CD$_x$: A Family of Real-time Java Benchmarks

Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek*, Ben Titzer, Jan Vitek
Purdue University and *INRIA Lille

ABSTRACT

Java is becoming a viable platform for hard 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 real-time benchmarks. The few benchmarks that exist are either low-level synthetic micro-benchmarks, or benchmarks used internally by companies, making it difficult to independently verify and repeat reported results.

This paper presents the CD$_x$ (Collision Detector) benchmark suite, an open source application benchmark suite that targets different hard and soft real-time virtual machines. CD$_x$ is, at its core, a real-time benchmark with a single periodic task, which implements aircraft collision detection based on simulated radar frames. The benchmark can be configured to use different sets of real-time features and comes with a number of workloads. We describe the architecture of the benchmark and characterize the workload based on input parameters.

1. INTRODUCTION

Driven by the popularity of Java, the availability of development tools, and the wide library support, the Real-Time Specification for Java (RTSJ) [10] is on the rise. It is used in avionics [4], shipboard computing [20], industrial control [17] and music synthesis [5, 24]. Multiple virtual machines implementing the RTSJ are available [34, 19, 1, 3, 28].

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 [32, 9, 31, 22, 26]. While these benchmarks can and have been used to evaluate the quality of real-time virtual machines [6, 8], they are not representative of real-time workloads and they may actually end up misleading developers and end users.

Traditional Java benchmarks are designed with throughput as the main aspect of performance quality. They aim to generate the highest sustainable load and they measure the mean performance under this load, neither of which suits real-time systems in general and hard real-time systems in particular. Hard 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. A hard real-time system, for example, has to be shown schedulable (i.e. guaranteed not to miss any deadline). This is typically done with a schedulability test [16] which assumes knowledge of the worst case execution time for every piece of a program. Since, worst case execution times are usually conservative estimates, real-time systems end up with substantial slack (idle time). A benchmark without slack may cause a real-time virtual machine to run out of memory (e.g. if it uses the Henriksson [18] algorithm for garbage collection). Another difference is that a real-time Java program will make extensive use of the RTSJ APIs.

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 dimensioned 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 suite, an open source application benchmark suite that can be targeted to different hard and soft real-time platforms. At its core, CD$_x$ has a periodic thread that detects potential aircraft collisions, based on simulated radar frames. 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, a RTSJ virtual machine with scoped memory or with real-time garbage collection. Furthermore, we can add non-real-time computational noise to create a more challenging workload.

In the rest of the paper, we describe the application logic of the benchmark (Section 2), highlight its architecture (Section 3), characterize two selected workload configurations (Section 4). In Section 5, we describe the supported measurement techniques and performance metrics. We also present sample results.

The source code of CD_x and scripts used to run experiments presented in the paper can be downloaded from http://www.ovmj.net/cdx/.

2. BENCHMARK DESCRIPTION

The CD_x benchmark was originally designed as a project in an undergraduate software engineering class at Purdue by Ben Titzer in 2001. The particular implementation, CD_x, is based on an earlier RTSJ version of the benchmark by Jeff Hagelberg and Filip Pizlo. The benchmark has been modified a number of times over the years. The 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 (`javac` or other benchmark from SPEC JVM 98) and pre-generate all the radar frames needed for a run of the CD_x, which is now the suggested usage scenario.

2.1 ATS: 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 (`MAX_FRAMES`). Thus, t_{max} is `MAX_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.

2.2 CD: 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 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. 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:

```
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 [11]. 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. We describe the test in Appendix A. To keep the memory requirements reasonable, and in particular independent on the dimensions of the 2-d detection space, the mapping of grid elements to aircraft that span it is implemented using a hash table, rather than a two-dimensional array.

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 Appendix B.

2.3 Interaction between the ATS and the CD

The ATS, which is a non real-time 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. 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 IO is not available, or for program analysis with tools that would be confused with the IO (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.

2.4 Noise Generators

The CD is by itself quite efficient in memory usage: it does not generate much garbage collection work by allocating memory or updating pointers in the heap. To allow scaling the GC work generated by the detector better, we added an optional noise generator which can run within the CD 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 (constants are capitalized in pseudo-code):

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

This simple algorithm allows to tune the allocation rate by tuning `ALLOC_SIZE` and the amount of reclaimed objects by tuning the size of the root array. On the other hand, the reuse 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;
int maxSizeIncrement = MAX_ALLOC_SIZE - MIN_ALLOC_SIZE;
for(i=0;i<OBJECTS_PER_RELEASE;i++) {
   rootArray[ (rootPointer++) % rootArray.length ] =
      new byte [ MIN_ALLOC_SIZE +
                     sizeIncrement % maxSizeIncrement ];
   sizeIncrement += ALLOC_SIZE_STEP;
}
```

Indeed, the downside is that the noise generator does not generate a realistic load for the GC.

In order to provide a more realistic source of allocation noise and also some background computational noise, we support the execution of a SPEC JVM 98 [30] benchmark as a task with the lowest priority in the system. The external benchmark is run using Java reflection, thus it needs not to be available at build time and the code base is completely independent. Although the allocation noise of this background benchmark is more realistic, it is still not representative of a real-time system. Creating a GC intensive realistic real-time application benchmark is an interesting problem that we do not solve by this work.

3. BENCHMARK IMPLEMENTATION

The CD_x benchmark is configurable to support different runtime environments. In particular, we wanted to be able to compare the efficacy of scoped memory vs. traditional garbage collection vs. real-time garbage collection. Another dimension of customization is whether to have computational noise or not. The following table summarize the main configuration options that are supported:

$\mathrm{CD}_{\mathtt{jg}ns}$	Plain Java with garbage collection
$\mathrm{CD}_{\mathtt{rs}ns}$	RTSJ with scoped memory
$\mathrm{CD}_{\mathtt{rg}ns}$	RTSJ with real-time garbage collection
$\mathrm{CD}_{\mathtt{ss0}s}$	SCJ with scoped memory (*upcoming*)

The value of n can be either 0 to indicate the absence computational noise, `j` for the SPEC JVM 98 `javac` benchmark or `s` for the ATS simulator. The value of s defines the ATS implementation: `a` is the ATS simulator, `b` is the version that reads the simulation from a binary file, and `e` is for the case where the simulation is encoded in a Java class.

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 RTSJ library can be removed by the wrappers, 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, as we detail in the next section, 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, but not of plain Java application.

The plain Java version of the benchmark can be both compiled and run with standard Java. The RTSJ Java libraries and a RTSJ VM are only needed to build and run the RTSJ version of the benchmark with immortal memory, scopes or RTGC. The RTSJ code has been tested with Sun's Java Real-Time System (RTS), IBM's WebSphere Real-Time (WRT), and Ovm.

3.1 Using Scoped Memory Areas

In the $\mathrm{CD}_{\mathtt{rs}ns}$ configurations, the ATS runs in the heap, the frame buffer is allocated in immortal memory, and the CD is allocated in scoped memory. 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 [2]. Thus, there are packages named `heap`, `immortal`, `immortal.persistentScope`, and `immortal.persistentScope.transientScope`. It is correct to pass references to subpackages, 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:

```
immortal.ImmortalEntry,
immortal.peristentScope.PersistentScopeEntry,
immortal.peristentScope.transientScope.TransientScopeEntry
```

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.

3.2 Code Complexity

To measure the complexity of the benchmark code, we use the Chidamber and Kemerer object-oriented programming (CK) metrics [13] measured with the ckjm software package [33]. We apply the CK metrics to the classes that the application actually loads. The results are shown in Table 1, separately for the CD and the ATS. The CD only uses selected collection classes from the Java libraries, which we isolated into `javacp.util` package. For the CD we thus also have the complexity metrics for standard libraries it uses. For the ATS, we exclude the standard libraries from the analysis.

We use the same tool and metrics as in the DaCapo benchmarks [9], which allows us to compare CD_x to non-real-time application benchmarks: SPEC JVM 98, DaCapo, and pseudojbb. With standard libraries excluded, the detector is comparable to the simplest SPEC JVM 98 benchmarks, db and compress, in WMC (Weighted Methods per Class), DIT (Depth of Inheritance Tree), and NOC (Number of Children). In CBO (Coupling Between Objects), it is still comparable to the db benchmark. The CD is however simpler in RFC (Response for a Class) and LCOM (Lack of Cohesion in Methods). The detector is simpler than the DaCapo benchmarks and the pseudojbb benchmark. The ATS is more complex than most of the SPEC JVM 98 benchmarks and the pseudojbb benchmark. It is slightly more complex that the simplest DaCapo benchmarks luseach and luindex, but simpler than the other DaCapo benchmarks. Indeed, the CD_x benchmark would typically be configured such that the simulation would not happen when measuring, and thus only the complexity of the CD would be relevant.

4. WORKLOAD CHARACTERIZATION

The CD_x benchmark is highly configurable. We describe two pre-configured workloads, named NOI and COL. The basic parameters of the two workloads are summarized in Table 2. In the following, we describe in detail the air traffic configuration of the two workloads.

	COL	NOI
VM	RTSJ, RTGC	RTSJ, RTGC
Period	10ms	4ms
Collisions	YES	NO
Detector Noise	NO	YES
Background Noise	NO	YES
Number of Aircraft	40	20
Duration	100s	80s

Table 2: Sample workloads summary.

Package Name	WMC	DiT	NOC	CBO	RFC	LCOM	Ce	NPM
Detector								
immortal	35	6	0	21	87	13	8	25
immortal.persistentScope	32	4	0	29	77	6	8	23
immortal.persistentScope.transientScope	196	17	0	41	93	530	58	87
javacp.util	936	113	68	508	1506	7003	474	687
Simulator								
command.*	607	45	53	452	1569	3763	206	611
heap	187	44	18	101	420	511	80	144

WMC	Weighted methods/class	CBO	Object class coupling
DIT	Depth inheritance tree	RFC	Response for a class
NOC	Number of children	LCOM	Lack of method cohesion
Ce	Afferent couplings	NPM	Number of public methods

Table 1: CK metrics for loaded classes.

4.1 Air Traffic

The air traffic configuration of NOI and COL workloads was selected to be intuitive and stress the system enough – have enough collisions. 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 1(b)), such that they could never collide with each other. Only the x coordinate changes in time (Figure 1(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 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 1(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 1(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 2. 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 1(a) (upper part).

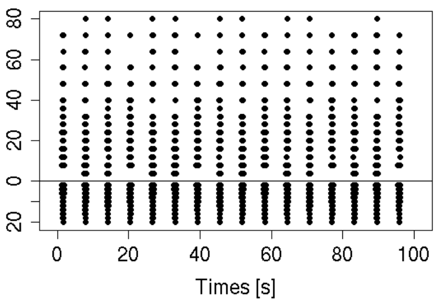


Figure 2: Number of collisions in COL workload. The upper graph shows the number of 3D collisions, the lower part gives 2D grid elements with collisions.

5. METRICS AND MEASUREMENTS

Hard real-time systems are designed not to miss deadlines. A secondary goal is to minimize resource requirements, CPU time (response times), memory or power. CD$_x$ allows to check that no deadline is missed and to compare the CPU time and heap memory requirements in different virtual machines. As deadline checking and the response time evaluation poses certain challenges, we describe it in more detail.

5.1 Properties of CDx

CD$_x$ has a single real-time periodic task. In configurations where the ATS pre-generates all frames, the CD task neither synchronizes, communicates with, nor is preempted by any other task. Technically, it could be preempted by a real-time garbage collector, but for simplicity of the description we assume it isn't.

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 3) 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

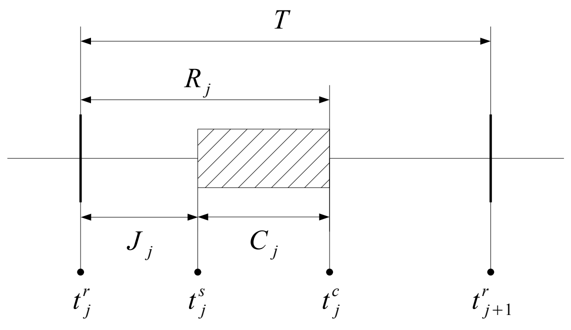
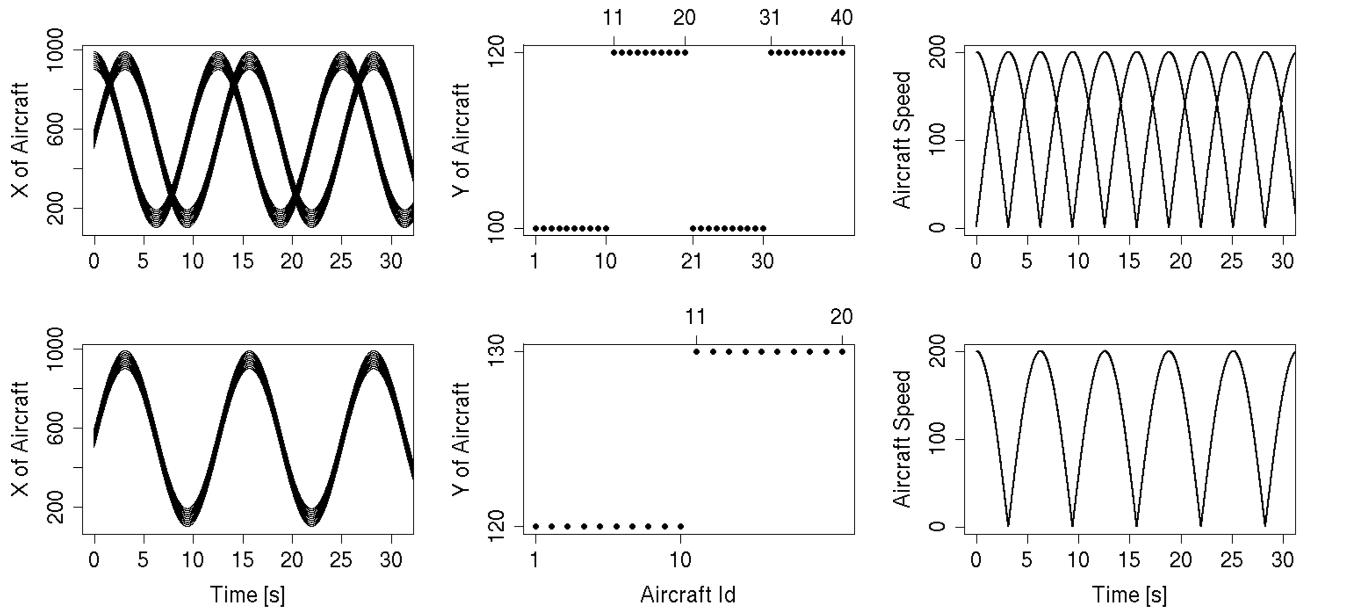


Figure 3: Metrics and measurements.



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

Figure 1: Aircraft coordinates and speed in COL (top) and NOI (bottom) workloads.

intervals when 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 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) 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 of the benchmark. 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 3 shows measurement points for each release j that allow to calculate metrics J_j, C_j, and R_j. These measurement points record ideal release time t_j^r, actual start time of detector thread t_j^s, and completion time t_j^c. The ideal release time is calculated from the system start time t_0^r, $t_j^r = t_0^r + jT$. The benchmark records absolute times at all these points, which allows 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: $C_j = t_j^c - t_j^s$. 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_k^r, start times t_j^s, and completion times t_j^c. 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 $|t_{j_0}^s - t_{k_0}^r|$. The mapping $k \leftrightarrow j$ is then given by $k - k_0 = j - j_0$ and for $j \leq j_0$, we have

$$J_j = t_j^s - t_{j-j_0+k_0}^r$$

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 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

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

Table 3: Sample results table.

Configuration	$\text{CD}_{\texttt{rgjb}}$
OS	Ubuntu Linux, RT-Kernel 2.6
VM	Sun RTS 2.1
CPU	1x Intel Pentium 4 3.8Ghz
Heap Size	300M
Reserved Memory	50M

Table 4: Platform settings.

there is no such j_m, there was no deadline miss, the mapping is sound and we and we accept the data with the measured R_j, C_j, and J_j.

5.3 Sample Results

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 Figure 4 and Table 3. The table 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. The figure then 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. The sample results were measured on a platform characterized in Table 4 with the COL workload, introduced in Section 4.

6. RELATED WORK

The open-source Suramadu benchmark suite [23, 29] 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. Similarly to Suramadu, the RTJPerf [14] benchmark suite provides micro benchmarks for selected real-time system-relevant features: allocation time, dispatch latencies, thread scheduling latencies or timer resolution. Although micro-benchmarks can be useful for identifying flaws in very specific parts of a system in isolation, and in real-time domain they allow to test for worst case latencies that are important for schedulability analysis, they do not realistically describe performance of todays complex system, where performance is determined by an inter-play of many different aspects [25, 21]. Application benchmarks are thus also needed for performance evaluations of systems. Our benchmark, being closer to an application benchmark than both Suramadu and RTJPerf, thus complements the two.

The non-realtime SPEC JBB 2005 [31] benchmark has been modified to allow evaluation of response time in a soft real-time setting [15]. The benchmark infrastructure is modified to allow response time measurements and to scale the load in a way more applicable to a real-time system. The benchmark also has an abstraction layer for real-time threads API, such that it can be run both in an RTSJ and non-RTSJ VM. While the application logic originating from SPEC JBB 2005 is far more complex than the logic of the collision detector, it is still a non-realtime logic with no RTSJ API calls (and in particular no use of RTSJ scoped or immortal memory). The benchmark is thus more suitable for evaluation of VMs for soft real-time Java systems than for hard real-time RTSJ applications. 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.

Earlier versions and modifications of the collision detector were used in [27, 35, 2, 7]. 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.

7. 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 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 hard 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 hard real-time Java benchmark publicly available. It is also the most complex freely available RTSJ code actually using scopes and immortal memory.

This said, CD_x is quite simple and definitely not a perfect application benchmark. To stress a real-time garbage collector, it requires an artificial air traffic (inputs) and additional synthetic noise generators. Also, its application logic has never been deployed to a real system. By pronouncing that such a simple benchmark is the most complex available, we call for further open-sourcing of real-time Java application benchmarks mentioned in other research studies. We also call for design of more complex real-time application benchmarks. Those should have multiple real-time tasks with in-

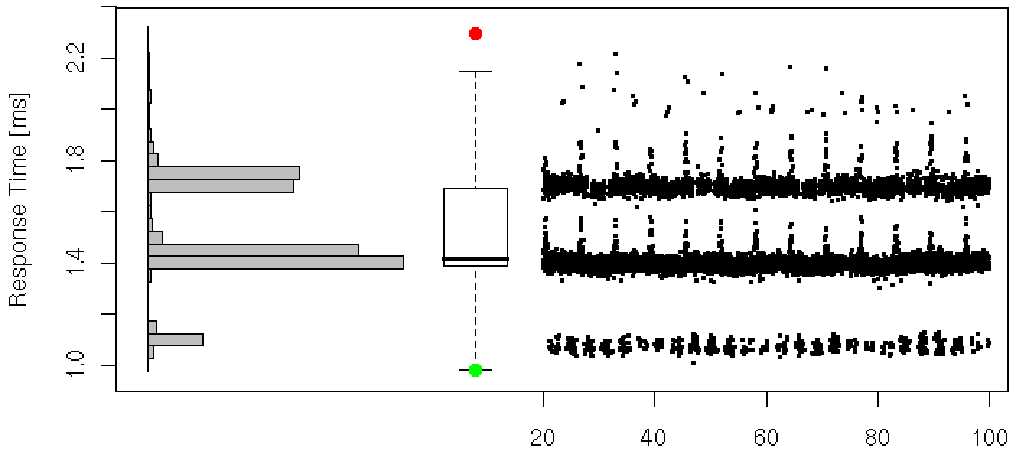


Figure 4: Sample response time plot: histogram, boxplot, and run-sequence plot.

puts realistic for the domain while enough challenging for the garbage collector, and with well tested application logic that has ideally been deployed to a real system.

8. REFERENCES

[1] AICAS. The Jamaica virtual machine. http://www.aicas.com.

[2] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time Java memory management. *Realtime Systems Journal*, 37(1):1–44, October 2007.

[3] AONIX. The PERC virtual machine. http://www.aonix.com.

[4] Austin Armbruster, Jason Baker, Antonio Cunei, David Holmes, Chapman Flack, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(1):1–49, 2007.

[5] Joshua Auerbach, David F. Bacon, Florian Bömers, and Perry Cheng. Real-time music synthesis in Java using the Metronome garbage collector. In *Proceedings of the International Computer Music Conference*, Copenhagen, Denmark, August 2007.

[6] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: democratic scheduling for real-time garbage collection. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, pages 245–254, October 2008.

[7] Joshua S. Auerbach, David F. Bacon, Rachid Guerraoui, Jesper Honig Spring, and Jan Vitek. Flexible task graphs: a unified restricted thread programming model for Java. pages 1–11. ACM, June 2008.

[8] Jason Baker, Antonio Cunei, Filip Pizlo, and Jan Vitek. Accurate garbage collection in uncooperative environments. *Concurrency and Computation: Practice and Experience*, 2009.

[9] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, volume 41, pages 169–190, October 2006.

[10] Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Programming with non-heap memory in the real-time specification for Java. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 361–369, 2003.

[11] J. E. Bresenham. Algorithm for computer control of a digital plotter. pages 1–6, 1998.

[12] Alan Burns, Ken Tindell, and Andy Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, 1995.

[13] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[14] Angelo Corsaro and Doug Schmidt. Evaluating Real-Time Java features and performance for real-time embedded systems. In *The 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2002.

[15] Brian P. Doherty. A real-time benchmark for java^{TM}. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 35–46, New York, NY, USA, 2007. ACM.

[16] Colin J. Fidge. Real-time schedulability tests for preemptive multitasking. *Real-Time Systems*, 14(1):61–93, January 1998.

[17] Sven Gestegard Robertz, Roger Henriksson, Klas

Nilsson, Anders Blomdell, and Ivan Tarasov. Using real-time Java for industrial robot control. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES)*, pages 104–110, 2007.

[18] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University, July 1998.

[19] IBM. WebSphere Real Time. http://www.ibm.com/software/webservers/realtime.

[20] IBM. DDG1000 Next generation navy destroyers. http://www.ibm.com/press/us/en/pressrelease/21033.wss, 2007.

[21] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[22] Java Grande Forum. Java Grande Forum benchmark suite. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html, 2001.

[23] Jet Propulsion Laboratories, Golden Gate Project. Suramadu benchmarking framework. http://www.opengroup.org/projects/suramadu/, 2006.

[24] Nicolas Juillerat, Stefan Müller Arisona, and Simon Schubiger-Banz. Real-time, low latency audio processing in Java. In *Proceedings of the International Computer Music Conference*, Copenhagen, Denmark, August 2007.

[25] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[26] NIST. SciMark 2.0 benchmarks. http://math.nist.gov/scimark2, 2000.

[27] Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, Vienna, Austria, May 2004.

[28] Purdue. The Ovm virtual machine, `www.ovmj.org`.

[29] David C. Sharp, Edward Pla, Kenn R. Luecke, and Ricardo J. Hassan II. Evaluating real-time java for mission-critical large-scale embedded systems. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 30, Washington, DC, USA, 2003. IEEE Computer Society.

[30] SPEC. SPECjvm98 benchmarks, 1998.

[31] SPEC. SPECjbb2000 benchmarks, 2005. http://www.spec.org/jbb2005.

[32] SPEC. SPECjvm2008 benchmarks, 2008. http://www.spec.org/jvm2008.

[33] D. D. Spinellis. ckjm Chidamber and Kemerer metrics Software, v 1.6. *Technical report, Athens University of Economics and Business*, 2005.

[34] Sun Microsystems. Sun java real-time system. http://java.sun.com/javase/technologies/realtime, 2008.

[35] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, Lisbon, Portugal, December 2004.

APPENDIX

A. REDUCER TEST FORMULATION

The test performed by the reducer is a general intersection test for a line segment and a line. We have invented this test, simplicity of which is appealing as we do not need to calculate intersections with individual square segments. We are not aware that it was published before, so we formulate it precisely.

Notation

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

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 $(\vec{i}, \vec{f})$ intersects square $(x_l, y_l, x_h, y_h) = (x_e - r/2, y_e - r/2, x_e + s + r/2, y_e + s + r/2)$, `FALSE` otherwise.

Step 1

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

$$\begin{aligned} x^t &\mapsto \tfrac{x - x_i}{x_f - x_i} \\ y^t &\mapsto \tfrac{y - y_i}{y_f - y_i} \end{aligned}$$

By this transformation we get

$$\begin{aligned} \vec{i^t} &= (x_i^t, y_i^t) &= (0, 0) \\ \vec{f^t} &= (x_f^t, y_f^t) &= (1, 1) \end{aligned}$$

$$\begin{aligned} x_l^t &= \frac{x_e - r/2 - x_i}{x_f - x_i} & x_h^t &= \tfrac{x_e + s + r/2 - x_i}{x_f - x_i} \\ y_l^t &= \frac{y_e - r/2 - y_i}{y_f - y_i} & y_h^t &= \tfrac{y_e + s + r/2 - y_i}{y_f - y_i} \end{aligned}$$

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). WLOG, we assume that $x_l^t \leq x_h^t$, $y_l^t \leq y_h^t$. We now rule out the collision in the simple cases. If any of the following conditions hold, there is no intersection (`FALSE`):

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

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

Step 3

Otherwise, we know that at least one corner of the rectangle is within the unit square (0,0,1,1). Given this, we know that 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 $(x_l^t \leq y_l^t \wedge y_h^t \leq x_h^t)$

2. LL corner is below the line and HR is high enough so that there is intersection (HR can be both below and above the line)
$(y_l^t \leq x_l^t \wedge y_h^t \geq x_l^t)$

3. HR corner is above the line and LL is low enough so that there is intersection (LL can be both below and above the line)
$(x_h^t \leq y_h^t \wedge y_l^t \leq x_h^t)$

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

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:

$$x_i \quad < x_e - r/2, \quad x_i \quad > x_e + s + r/2, \qquad (3)$$

$$y_i \quad < y_e - r/2, \quad y_i \quad > y_e + s + r/2 \qquad (4)$$

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`).

B. COLLISION CHECKER ALGORITHM

We include the algorithm for completeness of description of the application logic of the benchmark.

Notation

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

Input

$\vec{i_1}$, $\vec{i_2}$, r

Output

`TRUE` if $\exists t, d(\vec{p_1}(t), \vec{p_2}(t)) <= r$, `FALSE` otherwise.

Step 1

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

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

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

$$\vec{p_1}(t) \quad = \vec{i_1} + t\vec{v_1}$$

$$\vec{p_2}(t) \quad = \vec{i_2} + t\vec{v_2}$$

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

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

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

Step 2

If the equation is not quadratic, $((\vec{v_1} - \vec{v_2}) \cdot (\vec{v_1} - \vec{v_2}) = 0)$, 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 $d\left(\vec{i_1}, \vec{i_2}\right) \leq r$, `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 $0 \leq t \leq 1$) must be r. Otherwise, there would have been two solutions. This means that the points got to the distance r for $0 \leq t \leq 1$ (within the line segments) iff $0 \leq t_0 \leq 1$. So we return `TRUE` if $0 \leq t_0 \leq 1$, `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 (earliest) collision.

C. INSTALLATION INSTRUCTIONS

We provide open-source distribution of CD$_x$ at http://www.ovmj.net/cdx/, where documentation on how to to build, run, and modify CD$_x$ is also available.

C.1 Building CD$_x$

To build the benchmark, various VMs specified in the `compiler.properties` files can be used. The benchmark is distributed with support for regular Java VM, Sun RTS, and IBM WRT. However, a new Java compiler can be easily added by defining a new property file; see the `properties/` folder for examples. An Ant `build.xml` file is provided to implement build and distribution tasks. Execute:

```
ant -Drt=propertyFileName dist
```

to build CD$_x$ benchmark, where `propertyFileName` is the name of the `compiler.properties` file specifying the compiler (i.e. `rts`).

C.2 Running CD$_x$

The two workloads presented in this paper can be easily run by scripts `run_rts_noi.sh` (NOI workload) and `run_rts_col.sh` (COL workload). Detailed instructions on running CD$_x$ with modified parameters and workloads can be found at http://www.ovmj.net/cdx/Running/.