A Family of Real-time Java Benchmarks

Tomas Kalibera*, Jeff Hagelberg, Petr Maj, Filip Pizlo,
Ben Titzer, Jan Vitek

*Purdue University, West Lafayette, IN 47907, USA; *Charles University, Prague, 147 00, Czech Republic*

SUMMARY

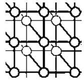
Java is becoming a viable platform for real-time computing. There are production and research real-time Java VMs, as well as applications in both military and civil sector. Technological advances and increased adoption of Real-time Java contrast significantly with the lack of benchmarks. Existing benchmarks are either synthetic micro-benchmarks, or proprietary, making it difficult to independently verify and repeat reported results. This paper presents the CD_x benchmark, a family of open source implementations of the same application that target different real-time virtual machines. CD_x is, at its core, a real-time benchmark with a single periodic task, which implements an idealized aircraft collision detection algorithm. The benchmark can be configured to use different sets of real-time features and comes with a number of workloads. It can be run on standard Java virtual machines, on real-time and Safety Critical Java virtual machine, and a C version is provided to compare with native performance.

1. Introduction

Driven by the popularity of Java, the availability of development tools, and wide library support, the Real-Time Specification for Java (RTSJ) [1] is on the rise. It is used in avionics [2], shipboard computing, industrial control [3] and music synthesis [4]. Real-time Java programs have different characteristics and requirements from traditional Java programs. While throughput remains important, it is predictability that is critical for real-time applications. Therefore many of the engineering tradeoffs that are an integral part of the design of a virtual machine have to be revisited to favor predictability over throughput. In order for virtual machine developers to understand the impact of design decisions, and for end users to select the technology that suits the requirements of a particular application, comprehensive and meaningful benchmarks are needed.

There are many Java benchmarks, ranging from synthetic micro-benchmarks to complex applications (e.g. SPECjvm, Dacapo [5], Java Grande, and SciMark). While these benchmarks can

*Correspondence to: Tomas Kalibera, Department of Software Engineering, Charles University, Malostranske nam. 25, 118 00 Praha 1, Czech Republic



be used to evaluate the quality of real-time virtual machines, they are not representative of real-time workloads. They aim to generate the highest sustainable load and they measure the mean performance under this load, neither of which suits real-time systems. Real-time systems are designed such that deadlines are not missed. The ability to meet deadlines depends on the worst-case computation times of periodically scheduled tasks, which are not captured by mean performance metrics. Moreover, programming styles for real-time systems are very different from non-realtime throughput targeted systems. The usefulness of benchmarks for performance evaluation is that the benchmarks, being realistic models of real applications, put the system of interest under a workload similar to those real applications. They allow us to capture and evaluate performance characteristics caused by many aspects of program execution, some of which we may not be aware of or be able to predict. For real-time Java, we thus need benchmarks that actually model real-time systems, have deadlines, use RTSJ, run on real-time OS kernels, use high precision timers, and measure workloads configured to never miss a deadline. Unfortunately, it is notoriously difficult to find real-time applications in the wild. Most real-time systems are proprietary and are tied to some hardware/OS platform. Real-time Java being a relatively young technology does not help.

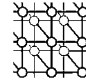
This paper presents the CD_x benchmark, a relatively small-sized (33KLOC), open source, application benchmark that can be targeted to different real-time platforms.[†] At its core, CD_x has a periodic thread that detects potential aircraft collisions, based on simulated radar frames. Other optional threads are used to generate simulated aircraft traffic, and create computational noise. The benchmark can thus be used to measure the time between releases of the periodic task as well as the time it takes to compute the collisions. This gives an indication of the quality of the virtual machine and the degree of predictability that can be expected from it. CD_x is configurable, it can be used with a standard Java virtual machine, an RTSJ virtual machine with scoped memory or with real-time garbage collection and a Safety Critical Java virtual machine. Finally a C implementation is provided to allow for performance comparison with native compiled code.

Another potential use of CD_x is linked to verification [18]. The increasing complexity of real-time systems is building up demand for automated verification tools. To develop these for Java, test cases are necessary. Similarly to benchmarking, plain Java programs are not enough, because they do not use the real-time API and are too complex for worst-case execution time (WCET) analysis. The real-time API introduces new behaviors and new error modes. In particular, these are the memory assignment errors in systems with scoped memory, but also incorrect sizing of scopes must be caught by these tools. CD_x can be easily used as test case for verification challenge problems. The RTSJ version can be a test for the detection of memory assignment errors. Any version can be used to test maximum allocation per release and the GC/RTGC version to test the RTGC overheads analysis. Bounds on WCET found by tools can then be verified against values measured with the benchmark. The plain Java version also makes it easier to get started with some types of analysis, gradually adding more and more RTSJ semantics to the verification, as current Java verification tools generally do not support RTSJ.

Earlier versions and modifications of the collision detector were used in [19, 20, 9, 21]. This work presents a first open-source version of the benchmark with improved instrumentation, several bug fixes, unification of a plain Java and RTSJ code, and a description of the application logic.

[†] The source code can be downloaded from http://www.ovmj.net/cdx/.

Copyright © 0000 John Wiley & Sons, Ltd.



2. Benchmarking Real-time Java Applications

Understanding the performance characteristics of a rich programming platform such as a Java virtual machine is a complex task, and even more when real-time constraints are added. It is thus highly unlikely that a single benchmark will ever provide all the information that is needed by developers. To start with, Java benchmarks are used for different purposes, including:

- understanding the performance of a particular feature or algorithm in the virtual machine, for instance, the cost of different implementation of locking or garbage collection,
- comparing the quality of virtual machines, for the purpose of selecting a vendor,
- evaluating the suitability of Java for a particular application and a particular deployment platform.

At the end of the day, a given benchmark is only meaningful if its workload is representative of applications that are relevant to end-users. This section establishes a list of features that should be covered by a real-time Java benchmark. It is not necessary for any benchmark to address all of these issues, different benchmarks addressing different subsets of the points listed here can be used to give a comprehensive picture of the quality of a real-time Java virtual machine.

- **Object-oriented features:** To be representative of idiomatic Java programs, object-oriented features of the language should be exercised. These include: inheritance, interfaces, virtual and interface dispatch.
- **Memory management:** Allocation and de-allocation of heap memory is a key feature of Java. This feature should be exercised with objects of different size classes. A real-time benchmark should allow developers to contrast the RTSJ's scoped memory management API with plain Java garbage collection, real-time garbage collection, and possibly traditional hand-coded object pooling.
- **Code size and complexity:** The size and complexity of the source code has an impact on performance in many different ways. Benchmarks should cover the range of program sizes and complexities, from micro-benchmarks that can easily be optimized by an ahead-of-time compiler, to more complex programs which are not as easy to optimize (e.g. for which the compiler can not de-virtualize all calls).
- **Multi-threading and synchronization:** A real-time benchmark should exercise the scheduler with multiple threads running at different priorities and with different release times. Synchronization and priority avoidance are key features of the RTSJ, benchmarks should exercise these features in a meaningful way with a mixture of contended and un-contended locking operations.
- **Other features:** A number of other important features in the language should be exercised, these include but are not restricted to: floating point operations, array accesses, exception handling, and reflection.
- **RTSJ API coverage:** A real-time Java benchmark should exercise the RTSJ APIs beyond the creation of threads and memory management. Some important features that should be exercised include: timers, asynchronous signals, and raw memory.
- **Predictability measurements:** Accuracy of release times and predictability of completion times are critical in real-time systems. A benchmark should have support for measuring

Copyright © 0000 John Wiley & Sons, Ltd.

the predictability of the virtual machine. This should be performed at different levels of computational load and with interference from unrelated low-priority tasks.
- **Start-up jitter measurements:** Some applications require low latency start-up times. A real-time benchmark should be set up so that it is possible to obtain a simple measurement of virtual machine start-up time.
- **Throughput measurements:** Predictability must also be correlated with throughput, as it is easy to trade one for the other. A benchmark should support some form of throughput measurement.
- **Self-checking:** The benchmark should include self-tests for correctness to ensure that results are only reported for correct runs of the benchmark.
- **Open source:** Free availability of source code for a benchmark, while not essential, is an enabler for wider adoption.
- **Portability:** A desirable feature is to be able to compare results across languages, operating systems, and hardware platforms.
- **Documentation:** The behavior, goals, and measurements performed by the benchmark should be clearly documented, so that end-users can understand which parts of the platform are exercised and the meaning of a particular result.

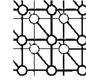
The importance of documentation should not be under-appreciated. Any result obtained from a benchmark can only be understood in the context of the operation performed by that benchmark. For instance, it is well known, though not properly documented, that the SPECjvm98 Jess benchmark is dominated by the cost of exception handling, and that SPECjbb spends most of its time acquiring and releasing uncontended fine-grained locks. If either of these operations is slow in one particular virtual machine, the performance results for that benchmark will appear to seriously lag behind competitors. The purpose of this paper is thus to ensure that users of CD_x understand what is being measured and what meaning to ascribe to results obtained by running it.

2.1. Qualitative Comparison of Benchmarks

We are aware of three other benchmarks that have been used to evaluate real-time and embedded Java programs. This section provides a qualitative comparison of these benchmarks based on publicly available documentation. Table I summarizes our impressions.

SPECjbbRT. SPECjbbRT [17] is based on the industry standard SPECjbb benchmark. The basic benchmark is written in an idiomatic object-oriented style and uses inheritance, interfaces and virtual dispatch liberally. Some standard Java collection classes are also used. The memory management policy is purely garbage-collected (both plain and real-time) with high-allocation rates that cause the GC to run regularly. The code base is medium sized, several thousand lines, of reasonable complexity. The benchmark can be configured to run with multiple threads and it employs standard Java synchronized statements to protect shared objects at a rather fine-grained level (there are roughly 125 synchronized blocks in the benchmark which are called often). The RTSJ API coverage is minimal, the main change from the original benchmark is the addition of real-time threads. The benchmark does not measure the predictability of releases, but rather the jitter in completion times. This is mostly useful to estimate execution time hazard introduced by the GC and JIT. The benchmark is thus more suitable for evaluation of VMs for soft real-time Java systems than for hard real-time RTSJ applications. Throughput measurement can be obtained with the number of transactions completed. There is no

Copyright © 0000 John Wiley & Sons, Ltd.



Features	SPECjbbRT	JBEmbedded	Suramadu	CD_x
Object-oriented	Yes	No	No	Yes
Memory management	GC/RTGC	–	Scopes/GC/RTGC	Scopes/GC/RTGC/Man
Code size	Medium	Small	Small	Small
Multi-threading	Yes	No	Yes	Yes
Synchronization	Sync	–	Sync/WF	WF
Other	–	–	Int/FP	FP/Array
RTSJ coverage	Small	–	High	Small
Predictability	Completion	–	Release/Completion	Release/Completion
Throughput	Yes	Yes	Yes	Yes
Self-checking	No	No	No	No
Open source	No	Yes	Yes	Yes
Portability	RTSJ	Java	RTSJ	RTSJ/Java/C
Documentation	Yes	Yes	Yes	Yes

Table I. Qualitative comparison.

meaningful self-checking, we have experimented with removing all synchronization from the original SPECjbb and have never seen a failing run on a 8-core desktop. Also, the benchmark is not open-source. To this day, it has neither been adopted as a SPEC benchmark by the SPEC Corporation, nor otherwise been made available. Portability is limited to Java and environments that support GC and have sufficient memory. Some documentation is available. CD_x complements **SPECjbbRT** in that it also exercises scoped memory and supports manual memory allocation (the C version), it is available for multiple platforms, it models a real-time application, it allows to measure predictability of releases and to detect deadline misses, it exercises floating point unit and arrays, and it is publicly available and open-source. On the other hand, **SPECjbbRT** has larger code base and exercises synchronization and Java collection classes to a larger degree.

JavaBenchEmbedded. This benchmark suite is made up of a series of micro-benchmarks and kernels that can be deployed on small embedded devices. The benchmarks do not use object-oriented features, there is no inheritance, no interfaces, and minimal use of virtual dispatching. The micro-benchmarks attempt to measure latencies of individual byte-code instructions, which only makes sense on non-compiling VMs. On a compiling VM, compiler optimizations can arbitrarily distort the results, and thus the measured values do not represent durations of individual instructions. The benchmarks do not allocate substantial amounts of memory and thus do not exercise the memory management subsystem. The code base is small and of limited complexity. The benchmark suite is single threaded. The RTSJ APIs are not invoked. Measurements are limited to throughput. Portability is limited to Java. The benchmark is open source and some documentation is available.

Suramadu. The open-source Suramadu benchmark suite [16] includes benchmarks that focus on low-level measurement of jitter, throughput, and latency of various RTSJ operations. The original suite also probably included one computational kernel throughput benchmark, but the core part of the code is missing in the open-source release. The micro-benchmarks test individual features of the RTSJ for performance and predictability. The benchmarks do not rely on object-oriented features or libraries. The benchmarks test allocation in scoped memory or garbage collected memory. The suite measures context switch latency, class loading costs, asynchronous event latency, cost overrun,

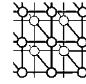
Copyright © 0000 John Wiley & Sons, Ltd.

interrupt latency, JNI overhead, priority inheritance latency, synchronization latency, wait free queues, floating point, integer, and shift operations. The code size is small and the complexity is minimal. The benchmark is not self-checking. The suite is open source. Its portability is limited to RTSJ. Documentation is available. The strongest weakness of the suite is that it only contains simple synthetic micro-benchmarks. Particularly the throughput micro-benchmarks are of little use, as they repeatedly measure an arbitrary hard-coded sequence of Math operations (one for floating point, one for integer operations, and one for shifting). The isolated execution of these sequences cannot reveal throughput performance of real applications, which include a mix of different types of instructions and indeed different sequences. Moreover, the instruction sequences heavily use constants, leaving most work to the compiler in the compiling VMs. Representative throughput performance measurements can only be obtained by application benchmarks, such as CD_x. **Suramadu** exercises the memory management using a simple sequence of allocation requests. CD_x includes a realistic allocation sequence in the application logic and a synthetic, yet more configurable, allocation sequence in its noise generators. The **Suramadu** micro-benchmarks that measure RTSJ related latencies in synthetic workloads can provide useful results for worst-case execution time estimates. **Suramadu** allows to measure predictability of releases and completions again using a trivial synthetic workload. CD_x can measure these using more complex application workload, allowing to take into account additional aspects of the VM, such as garbage collection pauses. The advantage of CD_x is also that it has a plain Java and C version.

$\mathbf{CD_x}$. This is an application level benchmark. It has been written in an object-oriented style using inheritance, interfaces, virtual dispatching and some collection classes from the standard library. Different versions of the benchmark allocate memory in scopes, on the heap, and even with malloc/free. The total code size is medium sized with a reasonable complexity, the part that exercises real-time capabilities of Java is relatively small (about 5 thousands LOC). The benchmark is multi-threaded and uses wait-free queues for communication. The benchmark uses arrays and floating point operations extensively. It is set-up to allow measurement of both release time and completion time of the main periodic thread. Additional computational noise can be configured in non-realtime threads. Additional allocation noise can be configured in the real-time threads. Throughput can be calculated from the measured completion times, i.e. as a sum or average. The output of the application logic of the benchmark is deterministic, it is thus possible to make the benchmark self-checking, though this remains to be done. The benchmark is open source and has been ported to plain Java, RTSJ, SCJ and C code. It has been run on Linux, OS/X and RTEMS.

Overall, CD_x covers a combination of features that was missing in previous work. While micro-benchmarks are well suited to stress test individual features in isolation, they do not provide a workload that it representative of real-world application and can magnify differences that do not show up in deployed systems. CD_x complements them by providing a larger, application, benchmark. SPECjbbRT is comparable in size, and exercises threading and synchronization but is mostly a throughput benchmark. Finally, it is the only benchmark that allows comparison across both different variants of Java (Java, RTSJ, RTSJ+RTGC, SCJ), different operating systems (Linux, RTEMS...) and even different programming languages (C / Java).

Copyright © 0000 John Wiley & Sons, Ltd.



3. $\mathsf{CD_x}$ Benchmark Algorithmic Description

The collision detector application was designed by Titzer and Vitek as a software engineering project. CD_x is based on an implementation of this project done by Hagelberg and Pizlo at Purdue in 2001. The benchmark was modified many times over the years. Its key components are an *air traffic simulator* (ATS), which generates radar frames based on user-defined air traffic configurations, and a *collision detector* (CD), which detects potential aircraft collisions. The program was designed so that collision detection and air traffic simulation could be performed independently. Indeed, in the original design the CD was a real-time thread while ATS was a plain Java thread and communication between the two was performed by a non-blocking queue. In that design we relied on the simulator to create computational noise and to occasionally trigger garbage collection. The version of the benchmark presented here can also use an external program to create computational noise and pre-generate all the radar frames needed for a run of the CD_x.

3.1. Air Traffic Simulator

The ATS generates radar frames with aircraft positions, based on a user-defined configuration. A *radar frame* is a list of aircraft and their current positions. An *aircraft* is identified by its call sign (a string). A *position* is a three dimensional floating point vector in the Cartesian coordinate system. The simulation runs for t_{max} seconds. Radar frames are generated periodically, providing a user-defined number of radar frames per second (fps) and number of frames in total (frames). Thus, t_{max} is frames/fps. The set of aircraft does not change during the simulation (i.e. none of the aircraft lands, takes-off, crashes, or otherwise enters or leaves the covered area). With respect to detected collisions, the semantics can be optimistically explained such that the pilots always avoid the collision in the end. The ATS is configured by a textual file, where each line describes a single aircraft. Each line contains the call sign of the aircraft and three columns with expressions giving the aircraft coordinates x, y, z as functions of time. The expressions thus use "t" as a variable and then common mathematical operations: arithmetics with brackets, trigonometric functions, logarithms, absolute value, etc. Coordinates can also be constants, i.e. aircraft can fly at constant altitude.

3.2. Collision Detector

The CD detects a collision whenever the distance between any two aircraft is smaller than a pre-defined *proximity radius*. The distance is measured from a single point representing an aircraft location. As aircraft location is only known at times when the radar frames are generated, it has to be approximated for the times in between. The approximated trajectory is the shortest path between the known locations. Another simplification is that constant speed of aircraft is assumed between the two consecutive radar frames. For these assumptions to be realistic, the frequency of the radar frames should be high (we typically run the benchmark at 100 HZ). To allow such high frequency, the detection has to be fast. This is achieved by splitting it into two steps. First, the set of all aircraft is reduced into multiple smaller sets of aircraft that have to be checked for collision (*reduction*). This step allows CD to quickly rule out collisions of aircraft that are very far from each other. Second, for each of the identified sets, every two aircraft are checked for collisions (*collision checking*). This step would functionally be sufficient, as it could be run on the set of all aircraft seen by the radar, but the computation would take too long.

Copyright © 0000 John Wiley & Sons, Ltd.

Notation

Position in previous frame	$\vec{i} = (x_i, y_i)$	Proximity radius (constant)	r
Position in current frame	$\vec{f} = (x_f, y_f)$	Size of grid element (constant)	s
Lower-left end of grid element	$\vec{e} = (x_e, y_e)$		

Input $\vec{i} = (x_i, y_i)$, $\vec{f} = (x_f, y_f)$, $\vec{e} = (x_e, y_e)$, s, r

Output TRUE if line segment $(\vec{i}, \vec{f})$ might have an intersection with square (x_l, y_l, x_h, y_h), $(x_l, y_l, x_h, y_h) = (x_e - r/2, y_e - r/2, x_e + s + r/2, y_e + s + r/2)$, FALSE if impossible.

Step 1. Assume $x_f \neq x_i$ and $y_f \neq y_i$. We transform the coordinates such that the line segment is a line from $(0,0)$ to $(1,1)$:

$$x^t \quad \mapsto \tfrac{x - x_i}{x_f - x_i} \qquad\qquad y^t \quad \mapsto \frac{y - y_i}{y_f - y_i}$$

By this transformation we get

$$\vec{i^t} \quad = (x_i^t, y_i^t) = (0,0) \qquad \vec{f^t} \quad = (x_f^t, y_f^t) = (1,1)$$

$$x_l^t = \frac{x_e - r/2 - x_i}{x_f - x_i} \quad x_h^t = \frac{x_e + s + r/2 - x_i}{x_f - x_i} \qquad y_l^t = \frac{y_e - r/2 - y_i}{y_f - y_i} \quad y_h^t = \frac{y_e + s + r/2 - y_i}{y_f - y_i}$$

Step 2. Now the problem is reduced to the detection of intersection of rectangle $(x_l^t, y_l^t, x_h^t, y_h^t)$ with line segment (0,0,1,1). Assume $x_l^t \leq x_h^t$, $y_l^t \leq y_h^t$, if any of the following conditions hold, there is no intersection:

$$max(x_l^t, x_h^t) \quad < 0, \quad min(x_l^t, x_h^t) \quad > 1, \tag{1}$$

$$max(y_l^t, y_h^t) \quad < 0, \quad min(y_l^t, y_h^t) \quad > 1 \tag{2}$$

Step 3. Otherwise, assuming at least one corner of the rectangle is within the square, the rectangle intersects the line segment iff it intersects line $y = x$. There is such an intersection, if any of the following holds:

1. LL corner is above the line and HR is below the line: $x_l^t \leq y_l^t \wedge y_h^t \leq x_h^t$ (Figure 2(a))
2. LL corner is below the line and HR is high enough: $y_l^t \leq x_l^t \wedge y_h^t \geq x_l^t$ (Figure 2(b))
3. HR corner is above the line and LL is low enough: $x_h^t \leq y_h^t \wedge y_l^t \leq x_h^t$ (Figure 2(c))

Note that all relative positions of the rectangle corners and the line are covered:

	HR below	HR above
LL above	1	3
LL below	2	2,3

Note that if no corner of the rectangle is within the square, we may report an intersection with the line segment even if there is in fact only intersection with line $y = x$. This is later detected by the checker.
It remains to be shown how to handle the case when $x_f = x_i$ or $y_f = y_i$. We perform Step 1 only for coordinates that allow it. Then, we modify Step 2. For $x_f = x_i$, we replace conditions 1 by 3. For $y_f = y_i$, we replace conditions 2 by 4:

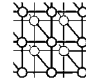
$$x_i \quad < x_e - r/2, \quad x_i \quad > x_e + s + r/2, \tag{3}$$

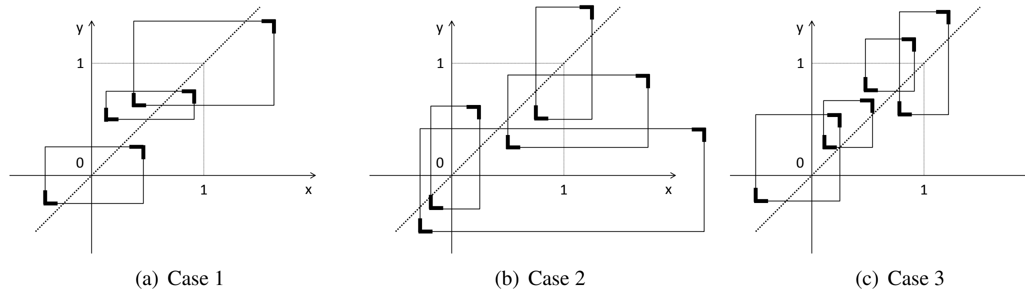
$$y_i \quad < y_e - r/2, \quad y_i \quad > y_e + s + r/2 \tag{4}$$

If all the conditions hold, we know there is intersection (TRUE). We do not perform Step 3: if any condition does not hold, there is no intersection (FALSE).

Figure 1. Reducer algorithm.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls





(a) Case 1 (b) Case 2 (c) Case 3

Figure 2. Rectangle positions for reducer algorithm.

Both the reduction and the checking operate on motions. A *motion* is a pair of 3-d vectors describing the initial position, $\vec{i}$, and the final position, $\vec{f}$, of an aircraft ($\vec{i}$ is from the previous frame, $\vec{f}$ is from the current frame). The frame also contains the call sign of the aircraft, which identifies the aircraft. A *motion vector* $\vec{m}$ is then defined as $\vec{m} = \vec{f} - \vec{i}$.

Reduction. Reduction is already collision detection, but of much less precise form than the one performed during collision checking. The 3-d detection space is reduced to 2-d and the conditions for detecting a collision are relaxed. These two simplifications are designed such that all collisions are still detected, but some of the collisions detected may not be really collisions in the 3-d space (false positives). The advantage is reduced complexity. The reduced 2-d space is created from the original 3-d space simply by ignoring the altitude (the z coordinate). The 2-d space is divided into a grid; a collision is detected whenever two aircraft span the same grid element. For each grid element with a collision, the reducer then outputs the set of aircraft that spanned the element. Each of these sets is then checked by collision checker to filter out false positives. The reducer maintains a mapping from a grid element to a set of motions that span the element. The reducer proceeds as follows. Starting with an empty mapping, it keeps adding motions to the map:

```
void mapGridElementToMotion(gridElement, motion, mapping) {
  if (motion.spansElement(gridElement) && !mapping(gridElement).contains(motion)) {
    mapping.put(gridElement, motion);
    foreach( e in gridElement.adjacent()) mapGridElementToMotion(e, motion, mapping);
  }
}
```

The code above could be improved to avoid checking of some grid elements and redundant checking of some of grid boundaries using algorithms common in the ray tracing domain or simply with the Bresenham's line drawing algorithm [7]. It should be easy to plug an implementation of a better algorithm into the benchmark. The key test in the procedure is spansElement. It checks whether a particular motion spans a given grid element, which is extended by half of the proximity radius at each side. The test is implemented as a geometric test for intersection of a line segment and a square. To keep the memory requirements reasonable, and in particular independent on the dimensions of the 2-d

Copyright © 0000 John Wiley & Sons, Ltd.

Notation

Position of aircraft n in previous frame	$\vec{i_n}$	Position of aircraft n at time t	$\vec{p_n}(t)$
Position of aircraft n in current frame	$\vec{f_n}$	Euclidean distance of points	$d(\vec{p_1},\vec{p_2})$
Proximity radius (constant)	r	Dot product of vectors $\vec{a}$,$\vec{b}$	$\vec{a}\cdot\vec{b}$

Input $\vec{i_1}$, $\vec{i_2}$, r
Output TRUE if $\exists t, d\left(\vec{p_1}(t),\vec{p_2}(t)\right) <= r$, FALSE otherwise.

Step 1. We are first looking for time t, such that $d\left(\vec{p_1}(t),\vec{p_2}(t)\right) = r$. By the properties of dot product and distance

$$d\left(\vec{p_1}(t),\vec{p_2}(t)\right) = \sqrt{\left(\vec{p_1}(t)-\vec{p_2}(t)\right)\cdot\left(\vec{p_1}(t)-\vec{p_2}(t)\right)} \tag{5}$$

To express the distance of the two points in (5), we define $\vec{v_1} = \vec{f_1} - \vec{i_1}$, $\vec{v_2} = \vec{f_2} - \vec{i_2}$. It follows that

$$\vec{p_1}(t) = \vec{i_1} + t\vec{v_1} \qquad \vec{p_2}(t) = \vec{i_2} + t\vec{v_2}$$

are the positions of the aircraft between the two radar frames for $0 \le t \le 1$. Then, $\vec{p_1} - \vec{p_2} = \left(\vec{i_1} - \vec{i_2}\right) + t\left(\vec{v_1} - \vec{v_2}\right)$. From the properties of dot product (we write $\vec{p_\bullet}$ instead of $\vec{p_\bullet}(t)$):

$$\left(\vec{p_1}-\vec{p_2}\right)\cdot\left(\vec{p_1}-\vec{p_2}\right) = t^2\left(\vec{v_1}-\vec{v_2}\right)\cdot\left(\vec{v_1}-\vec{v_2}\right) + 2t\left(\vec{i_1}-\vec{i_2}\right)\cdot\left(\vec{v_1}-\vec{v_2}\right) + \left(\vec{i_1}-\vec{i_2}\right)\cdot\left(\vec{i_1}-\vec{i_2}\right)$$

By combining with (5) we get an equation for variable t.

Step 2. If the equation is not quadratic, $(\left(\vec{v_1}-\vec{v_2}\right)\cdot\left(\vec{v_1}-\vec{v_2}\right) = 0)$, we have that $\vec{v_1} = \vec{v_2}$. This corresponds to the situation when the aircraft are moving in parallel and at the same speed. This means that their distance is constant. We thus return TRUE if $d\left(\vec{i_1},\vec{i_2}\right) \le r$, FALSE otherwise.

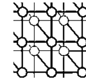
Otherwise we have a quadratic equation. If the equation has no solution, aircraft are far and we return FALSE. If the equation has only one solution (t_0), the aircraft are moving in parallel at different speeds (one of them may not be moving at all). The minimum distance they could have (for any t, not only $0 \le t \le 1$) must be r. Otherwise, there would have been two solutions. This means that the points got to the distance r for $0 \le t \le 1$ (within the line segments) iff $0 \le t_0 \le 1$. So we return TRUE if $0 \le t_0 \le 1$, FALSE otherwise. If the equation has two solutions ($t_1 < t_2$), the aircraft may or may not be moving in parallel. In both cases, however, there is an intersection at time $\frac{(t_1+t_2)}{2}$. For $t < t_1$ and $t > t_2$, the aircraft are farther from each other than r. For $t_1 < t < t_2$, the aircraft are closer than r (in a collision). So, we can rule out a collision (return FALSE) if $max(t_1, t_2) < 0$ or $min(t_1, t_2) > 1$ (the aircraft would collide only outside the studied segments). Otherwise, we know there is a collision and we return TRUE. Note, that based on t_1 and t_2, we can also calculate the location of the collision.

Figure 3. Collision detector algorithm.

detection space, the mapping of grid elements to aircraft that span it is implemented using a hash table, rather than a two-dimensional array. The reducer algorithm is described Figure 1.

Collision Checking. Collision checking is a full 3-d collision detection. The checker detects collisions of all pairs of aircraft belonging to each set identified by the reducer. The algorithm is based on checking the distance of two points (centers of the aircraft) traveling in time. If these points ever get closer than the proximity radius, a collision is detected. The test assumes that the speed of each of the aircraft is constant between two consecutive radar frames and that the aircraft trajectories are line segments. The calculations involved in the algorithm are described in Figure 3.

Copyright © 0000 John Wiley & Sons, Ltd.



3.3. Interaction between the ATS and the CD

The ATS, which is a non-realtime task, needs to transfer the generated frames to the CD, which is a real-time task. This is done through a frame buffer of fixed size, implemented as a wait-free queue. The simulator copies frames to the buffer, where the detector can read them. The CD is a periodic task. When released, it reads the next frame from the buffer. If a frame is available, it runs the detection algorithm, otherwise it does nothing. Three modes of interaction between the ATS and the CD are supported: pre-simulation, concurrent simulation, and synchronous simulation. With *pre-simulation*, the simulator first generates all frames and stores them in the buffer, which is set large enough to hold them all. This simplifies the analysis by avoiding any dependencies of the detector on the simulator. In *concurrent simulation*, the simulator runs concurrently with the detector, adding some background noise to the system and reducing memory requirements of the frame buffer. The speed of the simulator has to be configured carefully: if the simulator is too fast, frames may not fit into the buffer and be dropped. If it is too slow, frames will not be ready when required by detector. The speed of the simulator is controlled by command line arguments. In *synchronous simulation*, the detector waits for the simulator to generate a frame, as well as the simulator waits for the detector to finish processing the previous frame. This mode is intended only for debugging. The ATS can also store the generated air traffic into a binary file for later use. The benchmark can then run with a simplified version of the simulator that only reads data from this binary file, storing them into the buffer before CD starts. As a step towards benchmarking on embedded systems with further reduced resources, the binary dump of the air traffic can also be converted into Java source code. Thus, we can generate a simulator for a particular workload and use it on systems where file I/O is not available, or for program analysis with tools that would be confused with the I/O (such as a model checker). The binary dump of the air traffic can also be converted into a CSV file for further analysis with statistical software.

4. Benchmark Implementation

The CD_x benchmark is configurable to support different runtime environments and programming languages. The benchmark comes in three major versions, listed in the table below. CD_j is the Java version, it can be linked against the RTSJ APIs or against a placebo library to allow for execution on a standard Java virtual machine, CD_s is a version of the benchmark written against the upcoming Safety Critical Java specification, and CD_c is an idiomatic ANSI-C version of the benchmark.

CD_j	For both Java and RTSJ.
CD_s	For the Safety Critical Java version.
CD_c	For the ANSI C equivalent.

The benchmark can be run with pre-generated data, or can simulate frames online. Online simulation is only available in CD_j and can be done (i) concurrently, in a low-priority thread, (ii) synchronously with the main detector thread, or (iii) before the main detector thread is started. There is a version of CD_j for each of these configurations. CD_s and CD_c only support pre-generated data, CD_c reads the data from a binary file, while CD_s uses compiled-in data. The data can be generated and stored by CD_j, either to a binary file or to a Java source file.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

PDSK	All frames are pre-simulated offline and stored on disk.
PBIN	All frames are pre-simulated offline and linked into the binary.
PMEM	All frames are pre-simulated online and stored into memory.
SSYN	Frames are simulated on-line, synchronously with main detector thread.
SCON	Frames are simulated on-line, concurrently with main detector thread.

The memory management options for CD_j are scoped memory or garbage collection, be it plain Java garbage collection or real-time garbage collection. CD_s can only use scoped memory as it is the only option supported by Safety Critical Java. CD_c uses malloc() and free() for manual memory management.

SCP	Scoped memory is used for all dynamically allocated data.
GC/RTGC	Standard/real-time garbage collection is used to reclaim memory.
MAN	Manual memory management using malloc() and free().

Lastly, computational noise can be configured in the CD_j benchmark by adding a plain Java thread that runs an unrelated program for the purpose of stressing the virtual machine. Also, simple synthetic allocation noise can be added to the main detector thread of CD_j. The algorithm of this noise generator is described in the next section.

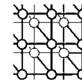
CNG	Computational "noise" is generated by a non-realtime thread.
ANG	Allocation "noise" is generated in the main detector thread.

The plain Java version of CD_x is obtained through wrapper functions that provide plain Java implementations of the requested RTSJ functionality. While the dependency of the benchmark code on the RTSJ library can be removed by the wrappers and the benchmark indeed run by a plain Java virtual machine, the impact of RTSJ memory semantics on the architecture could not be abstracted out. The use of scopes and immortal memory by itself requires additional threads in the application. Also, memory assignment rules sometimes lead to the need of copying arguments passed between memory areas (i.e. heap to scope, inner scope to outer scope). Even more, we also structured the code to make it is easier for programmers to keep track of which objects live in which memory areas. Thus, the architecture is representative of an RTSJ application. The plain Java version of the benchmark can be both compiled and run with standard Java. The RTSJ Java libraries and an RTSJ VM are only needed to build and run the RTSJ version of the benchmark with immortal memory, scopes or GC/RTGC. The RTSJ code has been tested with Sun's Java Real-Time System (RTS), IBM's WebSphere Real-Time (WRT), Fiji VM, and Ovm. The Safety Critical Java version has been run with Ovm.

4.1. Noise Generators

The CD component of CD_x is by itself quite efficient in memory usage. To allow scaling the GC work generated by the detector better, we added an optional synthetic allocation noise generator which can run within the main collision detector thread. The generator has an array of references (root array), which is initialized to null references at start-up. The array implements a write-only cyclic buffer. Pointers to newly allocated objects are stored to the array, overwriting the oldest ones. During each release of the detector, a constant number of objects is allocated:

Copyright © 0000 John Wiley & Sons, Ltd.



```
for (i=0; i< objectsPerRelease; i++)
    rootArray[rootPointer++ % rootArray.length] = new byte[size];
```

This simple algorithm allows the tester to tune the allocation rate by tuning size, and the amount of reclaimed objects by tuning the size of the root array. On the other hand, the re-use of objects of constant size can be very easy for a garbage collector, adding relatively small amount of GC work per computation – the computation time could easily be the bottleneck with such a noise generator. We thus add an option to vary the object size: there is a minimum and maximum object size and a step by which the object size is increased after each allocation:

```
int sizeIncrement = 0, maxSizeIncrement = maxSize − minSize;
for (i=0; i<objectsPerRelease; i++) {
   rootArray[(rootPointer++) % rootArray.length] = new byte[minSize + sizeIncrement % maxSizeIncrement];
   sizeIncrement += allocSizeStep;
}
```

In order to provide a more realistic source of allocation noise and/or some background computational noise, we support the execution of an external benchmark in low-priority threads. The external benchmark is run using Java reflection, thus it needs not be available at build time and the code base is completely independent. For our experiments, we used SPECjvm98 javac benchmark for its non-trivial memory use that includes fragmentation. Indeed, although the allocation noise of this background benchmark is far more realistic than that of the noise generator, it is still not representative of a real-time system.

4.2. Using Scoped Memory Areas

In the SCP configuration of CD_j, the ATS runs in the heap, the frame buffer is allocated in immortal memory, and the CD is allocated in scoped memory. The SCP configuration of CD_s uses the same memory areas except for ATS, which it does not have. We use two scoped areas, the first is for persistent detector data (stored locations of aircraft) which we call the *persistent scope*, and the second is a nested scope used as a scratch pad for each iteration of the algorithm, which we call the *transient scope*. The persistent scope is entered once before the first detector release and left when the benchmark exits. The transient scope is re-entered for every frame. To assist in keeping track of where objects are allocated, we reflect their allocation context in the package structure of the code following the approach described in [9]. Thus, there are packages named heap, immortal, immortal.persistentScope, and immortal.persistentScope.transientScope. It is correct to pass references to sub-packages, but data have to be copied when they have to be passed to parent packages. There are two exceptions to the rule for placement of classes into packages: entry threads and parameter copying. Each of the non-heap areas is entered through its singleton *entry thread* object. An entry thread is sometimes a multi-area object, which means that some methods, such as the constructor, execute in a different area from the other. Still, we always place an entry thread into the package of the scope that is being entered. In order to copy parameters to a memory area, we again use a multi-area object, because code that does the allocation of the target buffer for the copy needs to run in the target memory area, while the code that does the actual copy has to run in the source memory area. An example is storing a transient motion vector into persistent state. This is handled by immortal.persistentScope.StateTable.put() method which runs in the transient detector scope, but the StateTable lives in the persistent scope.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

Table II. Complexity of simulator and detector.

	WMC			DIT			NOC			CBO		
	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum
All code	5	95	2109	2	5	564	0	28	128	5	77	2004
No libraries	5	46	1140	2	5	379	0	28	77	5	77	1258
Detect. only	**4**	**27**	**217**	**1**	**5**	**69**	**0**	**2**	**11**	**4**	**23**	**243**
Simul. only	5	46	914	3	5	305	0	28	66	6	77	1002
Libraries only	6	95	969	2	4	185	0	8	51	5	48	746

	RFC			LCOM			LOC		
	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum
All code	11	117	4294	1	4465	12969	53	6023	50848
No libraries	10	94	2566	1	979	5392	57	6023	33761
Detect. only	**7**	**53**	**498**	**0**	**351**	**563**	**46**	**652**	**5281**
Simul. only	11	94	2003	3	979	4818	59	6023	27595
Libraries only	12	117	1728	0	4465	7577	49	4151	17087

Table III. Complexity of detector with pre-simulated radar frames.

	WMC			DIT			NOC			CBO		
	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum
All code	4	95	676	1	5	121	0	2	18	4	39	475
Detect. only	**4**	**27**	**217**	**1**	**5**	**69**	**0**	**2**	**11**	**4**	**23**	**241**
Libraries only	14	95	417	1	4	43	0	2	7	6	39	213

	RFC			LCOM			LOC		
	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum
All code	10	117	1338	1	4465	7417	72	4151	15769
Detect. only	**7**	**53**	**498**	**0**	**351**	**563**	**46**	**652**	**5281**
Libraries only	16	117	694	28	4465	6646	25	4151	8819

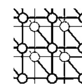
Table IV. Source lines of code of CDx.

With simulator			Pre-simulated		
	All code	37875		All code	20127
	No libraries	17741		**Detect. only**	**2761**
	Detect. only	**2761**		Libraries only	16578
	Simul. only	14633			
	Libraries only	20134			

4.3. Code Metrics

Complexity metrics can provide an objective and compact characterization of the source code of a benchmark. In our case, we are interested in more than just the code size. We also want to know whether the benchmark uses object-oriented features in a sophisticated or trivial way. For this, we measure the complexity of the benchmark code with the Chidamber and Kemerer object-oriented programming (CK) metrics [10], with the ckjm tool [11], which has also been used to evaluate the DaCapo benchmarks [5]. We apply the CK metrics to the classes that the application loads. Each metric is defined for a class. Weighted methods per class (**WMC**) is the number of methods in a class. Depth of inheritance tree (**DIT**) is the number of ancestor classes of a class. Number of children (**NOC**) is the number of direct subclasses of a class. Coupling between object classes (**CBO**) is the number

Copyright © 0000 John Wiley & Sons, Ltd.



of classes coupled to a class. Two classes are coupled if one of them uses the methods or fields of the other class. This includes inheritance, method arguments, method types, and exceptions. Response for a class (**RFC**) the number of methods that can be called, recursively, from any method of the class. The tool uses an approximation, where it only counts the number of methods of the class plus the number of directly called external methods. Lack of cohesion (**LCOM**) quantifies the number of methods of a class that do not share instance variables. The resulting number is calculated as $|P| - |Q|$, where P is the set of all method pairs that do not share any variable, and Q is the set of all pairs that do. Finally, lines of code (**LOC**) is the normalized number of lines of code of a class. While not a Chidamber and Kemerer metric, it has a straightforward interpretation. The ckjm tool counts the number of lines of normalized source code that could be generated from a byte-code representation of the class: it sums up the number of methods, number of fields, and number of byte-code instructions of methods.

The results are shown in Table II for CD_x configurations with both full simulator and detector (SSYN,SCON), and in Table III for configurations that use pre-simulated radar frames (PDSK or PBIN excluding the frames themselves). We summarize each metric using median, maximum, and total. The median and maximum seem more natural to most metrics and were used in [10], while the total has been used in [5] and is natural for LOC and WMC. The libraries used by the program are mostly collection classes from java.util. The source code includes implementation of these classes taken from GNU Classpath, so that one can reduce the impact of class libraries on performance when comparing different virtual machines. The tables show the complexity of these selected collection classes separately from the complexity of the rest of the program. In Table IV we also provide the raw source lines of code (SLOC) summarized over all source input files from a particular version. This metric is influenced by the particular formatting of the source code we use. We use the same tool and metrics as in the DaCapo benchmarks [5], which allows us to compare CD_x to non-realtime application benchmarks: SPECjvm98, DaCapo version beta-2006-08, and pseudojbb. With standard libraries excluded, the CD_x version with simulator (Table II) is more complex than SPECjvm98 benchmarks and pseudojbb (only javac has higher RFC and jack has higher LCOM). It is more complex than luindex and lusearch benchmarks from DaCapo, and sometimes it beats another of DaCapo benchmarks in some metric of complexity. But it is significantly simpler than eclipse, the most complex DaCapo benchmark.

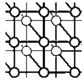
4.4. Workload Characterization

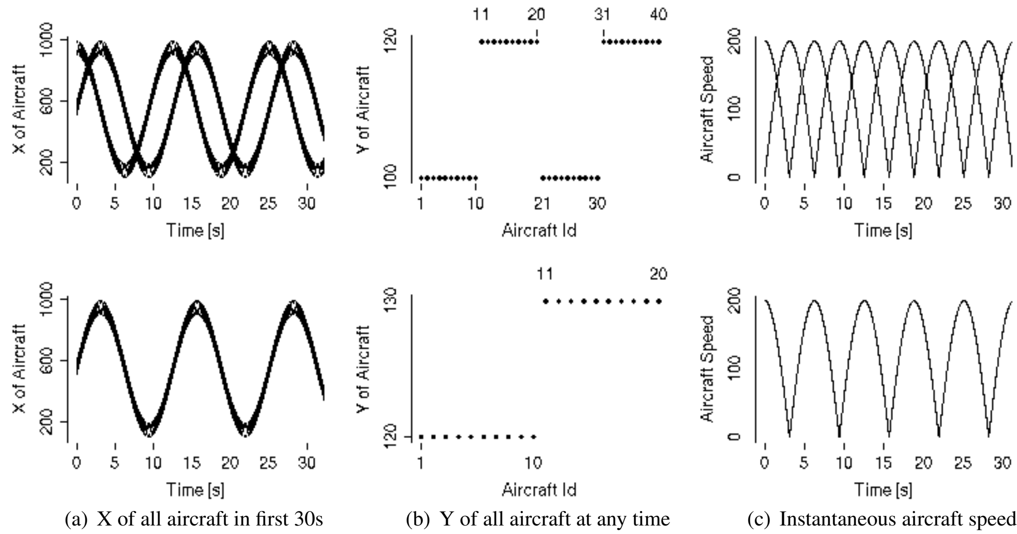
The CD_x workload is highly configurable. We describe two pre-configured workloads, named NOI and COL. Other workloads are used when necessary. The basic parameters of the two workloads are summarized in Figure 5. The air traffic configuration of NOI and COL workloads was selected to be intuitive and stress the system enough – have enough collisions (COL) or enough artificial noise (NOI). It is by no means a realistic air traffic. All aircraft fly at the same altitude at all times. The y coordinate of each aircraft is constant, but different aircraft sometimes have it set differently (Figure 4(b)), such that they could never collide with each other. Only the x coordinate changes in time (Figure 4(a)). The NOI workload (the lower part of the figure) has 20 aircraft, first ten of them flying at $y = 120$ [‡], the other ten flying at $y = 130$. The x coordinates are set such that the aircraft never collide – the aircraft fly

[‡]We omit units for lengths and speeds as they do not have realistic physical meaning.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls





(a) X of all aircraft in first 30s (b) Y of all aircraft at any time (c) Instantaneous aircraft speed

Figure 4. Aircraft coordinates and speed in COL (top) and NOI (bottom) workloads.

	COL	NOI
VM	RTSJ,GC/RTGC	RTSJ,GC/RTGC
Period	10ms	4ms
Collisions	YES	NO
Detector Noise	NO	YES
Backgr. Noise	NO	YES
No. of Aircraft	40	20
Duration	100s	80s

Figure 5. Sample workloads summary.

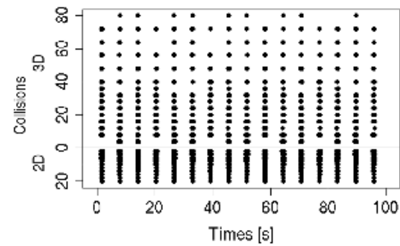


Figure 6. Collisions in COL workload. The upper graph shows 3D collisions, the lower is the number of 2D collisions.

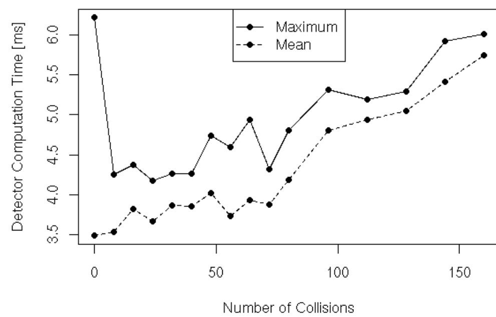


Figure 7. Computation time relative to number of collisions.

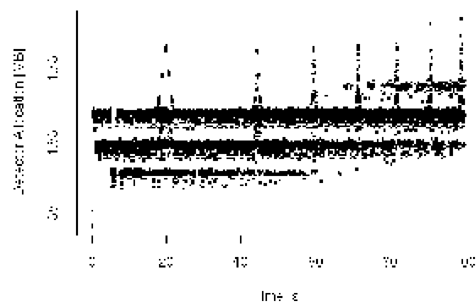
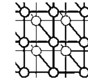


Figure 8. Allocation rate.

Copyright © 0000 John Wiley & Sons, Ltd.



in parallel far enough from each other. The COL workload has 40 aircraft, 20 of which fly at $y = 100$ and the other 20 at $y = 120$. The x coordinates are set such that there are regularly massive collisions, as visible graphically in Figure 4(a). In the NOI workload, all aircraft fly at the same speed, which is however not constant in time. The speed is shown in Figure 4(c) (the lower part). The COL workload has two groups of aircraft, 40 of them fly at the same speed as the aircraft in the NOI workload, the other 40 at the speed shown in the upper part of the figure. The structure of COL workload collisions and their occurrence in time is shown in Figure 6. The upper part of the figure are numbers of detected collisions by the collision checker (numbers of pairs of colliding aircraft in 3-d). The lower part is the number of grid elements of the 2-d grid that were occupied by two and more planes, as identified by the reducer. The peaks of collisions well align with the x coordinates of the trajectories in Figure 4(a) (upper part).

We conducted some experiments to investigate how the computation time and allocation rates are related to collisions. The computation time of the detector depends on the amount of work the reducer and the collision checker have to do, which in turn depends on the number of collisions in the workload. Figure 7 shows the relation between computation time (max and mean) and the number of collisions. The dependency is largely linear. As the case with no collisions is more frequent than with nonzero collisions, interference from the OS or hardware is more likely, and thus the measured maximum is higher. Figure 8 shows memory allocation per detector invocation throughout the execution of the benchmark. The peaks in allocation correspond to presence of collisions in the workload.

5. Metrics and Measurements

This section describes in detail what kind of data can be obtained by running CD_x and gives examples of the benchmark use on a number of configurations.

5.1. Properties of CDx

Measurements in CD_x focus on the periodic real-time task that performs the detection. The task has period T given by the number of frames produced per second: $T = 1/FPS$ (e.g., 10ms). The deadline for the task is its period, $D = T$. The important performance metrics for such a task (Figure 9) are *release jitter* J_j, *computation time* C_j, and *response time* R_j (j is the invocation index). The *release jitter* is influenced mainly by the system timer implementation, scheduling overheads, and incrementality of the VM runtime, mostly the garbage collector. A particular problem that has to be taken care of is phase shift. The *phase shift* is present in systems with tick schedulers [12], where tasks can only be re-scheduled at specified periodic intervals when the system timer ticks. With the single task in our case and with a period T being a multiple of the system timer period, the phase shift would be zero for the start time t_0^r at a system timer tick, up to the timer period for unlucky time t_0^r. As the system timer can run at periods around 500 μs or even more, with a naive (random) choice of t_0^r the phase shift dominates the release jitter in the benchmark, rendering the other overheads in release jitter unmeasurable. The benchmark thus sets t_0^r to start at absolute time rounded-up to a single benchmark period T, making the phase shift more deterministic. Typically, T is also a multiple of the system timer period, and thus this also reduces the phase shift to scheduling overhead and overhead of the set-up code. This trick indeed depends on more technical subtleties, as there can be multiple timers (OS,VM)

Copyright © 0000 John Wiley & Sons, Ltd.

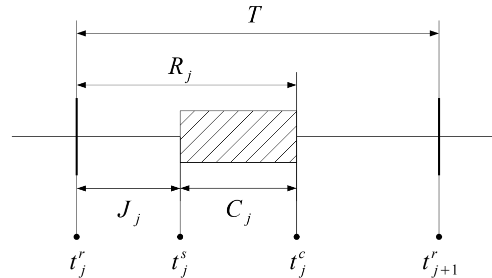


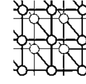
Figure 9. Metrics and measurements.

and multiple clocks in the system. We have successfully tested it empirically with Ovm, RTS, and WRT on Real-Time Linux. The *computation time* is mainly influenced by the implementation and workload configuration. Indeed it also depends on the (non-realtime) performance of the complete system. The *response time* is in our case just the sum of *release jitter* and *computation time* of each detector release.

5.2. Measurement Technique

Figure 9 shows measurement points for each release j that allow the tester to calculate metrics J_j, C_j, and R_j. These measurement points record ideal release time t^r_j, actual start time of detector thread t^s_j, and completion time t^c_j. The ideal release time is calculated from the system start time t^r_0, $t^r_j = t^r_0 + jT$. The benchmark records absolute times at all these points, which allows the tester to map anomalies to other activities identified by absolute times, such as various GC events. All timestamps are stored in a pre-allocated (immortal) memory buffer and are dumped after the measurement is over. Calculating C_j is simple: $C_j = t^c_j - t^s_j$. Once we have J_j, $R_j = J_j + C_j$. Calculating J_j is however more subtle. The problem is that we want to measure real-time performance at steady state, allowing missed deadlines during a fixed number of initial releases (warm-up). This might not be needed on a real-time OS with ahead-of-time compilation, but we want to be able to run on Real-Time Linux with RT JVMs with JIT, in particular in WRT and RTS. We have found experimentally that these cannot meet the deadlines reliably from the beginning in this benchmark in a configuration sufficiently challenging at steady-state. The problem with the missed deadlines during initialization is that we have to map the steady state release times to start times: with missed deadlines at initialization, we have release times t^r_k, start times t^s_j, and completion times t^c_j. We thus need to find a mapping $k \leftrightarrow j$ to calculate J_j and R_j. This mapping is influenced by missed deadlines, which can be either reported via the RTSJ API (waitForNextPeriod returns false) or unreported by the VM. Let's assume that we have verified that the benchmark warms-up well within Tk_0 seconds. Now, if there was any reported deadline miss after this time, we reject the data and do not need the mapping. Otherwise, we find (the smallest) j_0 that minimizes the offset of a measured task start from the ideal release $|t^s_{j_0} - t^r_{k_0}|$. The mapping $k \leftrightarrow j$ is then given by $k - k_0 = j - j_0$ and for $j \leq j_0$, we have

$$J_j = t^s_j - t^r_{j-j_0+k_0}$$



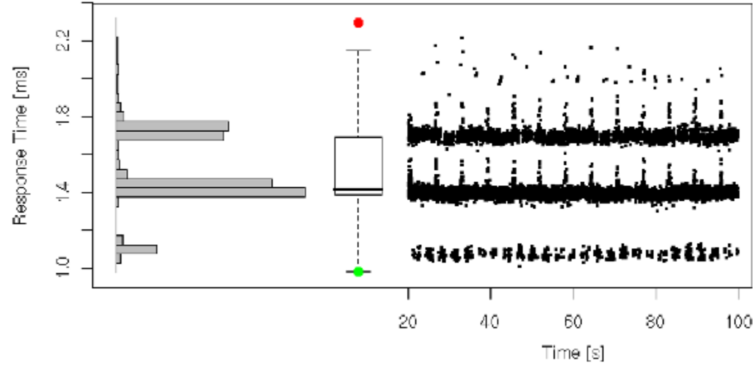


Figure 10. Sample response time plot: histogram, boxplot, and run-sequence plot.

[ms]	Min	Avg	StdDev	Max
Response	0.980	1.489	0.193	2.294
Computation	0.969	1.460	0.192	2.250
Jitter	0.006	0.029	0.008	0.532

Figure 11. Sample results table.

Configuration	$\mathrm{CD}_{\mathtt{rgjb}}$
OS	Ubuntu RT Linux 2.6
VM	Sun RTS 2.1
CPU	1x Intel Pentium 4 3.8Ghz
Heap Size	300M
Reserved Mem.	50M

Figure 12. Platform settings.

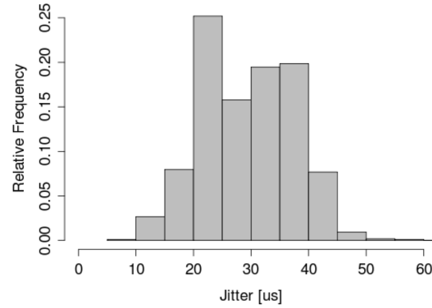


Figure 13. Relative frequency histogram of release jitter.

and we can compute R_j. If $\exists j_m, j_m \geq j_0 \wedge R_{j_m} \geq T$, there is an unreported deadline miss and we reject the data. Note, however, that the test does not allow the tester to reliably find out how many deadlines were missed or when the misses took place, because the mapping $k \leftrightarrow j$ does not have clear semantics in the presence of missed deadlines. On the other hand, if there is no such j_m, there was no deadline miss, the mapping is sound and we accept the data with the measured R_j, C_j, and J_j.

5.3. Experimental Result Format

Once the metrics are calculated and results shown to have no missed deadlines, some summarization and presentation of the data is needed. This has to include results from multiple executions of the benchmark, as to account for random effects at various levels in the measured system. A sample data presentation is provided in Figures 10 and 11 (CD_j COL workload, PDSK GC/RTGC, SUN RTS, Linux, x86). Recall that PDSK states that all frames are pre-simulated offline and stored to disk.

Copyright © 0000 John Wiley & Sons, Ltd.

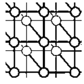


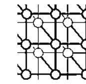
Figure 11 shows minimum, mean, standard deviation, and maximum of the response time, computation time, and jitter. The values are taken from 50 executions of the benchmark, skipping a safe amount of initial measurements to allow the system to stabilize. Based on manual inspection of the measured data, we decided that the system stabilized well within the first 20 seconds of execution, and thus we dropped the initial 2000 measured values. Figure 10 shows a histogram, boxplot, and run-sequence plot for the response time. The histogram and the boxplot use the same values from all of the 50 executions. In the boxplot, the red and green dots are extremes. We use the default boxplot definition from R statistical software: the central bold line marks the median, the hinges mark the quartiles and the whiskers are each up to 1.5x the inter-quartile-range from the closer quartile. The run-sequence plot only shows values from a single execution of the benchmark. The horizontal axis of the run-sequence plot is experiment time in seconds (it starts at 20, as the initial 20 seconds were assumed to be the warm-up). The same plot could indeed be created also for computation time and the jitter. Figure 13 then shows a relative frequency histogram of the release jitter. Four outliers of about 530 μs were excluded from the plot. The sample results were measured on a platform characterized in Figure 12.

5.4. Stress Tests

The noise generators included in the benchmark lend themselves well to stress testing virtual machines. Stress tests are needed to evaluate scalability as well as discovering breaking points of the measured system. The natural parameter to vary is the number of allocated objects: one can measure the system for a varying number of allocated objects per release. The noise generator that uses variable object size stresses both the garbage collector (if present) and the memory manager. The noise generator with the constant object size then usually stresses the memory manager more than the garbage collector, because the memory for the objects can easily be re-used, as the noise generators maintain a constant number of objects reachable from their structures. A sample result of a stress test is shown in Figure 14. The test used the noise generator with fixed allocation size, which was set to 64 bytes. To allow the system to run successfully with different stress levels, we have used a custom workload with a 10 ms period, 19 aircraft, no actual collisions, and no background noise. For every bullet shown in the figure, we have executed the benchmark 3 times, processing 10,000 frames in each execution. The figure shows clear linear dependency of the minimum and the median response time on the number of allocated objects in the noise generator, which is the expected behavior. For larger number of allocated objects, the maximum response time has occasional peaks of about 2ms, which suggests infrequent but present slow-paths in the memory allocator. The test was run with Sun RTS's RTGC, which was configured to behave like Henriksson's GC [13]. Therefore, the collector has only been running when the application was idle, and thus couldn't have been the direct cause of the peaks. The two vertical grey lines in the plot show two significant breaking points discovered by the experiment. The first one denotes the largest number of allocated objects for which all executed experiments finished successfully without missing a deadline. For even larger numbers of allocated objects, some of the experiments have failed. The second grey line then denotes the smallest number of allocated objects for which all experiments have failed (we have run tests for up to 40,000 allocated objects, which are not shown in the plot). In this particular experiment, the failures were all due to running out of memory: the benchmark allocated more memory than the collector was able to reclaim during the idle time. Note that with the increasing number of allocated objects, the idle time was decreasing as well as the amount of allocation increasing, both contributing to greater challenge for the collector. This experiment is useful to understand the

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



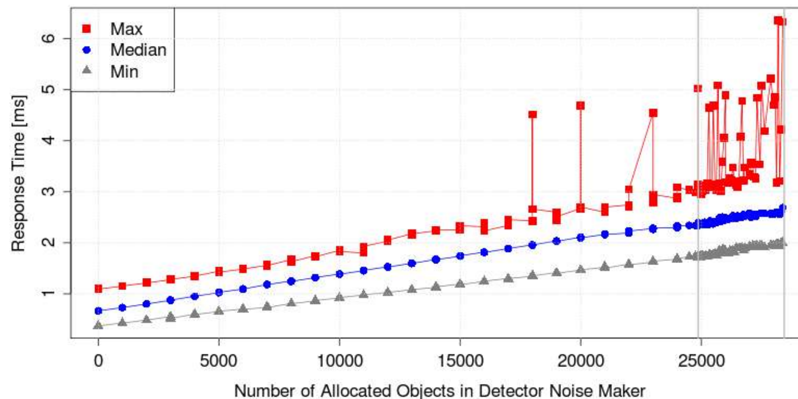


Figure 14. Sample stress test: response time for an increasing amount of allocation noise on RTS.

impact of allocation on the real-time collector. It demonstrates that after 15K objects the worst case time starts to experience outliers due to GC and after 25K objects the garbage collector starts to fail.

5.5. Start-up Time

In many real-time applications start-up behavior is important. CD_x can be set up to measure the start-up behavior of virtual machines. In this experiment we contrast the behavior of ahead-of-time compiler with that of just-in-time compiler (JIT). We typically ignore, or drop, the completion of the initial frames from our measurements. However, to understand the impact of a JIT, which may induce pauses in the early stages of the computation, we analyze those initial measurements more carefully. In particular, we will look at the worst case completion time (more detail on this test is in [14]). Fig. 15 illustrates the evolution of the worst observed time as we remove initial iterations of the benchmark. By this we mean that position 0 on the X-axis shows the worst observed case for all 10,000 iterations of the algorithm. This measure is dominated by the cost of just-in-time compilation. At offset 100, for example, the graph shows the worst-case observed between iterations 101 and 10,000. Finally, the far right of the graph shows the worst-case times when the first 400 iterations are ignored. At that point the worst-case time is dominated by GC. It is interesting to observe that the costs of JIT compilation are highest in Hotspot Server and they take longer to stabilize. Hotspot Client is less aggressive and reaches fixpoint in around 60 iterations of the benchmark. WebSphere tries to compile code quickly, but the data shows that some compilation is still happening until around 200 iterations. Unsurprisingly, Fiji VM has no start up jitters as it is an ahead-of-time compiler.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

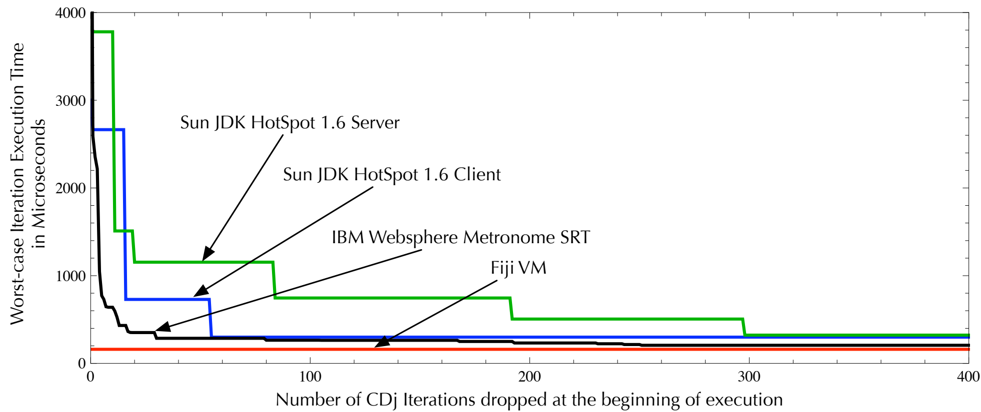
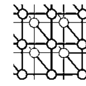


Figure 15. Start-up costs. The Y-axis is the worst-case observed execution time, while the X-axis shows iterations dropped from the 10,000 performed for each VM. The initial iterations are dominated by the just-in-time compiler. If roughly 300 or more iterations are dropped, the worst-case is dominated by garbage collection.

5.6. Evaluating an Embedded Configuration

In this section we illustrate that CD_x can indeed be used on embedded platforms. For this example run we selected the LEON3 – a SPARC-based architecture that is used both by the NASA and the European Space Agency [15] – and the RTEMS real-time operating system. Our experiments were run on a GR-XC3S-1500 LEON development board. The board's Xilinx Spartan3-1500 field programmable gate array was flashed with a LEON3 configuration running at 40MHz. The development board has an 8MB flash PROM and 64MB of PC133 SDRAM split into two 32MB banks. The version of RTEMS is 4.9.3. The experiment compares the performance of CD_s, the Safety Critical Java version of CD_x, running on the Ovm virtual machine with scoped memory, to CD_c, the ANSI-C version of the same benchmark. We configured the benchmark to run the real-time periodic task every 120 milliseconds. The benchmark tracks 6 airplanes and executes the algorithm for 10,000 iterations. The computation took roughly 20 minutes to complete on the LEON3 platform. The raw runtime performance of CD_c compared to CD_s is presented in Figure 16. As we can see, in most cases the execution time of one iteration in CD_c is around 53 milliseconds, while for CD_s it is around 78 milliseconds. The median execution time for CD_c is 50% smaller than the median for CD_s. For real-time developers, the key metric of performance is the worst observed time. In our benchmarks, SCJ is 50% slower than C in the worst-case. No deadlines were missed in any executions. A more detailed view of the performance of CD_c and CD_s for a subset of the iterations is presented in Figure 17. The graph clearly indicates that there is a strong correlation of execution times between CD_c and CD_s, and that indeed the workload is highly deterministic.

Copyright © 0000 John Wiley & Sons, Ltd.



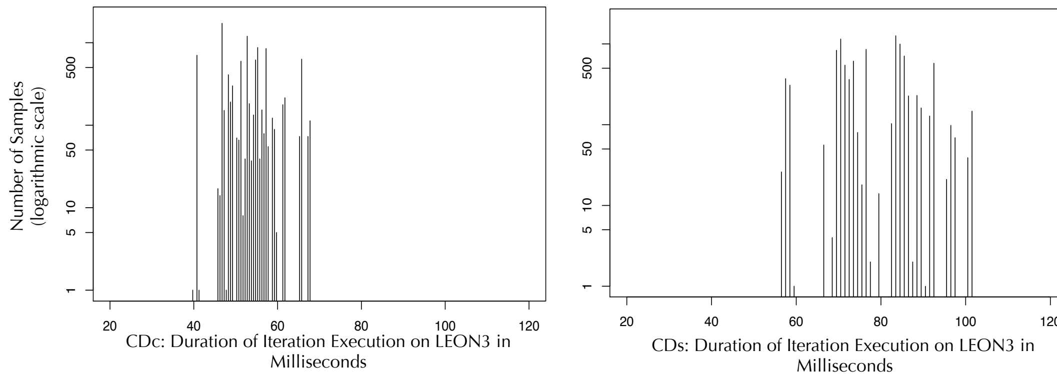


Figure 16. Histograms of iteration execution times for CD_c and CD_s on RTEMS/LEON3. SCJ's worst observed case is 50% slower than C, and the median is also 50% slower.

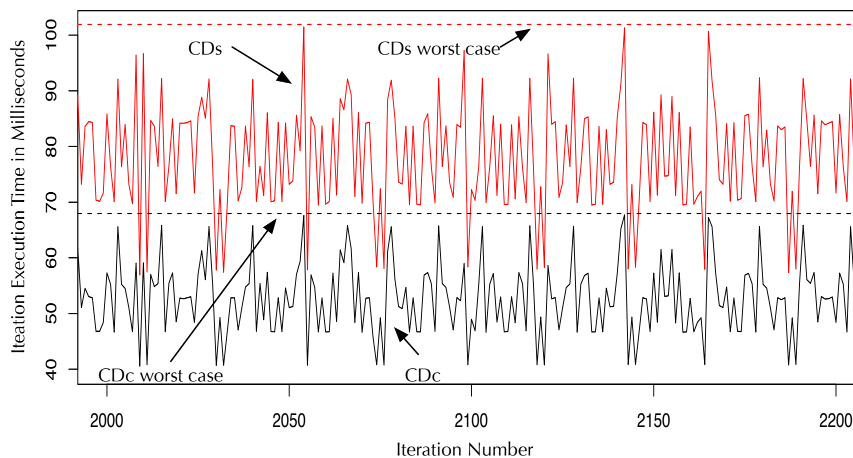


Figure 17. A detailed runtime comparison of CD_c and CD_s for 200 iterations on LEON3. SCJ and C execution times are closely correlated.

5.7. Comparing Virtual Machines

Our last example use of CD_x is a comparison of the predictability of different virtual machines. In this experiment we compare WebSphere SRT, Hotspot Client, Hotspot Server, and Fiji VM using CD_j, PDSK GC/RTGC running on a multicore. CD_j was configured to use up to 60 planes with a 10 milliseconds period and 10,000 iterations. All VMs were given maximum heap sizes of 35MB to execute CD_j and were run with default options. The goal of the experiment is to have a rough

Copyright © 0000 John Wiley & Sons, Ltd.

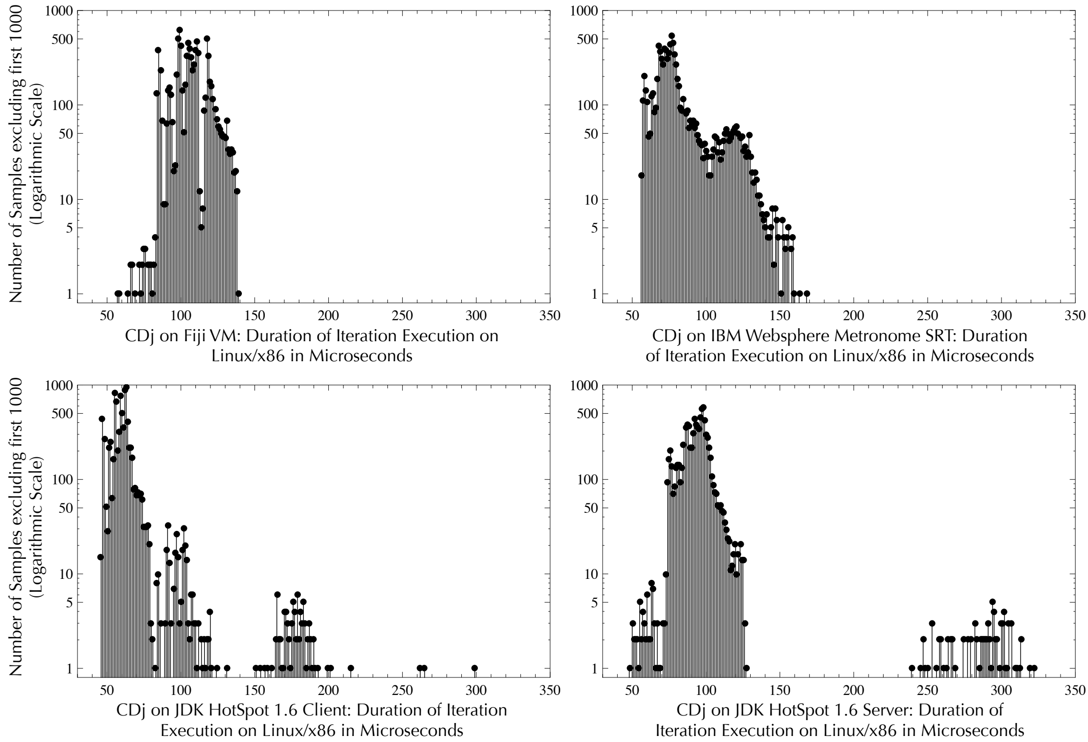


Figure 18. Histograms of iteration execution times for CD_j on Linux/x86.

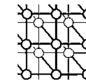
idea of the difference in predictability between VMs. The histogram of Fig. 18 show the frequency of execution times for each VM with the first 1,000 iterations of the algorithm dropped to avoid bias due to the just-in-time compiler. The data demonstrates that average case performance is better for Hotspot. Specifically, Hotspot Server is 37% faster than, e.g., Fiji VM. The worst observed case is more important for real-time developers. There Hotspot performs between 185% and 200% worse than, e.g., Fiji, these difference are caused by garbage collection pauses. Fiji VM has the tightest distribution (i.e. least deviation from peaks to valleys) of any virtual machine for this benchmark.

6. Conclusion

Publicly available real-time Java benchmarks are needed for repeatable and trusted comparisons of real-time Java products and for decisions in their design. The only available (freely or commercially) benchmarks to this end are micro-benchmarks measuring various real-time latencies in isolation under a purely synthetic workload. Application benchmarks, which could measure real-time aspects in more

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



realistic settings, are only used internally by companies and universities, making results non-repeatable, unverifiable, and hard to interpret. We present CD_x, an open-source real-time Java benchmark family that models a real-time aircraft collision detection application. For comparing the quality of RTSJ implementations, it utilizes RTSJ scopes and immortal memory features. For comparing the quality of real-time garbage collectors, it supports a mode with heap only allocations and RTSJ timers and threads. For the ease of development and educational purposes, it also runs in a plain Java VM. To our knowledge, CD_x is the only application-level real-time Java benchmark publicly available. It is also the most complex freely available RTSJ code actually using scopes and immortal memory.

Acknowledgments

The authors thank G. Haddad for work on the C benchmark, L. Zhao and D. Tang for their work on the SCJ version, and A. Plsek for the initial version of the code complexity metrics.

REFERENCES

1. Bollella G, Gosling J, Brosgol B, Dibble P, Furr S, Turnbull M. The Real-Time Specification for Java. *Addison-Wesley*, 2000.
2. Armbruster A, Baker J, Cunei A, Holmes D, Flack C, Pizlo F, Pla E, Prochazka M, Vitek J. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems* (TECS) 2007; **7**(1):1–49.
3. Gestegard Robertz S, Henriksson R, Nilsson K, Blomdell A, Tarasov I. Using real-time Java for industrial robot control. *Proceedings of the Workshop on Java Technologies for Real-time and Embedded systems* (JTRES), 2007; 104–110.
4. Juillerat N, Müller S, Schubiger-Banz S. Real-time, low latency audio processing in Java. *Proceedings of the Computer Music Conference*, 2007.
5. Blackburn SM, *et al.*. The DaCapo benchmarks: Java benchmarking development and analysis. *Proceedings of the Conference on Object-Oriented Programing, Systems, Languages, and Applications* (OOPSLA), 2006; 169–190.
6. NIST. SciMark 2.0 benchmarks. http://math.nist.gov/scimark2, 2000.
7. Bresenham JE. Algorithm for computer control of a digital plotter 1998; :1–6.
8. SPEC. SPECjvm98 benchmarks, 1998.
9. Andreae C, Coady Y, Gibbs C, Noble J, Vitek J, Zhao T. Scoped types and aspects for real-time Java memory management. *Realtime Systems Journal*, 2007; **37**(1):1–44.
10. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 1994; **20**(6):476–493.
11. Spinellis DD. ckjm - A Tool for Calculating Chidamber and Kemerer Java Metrics. 2009.
12. Burns A, Tindell K, Wellings A. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 1995; **21**(5):475–480.
13. Henriksson R. Scheduling garbage collection in embedded systems. PhD Thesis, Lund University Jul 1998.
14. Pizlo F, Ziarek L, Blanton E, Maj P, Vitek J. High-level programming of embedded hard real-time devices. *Proceedings of EuroSys 2010*, 2010.
15. Kalibera T, Prochazka M, Pizlo F, Decky M, Vitek J, Zulianello M. Real-time Java in Space: Potential Benefits and Open Challenges. *Proceedings of Data Systems in Aerospace (DASIA)*, 2009.
16. Jet Propulsion Laboratories, Golden Gate Project. Suramadu benchmarking framework. 2006.
17. Doherty B. A real-time benchmark for Java. *Proceedings of the Workshop on Java technologies for Real-time and Embedded Systems* (JTRES), 2007; 35–46.
18. Kalibera T, Parizek P, Haddad G, Leavens GT, Vitek J. Challenge benchmarks for verification of real-time programs. *Proceedings of the 4th workshop on Programming languages meets program verification (PLPV)*, 2010; 57–62.
19. Pizlo F, Fox J, Holmes D, Vitek J. Real-time Java scoped memory: design patterns and semantics. *Proceedings of the International Symposium on Object-oriented Real-Time Distributed Computing* (ISORC), 2004.
20. Zhao T, Noble J, Vitek J. Scoped types for real-time Java. *Proceedings of the Real-Time Systems Symposium* (RTSS), 2004.
21. Auerbach JS, Bacon DF, Guerraoui R, Honig Spring J, Vitek J. Flexible task graphs: a unified restricted thread programming model for Java. *Proceedings of Conference on Languages Compilers and Tools for Embedded Systems* (LCTES), 2008; 1–11.

Copyright © 0000 John Wiley & Sons, Ltd.