Dynamic Memory Management in High-Integrity Java Systems

Kelvin Nilsen, Ph.D.

Aonix North America
125 E. Main St., #501
American Fork, UT 84003
(+1) 801-756-4821
kelvin@aonix.com

Abstract: The Java programming language [1], widely recognized as the preferred programming language for all varieties of information processing applications, typically manifests a two-fold improvement in developer productivity and a five- to ten-fold improvement in software reuse, integration, and maintenance activities in comparison with legacy systems built using C or C++. Java is especially relevant to multiprocessor applications, as the language has built-in support for multithreading, synchronization, and shared access to common objects.

The full power of Java is relevant to many high integrity systems, including applications in ballistic missile defense, radar subsystems in support of air traffic control, and rail traffic scheduling. But using Java in these sorts of high integrity applications requires special attention to selection of an appropriate Java virtual machine and special development methodologies [2, 3].

This paper discusses some of the special challenges of using Java in high integrity real-time systems, along with recommended practices for dealing with these challenges.

Keywords: Java, real-time Java, hard real time, soft real time, safety certification, high-integrity Java, real-time specification for Java

1. Relevance of Dynamic Memory Management

The designers of the object-oriented Java language made a distinction between primitive types and objects. Variables representing the primitive types, which include int, char, float, and double, are always allocated on the Java thread's stack. Java variables representing objects are really just references (pointers). The Java objects referred to by Java reference variables are generally allocated from a garbage-collected heap.[1] The syntax of Java makes it very clear when objects are allocated. However, a cursory review of Java source code may overlook many of the allocations that it embodies. Each of the Java source code examples shown below results in allocation of at least one object:

```
// Explicit allocation of HashMap
HashMap m = new HashMap();

// Implicit allocation of StringBuilder and String
System.out.println("the answer is " + x);

// Implicit allocation of Integer(13) if foo() is declared to expect
// an object argument. This is known as auto-boxing.
//     void foo(Integer);
foo(13);

// Implicit allocation and initialization of an Object[3] array, and
// the three Integer objects to be represented by this array if
// the baz() method is declared to expect a variable number
// of arguments. This also is known as auto-boxing.
baz(1, 2, 3);

// Probably does an allocation of internal data "record" to
// represent the pairing between key and value within the
// hash table represented by the variable m. The specification
// for this method does not make clear when internal data
// structures are allocated.
m.put(key, value);
```

In Java, it is not considered important to syntactically identify every object allocation because the programmer who writes code to force creation of objects is not responsible for reclaiming the object's memory when the object is no longer required. Instead, the tracing garbage collector takes full responsibility for automatically detecting the death of objects and reclaiming their memory.

The Java style of dynamic memory management has numerous benefits over the more traditional approach of explicitly allocating and deallocating objects as is done in Ada, C, and C++. Particular benefits of the Java approach are identified below:

No dangling pointers. The term *dangling pointer* refers to a situation in which a pointer to an object persists longer than the object itself. Once the object is deallocated, any pointers that still refer to that object are considered to "dangle", because they point to memory that is no longer dedicated to the intended purpose.

1. Sophisticated compiler optimizations are able to detect that many temporary objects can be allocated and reclaimed from the thread's stack memory.

Full type safety. Unlike Ada, C, and C++, Java guarantees that a variable declared as a reference to a particular type only points to objects of that type. Languages that permit dangling pointers generally allow the possibility that a deallocated object will be reallocated as a different type that is incompatible with the original object. After the deallocated object's memory is reallocated, dangling pointers to the original object now refer to the "wrong type".

Avoidance of memory leaks. The term *memory leak* represents a situation in which allocated objects are never reclaimed, even after they no longer serve a useful purpose. With Java, an "accurate" garbage collector guarantees to find all garbage, where garbage is defined as objects that are not reachable by following a chain of pointers starting with one of the system's root pointers and comprising zero or more non-garbage objects. Not all Java garbage collectors promise to accurately identify all garbage, but those designed for mission critical operation in limited memory embedded systems generally do. It is important to recognize that memory leaks may exist even in systems that incorporate an accurate garbage collector. This is because certain objects which are considered live (reachable) by the garbage collector may actually have no useful role in ongoing computations. Though Java programmers do not explicitly deallocate dead objects, it is important that they overwrite references to objects no longer needed with null in order to enable the garbage collector to reclaim their memory.

Memory defragmentation. With explicitly managed memory heaps, the allocation pool may become fragmented over time. Once the heap has become fragmented, allocation requests may fail even though there may exist an abundance of available memory. The integration of a garbage collector within the implementation of Java makes it possible to implement memory defragmentation as a peripheral benefit of garbage collection. Not all Java garbage collectors defragment memory, but those designed to support mission critical operation in limited-memory embedded devices generally do.

Ease of integration. An important attribute of object-oriented programming environments is the ability to easily integrate independently developed software components. With legacy languages, a difficulty of integrating independently developed components is that each new integration requires the design and implementation of a tailored protocol to allow deallocation of the objects allocated by one component but accessed by other components after the objects are no longer needed by any components. Designing, implementing, and debugging these protocols is very difficult, and this represents a significant impediment to the creation of large software systems through modular composition of independently developed components.

2. High Integrity Garbage Collection

When mission-critical systems are deployed as traditional Java applications, the underlying Java virtual machine's garbage collection system plays a critical role in overall reliability, real-time responsiveness, and ultimately in the system's ability to fulfill its mission.

Garbage collectors designed to support mission-critical operation must address the following issues:

1. Preemptible: if a higher priority activity needs to run while garbage collection is active, the higher priority activity must be able to preempt garbage collection within a predictably small amount of time.
2. Incremental: Whenever garbage collection is preempted, it is important that the increments of work completed prior to the preemption are preserved and the garbage collector's progress continues to advance when garbage collection resumes. Otherwise, it is not possible to guarantee forward progress, which is necessary to assure availability of memory to satisfy future allocation requests.
3. Accurate: The garbage collector must guarantee to accurately find and reclaim all of the dead memory in the system. If a dead object is not reclaimed by the garbage collector, all of the objects reachable from that dead object, many of which may also be dead, will necessarily be retained as if they were alive. Thus, an inaccurate (also known as *conservative*) garbage collector cannot guarantee availability of memory to satisfy future allocation requests.
4. Defragmenting: Over time, the interleaving of memory allocation and deallocation operations may create a situation in which the memory allocation pool consists of a large number of small free segments separated from one another by in-use objects. In the presence of memory fragmentation, the allocation pool may contain many megabytes of available memory, but an allocation request for an object of size 256 Kbytes could fail because no single segment of free memory is sufficiently large. A garbage collection system designed to support mission-critical execution must defragment memory or provide some mechanism to mitigate the effects of memory fragmentation.
5. Paced: To support real-time operation of mission-critical threads which depend on the ability to allocate memory in fulfilling their mission-critical objectives, it is essential that the garbage collector reclaim memory at a pace that is consistent with the application's appetite for new memory allocation. Otherwise, the mission-critical thread that attempts to allocate memory could become blocked waiting for the garbage collector to reclaim enough memory to satisfy its allocation request.

3. A Mission-Critical Garbage Collector

The PERC® Ultra virtual machine has been performing real-time garbage collection in a large number of highly available applications for over a decade. Here, we provide a high-level overview of real-time garbage collection as it is performed by the PERC Ultra virtual machine.

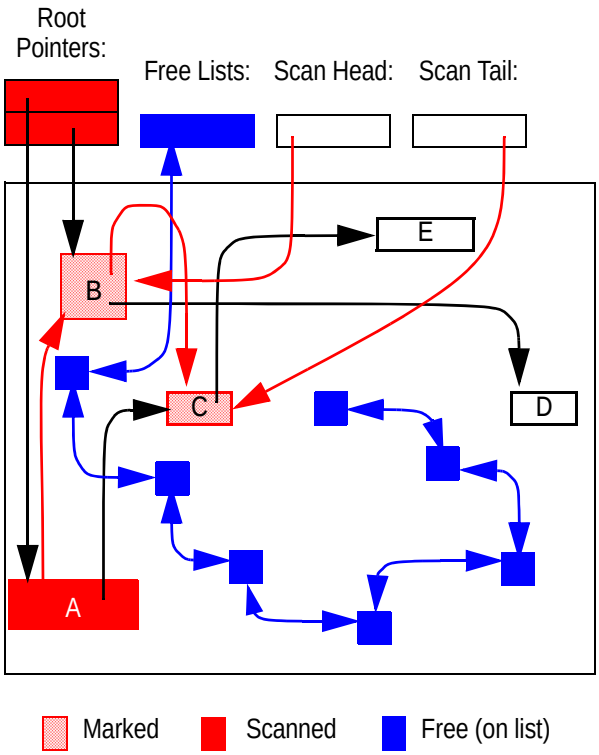

Figure 1. Incremental Mark and Sweep Garbage Collection

The PERC garbage collector uses a combination of mark-and-sweep and copying garbage collection techniques. Mark-and-sweep garbage collection is illustrated in Figure 1. In this illustration, the blue objects represent the free pool. The PERC allocator maintains several free lists, each one dedicated to different size ranges. For illustrative purposes, only one free list is shown here. The free lists are doubly linked to allow free segments to be removed, coalesced with newly found neighboring garbage, and inserted onto an alternative free list in constant time.

Black arrows represent references (pointers) from one object to another. Red arrows represent links on the scan list, a list that represents all of the objects that the garbage collector has marked as being in use. The garbage collector starts its search for in-use objects by marking the objects that are referenced directly from the root pointers. Then it scans those objects to see what they point to, and marks each of those objects as in-use also. It continues this process until all marked objects have been scanned. At this point, any memory that is not on the free list and is not marked is treated as garbage. All such memory is linked onto the free list.

The red objects represent objects that the garbage collector has identified as being in use (live or reachable, and thus retained). Every object recognized as live is placed on a scan list linked through the scan-list field within the object's header. The act of placing an object on the scan list is known as marking. A marked object is identified by a non-null value in the scan-list field of the object's header.

The scan list is represented by head and tail pointers. When a new object is marked, it is added to the tail of the existing list. For each object on the scan list, the garbage collector examines (scans) the object's contents to determine which additional objects might be referenced by this object. The garbage collector marks each object referenced from this marked object if that referenced object is not already marked. Figure 1 distinguishes objects that have been scanned from those that have been marked but not yet scanned.

Figure 1 represents a snapshot of the garbage collector's incremental progress. At the time of the snapshot, object A is considered marked and scanned. Objects B and C are marked but not yet scanned. The scan list holds only objects B and C. When garbage collection resumes, the collector will scan object B, which results in the marking of object D.

With traditional stop-and-wait garbage collectors, all application threads are suspended while garbage collection is performed. Though this simplifies garbage collection, the delays on application thread processing imposed by stop-and-wait garbage collectors can be quite long, lasting tens of seconds for gigabyte-sized heaps. The PERC garbage collector avoids these delays by allowing application threads to preempt the garbage collector.

Suppose, while the garbage collector is suspended in the state illustrated by Figure 1, that an application thread performs the following sequence of actions:

1. Fetch the reference to object D into a local variable by reading the relevant field of object B.
2. Overwrite the same field of object B with the value *null*.
3. Copy the object D reference from its local variable into a field of object A.
4. Overwrite the local variable that refers to object D with a null value.

Following this sequence, the object D is still live, because it is reachable from A, which is live. However, when the garbage collector resumes operation, it will not scan object A, because object A previously scanned. To make the mark-and-sweep garbage collection technique incremental, the PERC virtual

machine incorporates a write barrier. Each time a pointer field of an object is overwritten, the virtual machine marks the object referenced by the new value of the pointer field.

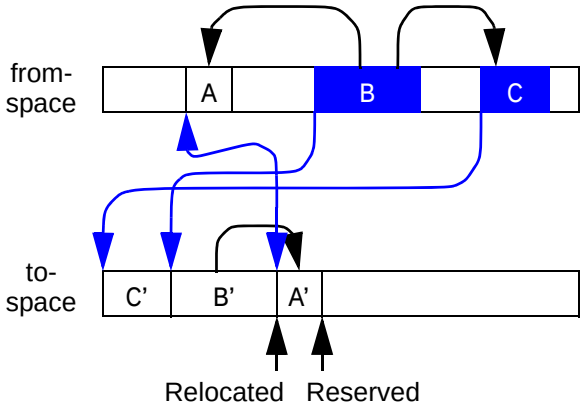

Figure 2. Fully Copying Garbage Collection

Copying garbage collection is shown in Figure 2. With copying garbage collection, live objects are incrementally copied out of one region of memory, named *from-space*, into another region of memory, named *to-space*. In this illustration, there are three live objects A, B, and C. Objects B and C have been copied to consecutive memory locations in *to-space*, represented by the objects labeled B' and C' respectively. Memory has been reserved for copying object A to the location labeled A', but that object has not yet been copied.

After all live objects have been copied out of *from-space*, we exchange the names given to *to-space* and *from-space*, and satisfy subsequent allocation requests from the new *to-space*. This has the effect of defragmenting the free pool. Note that the four unused memory segments in the original *from-space* are now combined into one large contiguous free segment in the new *to-space*.

If a high-priority thread preempts copying garbage collection in the state illustrated by Figure 2, the PERC virtual machine will redirect that thread's attempts to access object A' to object A. Likewise, it will redirect all attempts to access objects B or C to objects B' and C' respectively. The PERC virtual machine uses both read and write barriers to make this incremental copying garbage collection work reliably.

A key benefit of copying garbage collection is that it fully defragments memory with each pass of the garbage collector. However, the cost of providing this benefit is that memory utilization can never be any higher than 50%. In contrast, mark-and-sweep garbage collection typically achieves utilization of 60-75%. However, because it does not defragment memory, the worst-case utilization of a mark-and-sweep garbage collector may be arbitrarily poor (less than 5%). To combine the good average-case utilization of

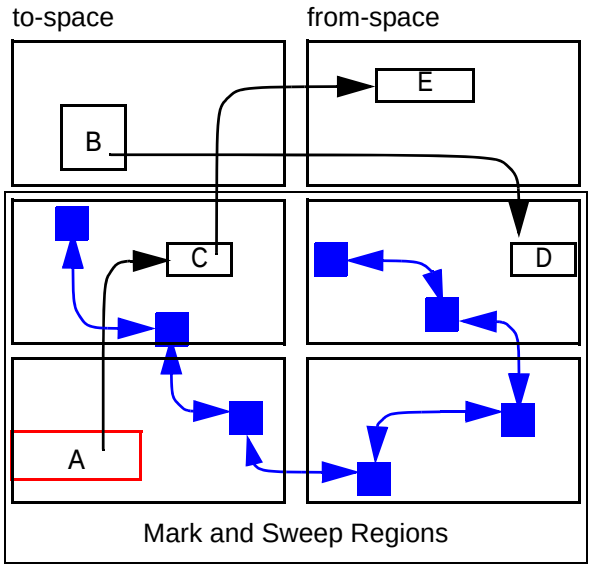

Figure 3. Incremental Mark and Sweep Garbage Collection

mark-and-sweep garbage collection with the reliability guarantees of copying garbage collection, the PERC virtual machine uses a mostly stationary garbage collection technique, illustrated in Figure 3. All of memory is divided into N equal-sized regions. Each time the PERC virtual machine begins another garbage collection pass, it applies a number of heuristics to select one of the regions to be defragmented. When configuring the PERC virtual machine, system integrators can set the number of regions to two so that all of memory is defragmented on each garbage collection pass, or can disable defragmentation entirely to maximize expected memory utilization. This allows system integrators to tune the PERC virtual machine for the levels of reliability and memory efficiency that are most appropriate for their given application.

4. Pacing Garbage Collection

The goals of real-time garbage collection are to make sure memory is available for allocation at the times the application needs to allocate without interfering with any task's compliance with real-time constraints. A real-time garbage collector must work incrementally, dividing its total effort into many small bursts of work. These bursts of work must be scheduled using the real-time scheduler so as to make sure that the real-time garbage collector makes timely forward progress while at the same time making sure that the garbage collector's CPU-time utilization does not intrude upon times set aside for execution of application software.

In order to make sure that garbage collection makes adequate forward progress, the total effort required to perform complete garbage collection

must be understood. Suppose, for example, that the total available memory is *M* bytes and that complete garbage collection is known to require *S* seconds of CPU-time. Suppose further that the total memory requirement of the system's real-time activities is *U* total bytes, and that the combined allocation throughput is *V* total bytes of allocation per second. Finally, let *R* represent the fraction of the CPU time that is dedicated to garbage collection. Note that the real time required to complete incremental garbage collection is *S / R*.

Consider the state of memory immediately following completion of garbage collection. In the worst steady-state case, there are a total of *U* bytes of live memory and *V (S / R)* bytes of dead memory currently occupying the heap. If we start the next garbage collection pass as soon as the first has completed, an additional *V (S / R)* bytes of memory will be allocated while this garbage collection pass is executing. Thus, the size of the space required to support this workload, *M*, measured in bytes, must be greater than or equal to *U + 2V(S/R)*. Based on the combined total memory requirement and maximum allocation rates described above, the minimum fraction of CPU time that must be spent in garbage collection is given by:

$$R \geq (2VS)/(M-U)$$

Note that *R* is proportional to the maximum rate at which memory is allocated multiplied by the total time required to perform a stop-and-wait garbage collection pass. *R* is inversely proportional to the difference between the total amount of available memory and the maximum amount of live memory.

When targeting a virtual machine designed to support deployment of high integrity Java applications, it is important that the virtual machine vendor be able to quantify upper bounds on the amount of CPU time required to perform a complete garbage collection, and offer the ability to control when this CPU time is consumed, at what priorities, and at what preemption latency.

At the same time, a complete analysis of the system's real-time behavior also requires that developers obtain upper bounds on the amount of memory retained as live by the application, and on the rate at which new objects are allocated.

The objective of garbage collection pacing is to make sure that garbage collection gets enough

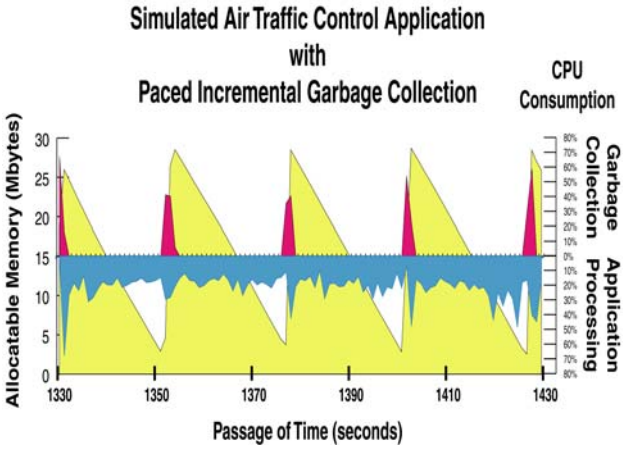

Figure 4. Pacing of Garbage Collection

increments of CPU time to make sure it consistently replenishes the allocation pool before the available supply of memory has been exhausted. Figure 4 shows a simulated air traffic control workload with real-time garbage collection running under the direction of a real-time pacing agent. This particular application is running in a fairly predictable steady state as characterized by the following observations. First, the slope of the yellow chart, which represents the amount of memory available for allocation, is roughly constant whenever garbage collection is idle. This means the application's allocation rate is approximately constant. Second, the heights of the yellow chart's peaks are roughly identical. This means the amount of live memory retained by the application is roughly constant. In other words, the application is allocating new objects at approximately the same rate it is discarding old objects. Finally, the percentage of CPU time required for application processing is well behaved, ranging from about 20% to 50%.

Note that garbage collection is idle most of the time. As memory becomes more scarce, garbage collection begins to run. When garbage collection runs, it interferes with some, but not all, of the real-time application processing. For this reason, a slight dip in application processing each time the garbage collector, represented by the occasional red upward spikes, runs is evident. Also evident is a tendency for application processing to briefly increase following each burst of garbage collection. This is because the preempted application needs to perform a small amount of extra work to catch up with real-time scheduling constraints following each preemption. If properly configured, the pacing agent will carefully avoid

delaying the application threads by any more than the allowed scheduling jitter.

5. Multiprocessor Considerations

The Java language specification includes compromises intended to enable portable deployment of the Java run-time environment on many different CPU architectures and operating systems. One such compromise is the restriction to only ten priorities, with no guarantee that higher priority threads will be scheduled ahead of lower priority threads. Further, the Java language specification does not guarantee that Java's built-in synchronization locks will implement priority inheritance.

Vendors of Java virtual machines targeted to high integrity real-time applications generally constrain scheduling and synchronization behavior beyond what is required by the Java Language Specification. Nevertheless, it is important to understand the scheduling model implemented by a particular Java Virtual Machine and tailor application code to that model.

1. How many priorities does the virtual machine support?
2. Does the virtual machine's scheduler strictly honor priorities?
3. Does the virtual machine provide API support to bind particular Java threads to specific processor cores or core subsets?
4. Does the virtual machine allow developers to bind particular threads to particular processors?
5. Does the virtual machine implement priority inheritance? How does the virtual machine characterize priority inheritance behavior on a multiprocessor system if some threads are bound to particular cores?
6. How concurrent is garbage collection? For a given heap size, does multi-core garbage collection of that heap run N times faster than single-processor garbage collection, where N is the number of processor cores? If not N times faster, what are the limiting factors?
7. Are all of the virtual machine's background infrastructure activities, including garbage collection and JIT compilation, easily understood and analyzable as part of the complete system's real-time workload?
8. Depending on the underlying scheduling mechanism, what is the theory for analysis of schedulability?

Note that analysis of real-time scheduling becomes much more difficult on a multiprocessor system and considerable care must be exercised to assure compliance with hard real-time deadlines. One difficulty results from tension between the main-stream objective of optimizing throughput and the real-time objective of assuring predictable scheduling behavior. To achieve optimal throughput, it is desirable to reduce the frequent migration of threads from one processor to another. However, this objective contradicts the neeed of real-time scheduling to always dedicate resources to the highest priority threads that are ready to run. Further, the analysis of a thread's worst-case execution time may be significantly affected if the thread is allowed to migrate from one processor to another. One way to address these tensions is to allow programmers to bind certain threads to particular processors. This reduces the scheduling uncertainty that results from automatic load balancing. The analysis of real-time schedulability can be done independently for one processor at a time. However, this approach introduces a different problem. Specifically, if a thread bound to one processor attempts to lock a resource that is shared with other processors, there is a risk of priority inversion. If a low-priority thread running on one processor holds a lock on a resource that is required by a high priority thread running on a different processor, the traditional approach to resolving this contention uses priority inheritance to boost the priority of the locking thread so that it can quickly get out of the way of the high-priority thread that desires access to the locked resource. When each thread is bound to a different processor, traditional priority inheritance does not work.

There is not universal agreement on the single best way to structure real-time threads for predictable and deterministic operation on a multiprocessor platform. The most important consideration is that these issues cannot be ignored. Architects, designers, and system integrators need to carefully consider the structure of their application code and the precise semantics of the services provided by the underlying platform in order to assure a reliable and maintainable integration. Additional discussion on topics relevant to multiprocessor embedded Java deployments is available in reference [4].

6. Garbage Collection in Hard Real-Time and Safety Critical Systems

In both hard real-time and safety critical systems, there is a general expectation that the system is proven to operate within real-time constraints. All timing constraints will be satisfied, and all specified actions will be performed within the specified timing constraints, even actions that require allocation of temporary objects.

Traditionally, hard real-time and safety critical systems have avoided all use of a dynamically managed memory allocation heap because the analysis of memory management services is very difficult. In particular, allocating an object of a particular size may require a search for a best fit or a first fit within a very long list of available free segments. The time required to successfully find the desired free segment is unpredict-

able. And even if the developers have carefully proven that a sufficient amount of memory has been deallocated to support a subsequent memory allocation request, there is usually no guarantee that the memory allocation request will succeed because memory may have become fragmented.

Since modern automatic garbage collection systems are capable of defragmenting memory, some garbage collection enthusiasts have promoted the use of automatic garbage collection in support of low-level hard real-time and safety-critical code. While their argument that a real-time garbage collector can be proven to be as safe and reliable as stack memory allocation is perfectly valid, this argument tends to ignore some of the larger issues associated with dynamic memory management.

In particular, to use garbage collection in a hard real-time or safety critical system, you must prove not only that the garbage collection implementation is fully deterministic, but you must also prove that the application's interaction with the garbage collector is fully consistent with the demands placed on the garbage collector by the application.

Proving that a real-time garbage collector behaves deterministically with respect to CPU time consumption, latency between object release and recycling of the object's memory, relocation and coalescing of independent free memory fragments, and the time required to respond to each allocation request is all relatively straightforward, though such proofs may be tedious and very costly. The more difficult proof deals with the application's behavior. All of the following considerations must be addressed:

1. Either the application needs to reallocate memory under real-time constraints or it does not.
   a. If the application performs no memory reallocation, then it is difficult to justify the presence of a real-time garbage collector because of the complexity it adds to the system, and the very high costs of certifying the correct behavior of the garbage collector. If the application does not reallocate memory, the entirety of garbage collection might be considered "dead code".
   b. If the real-time workload depends on an ability to reallocate memory, then the developer must "prove" that the application has made sufficient memory available to be reallocated, and has done so with sufficient lead time to allow the garbage collector to recycle the memory before the application demands to reallocate it.
2. One important attribute of the application that must be analyzed is the sizes and timings of each memory alloccation request. In a macro view, this can be represented as a simple ratio between bytes allocated per unit of wall-clock time. However, this macro view may hide potential fragmentation issues. With respect to memory fragmentation, allocating a 1,000-byte object every 10 ms is very different than allocating a single 100,000-byte object once per second.
   a. Understanding the allocation demand is comparatively easy. For any valid control flow through any of the workload's real-time tasks, if the flow includes an allocation operation, add the corresponding allocation to the cumulative allocation demand.
   b. Note, of course, that a tight bound on allocation demand requires significantly more effort than accumulating all of the allocation requests on all possible control paths through each real-time task. This is because many of the control paths may be mutually exclusive, but determining the conditions on which particular paths will be taken is generally very difficult.
3. The second important attribute that must be analyzed by the application developers is the sizes and timings of object releases. In a garbage collection environment, an object is considered to be released when no other non-released (live) object points to it. In a macro view, object releases can also be characterized as a simple ratio between bytes freed per unit of time. As with the allocation rate, this macro view may ignore certain memory fragmentation considerations. Releasing one 1,000-byte object every 10 ms may contribute to significantly more fragmentation than releasing a single 100,000-byte object once per second.
   a. Understanding the release of memory is much more difficult than understanding memory allocation. The Java assignment statement:

      pointer = null;

      indicates that the object previously referenced by the *pointer* variable is no longer referenced from *pointer*. However, in order to determine which object is referenced from *pointer*, and even to know what the type and size of that object is, we need to retrace the history of the *pointer* variable to determine the origin of its value. This sort of problem has been studied previously by computer scientists, and it is well known that the general solution is intractable. Thus, the problem cannot be solved by a static analysis system unless programmers restrict themselves to a style of programming that can be analyzed automatically.
   b. Even if it were known exactly which object is referenced by the *pointer* variable in the example statement above, removing *pointer*'s reference to the object does not necessarily make the object available to be reclaimed by the garbage collector. The garbage collector can only reclaim the object after all references to the object have

been cleared. Thus, the analysis of which objects are released at which time must also include a full aliasing analysis, to prove that all aliases referring to a particular object no longer refer to that object in order to assert that the garbage collector can reclaim a particular object at a particular time.

c. All of the analyses required to prove the timely release of objects are global in nature. In general, they span multiple methods, multiple class definitions, and multiple threads. The global nature of the analyses makes the analyses very difficult and very brittle. The analyses are brittle in the sense that once the analyses are completed, they are not robust to the many small incremental changes to the application that are typical of modern software maintenance activities. Changing a single line of code can have tremendous impact on the results of these analyses, totally invalidating any proofs of memory allocation behavior that might have been based on the original analyses.

For all of the reasons discussed above, Atego generally recommends against reliance on a garbage collector in hard real-time and safety critical code. Certainly, it is possible to build hard real-time software and even safety-critical software on top of a real-time garbage collector. But such a pursuit is mainly of academic interest. Either the programmers would be so restricted in their use of dynamic memory that they would not benefit from the full generality of automatic tracing garbage collection, or the certification and maintenance costs would be prohibitive.

7. Safe Stack Allocation for Hard Real-Time and Safety-Critical Systems

Even though traditional hard real-time systems have not made use of a dynamic memory allocation heap, they have generally made use of modern block structure to create local variables and structured data on each thread (or task) stack. A similar approach has been designed to allow Java programmers to specify through the use of certain programming annotations that certain objects are to be allocated on the run-time stack. The most well-known design of stack-allocated Java memory allocation system is found in the Real-Time Specification for Java [5]. The approach described below provides higher levels of abstraction. It is described more completely in [6, 7, 8]. Experience with this approach is reported in [9].

Programmers use annotations to associate attributes with the reference variables in a Java program. These annotations can be ignored when the code is compiled for deployment with a traditional Java virtual machine. However, when the code is deployed in a hard real-time or safety-critical environment, the annotations allow the safety-critical Java compiler to direct every allocation request to a specific region of memory. Regions are organized as a LIFO stack. And a special byte-code verification technique assures that no reference to a region-allocated object lives in a region that has a longer lifetime than the region that holds the object itself.

The following implementation of a complex number abstraction demonstrates the use of annotations to influence memory allocation behavior:

```
public class Complex {
   float real, imaginary;

   @ScopedThis public Complex(float r, float i) {
      real = r;
      imaginary = i;
   }

   @CallerAllocatedResult @ScopedPure
   public Complex multiply(Complex arg) {
      float r, i;
      r = this.real * arg.real - this.imaginary * arg.imaginary;
      i = this.real * arg.imaginary + arg.real * this.imaginary;
      return new Complex(r, i);
   }
}
```

In the above example, the constructor is annotated *@ScopedThis* to indicate that this constructor can be used to instantiate a *Complex* object within “scoped” memory. The verifier assures that, within this constructor, the value of the implicit *this* variable is never copied into an instance or static variable that might live longer than the scope-allocated *Complex* object itself.

The *@ScopedPure* annotation on the *multiply* method indicates that both the implicit *this* and explicit *arg* arguments may refer to objects residing in scoped memory, and the verifier assures that, within this method, neither is copied into an instance or static variable that might live longer than the scope-allocated objects they refer to.

The *@CallerAllocatedResult* annotation on the *multiply* method indicates that the object returned from this method shall be allocated in the scope indicated by the caller rather than within the method’s local scope. If it were allocated in the method’s local scope, the object would be reclaimed upon return from the method, and the reference value returned from the method would be considered a dangling pointer. Thus, the verifier would reject this implementation of the *multiply* method if the *@CallerAllocatedResult* annotation had been omitted from the method’s declaration.

A special byte-code verifier for hard real-time scopeallocated Java has been integrated within the Eclipse development environment, as illustrated in Figure 5. When programmers save their files, the Eclipse build system automatically invokes the verifier and any

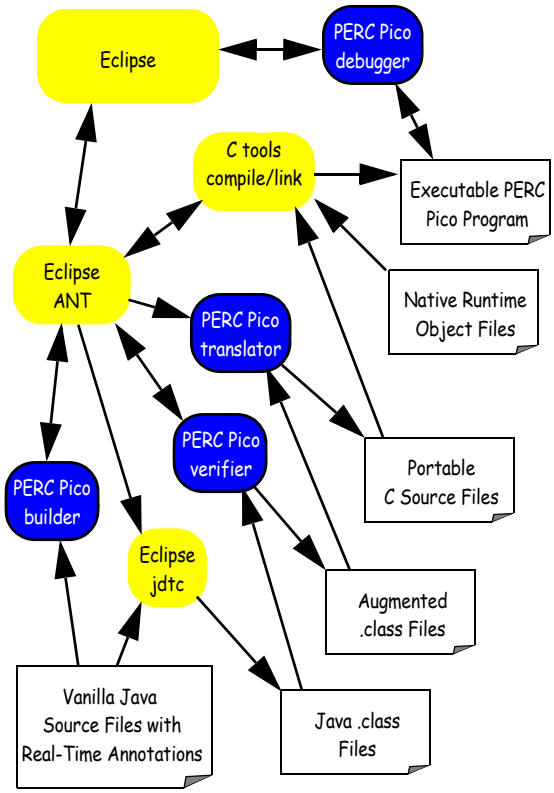

Figure 5. Hard Real-Time Java Development Environment

errors detected by the enhanced byte-code verifier are immediately displayed within the Eclipse edit window. This gives immediate feedback to developers of hard real-time Java code regarding possible violations of scope-oriented memory allocation protocols.

8. Summary

The full benefits of standard edition Java, including automatic garbage collection and traditional API libraries, are available to developers of soft real-time systems who choose to deploy their Java applications on a virtual machine that has been implemented to support real-time operation. The key enablers for this capability include real-time garbage collection, priority-based scheduling, priority-ordered thread queues and priority inheritance for synchronization.

Hard real time is characterized as systems that must be proven to satisfy all timing constraints. Tests and demonstrations of compliance with timing constraints do not generally constitute proofs unless the tests and demonstrations have been carefully analyzed to prove that they represent the worst-case execution scenarios. Java developers who need to satisfy hard real-time constraints or safety certification requirements are advised to avoid dependencies on tracing garbage collection because of the high costs of developing and maintaining proofs of compliance with specified constraints. Instead, such developers are advised to use a hard real-time version of Java which replaces automatic tracing garbage collection with safe stack allocation.

9. References

[1] K. Arnold, J. Gosling, D. Holmes. *The Java™ Programming Language, 4th edition*. 928 pages. Prentice Hall PTR. Aug, 2005.

[2] K. Nilsen. “Applying COTS Java Benefits to Mission-Critical Real-Time Software”, *Crosstalk The Journal of Defense Software Engineering*, pp. 19-24. June 2007.

[3] K. Nilsen. “Using Java for Reusable Embedded Real-Time Component Libraries”, *Crosstalk The Journal of Defense Software Engineering*, pp. 13-18. December 2004.

[4] “Six Design and Development Considerations for Embedded Multicore Systems”, Atego White Paper, July 21, 2009, www.atego.com.

[5] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, M. Turnbull, “The Real-Time Specification for Java”, Addison Wesley Longman, 195 pages, Jan. 15, 2000.

[6] K. Nilsen, “Applying Java to the Domain of Hard Real-Time Systems”, Conference Proceedings: Embedded Real-Time Software, Toulouse, France, May 2008.

[7] K. Nilsen. “Guidelines for Scalable Java Development of Real-Time Systems”, March 2006, available at http://research.aonix.com/jsc.

[8] “PERC Pico 1.1 User Manual”, April 19, 2008, available at http://research.aonix.com/jsc.

[9] M. Richard-Foy, T. Schoofs, E. Jenn, L. Gauthier, K. Nilsen. “Use of PERC Pico for Safety Critical Java”, Conference Proceedings: Embedded Real-Time Software and Systems, Toulouse, France, May 2010.