Reliable Real-Time Garbage Collection of C++

Kelvin Nilsen

kelvin@cs.iastate.edu

Department of Computer Science
Iowa State University

ABSTRACT

Most so-called real-time garbage collection techniques are incompatible with traditional real-time development methodologies. This paper provides a brief overview of standard real-time development techniques, with emphasis on issues that are relevant to garbage collection. It discusses several shortcomings of existing real-time garbage collection techniques. Then it summarizes the design of a C++ dialect that allows hardware-assisted real-time garbage collection of heap-allocated objects in a manner that is compatible with traditional real-time development methodologies.

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 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%) of the system's available computing resources. If performance is poor, the engineer cannot guarantee compliance with real-time deadlines.

Presented at the 1993 OOPSLA Workshop on Garbage Collection in Object-Oriented Systems.

- Because it is difficult to derive tight upper bounds on the worst-case times required to execute particular tasks, many modern real-time systems are comprised of a combination of essential and optional real-time tasks [1, 2]. 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 comprised of tasks that are prioritized according to importance [3]. 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 [4-6]. 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 [1, 3, 7-14]. 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 comprise the system. A variety of techniques for analysis of task execution times is available [15-23].

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 is an example of a soft real-time application. Users would generally prefer delayed responses over complete communication failure.

Existing real-time systems are often comprised 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.

Sample Real-Time Workloads

This section provides brief summaries of several actual working real-time systems [24-28]. The purpose of this presentation 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.

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%

Table 1

Note that 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 μs 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 commitments

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

to terminate within the amounts of time that they've claimed are their worst-case times. 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 [28]. 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 μs [29]), 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.

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.

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. Note that 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, they do so without even describing their notion of real-time computing. Note that 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 also that it is difficult for the real-time tasks described in table 1 to absorb the virtual-memory trap costs associated with typical "real-time" garbage collectors. Assume, for example, that the worst-case cost of each memory fetch and store operation is 500 μs. Suppose, further, that the 500 μs delay associated with these memory operations is uninterruptible. In both the HARM missile and the satellite control applications, several tasks have periods of approximately one 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-μs 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 accesses to the garbage-collected heap, the resulting system utilization would more than double. In fact, the resulting utilizations would exceed 100%. The Ada avionics application behaves similarly. Similar behavior is demonstrated by the Ada avionics application. Suppose that no more than two accesses to garbage-collected memory occur during each one-ms segment of code. This would double the worst-case execution times for all of the tasks in the Ada avionics application. Once again, the resulting task mix is unschedulable.

Garbage collection enthusiasts may claim that their real-time garbage collectors are not intended for these sorts of real-time applications. They will say, for example, that real-time tasks with these short execution times and high execution frequencies should not be allocating dynamic memory. But are they willing to allow these tasks to access dynamic memory allocated by other less time-critical tasks? If not, in what sense are their garbage collectors real time? How do they characterize the class of real-time systems to which their real-time garbage collectors apply?

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 [30, 31] 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/or 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 [31]. We found that the overhead imposed on all memory operations that refer to the garbage-collected heap more than doubled the cost of executing typical applications. Other researchers have reported similar results.
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. 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 [32, 33]. 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 [33]. 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, 35]. 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 [34]. 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, Pierce, and Haines describe a variant of Baker's real-time copying garbage collection system in which two valid copies of each live object are maintained during garbage collection. Read operations may fetch data from either copy [35]. But write operations must update both copies. As far as we know, performance measurements characterizing this algorithm's performance have not yet been published[1]. We are skeptical that this technique's overhead would be tolerably small in a typical C++ environment. The high frequency of memory writes in C++ and the high expected cost of performing write operations would likely combine to represent a significant percentage of the system's run time. Furthermore, it is quite likely that the performance of read operations would also be affected indirectly by this garbage collection scheme. Note that during garbage collection, it is necessary to maintain two identical copies of each live object. This will increase contention for available cache lines, resulting in lower cache hit rates for fetch operations. Were it desirable to port the Nettles *et al* garbage collection technique to C++, it would be necessary to develop efficient techniques for identifying all pointers that reside within registers, the stack, and the dynamic heap. In our experience, maintaining the tags necessary for copying garbage collection of C++ is the primary overhead associated with relocating garbage collection of C++ [36]. Another issue that would need to be addressed is the difficulty of bounding real-time response in the presence of arbitrarily large heap objects [37].

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 μs [29]. 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 [33, 38-40], 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 μs [39]. 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++. Though some researchers hint that simple refinements and/or straightforward enhancements to existing techniques would make real-time garbage collection of C++ possible, we remain unconvinced. Nobody has yet demonstrated a complete working system accompanied by careful analysis of worst-case execution times.

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

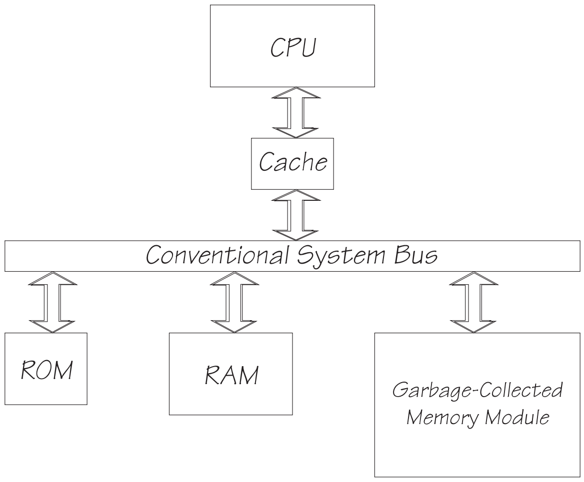
---

[1] Reference 35 describes the performance of a hybrid algorithm based on replicated objects to maintain coherence between garbage collection and mutation which uses virtual memory protection to determine when particular memory write operations must be replicated.

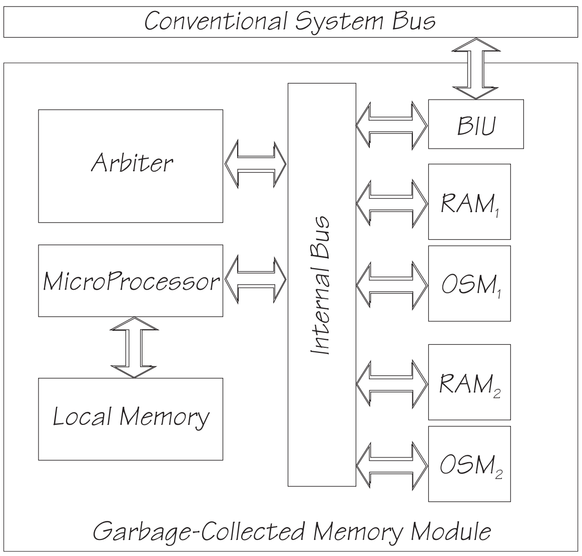
methodologies might be developed will need to give special care to the bursty nature of garbage collection performance. Regardless of how these conflicts are eventually resolved, it would be beneficial for real-time and garbage collection researchers to both become more familiar with each other's domains of expertise.

Hardware-Assisted Real-Time Garbage Collection

During the past several years, we have designed and evaluated a special intelligent memory architecture designed to facilitate high-performance garbage collection in hard real-time environments. The expansion memory module plays the role of traditional expansion memory within a standard bus-oriented system architecture, as illustrated below:



To the system bus, garbage-collected memory looks like a bank of traditional expansion memory accompanied by a small number of memory-mapped I/O ports. Logically, the garbage-collected memory module is organized as shown in the following sketch:



In this illustration, BIU is an abbreviation for *Bus Interface Unit*. The BIU provides an interface between the system bus and an internal bus used for communication between components of the GCMM. Each RAM module consists of traditional random access memory. The two independent RAM modules represent *to-* and *from-space* respectively. Each 32-bit word of RAM is accompanied by a one-bit tag that distinguishes pointers from non-pointers, and a one-bit write-protect tag that prevents the mutator from overwriting the garbage collector's internal data structures. OSM stands for object space manager. Each OSM module manages the contents of one RAM memory module by maintaining a data base of locations at which each object residing in the memory module begins [37]. Given a pointer to any location within a memory module, the corresponding OSM is capable of reporting the address of the start of the object that contains that address in approximately the same time required to perform a traditional memory fetch or store. The *arbiter* oversees access to the internal bus, and performs a number of important garbage collection activities using circuitry dedicated to supporting rapid context switches between low-priority garbage collection activities and the servicing of high-priority mutator requests. The *μprocessor*'s main responsibility is to supervise garbage collection, which is performed using a variant of Baker's real-time copying garbage collection technique [41]. The μprocessor oversees garbage collection by dividing the job into a large number of small straightforward activities and individually assigning each of these activities to the arbiter. The arbiter works on assignments from the μprocessor as a background activity, giving highest priority to servicing of BIU requests.

A detailed description of the system is beyond the scope of this paper. System details are provided elsewhere [36, 42, 43]. Briefly, we summarize the system's performance and capabilities here:

- The special hardware associated with the system is isolated within an expansion memory module that is portable between CPU and bus architectures. Values stored in the special memory module can be cached by the CPU.
- The cost of the expansion memory module is three to five times the cost of a traditional expansion memory module large enough to hold the application's worst-case dynamic memory needs, assuming that dynamic memory can be packed perfectly within the traditional memory module.
- The measured performance (average throughput) of the memory module is roughly equivalent to the performance of traditional memory systems on similar workloads.
- The worst-case time required to fetch memory is the time required to execute six traditional memory cycles (about 1 μs). The worst-case time required to store memory is three traditional memory cycles (about 500 ns). The large majority of memory fetch and store operations complete in a single traditional memory cycle.
- Assuming that dynamic memory is available, the worst-case time required to allocate memory, regardless of the object's size, is approximately 2 memory cycles (about 250 ns). If the dynamic memory is not available, garbage collection must be initiated. The time required to initiate garbage collection ranges between 10 and 500 μs, depending on system configuration.
- Intelligent components within the special garbage collection module run without intervention from the system CPU. Garbage collection typically initiates no more frequently than once per second.

We have ported version 1.37 of the GNU C++ compiler to a simulated implementation of the garbage collection architecture. In order to reduce the time required to initiate a garbage collection pass, the set of root pointers is contained entirely within the CPU's registers. 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[2]:

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

[2] View these requirements as principles of operation rather than hard-and-fast rules. There are many cases in which, for example, 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.

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.)

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[3]. Assignments by way of a pointer to the union field violate this constraint.

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

Our experience porting existing C++ code to the garbage-collected C++ implementation reveals that most existing 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.

Note that our language restrictions are fairly small. It is straightforward for programmers to write stylized C++ code that is portable between traditional C++ implementations and the C++ dialect that we designed to be garbage collected in hard real-time applications. Note also that our garbage collection environment is much more compatible with traditional techniques for task execution time analysis than most alternative real-time garbage collection techniques. Tasks that refer to garbage-collected memory may assume that each memory fetch and store operation that refers to the garbage-collected heap requires the worst-case time associated with the respective operation. Garbage collection is only initiated in response to a memory allocation request that cannot be satisfied from the existing free pool. Thus, the only tasks that must account for the cost of initiating garbage collection are tasks that allocate dynamic memory. Furthermore, in almost all cases, it can be assumed that garbage collection is initiated no more than once per execution of each task.

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. The traditional C++ implementation uses GNU `malloc` and `free` [44, 45]. In comparison with alternative implementations of `malloc` and `free`, the GNU allocator is very space efficient and exhibits average run-time performance. The total execution times of our garbage-collected implementations range from 25.8% higher to 7.5% lower than the times required to execute traditional implementations of the same programs. The program that performs worst on the garbage-collected hardware is a simple Lisp interpreter, designed for instructional uses, that heap allocates `cons` cells without freeing them. It is worth noting that the garbage-collected implementation recycles the memory that the original implementor did not bother to reclaim. As a result, the combined sizes of the garbage-collected system's *to-* and *from-space* heaps is 59 to 71% smaller than the sizes of the heap required to run traditional implementations of the same applications.

Space does not allow for complete descriptions of our experiments and their analysis. Other reports provide more thorough explanations [42]. The measurements we've briefly reported here demonstrate

[3] We can relax this requirement at a cost of tagging all assignments made by way of pointers to lvalues. We are currently investigating the performance impact of making this change.

that our custom memory architecture has the potential of supporting real-time garbage collection without degrading system throughput. Ongoing research is addressing limitations in our preliminary experiments, including investigation of several proposed performance optimizations.

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 briefly 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.

Acknowledgements

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 our special-purpose memory architecture and performed the experimental measurements that characterize our system's average-case throughput.

References

1. 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.

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

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

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

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

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

7. 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.

8. 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.

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

10. 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.

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

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

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

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

15. 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.

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

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

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

19. 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.

20. 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*, Dec. 1987.

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

22. 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.

23. 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.

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

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

26. 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*, Dec. 1991, 181-189.

27. 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.

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

29. 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.

30. C. Chambers, Cost of Garbage Collection in the SELF System, *1991 Workshop on Garbage Collection in Object-Oriented Systems of OOPSLA*, Phoenix, AZ, Oct 1991.

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

32. 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.

33. 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.

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

35. 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.

36. 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.

37. K. Nilsen and W. J. Schmidt, Cost-Effective Object-Space Management for Hardware-Assisted Real-Time Garbage Collection, *ACM Letters on Prog. Lang. and Systems*, to appear.

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

39. 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, Oct 1991.

40. H. Boehm, A. J. Demers and S. Shenker, ‘‘Mostly Parallel Garbage Collection’’, *ACM SIGPLAN Notices Conference on Programming Language Design and Implementation*, June 1991.

41. H. G. Baker Jr., ‘‘List Processing in Real Time on a Serial Computer’’, *Comm. ACM 21*, 4 (Apr. 1978), 280-293.

42. K. D. Nilsen and W. J. Schmidt, A High-Performance Hardware-Assisted Real-Time Garbage Collection System, submitted for publication.

43. 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.

44. B. Zorn and D. Grunwald, Evaluating Models of Memory Allocation, University of Colorado Tech. Rep. CU-CS-603-92, July 1992.

45. B. Zorn, The Measured Cost of Conservative Garbage Collection, University of Colorado Tech. Rep. CU-CS-573-92, April 1992.