this:

- Without systematic real-time design, it is difficult to characterize the worst-case workloads to which the system should be subjected in order to characterize its performance limitations. If testing reveals shortcomings in the real-time performance, there will be no systematic mechanism to isolate and modify the offending sections of code.

- Without formal analysis, exhaustive testing is required to determine the system's throughput and response time limitations. Rarely is it practical to test all possible workload and input conditions. Thus, it is difficult to quantify the likelihood that all deadlines will be satisfied.
- Whenever the soft real-time system is modified and/or integrated with other software, its real-time behavior is likely to be affected. Without the use of formal real-time analysis techniques, extensive testing would be required to build confidence that the revised system continues to function correctly.
- Whenever the soft real-time system is ported to new architectures, its real-time behavior must be reanalyzed. This is especially troublesome if the software system is ported to a less powerful architecture than the one on which it was originally developed, a common practice amongst developers of embedded real-time systems.

In summary, software engineers who want to control the real-time behavior of software need to organize their systems so as to facilitate systematic analysis of the system's real-time behavior. Software engineers who want merely to write software that runs as fast as is feasible need not bother with real-time methodologies.

## *2.2. Sample Real-Time Workloads*

This section provides brief summaries of several actual working real-time systems [7,18,19,22,27]. The purpose of this discussion is to indicate the timing resolution that is typical in today's state-of-the-art real-time systems. We focus here only on task execution times and scheduled processor utilization. Our summary presentation ignores interprocess precedence and exclusion constraints, preemption constraints, and scheduling techniques.

Each real-time system is organized as a collection of repetitive tasks. The times required to execute the tasks that run on the HARM missile guide processor range, for example, from 50 $\mu s$ to 6.83 ms. Each task has a different execution frequency. The periods of execution for the tasks that run on the HARM missile's guide processor range from 1 to 80 ms.

It is especially important in a real-time environment for each task to honor its advertised worst-case execution time. If a particular task exceeds its allotted time segment, every real-time task in the system is likely to be compromised. The analyses that guarantee compliance with real-time deadlines assume that all tasks are well behaved. Note that most of the applications summarized in Table 1 have tasks with execution times much shorter than a typical time-slice tick. Thus it is difficult for an operating system kernel to enforce that tasks terminate within the time frames that they've advertised as their worst-case execution times. With special hardware, it is possible to force real-time tasks to honor their time commitments. The integrated MWave/OS system, for example, has hardware that counts the machine cycles required by each task, and interrupts the task if it exceeds the time allotted for execution of the task [19]. Because a finite amount of time is required to handle a timeout interrupt and effect a fully general context switch (In the best of circumstances, context switches cost from 10 to 50 $\mu s$ [1]), the scheduler must increase the amount of time allotted to whatever tasks it might need to interrupt. Note that these context-switching costs can represent a large percentage of a system's workload, since several of the applications described in Table 1 have tasks whose execution times are less than or equal to the time required to perform kernel-induced context switches.

Table 1. Representative Real-Time Workloads.

| Application | Number of Tasks | Execution Times (ms) | Periods (ms) | Scheduled Utilization |
|---|---|---|---|---|
| HARM missile—Guide processor | 18 | 0.05 – 6.83 | 1.0 – 80.0 | 86% |
| HARM missile—Trace processor | 7 | 0.05 – 3.25 | 1.0 – 20.0 | 85% |
| HARM missile—Sort processor | 7 | 0.05 – 2.14 | 1.0 – 10.0 | 85% |
| Robot elbow manipulator (version 1) | 103 | 0.01 – 0.57 | 16.7 | 28%† |
| Robot elbow manipulator (version 2) | 90 | 0.073 – 0.7496 | 16.7 | 35%† |
| Ada avionics | 17 | 1.0 – 9.0 | 25.0 – 1000 | 85% |
| Satellite Control | 14 | 0.18 – 52.84 | 0.96 – 1000 | 85% |
| Multimedia | 24 | 0.0024 – 0.491 | 1.6 – 118 | 40% |

† The robot elbow manipulator is implemented by five processors, with average utilizations for each processor as reported above.

In summary, if a real-time system is configured to enforce worst-case task execution times, schedulable utilization will decrease (due to the overhead of timeout-induced context switching), hardware costs will increase (in order to pay for the high-resolution timeout device), and reliability will suffer (because it is difficult to predict how frequently important tasks might have to be interrupted before completing the work assigned to them). In general, the developer of cost-effective reliable hard real-time systems prefers to derive reliable upper bounds on the times required to execute each task so that she can trust each task to complete its work and relinquish the CPU prior to termination of its allotted time segment.

## 3. Real-Time Garbage Collection

Numerous proposals for real-time garbage collection have been put forth by the garbage collection research community. However, very few of the proposed techniques are compatible with existing methodologies for implementation of reliable real-time systems. The real-time engineer must be able to derive the worst-case execution time for each task in the system. Tasks interact with the garbage collector by allocating new memory and by fetching and storing to memory locations contained within heap-allocated objects. The delays associated with memory allocation and memory access must be bounded by constants that are small enough to allow the real-time schedule for the complete system to be analyzed or precomputed. If the garbage collector runs on the system CPU, its computation must be modeled as a periodic task with bounded execution time and a fixed period of execution. The performance impact of garbage collection on the complete system must be small enough that it does not prevent the system from meeting its real-time deadlines.

Most of the garbage collection researchers who have proposed so-called real-time garbage collectors rely on personalized definitions of real-time computing, usually without formalizing their definitions. Real-time practitioners need absolute worst-case bounds on the times required to fetch, store, and allocate memory. Practitioners cannot build reliable hard real-time systems based on average- or expected-case costs. Note from examination of Table 1 that it is difficult for many real-time workloads to absorb the virtual-memory trap costs associated with certain "real-time" garbage collectors [10]. Assume, for example, that the worst-case cost of each memory fetch and store operation is 500 $\mu s$. Suppose further that the 500 $\mu s$ delay associated with these memory operations cannot be interrupted by other tasks that make use of the dynamic memory heap. In both the HARM missile and the satellite control applications, for example, several tasks have periods of approximately 1 ms. These tasks would be unable to meet their deadlines if certain other tasks are delayed on consecutive memory operations. Consider also the impact of occasional 500- $\mu s$ trapped memory accesses on the calculation of worst-case execution times. Note that all of the worst-case execution times in the robotics and multimedia applications are less than 1 ms. If each of these tasks were to incur the worst-case overhead of only two memory accesses to the garbage-collected heap, the resulting system utilization would more than double. In fact, the resulting utilizations would exceed 100 percent.

If we allow the community of real-time scientists and practitioners to characterize the operating constraints under which a real-time manager of dynamic memory must operate, then we must conclude that most so-called real-time garbage collectors are not real-time at all. Developers of reliable real-time systems need to know the worst-case execution time of each real-time task that composes the system. This requires upper bounds on the times required to allocate and access dynamic memory. Furthermore, these bounds must be tight, so that the expected performance is as close as possible to the worst-case performance. Otherwise, the real-time functionality that the system designer guarantees to support is so low that the complete system would likely be rejected on the grounds that it is not cost effective.

### *3.1. Stock-Hardware Real-Time Garbage Collection*

Real-time garbage collectors allow application processing to continue while garbage collection is performed. Because the application continues to execute while garbage collection is carried out, it is necessary to coordinate the efforts of ongoing application processing with garbage collection. Often, the garbage collector desires to relocate and/or modify the memory structures that application processing desires to make use of. The frequent synchronization that is required to coordinate shared access to the garbage-collected heap significantly reduces system throughput [8,36] in comparison with traditional stop-and-wait garbage collection techniques. Without special hardware, there are several synchronization techniques that are available. These include:

1. The code generator can emit special range-checking code to accompany individual memory fetch and store operations. At run time, the generated code provides special handling whenever it detects a memory access that may require coordination with the garbage collector. This technique was implemented in an earlier software-only version of the garbage collection algorithm that we've now implemented in hardware [36]. We found that the overhead imposed on all memory operations referring to the garbage-collected heap more than doubled the cost of executing typical applications. Other researchers have reported similar results [6,10].
2. Custom microcode can automatically perform the requisite checks each time memory is fetched or stored. This is the technique used in certain Lisp machines [51]. On typical CISC processors, this offers higher performance than option 1. However, the run-time overhead is still so large (system throughput is reportedly slowed by 30%) that most users preferred to disable real-time garbage collection in favor of more traditional batch

techniques [10,51]. Note that this option is not available on today's high-performance RISC processors.

3. Virtual memory protection can be configured to generate page faults on any memory accesses that refer to regions requiring synchronization with the garbage collector. This technique was first proposed by Appel, Ellis, and Li [10]. Though this technique has demonstrated high performance running in a custom multi-threaded environment, the cost of servicing page faults in traditional Unix processes is so high that overall throughput suffers. Moreover, the resolution of page boundaries is too coarse to support tight latencies on all memory operations. Appel, Ellis, and Li report worst-case latencies measured in tenths of seconds.

Options one and two show the greatest potential of providing tight worst-case bounds on the time required to perform memory operations, but few applications can afford their high overheads. Two recent publications describe real-time garbage collection techniques that attempt to reduce system overhead by allowing all read operations to proceed unchecked [34,55]. Since reads are much more frequent than writes, especially for programs written in applicative languages like Lisp and SML, these techniques show some promise of improving upon prior experience with explicitly checked memory operations. However, the potential performance gains are not as large for applications that frequently mutate existing memory cells, as is common with programs written in imperative and object-oriented languages such as Icon, Smalltalk, and C++. Yuasa's technique targets Lisp [55]. As such, Yuasa recommends that the dynamic heap be divided into regions, each dedicated to allocation of fixed-size objects of a different size. Yuasa's garbage collection system does not compact live memory, and does not support dynamic reconfiguration of the different heap regions. This artificially imposed memory fragmentation reduces memory utilization, constrains application flexibility, and complicates the software engineer's analysis of the system's worst-case memory requirements. Nettles, O'Toole, et al. have recently described a replication-based copying garbage collector that maintains two copies of every live object throughout garbage collection [33,34]. The algorithm resembles Baker's copying technique except the invariants that govern sharing of information between the application and the garbage collector are much more flexible. In theory, read operations may fetch from either copy of the object, and writes must update both. In practice, read operations are constrained to refer to the *from-space* objects, and write operations are logged so that *to-space* objects can be updated in batch. This garbage collector is real time in the sense that the time required to perform an atomic increment of garbage collection is bounded by a tunable constant. Nettles and O'Toole have demonstrated measured worst-case times of 50 $\mu s$ per atomic action [34]. Additionally, this technique has exhibited good average-case performance on several sample workloads. For applications written in SML, Nettles and O'Toole report that their replication-based garbage collector slows overall execution by less than 25 percent in comparison with stop-and-copy garbage collection techniques for SML [34]. However, the worst-case real-time behavior of the current implementation has not yet been analytically determined. Though replication-based garbage collection shows promise of efficiently supporting reliable stock-hardware garbage collection, the current analysis of its real-time behavior is incomplete. Furthermore, this garbage collector's average-case performance has not yet been measured on C++ code, which is likely to contain a much higher percentage of memory write operations than is typical of SML applications. Finally, it is important to recognize that a high percentage of the overhead associated with accurate garbage collection of C++ is the cost of maintaining run-time type tags [45], regardless of how inexpensive memory fetches are, or how efficient garbage collection may be. Given these circumstances and difficulties, it appears unlikely that stock-hardware real-time replicating garbage collection of C++ will be able to offer the same levels of general-purpose performance and real-time reliability as are provided by the hardware-assisted copying garbage collection system described in this paper. Nevertheless, real-time replicating garbage collection shows promise of effectively serving the needs of many specialized applications.

The third alternative mentioned above shows potential for high performance, but suffers from very high variance in the time required to perform read and/or write operations. Each memory read or write operation may incur the overhead of a page trap. Recent measurements of idealized trap handlers running on current RISC processors reveal that trap handlers require an average of 10-50 $\mu s$ [1]. The worst-case costs would, of course, be much higher. In particular, the worst-case trap handler would miss its cache on every memory operation, would need to wait for floating point pipelines to flush before servicing the trap, would need to save and restore a large number of machine registers, and would need to modify virtual memory protection levels. Note that current trends in computer architecture, which are characterized by increasingly large register files and caches combined with deeper pipelines and multiple instruction issue, exacerbate the problem of minimizing the variance in memory operation processing times. Though numerous garbage collection systems make use of virtual memory protection to improve system throughput [5,10,12,16], we are not aware of any that provide rigorous analysis of the worst-case times required to both allocate and access dynamic memory. Of the systems that have been carefully measured, the best response latency that we are aware of is 500 $\mu s$ [12]. It is important to recognize that real-time engineers are unlikely to accept measured performance as a reliable indication of a component's worst-case run time. They have learned through years of hard-earned experience that worst-case execution times are rarely seen in the laboratory. Rather, worst-case performance of individual components is usually seen only when the complete system is most heavily stressed. Unfortunately, it is when the system is most heavily stressed that reliable operation of the complete system is most critical. It seems unlikely that future garbage collection systems that rely on virtual memory protection will be able to provide sufficiently small rigorous bounds on worst-case response latencies without sacrificing overall throughput.

In summary, none of the stock-hardware garbage collection systems currently available offers the combination of capabilities required for reliable real-time garbage collection of C++. Nobody has yet demonstrated a complete working system accompanied by careful analysis of worst-case execution times.[2]

It would appear that existing stock-hardware garbage collection techniques are more suitable for interactive applications than for hard real-time programming. It is possible that some existing garbage collection techniques might generalize to soft real-time application domains. However, this will necessitate the creation of new real-time design and development methodologies. Furthermore, whatever methodologies might be developed will need to give special care to the bursty nature of garbage collection performance. Regardless of how these conflicts are eventually resolved, it would be beneficial for real-time and garbage collection researchers to both become more familiar with each other's domains of expertise.

## *3.2. Accurate Garbage Collection of C++*

We use the term *accurate garbage collection* to describe garbage collection techniques in which the garbage collector has full knowledge of which memory cells contain pointers and which don't, and uses this knowledge to accurately trace only the dynamic objects that are referenced by memory cells known to contain pointers. We emphasize this difference because most garbage collectors for C++ use conservative garbage collection techniques, in which every memory cell that holds a value representing a legal address is treated as a pointer [4]. The real-time garbage collector described in this paper performs accurate garbage collection with

2. Paul Wilson is currently pursuing a stock-hardware real-time garbage collector for C++ [52]. His technique shows promise of supporting tight bounds on memory access times, but makes no attempt to defragment memory in order to bound the time required to allocate new objects (or to bound the memory required to support a particular application). Ongoing research attempts to characterize the run-time overhead of this system. It seems unlikely that the overhead will be as low as what is currently provided by the hardware-assisted real-time collector described in this paper.

low synchronization and tolerable pointer tagging overheads, while guaranteeing to complete all memory fetch, store, and allocate operations in less than 1 $\mu s$. The system makes use of special hardware placed within an expansion memory module to achieve high throughput and tight worst-case bounds on the time required to perform particular operations. If the special hardware is mass produced, we estimate the cost of a hardware-assisted memory module to be three to five times the cost of the memory required to represent the same total amount of live data, assuming that the data is perfectly packed[3] [39]. But it is rare in practice for high-performance memory managers to achieve 100 percent utilization of memory. Rather, typical memory utilizations range from 17 to 85 percent [59]. Cost comparisons between the hardware-assisted memory manager and traditional memory management techniques must take these factors into account.

Our system garbage collects all of the objects allocated within the C++ dynamic heap without requiring any changes to C++ syntax. There are a few C++ practices that must be avoided in order for the garbage collector to operate correctly[4]:

1. Pointers should not be coerced to integers. A common practice is to use object addresses as hash values. Since our garbage collector relocates objects to eliminate fragmentation, object addresses are not constant.
2. Integers should not be coerced to pointers. Whenever the garbage collector relocates an object, it automatically updates all pointers that refer to that object so that the pointers refer to the object's new location. Any pointers hidden within variables declared as integers will not be updated.
3. All pointers need to refer to addresses contained within the objects they point to. In case a pointer refers to an array, the pointer may point to an imaginary element appended to the end of the allocated array. (This restriction is already specified in the C++ standard; we mention it here for emphasis.)

3. We have recently redesigned the hardware-assisted memory module so that it supports a hybrid garbage collection algorithm that collects garbage in certain regions using incremental mark-and-sweep methods and collects garbage in other regions using real-time copying techniques [38]. Though we have not yet measured this new algorithm's throughput, we expect that overall performance will be roughly equivalent, if not superior, to the fully copying real-time garbage collection technique that we have already measured. The main motivation for this alternative garbage collection system is to reduce the amount of real memory required to support a particular application's dynamic memory needs. Whereas the original design costs three to five times more than the cost of the perfectly packed memory required to represent an application's worst-case memory needs, the revised design is likely to cost only 20 to 50% more than the perfectly packed memory.
4. View these requirements as principles of operation rather than hard-and-fast rules. For example, there are many situations under which it is perfectly reasonable to coerce between integer and pointer types. In these cases, it is the programmer's responsibility to verify that the code does not violate the integrity of the garbage collector.

4. Every assignment to a union field that represents both pointer and non-pointer data must be visible to the compiler as a union field assignment. Assignments by way of a pointer to the union field violate this constraint.

5. The structure of every heap-allocated object must be represented by the argument to `new`.

Our experience porting existing C++ code to the garbage-collected C++ implementation suggests that most existing C++ code already complies with these restrictions. For example, the troff component of James Clark's groff implementation is compiled from over 23,000 lines of C++ code. To make this compatible with our garbage collector, we rewrote only four lines. In each case, the original code `new` allocated an object as an array of characters and coerced the resulting address to a structure pointer, violating the fifth constraint listed above.

Recently, we have also ported several allocation-intensive C programs to our garbage-collected C++ dialect. The programs are:

| | |
|---|---|
| `cfrac` | Factors arbitrarily large integers using the continued fraction method. |
| `cham` | Routes channels for multi-level printed circuit boards. |
| `espresso` | Optimizes binary logic expressions. |
| `gawk` | An interpreter for the AWK programming language. |
| `ptc` | A translator from pascal to C. |

These programs were obtained from Benjamin Zorn (`ftp.cs.colorado.edu`), who has studied them in comparing the performance of several alternative dynamic memory management implementations [57,59]. In order to analyze these programs in our environment, it was necessary to:

1. Transform existing Kernighan and Ritchie code to the ANSI C standard. We used the tools `protoize` and `cproto` to automate most of this effort.

2. Rename identifiers that collide with C++ keywords, such as `new`, `delete`, and `class`.

3. Replace all occurrences of `malloc` with appropriate invocations of `new`.

4. Replace all occurrences of `free` with `delete`.

In addition to the work required to convert C to C++, summarized above, additional effort was required to enforce compliance with the garbage collector's operating constraints. This work consisted of:

Identifying code that might violate garbage collection constraints:
: In theory, it would be possible for a lint-like tool to assist with this effort by locating all implicit or explicit coercions between pointer and integer types, and between pointer types possessing different type signatures. This same tool would also be able to identify all source code locations where a union field's address is taken. The lint-like tool would not be able to identify subscript-out-of-bounds errors.

Revising the code so that it is compatible with the garbage collector:
: Though this has been straightforward for many applications, the effort required to rewrite legacy code is sometimes very large.

Our experience porting C code to our garbage-collected C++ dialect has not been as positive as our experiences with C++ code. In particular, C programmers appear to be much more willing than C++ programmers to abuse the language's type system. For example, the author of `cfrac` implemented his own dynamic memory manager to improve upon the performance of `malloc` and `free`. His code to free the unneeded object located at address `u` includes the following:

```
typedef struct precisionType {
  short alloc, size, sign;
  char value[1];
} *precision;

typedef struct {
  struct precisionType *next;
  short count;
} *cachePtr;

cachePtr kludge;
precision u;

...

((cacheType *) u)->next = kludge->next;
```

Before the above assignment, the first word of the object referenced by `u` represents two 16-bit integers. After the assignment, the word represents a pointer. This form of type coercion is not visible to the compiler, and thus confuses the garbage collector.

We discovered similar problems with the implementations of `ghostscript` and `perl`, two additional programs whose use of dynamic memory has also been studied by Zorn. In these programs, we considered the amount of work required to enforce compliance with the garbage collector's constraints to be more than we were willing to commit. The difficulties were both in trying to understand the offending code and in trying to find straightforward ways to rewrite it in a way that does not violate the garbage collector's operating constraints. In `ghostscript`'s implementation, for example, we found several situations in which the type of an object to be dynamically created was separated from the corresponding `malloc` invocation by several levels of function calls. We also found several examples of various memory allocation functions being invoked through function pointers.

Though our particular implementation of C++ garbage collection uses special hardware support to provide hard real-time response, the requirements enumerated in this section are sufficient to allow efficient software garbage collection as well. In order to experiment with alternative code generation strategies for stack activation frames, we are currently modifying version 2.4.5 of the GNU C++ compiler to support stock-hardware garbage collection of the same C++ dialect that is currently supported by our hardware-assisted garbage collection system. None of the current designs for software garbage collection is capable of honoring the same real-time constraints as our hardware-assisted system. However, many software garbage collection systems offer overall performance that appears to be as good or even better than the performance of our hardware-assisted system [5,10,33,50,58].

### *3.3. Standardization Efforts Regarding Garbage Collection of C++*

Recently, John Ellis and David Detlefs proposed a standard for garbage collection of C++ [11]. Their proposal is not accompanied by an implementation. The requirements for use of our garbage collection system are much simpler than the proposed standard. The design constraints outlined by Ellis and Detlefs specify that C++ garbage collection must require *minimal changes* to the C++ language, its implementations, or its programming styles; must *coexist* with components written in other languages such as C or Fortran; must be *safe* in the sense that the rules for garbage collection must be clearly defined and enforceable at compile time; must be *portable* in the sense that programs that comply with the rules of garbage collection safety should run correctly on any C++ implementation that supports the garbage collection standard; and must be *efficient*, in the sense that the standard does not require semantics that is costly to implement. Below, we compare our garbage collection technique with the proposed standard in terms of these five criteria.

Minimal Changes: The standard proposed by Ellis and Detlefs introduces two new keywords and overloads the semantics for object destruction. The proposed standard's definition of GC safety includes our requirements 1, 2, 3, and 5. The standard does not impose our requirement 4, arguing that portable support for unions is too difficult to implement efficiently. Rather, they suggest that union fields must be scanned conservatively. In effect, the proposed standard disallows accurate garbage collection of programs that make use of unions containing both pointer and non-pointer fields. In terms of impact on the C++ language definition, our proposed garbage collection technique is at least as minimal as the Ellis/Detlefs proposal.

On the other hand, the C++ implementation described in this paper requires significant modification to the compiler's code generation and optimization components, and depends on special hardware. We argue that this is the only way to provide reliable high-performance real-time garbage collection of C++. It is important to note that software implementations of our garbage-collected C++ dialect are possible. We are currently implementing a stock-hardware accurate garbage collector for C++. Furthermore, the Boehm-Weiser C++ garbage collector is capable of garbage-collecting our C++ dialect with even fewer changes to a C++ compiler than are required to implement the Ellis/Detlefs standard.

Coexistence: The Ellis/Detlefs proposal suggests that it is necessary to support "libraries written without garbage collection or written in other languages such as C or Fortran." Thus, Ellis and Detlefs have specified that the implementation must support at least two virtual heaps, one that is garbage collected and the other that is managed explicitly. Though not discussed in detail in their proposal, they presume that the garbage collector has intimate knowledge of the memory and register usage of these foreign libraries. Note that foreign libraries introduce loopholes into the safety net. Both the programmer and the compiler of the foreign library may introduce subtle bugs that prevent reliable operation of the garbage collector.

We make no attempt in our system to support code that ignores the operating constraints of the garbage collector. Rather, we observe that the work required to revise non-comforming C code to make it compliant with the garbage collection constraints is manageable. We support the use of Fortran and other foreign libraries only if they do not require access to shared memory.

Note that a Boehm-Weiser implementation of our garbage-collectable C++ dialect coexists with foreign libraries as well as any of the alternative techniques discussed in the Ellis/Detlefs proposal.

Safety: The Ellis/Detlefs proposal speaks of several layers of safety. First, it discusses a set of safe-use rules that define the language subset that is compatible with garbage collection. The safe-use rules for our garbage collector, which we have listed above, are very similar to the safe-use rules described by Ellis and Detlefs. With either the Ellis/Detlefs standard or our garbage-collected C++ dialect, lint-like tools are capable of assisting programmers in searching out and correcting all potential violations of the safe-use rules. Second, Ellis and Detlefs discuss issues of code-generator safety. Certain code generation techniques may transform source code that complies with the safe-use rules into non-comforming machine code. Our requirements for code-generator safety are more stringent than what has been suggested by Ellis and Detlefs. However, we do not discuss them here because we are not proposing a standard implementation technique. For more information on our code generation techniques, see references 45 and 39. Finally, Ellis and Detlefs describe a safe subset of C++, use of which guarantees to programmers that they will not suffer from any of the following storage-related bugs: dereferencing of dangling pointers, memory smashes, dereferencing of a null pointer, and array subscripts out of bounds. Use of the subset is not required for reliable operation of their garbage collectors. Thus, their definition of the safe subset is really more of a style guideline than an implementation or standardization technique for garbage collection of C++. Use of this safe subset decreases the likelihood of dynamic-memory-related programming errors, regardless of the implementation technique.

Efficiency: Ellis and Detlefs state: "To be successful, garbage collection needn't be quite as efficient as programmer-written deallocation, since many programmers would gladly sacrifice a little extra run time or memory to eliminate storage bugs quickly and reliably." They point out that recent measurements by Zorn suggest that conservative garbage collection techniques can run as fast as explicit memory deallocation [58]. The measurements that we report in this paper demonstrate that our system can likewise run as fast as explicit memory deallocation.

In summary, our garbage collectable dialect of C++ satisfies most of the same design constraints as the Ellis/Detlefs proposal. However, our implementation technique is not 100 percent compatible with the proposed Ellis/Detlefs standard.

Besides providing support for automatic reclamation of dynamic memory in C++ programs, the Ellis/Detlefs proposal extends the programmer's repertoire of expressive mechanisms. In particular, it adds support for weak pointers and for automatic finalization of garbage-collected objects. Our completely general garbage collection system supports these mechanisms [37,45], but our C++ garbage collector does not, since traditional C++ does not offer these capabilities. One of our goals in designing garbage collection for C++ was to demonstrate how small an impact garbage collection would have on the language.

It is important to note that there is no known garbage collection technique that complies both with the Ellis/Detlefs proposal and with the requirements of reliable real-time garbage collection as outlined in this paper. We believe that a standard for garbage collection of C++ is important, and we believe that the Ellis/Detlefs proposal represents a valuable step in this direction. However, its adoption as a standard seems premature given the limited experience of the C++ user community with conservative garbage collection techniques and the proposal's lack of support for real-time programming.

## *3.4. Hardware-Assisted Real-Time Garbage Collection*

In the discussion that follows, we use the term *descriptor* to denote a pointer. By pointing to objects allocated elsewhere, each descriptor is capable of "describing" all conceivable kinds of information. We use the adjective *terminal* to characterize memory locations known not to contain pointers. If all live memory is represented as a directed graph in which nodes represent dynamically allocated objects and directed edges represent pointers from one object to another, the terminal nodes are those from which no directed edges emanate. The source nodes in this directed graph are pointers residing outside the garbage-collected heap. These source pointers, which are under direct control of the CPU, are called *root descriptors*.

During garbage collection, live objects are copied from one region of memory to another. At the moment garbage collection begins, the application process *tends* each of the root descriptors by communicating their current values to the garbage-collected memory module (*GCMM*) which in turn provides the application process with updated values to reflect the new locations of the objects they refer to. Internal to the GCMM, tending consists of checking whether the pointer refers to a *from-space* object, arranging for the *from-space* object to be copied to *to-space* if necessary, and updating the pointer to reflect the new *to-space* location of the object.

Application processes run on the CPU and certain garbage-collection tasks run on the GCMM. Application tasks are collectively referred to as the *mutator*, since, insofar as garbage collection is concerned, their only role is to modify (or mutate) the contents of heap-allocated memory.

The GCMM plays the role of traditional expansion memory within a standard bus-oriented memory architecture, as illustrated in Figure 1.

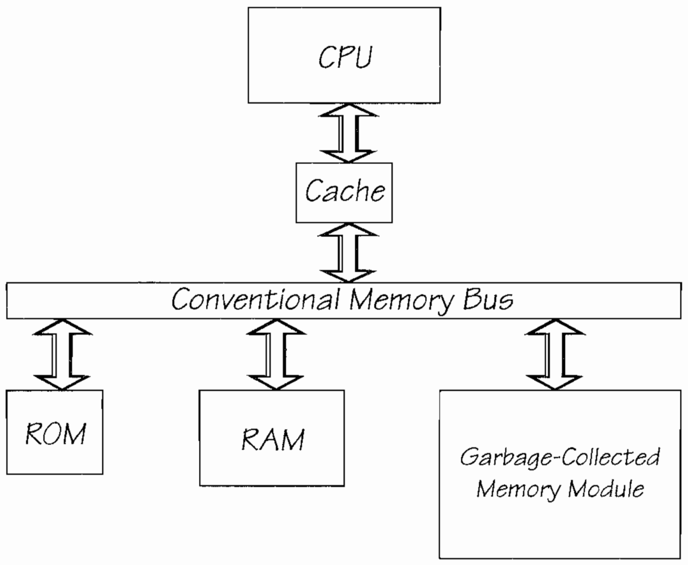


Figure 1. Proposed system architecture.

Logically, the GCMM looks like a bank of traditional expansion memory accompanied by a small number of memory-mapped I/O ports. The GCMM is like traditional memory except for the following special capabilities:

1. Each word of GCMM memory is accompanied by a *descriptor tag* which distinguishes between words holding pointers and those holding non-pointers.
2. Each word of GCMM memory is also accompanied by a *write-protect tag* which signifies that certain words cannot be overwritten by the mutator. The write-protect tag is used to ensure that the dynamic memory manager's internal data structures will not be corrupted by accidental or malicious C++ writes to out-of-bounds memory addresses.
3. The GCMM monitors all read and write operations. Depending on the current mode of operation, the GCMM provides special handling of certain read and write requests. For typical workloads, fewer than 1 percent of the memory operations that escape the cache require special handling [39]. The other 99 percent of memory operations are unimpeded by the GCMM.
4. The GCMM supports several direct-memory-access (DMA) operations to, for example, allow the mutator to initialize descriptor tags and to improve the performance of certain garbage collection activities.

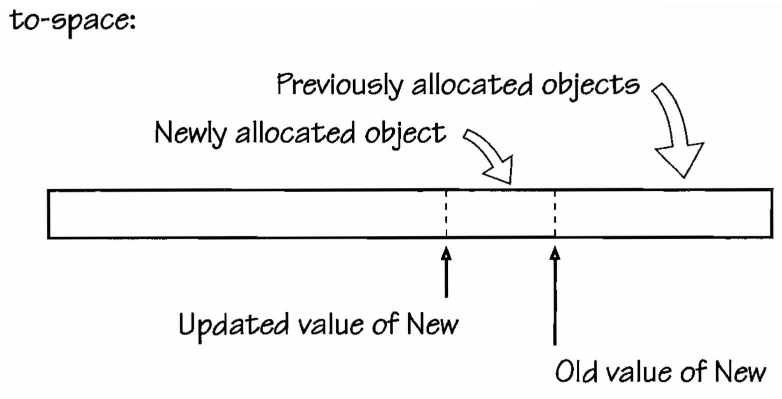


Figure 2. Allocation of memory within *to-space.*

A complete description of the GCMM is beyond the scope of this paper. For further information, see references 39 and 38.

Our garbage collection algorithm is derived from the real-time copying algorithm originally described by Baker [2]. The mutator allocates dynamic memory as independent objects. Each object occupies a contiguous segment of memory, the first word of which is a title describing the object's type and size. All dynamic objects are allocated from a large region of memory named *to-space.* Initially, *to-space* contains no objects. This allows very fast allocation of new objects. Within *to-space*, the **New** pointer initially points to the end of *to-space.* To allocate a new object, the system simply decrements the value of **New** by the object's size, as illustrated in Figure 2.

As execution proceeds, the **New** pointer eventually bumps against the end of *to-space.* When this occurs, garbage collection begins. The system allocates a new *to-space*, and the old *to-space* is renamed *from-space.* We call this a garbage collection flip. Garbage collection consists of incrementally copying live objects out of *from-space* into *to-space.* After all of the live objects residing in *from-space* have been copied into *to-space*, *from-space* contains no useful data. At the time of the next flip, the current *from-space* will be renamed *to-space.*

To minimize the real-time latency of the flip operation, the garbage collector does not copy all live objects at the time of the flip. Rather, it arranges to copy only those objects that are directly referenced by the system's root descriptors. For example, suppose Figure 3 represents live memory immediately before execution of the garbage collection flip. In this figure, there are two root descriptors, represented by address registers one and two, and three live objects, labeled **A**, **B**, and **C**. Note that object **A** is not directly referenced by the mutator. The mutator can only access **A** by first fetching its address from within object **B**. At flip

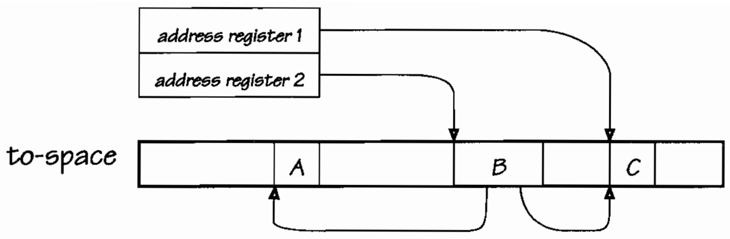


Figure 3. Memory immediately before garbage collection begins.

time, the garbage collector copies objects **B** and **C**, and the two root descriptors are updated to represent the new locations of these two objects. This is illustrated in Figure 4.

This illustration shows objects **B** and **C** having been copied verbatim (bitwise) into *to-space*. For each object copied into *to-space*, the garbage collector creates a forwarding pointer from the original object to the new copy of the object. These forwarding pointers are indicated in this illustration as dotted directed edges. Note that object **B′** contains pointers to the obsolete copies of the **A** and **C** objects that reside in *from-space*. These pointers need to be overwritten with pointers to the new locations of these objects. The garbage collector's handling of these obsolete pointers is described below.

The flip operation described above is really Baker's original flip algorithm. Baker was primarily interested in garbage collection of Lisp, in which all objects are the same size: two words. Compliance with real-time performance constraints requires that the flip operation execute within a small constant time bound. Note that the time required to execute Baker's flip operation depends on how much time is required to copy objects **B** and **C** out of *from-space*. This is unacceptable,

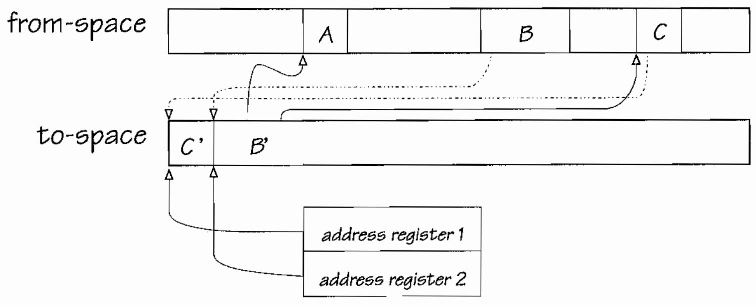


Figure 4. Memory following the garbage collection flip (Baker algorithm).

because either object may be arbitrarily large. Suppose, for example, that object **B** were a 512 KByte bitmap image. Then the flip would require at least 5 msec, the approximate time required to perform 32 K memory cycles, each transferring 16 bytes worth of information.

Rather than copy objects **B** and **C** at flip time, our algorithm simply reserves space within *to-space* into which the objects will eventually be copied. When the space is reserved, the garbage collector overwrites the first two words of the reserved space with the object's title and source location respectively. The title word of the *from-space* version of each object queued for copying is overwritten with a forwarding pointer to the object's new *to-space* location. This is illustrated in Figure 5.

In this figure, the **Reserved** pointer points to the end of memory reserved for copying of objects out of *from-space*. The **Relocated** pointer points to the end of memory already copied out of *from-space*. Following completion of the flip operation, the garbage collector repeatedly examines the object found at the location named by the **Relocated** pointer, incrementally copies that object into *to-space*, and updates **Relocated** to point to the end of the object that has just been copied. Pointers contained within the objects that are being copied are tended before they are written to *to-space*. Tending consists of checking whether the pointer refers to a *from-space* object, arranging for the *from-space* object to be copied to *to-space* if necessary, and updating the pointer to reflect the new *to-space* location of the object. During copying of **B** to **B'** in our example, two pointers are tended. First, the pointer to **A** is processed, which causes space to be reserved in *to-space* for **A**′. When the pointer to **C** is processed, the collector simply looks up the new location of **C** by examining the forwarding pointer that comprises **C**'s header. After both **C** and **B** have been copied, memory appears as shown in Figure 6.

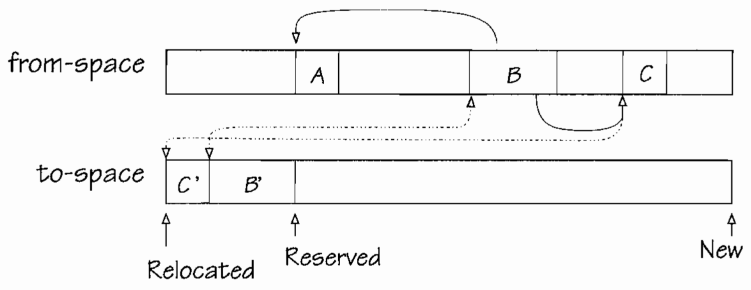


Figure 5. Memory following the garbage collection flip (Nilsen algorithm).

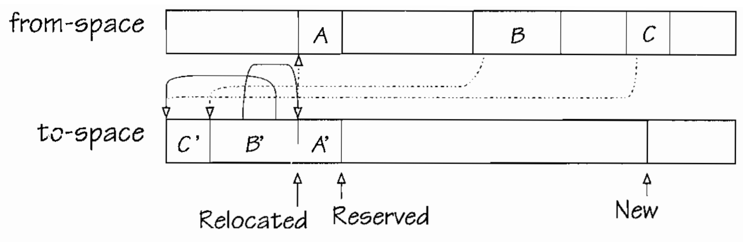


Figure 6. Memory after copying **B** and **C**, and reserving space for **A**.

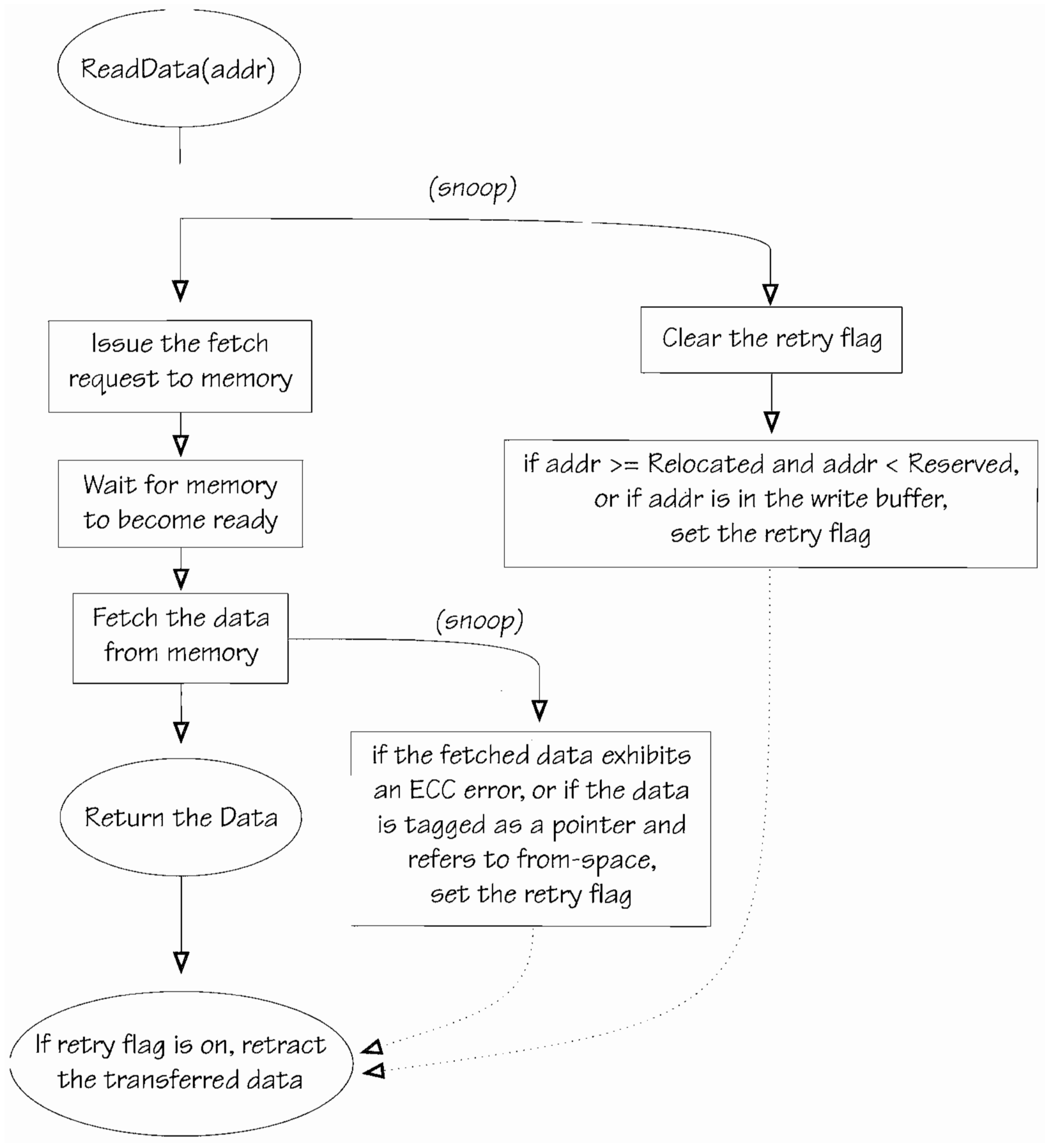
Note that the reverse link from the *to-space* objects to their *from-space* copies is destroyed as a side effect of copying the objects. Also, observe that the **New** pointer has been adjusted in this illustration to reflect that new memory is typically allocated while old memory is being garbage collected.

While garbage collection is active, certain mutator memory fetch and store operations require special handling. In particular, any attempt to fetch or store memory found between **Relocated** and **Reserved** must be redirected to the appropriate location in *from-space*. Furthermore, after redirecting a memory fetch to *from-space*, the garbage collector must tend the fetched word before making it available to the mutator if the word happens to be a descriptor. This interaction between mutator and garbage collector is the major overhead associated with software implementations of Baker's real-time copying garbage collection technique. The hardware-assisted garbage collection system avoids these performance penalties by performing the required checks in parallel with the memory and communication activities that comprise the interface between the CPU's cache and the memory subsystem, as illustrated by the flow chart in Figure 7.

In this flow chart, the data is returned to the CPU speculatively, as soon as it is available from the memory system [31]. The protocol for interaction with the CPU allows the memory subsystem to send a data retry signal on the clock that follows the data transfer. If the CPU receives this signal, it discards the transferred data and reissues its fetch request. The simulation results presented in reference 39 indicate that fewer than 1 percent of the memory fetches that miss the cache need to be retried. The worst-case delay associated with a fetch request that requires special handling is approximately 1 $\mu s$.

A large write buffer is built into the memory controller so the work required to handle write operations is generally amortized over time. The memory system's implementation of a write operation is illustrated in Figure 8.

The worst-case time required to wait for a slot in the write buffer is less than 1 $\mu s$. The write buffer is serviced by the independent process illustrated in Figure 9.



ReadData(addr)
(snoop)
Issue the fetch request to memory
Clear the retry flag
Wait for memory to become ready
if addr >= Relocated and addr < Reserved, or if addr is in the write buffer, set the retry flag
Fetch the data from memory
(snoop)
if the fetched data exhibits an ECC error, or if the data is tagged as a pointer and refers to from-space, set the retry flag
Return the Data
If retry flag is on, retract the transferred data

Figure 7. The memory subsystem's processing of fetch requests.

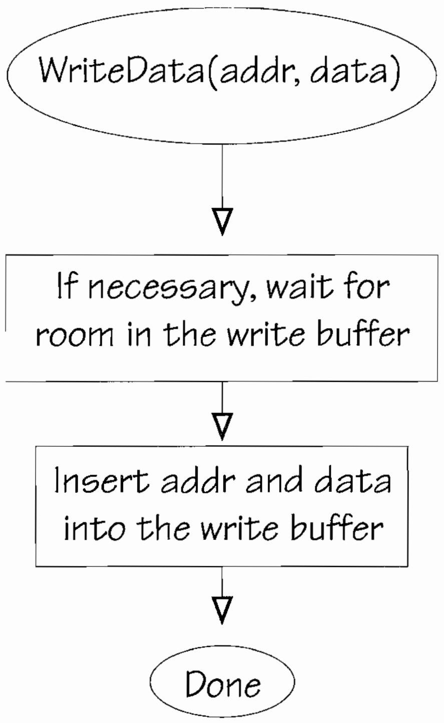

WriteData(addr, data)
If necessary, wait for room in the write buffer
Insert addr and data into the write buffer
Done

Figure 8. The memory subsystem's processing of store requests.

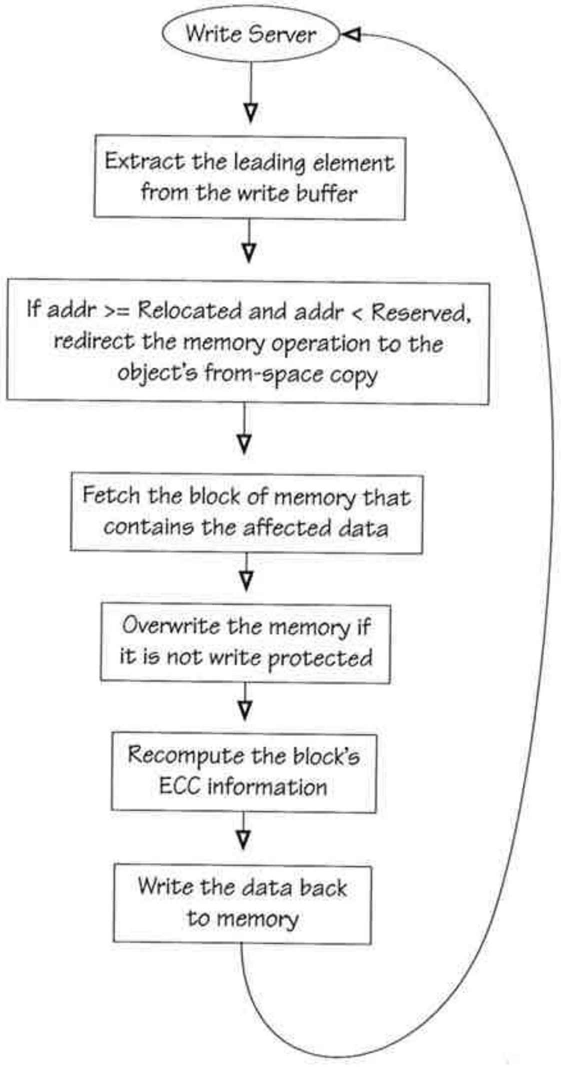

Write Server
Extract the leading element from the write buffer
If addr >= Relocated and addr < Reserved, redirect the memory operation to the object's from-space copy
Fetch the block of memory that contains the affected data
Overwrite the memory if it is not write protected
Recompute the block's ECC information
Write the data back to memory

Figure 9. The memory subsystem's handling of its write buffer.

## 4. Empirical Performance

Though we consider predictable real-time latency to be of primary importance, we also desire good average-case performance. To estimate the overhead of our garbage collection protocol in comparison with traditional dynamic memory management techniques, we have constructed a hardware simulator for our garbage-collected memory module and have customized an implementation of the GNU C++ compiler, version 1.37.1, to generate code that communicates with our hardware-assisted garbage collecting memory module for all dynamic memory allocation needs. We have conducted experiments to compare the performance of code generated by the customized C++ compiler running on the simulated garbage-collecting memory architecture with code generated by the traditional GNU C++ compiler as simulated on a traditional memory architecture. Both architectures depend on separate 32-Kbyte instruction and data caches, organized with two-way associative one-word cache lines. The data cache uses a write-back policy. We assume that both architectures use static-column RAM for all memory. This memory responds especially quickly to read and write requests that happen to access the same row of memory as was referenced by the previous request. The traditional C++ implementation uses GNU `malloc` and `free` [58,59]. This particular implementation of `malloc/free` is known to be very space efficient while exhibiting average run-time performance in comparison with alternative implementations.

We have ported three real-world C++ applications to the simulated environments. These applications are `sfft`, a sliding fast fourier transform; `lisp`, a lisp interpreter written in C++ by Tim Budd for instructional support of Kamin's comparative programming language text [17]; and `troff`, the typesetting component of James Clark's `groff` software. These three applications cover a broad spectrum of programming styles. `sfft` is floating-point intensive, performs no dynamic memory allocation except what is required to allocate a single I/O buffer, and performs a large amount of work between function calls. Since `sfft` does very little dynamic memory allocation, this program helps characterize the overhead of garbage collection on applications that do not benefit from the use of dynamic memory. On the other hand, `lisp` makes heavy use of dynamic memory allocation, never frees allocated objects, and does comparatively little work between function calls. `sfft` and `lisp` represent opposite ends of the programming spectrum. `troff` represents the middle of the road. The typical `troff` function is twice as long as the average `lisp` function size. And `troff` is careful to recycle dynamic memory after it is no longer needed.

Table 2 summarizes the behavior of the traditional implementations of the sample applications, each application running two different workloads.

Table 2. Traditional Implementations of Experimental Workloads.

| Test Case | Instructions Per Call | Instructions Executed | Cycles Executed | Bytes Allocated | Heap Size (bytes) | Cycles in `malloc/free` |
|---|---|---|---|---|---|---|
| `sfft`/small | 448 | 61,033,415 | 77,711,461 | 13,328 | 37,392 | 6,123 |
| `sfft`/medium | 448 | 243,298,164 | 309,626,744 | 13,328 | 37,392 | 6,123 |
| `lisp`/prune | 23 | 250,626,176 | 304,271,795 | 643,435 | 1,276,432 | 9,061,562 |
| `lisp`/db | 23 | 430,967,788 | 520,369,298 | 910,571 | 1,790,496 | 12,661,448 |
| `troff`/paper1 | 45 | 312,497,672 | 519,231,664 | 1,407,369 | 760,080 | 15,120,973 |
| `troff`/paper2 | 45 | 843,986,701 | 1,644,574,888 | 3,431,119 | 1,014,000 | 41,982,325 |

The first column of Table 2 represents the quotient of dividing the total number of instructions executed by the total number of function calls. These values are extrapolated from studies performed previously [45], and do not represent exactly the same workloads otherwise reported in this paper. The cycles reported for `malloc` and `free` invocations exclude the time occasionally required by the operating system to expand the `brk` region. The ratio between the number of cycles and the the number of instructions executed varies from 1.2 to 1.9, depending primarily on the likelihood of instruction and data cache hits. Instruction and data accesses are much less localized for `troff` than for `lisp` and `sfft`. Note that `lisp`'s heap size is approximately twice as large as the total number of bytes allocated. This is because each allocated object must be aligned with two-word boundaries, and each object is accompanied by a header that describes the size of the dynamic object. The size of `sfft`'s heap is almost three times as large as the number of allocated bytes. For this program, the size of the heap is determined by the default initial heap size rather than by the application's need for dynamic memory. Note that the heap size for the `troff` applications is much smaller than the total number of bytes allocated. This is because `troff` recycles memory by deleting objects whose useful lifetime has ended.

Table 3 summarizes the performance of the garbage-collected C++ implementations of the same experimental workloads.

Semispaces are constrained to power-of-two sizes by limitations in the current simulator implementation. The smallest heap configuration supported by the garbage-collected system is 256 KB per semi-space, which is 512 KB (524,288 bytes) total. This heap configuration is large enough to support the `sfft` and `lisp` workloads, but larger heaps are required to support the `troff` application. For the garbage-collected implementation, the cycles spent in allocation of new memory includes both the time spent allocating new objects, and the time spent initializing tag bits to enable the garbage collector to distinguish between words containing pointers and words known not to contain pointers. The last column of Table 3 reports the number of times that each experimental workload requires garbage collection.

Table 3. Garbage-Collected Implementations of Experimental Workloads.

| Test Case | Instructions Executed | Cycles Executed | Bytes Allocated | Heap Size (bytes) | Cycles in Allocation | Number of Flips |
|---|---|---|---|---|---|---|
| `sfft`/small | 61,345,060 | 78,714,908 | 53,650 | 524,288 | 31,512 | 0 |
| `sfft`/medium | 244,456,999 | 313,387,314 | 53,650 | 524,288 | 31,512 | 0 |
| `lisp`/prune | 304,464,457 | 378,763,456 | 779,281 | 524,288 | 1,711,143 | 4 |
| `lisp`/db | 525,611,056 | 654,509,971 | 1,104,109 | 524,288 | 2,429,129 | 6 |
| `troff`/paper1 | 339,538,043 | 507,428,938 | 1,598,820 | 2,097,152 | 2,251,436 | 2 |
| `troff`/paper2 | 912,908,256 | 1,514,830,175 | 3,654,093 | 4,194,304 | 5,994,677 | 2 |

Table 4 tabulates the comparative performance of the two implementation techniques.

Table 4. Garbage-Collected C++ vs. Traditional C++.

| Test Case | Instructions Executed | Cycles Executed | Bytes Allocated | Heap Size (bytes) | Allocation Cost |
|---|---|---|---|---|---|
| `sfft`/small | +5.1 percent | +1.3 percent | +302.5 percent | N/A | +414.6 percent |
| `sfft`/medium | +4.8 percent | +1.2 percent | +302.5 percent | N/A | +414.6 percent |
| `lisp`/prune | +21.5 percent | +24.5 percent | +21.1 percent | −58.9 percent | −81.1 percent |
| `lisp`/db | +22.0 percent | +25.8 percent | +21.3 percent | −70.7 percent | −80.8 percent |
| `troff`/paper1 | +8.7 percent | −2.3 percent | +13.6 percent | +175.9 percent | −85.1 percent |
| `troff`/paper2 | +8.2 percent | −7.9 percent | +6.5 percent | +313.6 percent | −85.7 percent |

In all cases, the garbage-collected implementations require more instructions to perform the same total amount of work. The increase in instruction counts is due primarily to the overhead of maintaining run-time type information to describe the contents of each function activation frame. The combination of prologue and epilogue code in the garbage-collected implementation of C++ executes a minimum of nine extra instructions per function call [39]. This is why the `lisp` workloads, which have the smallest average function size, exhibit the greatest increase in executed instructions. Even though the garbage-collected implementations execute more instructions than the traditional implementations, the garbage-collected implementation of troff requires fewer total machine cycles than does the traditional implementation. The traditional troff implementation uses a single bank of memory to represent all code and data. The garbage-collected implementation uses one bank of memory to represent code and static data known not to contain pointers, and a different bank of memory to represent the dynamic heap. Since each bank of memory is implemented using static-column DRAMs, localized memory accesses within each bank perform better than completely random accesses. By separating the dynamic heap and code into distinct memory banks, the garbage-collected implementation improves the locality of memory references within each bank. This is the primary reason that the garbage-collected implementation of troff runs faster than its traditional implementation. Note, however, that this is not the only factor that influences the comparative CPI (cycles per instruction) of each application. In the lisp applications, for example, the percentage increase in number of cycles is greater than the percentage increase in number of instructions. This is because the garbage-collected lisp implementation exhibits a worse cache hit rate than the traditional implementation. Remember that the garbage-collected system periodically invalidates the cache in order to initiate garbage collection. All of the garbage-collected implementations heap-allocate more memory than the traditional implementations of the same applications because the garbage-collected implementations heap-allocate stack activation frames [39]. We do not compare the heap sizes for sfft because this comparison would serve only to report the relationship of the smallest initial heap sizes for the two implementations. The garbage-collected lisp implementation uses a much smaller heap than the traditional implementation because the garbage collector automatically recycles memory that the system's developer did not bother to reclaim. Comparisons between the memory utilizations of the two troff implementations emphasize that copying garbage collectors require at least twice as much memory as the amount of live heap-allocated data. In the simulator's measured configuration, the expected ratio between real memory and live memory is 2.25 [39]. Using troff's traditional heap sizes as an estimate of the amount of live data, we would expect the garbage-collected memory requirements to be 1,710,180 and 2,281,500 bytes for the paper1 and paper2 workloads respectively. Note that the measured memory usage of the garbage-collected troff implementations is consistent with the values calculated by rounding 1,710,180 and 2,281,500 respectively up to the nearest power of two. In an actual hardware implementation, semispace sizes need not be equal to a power of two. The last column of Table 4 emphasizes the potential performance benefits of the hardware-assisted garbage collection technique in comparison with more traditional dynamic memory management techniques. For all of the workloads that make effective use of dynamic memory management, the garbage-collected implementation runs in less than 25 percent of the time required to explicitly manage the same dynamic memory.

Table 5 is of particular relevance to the real-time practitioner. This table reports the range of times, measured in terms of machine cycles, for each dynamic memory operation.

Table 5. Variation in Cycles Required to Manage Memory.

| Test Case | Traditional Implementation | | Garbage-Collected Implementation |
|---|---|---|---|
| | Cycles per `malloc`† | Cycles per `free` | Cycles per allocation‡ |
| `sfft`/small | 378 – 5,745 | N/A | 11 – 43 |
| `sfft`/medium | 378 – 5,745 | N/A | 11 – 43 |
| `lisp`/prune | 132 – 4,357 | N/A | 11 – 97 |
| `lisp`/db | 132 – 7,081 | N/A | 11 – 96 |
| `troff`/paper1 | 132 – 7,350 | 70 – 828 | 11 – 99 |
| `troff`/paper2 | 132 – 7,256 | 70 – 816 | 11 – 94 |

† Excludes the costs of system calls to expand the `brk` region.
‡ Excludes certain communication costs associated with the exchange of information between the CPU and the GCMM. This communication adds approximately 10 additional machine cycles per allocation.

The garbage-collected implementation was measured to perform all memory allocations in less than 100 machine cycles. Assuming a 50 MHz clock rate, this corresponds roughly to a worst-case memory allocation time of 2 $\mu s$. In contrast, the time to allocate a new object using a traditional implementation of `malloc` was measured to range from approximately 2 to 150 $\mu s$. Though traditional implementation techniques offer good average-case performance, occasional allocations result in unacceptably long latencies.

These simulations have demonstrated that the proposed architecture executes programs in time roughly comparable to that of traditional architectures. But it is important to remember that average-case performance is not the only issue here. The hardware-assisted garbage collection system provides the additional benefit of guaranteeing small upper bounds on the time required to read, write, and allocate an object in garbage-collected memory, and it automatically reclaims unused memory. The major shortcomings of the proposed memory architecture are that the system incurs a high overhead on function invocation, and the system requires more memory than traditional dynamic memory management implementations.

Continuing research focuses on constructing and evaluating a hardware prototype of the real-time garbage collection system. The system currently under development differs in several ways from the simulated design:

1. The code generation model for the new system will generate function prologues and epilogues that are, on average, nearly as time efficient as what is currently used in traditional C++ implementations. We achieve this by establishing a standard activation frame template that serves all functions. The layout specifies which fields within the activation frame contain pointers. Large activation frames are represented by several contiguous copies of the standard template. Functions whose activation frames must represent structured data that does not fit the standard activation-frame template heap-allocate the structured data using techniques similar to the techniques used for all activation frames in the current implementation of garbage-collected C++ [39].

2. The hardware protocol for communication between the CPU and the GCMM has been modified to reduce the need for the CPU to wait for GCMM operations to complete before proceeding. The new system allows the GCMM to pipeline the most commonly executed operations, only stalling the CPU when the pipeline fills. This new protocol will provide much better average-case performance for object allocation and descriptor tag initialization.

3. We have designed a hybrid garbage collection technique that uses copying garbage collection to compact certain segments of the heap while processing the remainder of the heap using mark-and-sweep techniques. This hybrid technique provides the benefits of compacting garbage collection without limiting heap utilization to less than 50 percent. The hardware prototype will support this newly designed hybrid garbage collection technique in addition to the fully-copying garbage collection technique that has already been simulated.

Considerable performance evaluation will be required to fully characterize the costs and benefits of the proposed hardware prototype. Nevertheless, extrapolation of current performance results suggests that the hardware prototype will provide much better performance than has been demonstrated by the simulated architecture.

## 5. Conclusions

Software engineers specializing in the development of real-time systems have invested decades of research and experience in the creation of methodologies that promote the creation of real-time software that reliably complies with all real-time constraints. Garbage collection researchers who undertake to develop techniques for real-time garbage collection must find ways to integrate their garbage collection systems with existing real-time development methodologies.

The technique for hardware-assisted garbage collection of C++ that is described in this paper is compatible with traditional real-time development techniques. Most other so-called real-time garbage collection techniques do not provide the fine granularity of timing behavior that is required for the creation of reliable real-time systems.

Though special hardware is required to achieve real-time performance, the experiments reported in this paper serve also to demonstrate that accurate garbage collection of C++ is feasible.

## Acknowledgments

I thank Benjamin Zorn for his thoughtful criticisms of an earlier draft of this document. His comments motivated me to broaden the scope of this paper to include more specific discussion of the shortcomings of existing stock-hardware real-time garbage collectors. I also thank the anonymous referees, whose comments served to motivate the comparison of our garbage-collected C++ dialect with the proposed Ellis/Detlefs standard for garbage collection of C++, and also to improve the section on empirical performance by requesting that we tabulate the system's measured performance rather than simply summarizing previous results.

This work was supported by the National Science Foundation under grant MIP-9010412, and by a grant from the U.S. Department of Commerce. William Schmidt ported the GNU C++ compiler to the special-purpose memory architecture and performed the experimental measurements that characterize the system's average-case throughput. I also thank Jim Lathrop and Craig VanZante, students at Iowa State University, for their assistance in developing the `sfft` and `lisp` test cases. Hong Gao, another Iowa State University student, is currently analyzing the performance of the `cfrac`, `cham`, `espresso`, `gawk`, and `ptc` applications. Her experience porting these C applications to the garbage-collected dialect of C++ is summarized in section 3 of this paper.

## *References*

1. T. E. Anderson, H. M. Levy, B. N. Bershad and E. D. Lazowska, The Interaction of Architecture and Operating System Design, *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, 1991, 108–119.
2. H. G. Baker Jr., "List Processing in Real Time on a Serial Computer," *Comm. ACM* 21, 4 (April 1978), 280–293.
3. S. Basumallick and K. Nilsen, Cache Issues in Real-Time Systems, *ACM SIGPLAN Notices, Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.
4. H. Boehm and M. Weiser, Garbage Collection in an Uncooperative Environment, *Software—Practice & Experience* 18, 9 (Sep 1988), 807–820.
5. H. Boehm, A. J. Demers and S. Shenker, "Mostly Parallel Garbage Collection," *ACM SIGPLAN Notices, Conference on Programming Language Design and Implementation*, June 1991.
6. R. A. Brooks, Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware, *ACM Symposium on LISP and Functional Programming*, August 1984, 256–262.
7. A. Burns, A. J. Wellings, C. M. Bailey and E. Fyfe, The Olympus Attitude and Orbital Command System: A Case Study in Hard Real-Time System Design and Implementation, YCS190, Department of Computer Science, University of York, 1992.
8. C. Chambers, Cost of Garbage Collection in the SELF System, *1991 Workshop on Garbage Collection in Object-Oriented Systems of OOPSLA*, Phoenix, AZ, October 1991.
9. S. Cheng, J. A. Stankovic and K. Ramamritham, Scheduling Algorithms for Hard Real-Time Systems—A Brief Survey, in *Tutorial on Hard Real-Time Systems*, J. A. Stankovic and K. Ramamritham (ed.), IEEE Computer Society Press, 1988, 150–173.
10. J. R. Ellis, K. Li and A. W. Appel, "Real-time Concurrent Collection on Stock Multiprocessors," *ACM SIGPLAN Notices, Conference on Programming Language Design and Implementation*, June 1988.
11. J. R. Ellis and D. L. Detlefs, Safe, Efficient Garbage Collection for C++, Digital Equipment Corporation Systems Research Center Report 102, June 1993.
12. S. L. Engelstad and J. E. Vandendorpe, Automatic Storage Management for Systems with Real-Time Constraints, *Oral presentation at 1991 Workshop on Garbage Collection in Object-Oriented Systems of OOPSLA*, Phoenix, AZ, October 1991.
13. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
14. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

15. S. Hong and R. Gerber, Compiling Real-Time Programs into Schedulable Code, *ACM SIGPLAN Notices, Conference on Programming Language Design and Implementation* 28, 6 (June 1993), 166–176.

16. R. Johnson, Reducing the Latency of a Real-Time Garbage Collector, *ACM Letters on Prog. Lang. and Systems* 1, 1 (March 1992), 46–58.

17. S. N. Kamin, *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, Reading, MA, 1990.

18. H. Kasahara and S. Narita, Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System, *IEEE Journal of Robotics and Automation* 1, 2 (June 1985), 104–113.

19. D. I. Katcher, J. K. Strosnider and E. A. Hinzelman-Fortino, Dynamic versus Fixed Priority Scheduling: A Case Study, submitted for publication.

20. K. B. Kenny and K. Lin, A measurement-based performance analyzer for real-time programs, University of Illinois at Urbana-Champaign Report UIUCDCS-R-90-1606, 1990.

21. K. B. Kenny and K. Lin, Building Flexible Real-Time Systems Using the Flex Language, *IEEE Computer*, May 1991, 70–78.

22. T. Kuo and A. Mok, SSP: A Semantic-Based Protocol for Real-Time Data Access, *IEEE Real-Time Systems Symposium*, December 1993.

23. P. S. Lavoie, Tool to Analyze Timing on 68020 Processor, Master's Project, University of Massachusetts-Amherst, 1991.

24. J. P. Lehoczky, Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines, *IEEE Real-Time Systems Symposium*, December 1990, 201–209.

25. C. L. Liu and J. W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *J. ACM* 20, 1 (January 1973), 44–61.

26. J. W. S. Liu, K. Lin, W. Shih, A. C. Yu, J. Chung and W. Zhao, Algorithms for Scheduling Imprecise Computations, *IEEE Computer*, May 1991, 58–68.

27. C. D. Locke, D. R. Vogel and T. J. Mesler, Building a Predictable Avionics Platform in Ada: A Case Study, *IEEE Real-Time Systems Symposium*, December 1991, 181–189.

28. H. Massalin and C. Pu, Fine-Grain Adaptive Scheduling using Feedback, *Computing Systems* 3, 1 (Winter 1990), 139–173.

29. F. W. Miller, The Performance of a Mixed Priority Real-Time Scheduling Algorithm, *Operating Systems Review* 26, 4 (October 1992), 5–13.

30. A. Mok, P. Amerasinghe, M. Chen and K. Tantisirivat, Evaluating Tight Execution Time Bounds of Programs by Annotations, *Sixth IEEE Workshop on Real-Time Operating Systems and Software*, May 1989, 272–279.

31. *Power PC 601 RISC Microprocessor User's Manual*, Motorola, 1993.

32. K. Narasimhan and K. Nilsen, *Portable Execution Time Analysis for RISC Processors*, ACM SIGPLAN Notices, Workshop on Language, Compiler, and Tool Support for Real-Time Systems, June 1994.

33. S. Nettles, J. O'Toole, D. Pierce and N. Haines, Replication-Based Incremental Copying Collection, in *Memory Management*, Y. Bekkers and J. Cohen (ed.), Springer-Verlag, 1992, 357–364.

34. S. Nettles and J. O'Toole, Real-Time Replication Garbage Collection, *ACM SIGPLAN Notices, Conference on Programming Language Design and Implementation* 28, 6 (June 1993), 217–226.

35. D. Niehaus, Program Representation and Translation for Predictable Real-Time Systems, *Proceedings of the Twelfth Real-Time Systems Symposium*, San Antonio, TX, December 1991, 53–63.

36. K. Nilsen, "Garbage Collection of Strings and Linked Data Structures in Real Time," *Software—Practice & Experience* 18, 7 (July 1988), 613–640.

37. K. Nilsen and W. J. Schmidt, Hardware-Assisted General-Purpose Garbage Collection for Hard Real-Time Systems, Iowa State Univ. Tech. Rep. 92-15, 1992.

38. K. Nilsen, Cost-Effective Hardware-Assisted Real-Time Garbage Collection, *ACM SIGPLAN Notices, Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

39. K. D. Nilsen and W. J. Schmidt, A High-Performance Hardware-Assisted Real-Time Garbage Collection System, *Journal of Programming Languages* 2, 1 (January 1994), 1–40.

40. C. Pu, H. Massalin and J. Ioannidis, The Synthesis Kernel, *Computing Systems* 1, 1 (Winter 1988), 11–32.

41. P. Puschner and C. Koza, Calculating the Maximum Execution Time of Real-Time Programs, *The Journal of Real-Time System* 1, 2 (September 1989), 159–176.

42. R. Rajkumar, L. Sha and J. P. Lehoczky, On Countering the Effects of Cycle-Stealing in a Hard Real-Time Environment, *IEEE Real-Time Systems Symposium*, December 1987.

43. J. Rawat, Static Analysis of Cache Performance for Real-Time Programming, Iowa State Univ. Tech. Rep. 93-19, Master's Thesis, Iowa State Univ., 1993.

44. P. Rovner, On Adding Garbage Collection and Runtime Types to a Strongly-Typed Statically-Checked, Concurrent Language, CSL-84-7, Xerox Palo Alto Research Center, 1984.

45. W. J. Schmidt, Issues in the Design and Implementation of a Real-Time Garbage Collection Architecture, Ph.D. Dissertation, Iowa State Univ. Tech. Rep. 92-25, 1992.

46. A. C. Shaw, Reasoning About Time in Higher-Level Language Software, *IEEE Transactions on Software Engineering* 13, 7 (July 1989), 875–889.

47. J. A. Stakovic and K. Ramamritham, The Design of the Spring Kernel, *IEEE Real-Time Systems Symposium*, December 1987.

48. K. Tindell, An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks, YCS189, Department of Computer Science, University of York, December 1992.

49. K. Tindell, A. Burns and A. Wellings, Allocating Real-Time Tasks (An NP-Hard Problem made Easy), *Real-Time Systems*, to appear.

50. D. Ungar, Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm, *ACM SIGPLAN Notices* 19, 5 (May 1984), 157–167.

51. S. Wholey and S. E. Fahlman, The Design of an Instruction Set for Common Lisp, *ACM Symposium on LISP and Functional Programming*, 1984, 150–158.

52. P. R. Wilson and M. S. Johnstone, Real-Time Non-Copying Garbage Collection, *1993 ACM OOPSLA Workshop on Memory Management and Garbage Collection*, Washington, DC, September 1993.

53. J. Xu and D. L. Parnas, Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations, *IEEE Transactions on Software Engineering* 16, 3 (March 1990), 360–369.

54. J. Xu and D. L. Parnas, On Satisfying Timing Constraints in Hard Real-Time Systems, *IEEE Transactions on Software Engineering* 19, 1 (January 1993), 70–84.

55. T. Yuasa, Real-Time Garbage Collection on General-Purpose Machines, *Journal of Systems and Software* 11 (1990), 181–198.

56. W. Zhao, K. Ramamritham and J. A. Stankovic, Scheduling Tasks with Resource Requirements in Hard Real-Time Systems, *IEEE Transactions on Software Engineering* SE-13, 5 (May 1987), 564–577.

57. B. Zorn and D. Grunwald, Empirical Measurements of Six Allocation-intensive C Programs, *SIGPLAN Notices* 27, 12 (December 1992).

58. B. Zorn, The Measured Cost of Conservative Garbage Collection, *Software—Practice & Experience* 23, 7 (July 1993), 773–756.

59. B. Zorn and D. Grunwald, Evaluating Models of Memory Allocation, *ACM Transactions on Modeling and Computer Simulation* 4, 1 (January 1994).