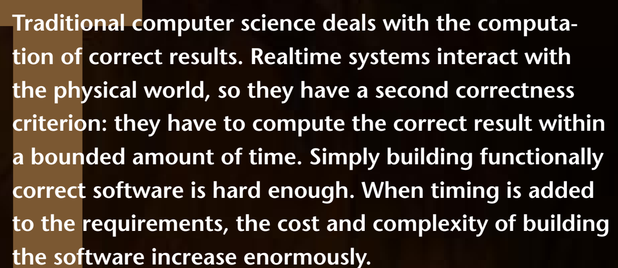
Realtime Garbage Collection

It's now possible to develop realtime systems using Java.



Traditional computer science deals with the computation of correct results. Realtime systems interact with the physical world, so they have a second correctness criterion: they have to compute the correct result within a bounded amount of time. Simply building functionally correct software is hard enough. When timing is added to the requirements, the cost and complexity of building the software increase enormously.

In the past, realtime systems have made up a relatively small portion of the total amount of software being produced, so these problems were largely ignored by the mainstream computer science community. The trend in computer technology, however, is toward an ever-increasing fraction of software having some kind of timing requirement.

Continued miniaturization has brought computers into a much larger range of physical systems. A car today has several networks and up to 100 processors. Software forms a larger and larger portion of the consumer value of a car, and by 2011 it is estimated that a new car will run 100 million lines of code—more than twice as much as in the Windows operating system. This trend will continue to move down the food chain: in the not-too-distant future, your pen might have a microprocessor, wireless networking, and run a 1-million-line application.

The other trend is for more traditional computer systems to take on more and more realtime properties. Spurred by VoIP, an increasing number of applications are making use of audio and video, enriching traditional business applications. Computer arbitrage of financial

DAVID F. BACON, IBM RESEARCH

Realtime Garbage Collection

instruments is now so time-critical that an advantage of a few milliseconds can translate into millions of dollars a year in profits.

This article describes a new technology developed at IBM that makes it possible to develop realtime systems, even those that have extremely demanding timing requirements, in standard Java. Called Metronome, this new technology has already been adopted for use in the Navy's new DDG-1000 destroyer, currently under development and scheduled for launch early in the next decade. At IBM, we are using Metronome as the underlying technology for a meter-scale helicopter called the JAviator, for which we've conducted initial flight tests. Since music is one of the most demanding realtime applications (the human ear is incredibly sensitive), we've also built a Java-based music synthesizer with response times of five milliseconds—as good as that achieved by hardware synthesizers.

SAFE LANGUAGES AND GARBAGE COLLECTION

The explosive growth of Java over the past 10 years has shown the enormous benefits to software productivity that come from a *safe* language. *Safety* in this sense means that it is impossible for the program to corrupt memory. Safety is achieved through garbage collection: instead of the programmer manually freeing memory, the garbage collector periodically scans the memory space of the application, finds unused objects, and reclaims their space.

Garbage collection has been in use since its invention for the Lisp programming language by John McCarthy in 1960.[1] In the late 1970s, as computer power and memory size continued to increase, garbage collection and object-oriented programming were brought together in Smalltalk, and a revolutionary new programming style was born based on dynamically allocated objects, the implementation and storage layout of which were independent of the interface that they implemented.

Of course, there's no free lunch. As with many techniques for improving programmer efficiency, garbage collection requires additional computation. But the real killer is that garbage collectors could not do that additional computation in little bits here and there throughout the computation, the way it is done in languages such as C or Pascal with the `free()` operator. Garbage collection is a global operation that is applied to the entire state of the memory. Not only does it slow the application down, it also introduces unpredictable and potentially long *pauses* into the execution of the application.

Although computer scientists made improvements in both the overhead and the delays caused by garbage collection, it continued to be a fringe technology until the arrival of the Worldwide Web in the mid-1990s. Everyone wanted dynamic, exciting Web pages, but no one wanted to allow an unknown Web server to start running arbitrary code on a personal computer. The explosive growth of Java resulted from its use of garbage collection to maintain language-level safety.

SOFT VERSUS HARD REALTIME

A distinction is often made between *soft realtime* and *hard realtime* applications. In the former, missing a deadline is undesirable; in the latter, missing a deadline is considered a catastrophic failure.

This is an oversimplification. In practice, there are three primary characteristics of realtime applications: response time, determinism, and fault-tolerance.

Response time is the amount of time within which the application must respond to an external event: a music synthesizer must respond to a person pressing on the piano keyboard within five milliseconds; a helicopter must respond to a change in the gyro reading within 50 milliseconds; a telecom server must respond to a session initiation message within 20 milliseconds.

Determinism is the predictability of the response time. The predictability often needs to be tighter when the response time is lower, but not always. For example, telecom servers can tolerate variance out to 50 milliseconds, whereas helicopter control can tolerate only a few milliseconds of variance.

rants: feedback@acmqueue.com

Fault-tolerance is the behavior of the system when a deadline is not met. As systems become more safety- or financially critical, they must be engineered not only to meet their deadlines but also to behave well in the event that the deadline is missed. Missed deadlines can be caused by software faults or (more commonly) by unpredictability in the environment, such as a network cable being cut or a memory parity error.

Building a well-engineered realtime system involves understanding all of these parameters and designing the system appropriately. For classical realtime control, simple but rigid scheduling disciplines such as rate-monotonic scheduling may be appropriate.

Telecom systems require approaches based on queuing theory. In particular, if the system can tolerate a variance of 30 milliseconds, this does not mean that it is acceptable to have a garbage collector with 30-millisecond pause times: such latencies will cause queues to back up and will disturb the response time of subsequent requests as well. When a number of requests come very close together, the system may not be able to maintain its overall responsiveness and determinism requirements.

METRONOME: COMING IN FROM THE COLD

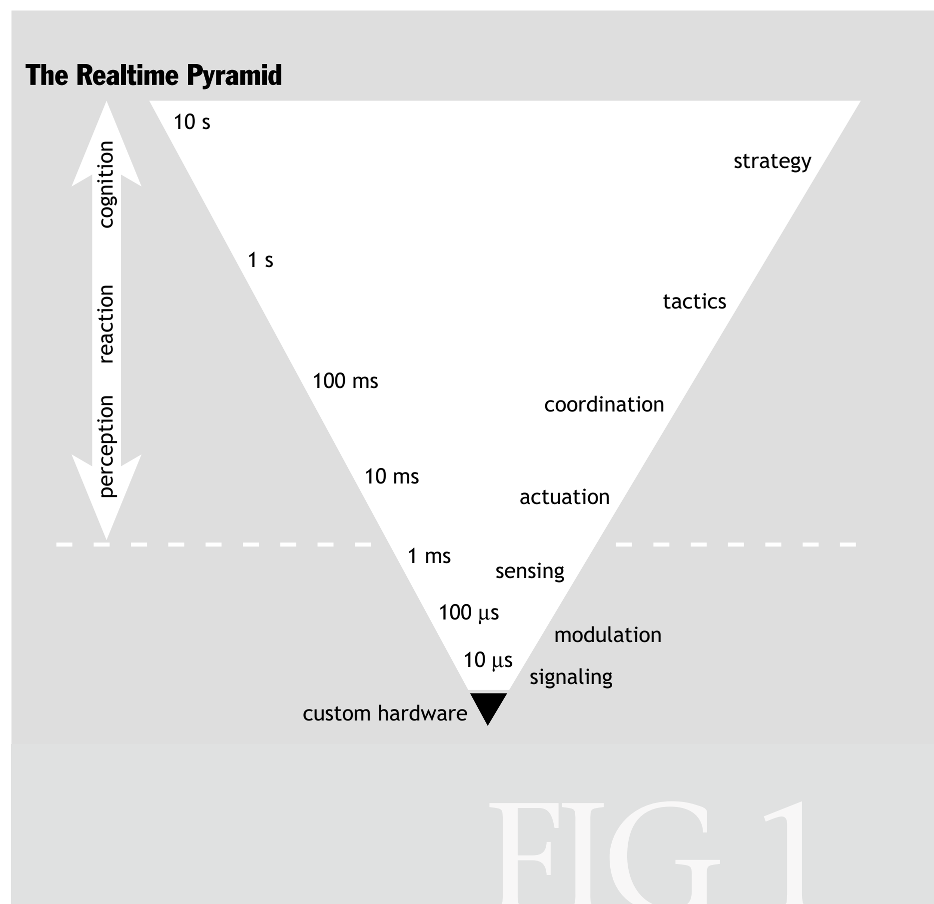
Because of the interruptions caused by garbage collection (up to several seconds), realtime programmers have been left out in the cold, unable to obtain the benefits of programming in a safe language such as Java. As realtime systems have become both larger and more pervasive, the inability to use garbage collection in a realtime application has become a critical problem.

The Metronome technology, developed at IBM Research and now available in production, solves this problem by limiting interruptions to one millisecond and spacing them evenly throughout the application's execution.[2] Memory consumption is also strictly limited and predictable, since an application can't be realtime if it starts paging or throws an out-of-memory exception. In other words, realtime behavior depends on predicting and regularizing all of a system's resource utilization.

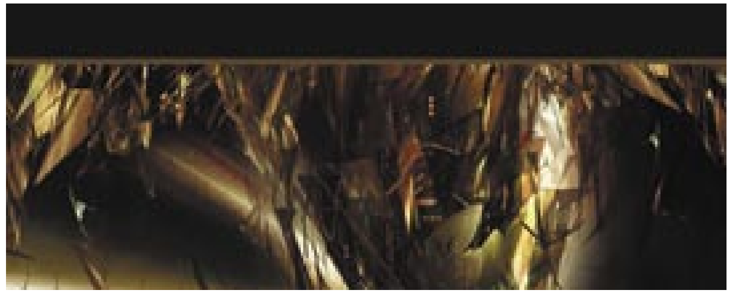
Realtime systems typically are constructed in an inverted software pyramid, as shown in figure 1. The most time-critical code is at the bottom, but it typically also represents a relatively small portion of the total application. Built on top of that is code with somewhat less critical timing constraints, which may make use of the bottom layer many times. A complex system may have several such layers.

Realtime systems are always limited by their weakest link. If there is a problem at the bottom of the pyramid, it will ripple up through the entire system, causing errors, forcing redesigns and workarounds, and generally making the system fragile and unreliable. Metronome allows Java to be used in systems whose timing requirements are as low as one millisecond, which covers the vast majority of realtime systems.

Other companies are also providing virtual machines



Realtime Garbage Collection



with some realtime characteristics; for example, BEA has introduced a virtual machine that can usually achieve worst-case latencies on the order of 40 milliseconds. This is good enough for some domains, such as certain telecommunications systems.

The result is a revolutionary productivity increase in the construction of realtime systems.

STOPPING THE WORLD

Garbage collectors typically stop the application because they need to scan the pointers in the heap. In figure 2, local variables on the stack point to objects A and B, which in turn point to B, C, D, E, and F. If the program destroys the pointer from B to E, and if there is no more memory at a subsequent allocation, the collector will be triggered. It will start with the pointers on the stack and follow all the reachable pointers in the heap, marking the objects it finds as *live*. This means that it will not find objects E and F, and they will be reclaimed for subsequent allocation.

The two basic kinds of collectors are *mark-and-sweep* and *copying*.[3] A mark-and-sweep collector marks the objects as it encounters them and then makes a pass in which it reclaims the unmarked objects. A copying collector copies live objects into a new region of memory when it encounters them for the first time, updating all pointers to use the new addresses of the copied objects.

Copying collectors have the advantage of requiring only work that is proportional to the size of the live data. They also have better locality because objects are allocated contiguously and kept densely. Mark-and-sweep collectors, on the other hand, don't suffer the overhead

Stop-the-World Garbage Collection

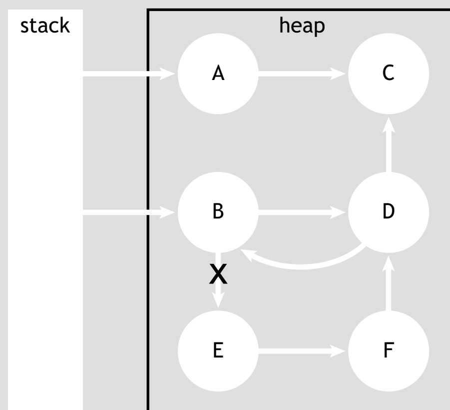


FIG 2

Lost Objects in Incremental Garbage Collection

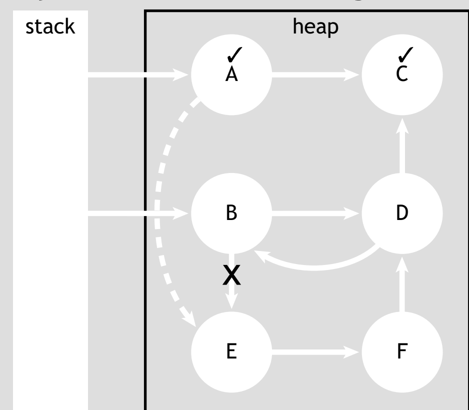


FIG 3

of copying and don't use nearly as much memory because they don't need an entire clean region into which to copy the objects.

COLLECTING ON THE FLY

Stopping the application completely while garbage collection takes place can introduce long delays, typically hundreds of milliseconds for the heap sizes that are typical today. With the introduction of 64-bit architectures and continued expansion of available RAM, the cost is likely to get worse.

Obviously, interrupting the application in small steps of collection would be better than requiring the entire operation to happen in a single stop-the-world step. This isn't easy, however, because the collector is examining the shape of the heap at the same time that the application is changing it.

People have avoided long pauses in two fundamental ways: *generational* and *incremental* collection. A generational collector partitions the heap into two regions: a nursery, where newly allocated objects are placed, and a mature space. When the nursery fills up, the objects are copied into the mature space, and their pointers are updated. In addition, during execution every pointer stored into the mature space has recorded mature-to-nursery references, so those can be updated quickly when the nursery objects are copied.

Collecting the nursery typically takes far less time than collecting the heap. Generational collection, however, only puts off the inevitable: sooner or later, the mature space will fill up and a full garbage collection will have to be performed. In other words, generational collection does not improve the *worst-case* behavior of collection, and for realtime systems it is the worst-case behavior that matters. Generational collection is now common in commercially deployed Java virtual machines.

An incremental approach is necessary for avoiding the need for stop-the-world collection, but this requires a solution to the problem of concurrent modification of the heap by the application.

The problem is illustrated figure 3, using the same heap from figure 2. Imagine that we are interleaving the collector with the application. The collector starts and marks A and C (shown by the check mark on the objects). Then the application runs, reads the pointer from B to E, stores it in A, and overwrites the pointer to E in B. Now the collector starts again, picking up where it left off, and marks B and D. But it hasn't marked E and F, even though they are still reachable from A. This is called the *lost object problem*.

NOT REALTIME? LET ME COUNT THE WAYS

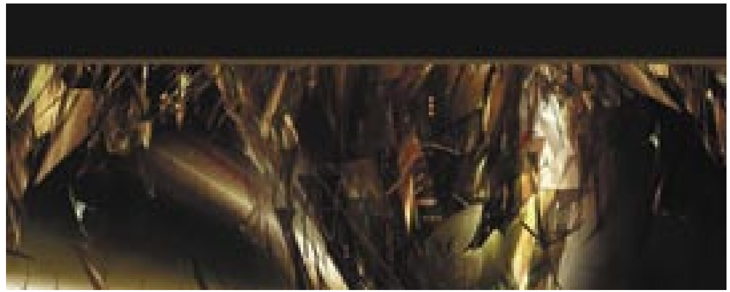
Incremental collectors that avoid the lost object problem have existed since the mid-1970s, but they have always suffered from one or more other problems that made them inappropriate for true realtime applications:

- Although the collectors were incremental, they were typically *work-based*, meaning that collector activity was triggered by allocation operations or other unpredictable application activity. That meant that an application could slow down unpredictably if the work trigger went off at a certain point, particularly if it suddenly allocated a lot of objects.
- Incremental collectors that compact memory are subject to *trap storms*. A trap storm occurs because during collection, the collector forces the application thread to copy an object into the new memory area before it can access it. Immediately after collection has started, none of the objects has been copied yet, so every single object access will cause a trap to code that copies the object. As a result, the speed of the application varies enormously, depending on how recently garbage collection was triggered.
- Many incremental collectors do not perform memory compaction. Over time this can lead to memory fragmentation, which can cause the application to consume many times as much space as it really needs. One of the key insights of the Metronome collector is that to achieve realtime behavior, one has to consider time and memory together as a single equation. I'll return to this in more detail later.
- Incremental collectors are often subject to *pathological behavior* in certain situations, such as when there are a large number of threads or when a large region of memory becomes unreachable as a result of the deletion of a single pointer.
- Incremental collectors do not solve the problem of *scheduling* the collector in such a way that the timing behavior of the application can be guaranteed.

SAYING WHAT YOU MEAN

The Metronome collector solves all of these problems and provides guaranteed behavior that is based on a simple characterization of the application. For the first time, it allows hard realtime applications to be written in a garbage-collected language.

The key to solving these problems is first to have a precise way of defining what we want to achieve. We have to precisely characterize how the execution of the collector will affect the execution of the application, so that the application developer knows exactly how it will behave.



We also have to characterize the memory consumption of the system, because if it uses too much memory, it will start paging and, therefore, cease to be realtime.

Previous incremental collectors focused on *worst-case pause time*, the single largest interruption of the application by the collector. Good research-quality incremental collectors were typically able to achieve worst-case pauses of 50 milliseconds or so.

Because of trap storms and work-based scheduling, however, pause times tell only part of the story. As an extreme example, imagine that the collector runs for 50 milliseconds, then the application runs for one millisecond, then the collector runs again for 50 milliseconds. The worst-case pause time is 50 milliseconds, but from the point of view of the application, if it needs to do more than one millisecond of work, it's as bad as a 100-millisecond pause.

The correct metric is MMU (minimum mutator utilization).[4] *Mutator* is what garbage-collection aficionados call applications, since from the point of view of the collector, the only relevant aspect of the application is that it mutates the heap.

Scheduling Affects Realtime Behavior

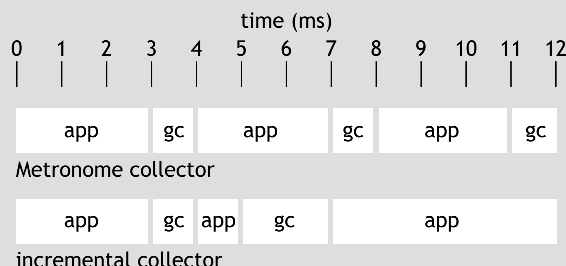


FIG 4

Metronome schedule is regularly spaced. Irregular schedule causes low CPU availability between times 3 and 7.

MMU measures the minimum (worst-case) amount of time that an application runs within a given length of time. For example, an MMU of 70 percent at 10 milliseconds means that the application is guaranteed to run for at least seven out of every 10 milliseconds.

Figure 4 shows the execution of an application, interleaved with the garbage collector. In the first execution (a), the collector runs for one millisecond after every three milliseconds of application execution. Its MMU (four milliseconds) is 75 percent. In the second execution, the collector performs exactly the same amount of work, but its MMU (four milliseconds) is only 25 percent, since in the four milliseconds from time 3 to 7, the application is running only 25 percent of the time.

MMU allows the requirements to be specified in terms that are meaningful to the application. An application generating video at 25 frames per second (one frame every 40 milliseconds), and requiring 20 milliseconds to generate a single video frame, requires an MMU (40 milliseconds) of 50 percent.

TIME IS MONEY, BUT SPACE IS TIME

Once you know the requirement of your application in terms of its MMU, you need to know whether it is achievable. We configure Metronome to run by default with an MMU (10 milliseconds) of 70 percent, with worst-case pauses of one millisecond. This works out of the box for most applications, but for critical systems the application must be properly understood so that the behavior can be guaranteed. Understanding how to provide guaranteed behavior is important to the application development process.

With Metronome, achieving an MMU goal depends on two application parameters: the maximum live memory that it uses and the maximum long-term allocation rate.

If this seems like a lot of information to provide, remember that even for a non-realtime application, you can't be sure that it will run without knowing its maximum live memory consumption: if you use more memory than the system has, you can't run. It's not

surprising that to guarantee the realtime behavior of the application, you need to know something about the rate at which it's consuming resources. If you allocate memory at a furious rate, it's clear that the collector will have to run frequently, which means that eventually the only way you can give the collector enough time is to lower your MMU requirement.

The amount of time required to perform a garbage collection depends on how much live memory the program consumes, since it has to trace through all of those objects (this is something of an oversimplification, but will serve to understand all of the key concepts). That collection time has to be spread out over the application's execution, and the MMU specifies how the collector is allowed to do this.

For example, if our video application uses, at most, 100 MB of memory, then on a particular piece of hardware it might be collected in two seconds. Since the requirement is MMU (40 milliseconds) at 50 percent, those two seconds must be spread over four seconds of actual time.

This is where the memory allocation rate comes into the picture: when the collector starts running, the system has a certain amount of free memory. That memory, however, cannot be exhausted before the collection finishes four seconds later, or else the entire system will block. Therefore, if the application is allocating 30 MB for every second of its execution, then it will allocate 60 MB while the collection is in progress. This means that the system requires at least 160 MB to achieve its MMU target and maintain realtime behavior.

This property allows increased space to be traded for better realtime behavior: by providing more memory, you provide a bigger "buffer," allowing the collector to take longer to finish, which means it can run with a higher MMU. If you don't have more memory, you can either reduce your MMU requirement or change your application to allocate memory more slowly.

Figure 5 shows how this all fits together. The graph on the top shows the amount of memory in use (red), the regions where garbage collection is active (yellow), the MMU (green), and the MMU requirement (blue). When the collector is inactive, the MMU is 100 percent. When the collector is running, the MMU drops, but remains above the target of 70 percent. When the collection finishes (end of the yellow region), memory utilization drops back down. The first collection doesn't free much memory; the second one frees a lot more. The amount of memory freed depends on the live memory in the heap at the time that collection starts.

The bottom graph in figure 5 shows the detailed behavior of the collector at the level of individual collector quanta. In this graph, time reads like a book: left to right and top to bottom. Where the figure is white, the application is running; where it is colored, the collector is running. Each rectangle is a single collector quantum of approximately 500 microseconds. The quanta occur

Garbage Collection Performance

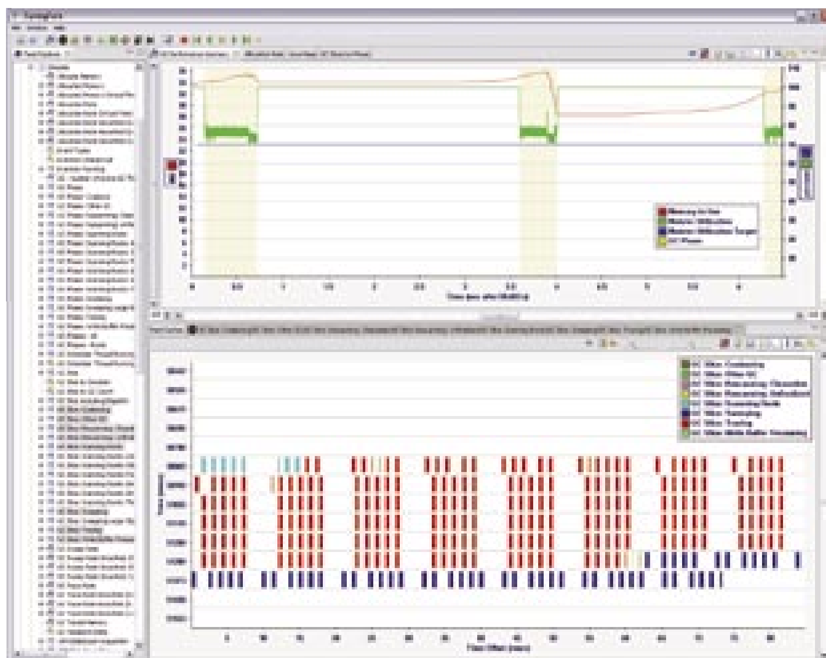


FIG 5

Realtime Garbage Collection

in groups of five or six, and there is one group every 10 milliseconds. This is because the system has been run at MMU (10 milliseconds) = 70 percent, with a target pause time of 500 microseconds.

If you are interested in understanding the behavior in more detail, these graphs are generated with an interactive realtime visualization tool called TuningFork, which is available from IBM alphaWorks (http://www.alphaworks.ibm.com/tech/tuningfork), along with some example Metronome traces.

COMPARISON WITH OTHER APPROACHES

A number of other less automatic approaches either guarantee or improve realtime behavior of garbage-collected languages. They can be roughly characterized as either object pooling or region-based memory management.

Object pooling is basically a form of manual memory management, implemented at the language level. A pool of objects is allocated at application startup, and objects are explicitly allocated from the pool and explicitly placed back into it. Pooling is essentially a higher-level form of `malloc/free`.

When implemented in Java, the pools are strongly typed, so freeing an object that is still in use will not result in memory corruption. It will still cause application failures, however, and the downside is that since the pool objects are strongly typed, the memory cannot be reassigned from one pool to another, so memory consumption may be increased by requiring many different pools.

For certain applications object pooling can be very effective, and if the data structures are simple, the danger of premature freeing may be low.

Region-based memory management divides memory into regions that are collected en masse. This is often coupled with a stack discipline: a memory region is associated with a particular function invocation, and that region can be accessed by that function and its callees. When the function exits, the region and all its objects are released.

Region-based memory management has the advantage that freeing memory is fast. It has the serious disadvantage, however, of constraining the structure of the application. In particular, a pointer from a region lower in the stack to a region farther up the stack could cause a dangling pointer when the upper region is freed.

There are several approaches to this. "Up-pointers" can be prevented dynamically using runtime checks, or a program analysis can assign objects to a region that is guaranteed to be low enough in the stack to avoid any up-pointers.

Automatic program analysis is safe and nonintrusive. In practice such analyses cannot do a very good job, especially in the presence of multithreading, so this approach is not yet mature enough for most environments.

Dynamic checking is the approach taken in RTSJ (Realtime Specification for Java), a variant of Java designed for realtime programming.[5] Since RTSJ was designed in the absence of realtime garbage collection, and since automatic region analysis was infeasible, it relies on regions (which it calls Scopes) and runtime checking. Every time a pointer is stored, the system checks whether it is an up-pointer from one Scope to another. If so, a runtime exception is thrown.

This has some serious disadvantages: most obviously, there is a significant performance cost to checking each pointer access for Scope containment. The real cost, however, is in program complexity: since Scope behavior is not documented in the interface of methods, the user of an API has no way reliably to predict its behavior in the presence of Scopes, especially when the Scope structure of the program involves multiple Scopes. The problem is especially acute with libraries developed without RTSJ in mind—the vast majority of all Java libraries. So, while Scopes buy determinism in terms of memory deallocation, they reduce overall flexibility and may cause unpredictable runtime failures.

BEYOND THE GARBAGE COLLECTOR

Although garbage collection is the largest source of nondeterminism in Java, there are other issues as well: class loading and JIT (just-in-time) compilation.

Java's dynamic semantics dictate that a class is not loaded and initialized until the first runtime reference to it. Since class loading involves file input and potentially long initialization sequences, large amounts of class loading can compromise realtime behavior. Our production realtime virtual machine includes the ability to preload a supplied set of classes and avoid this problem.

JIT compilation is also used extensively in Java runtime systems: the application initially runs in interpreted mode, and the system self-monitors, looking for "hot" methods. When a hot method is found, it is compiled, generally leading to a significant performance boost.

Using JIT in a realtime system has three problems, however. First, JIT compilation can be expensive and can therefore interrupt the application for a significant amount of time. Second, JIT compilation changes the performance of the code, which reduces predictability; performance usually improves, but, especially in systems that use multilevel optimization, some JIT compilations may actually slow the system down. Third, since JIT compilation is performed adaptively based on the program execution, it occurs at unpredictable points in the program's execution.

The combination of interruptions that occur at unpredictable times, last an unpredictable duration, and have an unpredictable effect on subsequent execution speed makes JIT highly problematic in a realtime system.

One solution is to use an AOT (ahead-of-time) compiler. An AOT compiler takes a unit of code (in our case, a JAR file) and compiles it into machine code. This can be done at application build time. Execution is then highly deterministic. Performance is sometimes lower than would be achieved with the best JIT compilation, but for very time-sensitive or critical applications this is generally the right trade-off to make.

For the synthesizer application that we built, for example, we cannot achieve glitch-free music synthesis with five-millisecond response time without AOT compilation. For the JAviator helicopter, which operates at a 50-millisecond period, JIT compilation is fast enough. Because of the criticality of the code, however, we use AOT compilation for production flights.

Another approach is to make JIT compilation more Metronome-like—that is, to have it perform its work in small quanta that are scheduled along with the collector to honor the application's MMU requirements. This provides much more predictable JIT behavior without the inconvenience of AOT compilation or the loss in performance resulting from lack of runtime information. For applications that cannot tolerate variable speed early in their execution, however, AOT compilation remains the best approach.

CONCLUSION

Garbage collection significantly simplifies programming and increases software reliability. With the advent of realtime garbage collection technology, these advantages can be applied in the realtime programming domain as well. As more and more applications are built with some kind of realtime requirement, these software engineering and productivity advantages become more and more important.

The Metronome technology is available now as a product, and over time you can expect to see more implementations of Java and other high-level object-oriented languages including garbage-collection technology that significantly increases determinism and reduces latency. Q

REFERENCES

1. McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3(4): 184–195.
2. Bacon, D. F., Cheng, P., Rajan, V. T. 2003. A Real-time garbage collector with low overhead and consistent utilization. *Conference Record of the Thirtieth ACM Symposium on Principles of Programming Languages* (New Orleans, Louisiana, January): 285-298.
3. Jones, R. E. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. New York: Wiley and Sons.
4. Cheng, P., Blelloch, G. 2001. A parallel, real-time garbage collector. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June). SIGPLAN Notices 36(5): 125–136.
5. Bollella, G., Gosling, J., Brosgol, B. M., Dibble, P., Furr, S., Hardin, D., Turnbull, M. 2000. *The Real-Time Specification for Java*. Reading, MA: Addison-Wesley.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

DAVID F. BACON is a research staff member at IBM's T.J. Watson Research Center, where he leads the Metronome project. His recent work focuses on high-level realtime programming, embedded systems, programming language design, and computer architecture. He received his Ph.D. in computer science from the University of California, Berkeley, and his A.B. from Columbia University.

© 2007 ACM 1542-7730/07/0200 $5.00