High-level Programming of Embedded Hard Real-Time Devices

Filip Pizlo Lukasz Ziarek Ethan Blanton Petr Maj† Jan Vitek†

Fiji Systems Inc. † Purdue University

Abstract

While managed languages such as C# and Java have become quite popular in enterprise computing, they are still considered unsuitable for hard real-time systems. In particular, the presence of garbage collection has been a sore point for their acceptance for low-level system programming tasks. Real-time extensions to these languages have the dubious distinction of, at the same time, eschewing the benefits of high-level programming and failing to offer competitive performance. The goal of our research is to explore the limitations of high-level managed languages for real-time systems programming. To this end we target a real-world embedded platform, the LEON3 architecture running the RTEMS real-time operating system, and demonstrate the feasibility of writing garbage collected code in critical parts of embedded systems. We show that Java with a concurrent, real-time garbage collector, can have throughput close to that of C programs and comes within 10% in the worst observed case on realistic benchmark. We provide a detailed breakdown of the costs of Java features and their execution times and compare to real-time and throughput-optimized commercial Java virtual machines.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—interpreters, run-time environments; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects; D.4.7 [*Operating Systems*]: Organization and Design—real-time systems and embedded systems.

General Terms Languages, Experimentation.

Keywords Real-time systems, Java virtual machine, Memory management.

EuroSys'10, April 13–16, 2010, Paris, France.

1. Introduction

Hard real-time software systems must operate in the presence of strong resource constraints. With applications ranging from satellite control systems to high-frequency algorithmic trading, hard real-time systems vary widely in hardware, operating system, and programming style. Yet there are important commonalities. Predictability of the software is paramount, with typical systems deployed on uni-processor architectures and hard real-time operating systems. The software is written to be analyzable, with simple control flow and no dynamic memory allocation; without the former analysis of the software is difficult and without the latter it is hard to make predictable. Therefore, it is common for real-time programmers to rely on static allocation of data and object pooling. The programming languages of choice are subsets of C or Ada and for the adventurous C++.

As the size of real-time code bases keeps increasing – million line systems are not unusual – factors such as productivity, reusability, and availability of trained personnel have spurred interest in Java as an alternative to low-level languages and motivated the development of real-time extensions [9]. While Java has had some success in soft real-time, every attempt that we are aware of to move into hard real-time domain ended in defeat. The reasons are multiple. The NASA/JPL Golden Gate project was too early, at the time performance was between 10-100 times slower than C. These conclusions were confirmed by an independent study conducted by Thales, which also observed that region-based memory management system offered at the time was difficult to use and error-prone. It was only when IBM researchers pushed real-time garbage collection into a production VM [2] that Java started gaining traction. Since then, real-time garbage collection has become part of all major real-time virtual machines. Unlike regular run-of-the-mill garbage collection, real-time garbage collection abstractly limits the impact that memory management can have on the execution of a thread; either by allowing the collector to be interrupted at any moment or limiting the collector to small quantums of work. Even with such advances, there is still significant reluctance to take Java seriously for the most demanding of real-time tasks.

Scaling modern, mainstream languages and their associated runtimes down to resource-constrained embedded set-

tings is difficult, especially when hard real-time guarantees must be met. This is because virtual machines are usually tuned and engineered for maximal throughput on a small set of common benchmarks. Implementation techniques often include fast paths for the common case, and expensive slow-paths for less frequent cases. Examples include inline caches on interface dispatch and type tests. Using benchmark suites to evaluate these optimizations can be misleading, as real-time workloads are unlikely to be represented by, e.g. SpecJVM98. Furthermore, to guarantee against deadline misses, one will have to make the pessimistic assumption that all slow paths will be taken, and as a result estimates of the worst-case execution time are likely to be needlessly pessimistic. Support for hard real-time requires an implementation that targets predictability over throughput and benchmarks that are representative of real-time workloads. Moreover, evaluations must be done in the context of representative hardware and operating systems.

In this work we have selected a platform that has been deployed on the Venus Express Mission by the European Space Agency (ESA). The platform's architecture hails from the LEON processor family, which implements the SPARC v8 instruction set, burnt on a Xilinx FPGA. The preferred real-time operating systems is RTEMS. The ESA use radiation hardened memory, either 32 or 64MB, and processors clocked at 40MHz. Their implementation language is usually C. Real applications are hard to get a hold of, so we use an idealized collision detection algorithm [15] as a workload. Collision detection algorithms have been implemented in hardware in radar systems. Our benchmark has similar characteristics to that of these real applications but is easier to retarget to different platforms. In the past, Java has not been deemed suitable for use for what ESA refers to as BSW (or Basic Software) since Java does not support interrupt handling, measured latencies were too high and the overhead of the Java Native Interface (JNI) was considered prohibitive [17].

This paper addresses the perceived shortcomings of Java in a hard real-time setting. The contributions of our work are threefold. First, we describe a new Java virtual machine implementation, called Fiji VM, which specifically targets real-time embedded systems. Our implementation maps Java bytecode directly to C leveraging knowledge of the entire program to generate efficient code. We give a detailed description of how Java is mapped to C from a systems perspective, including a review of overheads that must be introduced to maintain Java compatibility. Though the overheads in our VM are similar to other Java virtual machines, we are unaware of any concise review of these overheads in the literature. Thus, we hope to help real-time programmers in understanding the costs of utilizing Java. Second, we demonstrate the feasibility of running full, unmodified, Java code on an embedded LEON3 platform at high-speed while meeting hard real-time deadlines. To validate our claims of speed, we compare the performance of Java programs directly against C. To our knowledge, this is the first comparison of Java to C for a significant real-time benchmark that takes into account both throughput and predictability. Third, we demonstrate the predictability of our garbage collector. We want to convince real-time programmers to use dynamic memory management provided by garbage collection. This would be a major paradigm shift and a great simplification to real-time programming. The complexity of managing memory using object pools has substantial costs both in terms of productivity and ensuring correctness. We also include results comparing the performance of our VM to other production Java virtual machines to better position Fiji VM in the space of Java implementations.

2. State of the Art

Four commercial Java virtual machines support the Real-time Specification for Java (RTSJ) [9]. These are the IBM WebSphere Real-time VM [2], SUN's Java RTS [8], Aonix's PERC [26], and Jamaica from AICAS [30]. PERC is noteworthy in that it supports features similar to those offered by the RTSJ but uses its own APIs. All of these systems support real-time garbage collection, though the algorithms are markedly different ranging from time-based to work-based with varying degrees of support for concurrency. In addition, Oracle's WebLogic Real-Time and Azul Systems' virtual machine [11] both offer low-pause-time garbage collectors that approach real-time performance. The execution strategies of these systems range from ahead-of-time compilation to just-in-time compilation. PERC and Jamaica are the only other products currently targeting resource constrained embedded devices.

Ovm [1] is the most direct influence on the design of Fiji VM. Ovm is a Java-in-Java metacircular virtual machine that provides hard real-time guarantees. Like the Fiji VM it generates C code, but it also has an interpreter and just-in-compiler. It was used in the first Unmanned Aerial Vehicle flight using avionics software written in Java [1] and the first *open-source* real-time garbage collector with high throughput and good mutator utilization [21]. Ovm lacks functionality essential for embedded hard real-time systems. Many applications run on minimal hardware with a small real-time OS kernel and performance close to C. Ovm's performance and footprint prevented us from experimenting with a wide range of embedded devices. Furthermore, Ovm suffered from an overly complex design and we could not envision how the system could ever be certified. JRate was a contemporary of Ovm that was integrated into the GCC compiler [12]. There were early ahead-of-time compilers that translated Java to C [19, 24] which performed, broadly speaking, similar static optimizations to those included in Fiji VM, but real-time support was not part of their design goals and they targeted older versions of Java making performance comparisons difficult.

Real-time garbage collection has been investigated for many years in the context of Java. Nielsen [20], Baker's [4] and Henriksson's [14] early works inspired a number of practical algorithms including the ones used in all commercial VMs. The IBM Metronome collector uses periodic scheduling [3] to ensure predictable pause times. Jamaica uses a work-based techniques with fragmented objects [29] to get around the need for a moving collector to fight fragmentation. Java RTS [10] uses a non-moving variant of Henriksson with a scheduler that leverage the slack in real-time systems to collect garbage when no real-time task is active. For overviews of the main alternatives readers are referred to [16, 22]. The Real-time Specification for Java proposes a form of region-based allocation to avoid the costs of garbage collection. However, it has been found error-prone and incurs non-trivial runtime overheads due to dynamic memory access checks [22].

3. Designing a Real-time Virtual Machine

The goal of Fiji VM is to provide developers with an automated tool for converting high-level, memory safe, Java applications into small, efficient, and predictable executables for a wide range of embedded devices. We focus on ahead-of-time compilation as an execution strategy. A just-in-time compiler is under development but it will probably not be deployed on embedded devices. The virtual machine consists of a compiler, a runtime library, and a choice of open-source class libraries. In addition to supporting Java 6 (without dynamic class loading) and JNI 1.4, the Fiji VM has an on-the-fly concurrent real-time garbage collector and supports region-based allocation in the style of the Real-time Specification for Java.

3.1 An Ahead-of-time Java Compiler

Our VM parses Java 1.6 (or earlier) bytecodes and generates ANSI C. The generated code is automatically fed to the chosen C compiler for the target platform, and linked against the Fiji runtime, which consists of 10,000 lines of code. Many Java features are mapped directly to C, while most of the remaining features can be mapped directly, or through a light-weight wrapper, to OS functionality. The rest – including locking and type checking – is implemented by our VM. Fig. 1 shows a summary of the mapping (with some RTEMS operating system details added for concreteness). We proceed with a detailed description of the various overheads involved in compiling Java.

Java is a type-safe language. The implication for the programmer is two-fold: a program that compiles successfully will never cause memory corruption, and the compiler will insert checks which may incur a runtime performance penalty.

Null checks. Before any use of a pointer, the compiler checks that the pointer is non-null. On some platforms, null checks may be performed using virtual memory – but even then, some compiler-inserted null checks will remain. Fiji VM never uses virtual memory techniques for null checking in order to reduce the worst-case cost of a failing null check. Like most other VMs, we use control flow analysis to remove the majority of these checks. A null check is:

```
if (unlikely(!variable)) throwNPE();
```

The out-of-line helper function throwNPE throws a null pointer exception, while the intrinsic helper unlikely hints to the C compiler that the condition is unlikely to be true. On compilers such as GCC, we translate unlikely to a macro for that compiler's branch prediction pragma.

Array bounds checks. Before any access to an array, the compiler checks that the array index is within bounds of the array's range. Arrays in Java are always indexed in the range $[0, n-1]$, where n is the length given at allocation. As with null checks, most implementations will optimize away these checks – but bounds check removal is nowhere near as successful as null check removal. Thus, these checks typically incur a runtime penalty. A typical array bound check is:

```
if (unlikely((unsigned)index < (unsigned)array.length))
    throwABC();
```

Garbage collection checks. Ensuring type-safety requires either provably correct manual memory management or garbage collection. Although beneficial, garbage collection requires run-time checks. Fiji VM requires sync-points and store barriers. A sync-point checks if the stack of the current thread should be scanned. A store barrier ensures that modifications to the heap performed by the program are seen by the garbage collector, which may be operating concurrently. Sync-points are inserted using a policy that ensures that the worst-case latency between sync-points is bounded. This implies that every loop will have at least one, which hurts performance of tight loops. Optimizations such as loop unrolling can alleviate the overheads to some degree. A sync-point looks as follows:

```
if (unlikely(threadState->shouldSync)) synchronize();
```

The threadState pointer refers to the VM state for the current thread. This includes the boolean field `shouldSync`, which is true when the thread should yield. For high-priority threads which preempt the garbage collector, these checks never fire – if the collector is only running when the given thread is not running, then no additional synchronization is required. But, the presence of these checks will still result in a throughput hit. With virtual memory, these checks can be converted into the faster:

```
threadState->poison=0;
```

The volatile poison field is on page that is marked as write-protected whenever the VM wants the thread to yield. This is faster as it is a store rather than a load and memory stores are

Java Source Feature	*Java Bytecode Feature*	*C code implementation for RTEMS under Fiji VM*	*Qualitative performance comparison to equivalent C/C++ code*
Assignment (=)	pop, dup, load, store, etc.	Direct C translation (=), except for assignments to references, which are also copied into a stack-allocated Frame datastructure.	Java is slightly slower for pointer assignments, but equivalent for other assignments.
Simple Arithmetic (+, -, etc.)	iadd, isub, imul, etc.	Direct C translation (+, -, etc.)	*Same performance.*
Division, remainder	idiv, irem, etc.	Calls to helper functions (Java semantics differ from C semantics)	Java will be slower but not by much; division is expensive already.
Casts	f2i, d2i, i2c, etc., checkcast	Numeric casts are directly translated; checked casts include a type check.	*Same performance.* C++'s checked casts are no faster than Java's.
if, switch, for, while	if<cond>, goto, lookupswitch, tableswitch	if, switch, goto. Loops require sync-points (1 load, 1 branch).	Small loops will be slower in Java.
Method invocation	invokevirtual, invokeinterface, invokespecial, invokestatic	C function call with additional code at callsites, prologues, and epilogues	Java method invocation is considerably slower, but inlining alleviates this.
Throw/catch	athrow, exception handlers	if, goto	N/A for C; Java exceptions will be ***faster than C++ exceptions.***
Static initialization - implicit on first use of class	Implicit on invocation, static field access, instantiation	Run-time check (1 load, 1 branch) on first use	Slower than C/C++, but often optimized away to reduce performance burden.
Field access	putfield, getfield, etc.	Null check and direct C pointer arithmetic load/store; reference putfield results in addition code for GC barriers.	Often optimized by compiler to be as fast as C, except for reference field stores, which will incur a penalty.
Array access	iaload, iastore, etc.	Null check, array bounds check, array store check, and C pointer arithmetic load/store. As for fields, GC barriers inserted for reference array stores.	Array accesses are considerably slower in Java than in C due to safety checks. Even after optimization, many of these checks remain.
new	new, newarray, anewarray, etc.	C code for bump-pointer allocation. If this fails, call to helper function. The entire heap is pre-allocated by Fiji VM by a single call to malloc, and then managed internally.	***Faster than C/C++.*** A garbage collector enables very fast allocation; the work of organizing the heap is offloaded to a separate task, which can be run in parallel on a multi-core architecture.
synchronized	monitorenter, monitorexit	C code for fast lock acquisition, call to helper function on contention. Fiji VM implements its own locks for Java code; OS locks are only used internally.	***Faster than C/C++.*** Like other JVMs, Fiji VM has a locking implementation that outperforms OS-provided locking such as pthread_mutex.
Object.wait, Object.notify	Object.wait, Object.notify	Internal Fiji VM implementation written in C.	*Same performance.* Implemented similarly to OS condition variables.
java.lang.Thread API	java.lang.Thread API	rtems_task API, with additional Fiji VM per-thread data structures such as ThreadState	*Same performance.*
I/O libraries (java.io.*, java.nio.*, java.net.*, etc.)	I/O libraries (java.io.*, java.nio.*, java.net.*, etc.)	POSIX I/O	*Same performance.*

Figure 1. Java features and their mapping to C code and/or RTEMS operating system functionality under the Fiji VM.

much faster than loads, furthermore no branch is required. However, as RTEMS does not have virtual memory, we rely on the slower version of sync-points. Store barriers are the other source of overheads introduced by garbage collection. Primitive fields (integer, double, etc) need not be tracked, but all modifications to pointer fields are translated to:

```
if (source != null && source.gcState != marked)
    mark(source);
target.field = source;
```

The null-check on source is often optimized away, but even then, a load and a branch remain in the fast path, and a function call on the slow path.

Local variable assignment. Most local variable assignments found in Java source are either optimized away by copy propagation or translated to a C local variable assignment. However, for any local variable that contains a pointer that is live across an operation that may trigger garbage collection (a sync-point, a Java method call, or a heap allocation), the variable will be stored into a special data structure. This is needed because C compilers do not provide support for accurate stack scanning. Thus it is up to the VM to ensure that a stack scan will accurately see all live pointers. We accomplish this by having a stack-allocated data structure containing copies of local heap references as in [5, 13]. Given a local variable assignment a = b, where both are ref-

erence types, and a is live across a sync-point, method call, or allocation, the compiler emit the following:

```
a = b;
Frame.refs[index_of_a] = a;
```

Where Frame is a local struct variable, and *index of a* is a compiler-chosen index for the variable a in the Frame's list of references. This instrumentation is inserted rarely enough to not significantly impact performance.

Division and remainder. Java division has slightly different semantics than C division. It includes a division-by-zero check which is done similarly to null-checks, and benefits from similar optimizations. It has slightly different overflow rules which require a helper that performs some additional checks. We have found that the overhead of calling this helper is smaller than the cost of performing a division on most architectures. Therefore, this does not significantly contribute to performance overhead.

Loops. Loops require the insertion of sync-points. For small loops, this is a significant overhead – a loop that would have otherwise only had a single branch per iteration will now have two branches. For large loops, this overhead is negligible. To demonstrate the impact of sync-points, consider a program that sums an array of 100,000 integers on a Xeon 2.33 GHz machine running Linux, with the loop being re-executed 10,000 times. The C version needs 63.5 μs whereas Java runs in 108.7 μs. A 71% slow down for this pathological program. Luckily, benchmarks show, that this is not representative of the performance of larger programs.

Method invocations. A Java method invocation is translated to a C function call. Invocations have a number of indirect overheads: (*i*) spilling of some local variables to the GC map stored in the Frame, (*ii*) checks for exceptions following the C function's return, and (*iii*) additional code to link the Frame to enable stack scans. To mitigate these costs Fiji VM does aggressive de-virtualization and inlining. In practice, almost all small methods end up being inlined, and methods thought to be called frequently are also inlined provided that they are not too large. Thus, fewer method invocations will remain than what is apparent in the original program. Recursive code is penalized by our translation scheme, as recursive inlining is generally undesirable, but non-recursive code is usually as fast as the C equivalent.

Exception handling. The cost of throwing and catching exceptions is small. All exceptional control flow is converted to C-style if and goto statements.

Static initialization. Java prescribes that before a class is used, it must be initialized. Initialization checks are inserted before every static method call, static field access, and object instantiation. However, optimization is performed on static initializers. Initializers which are simple enough to be executed statically in the compiler are eliminated. No static initialization needs to be performed for classes that do not have static initializers. Control flow optimization is performed to eliminate redundant initialization checks. Additionally, care has been taken to ensure that the standard library classes make minimal use of static initialization, or else to ensure that the relevant initializers are called manually by the VM before program start, thus enabling the compiler to remove initialization checks for those classes. When a check remains, the code is:

```
if (unlikely(Klass.initState != initialized)) initClass(Klass);
```

`Klass` is a global struct variable containing data about the class in question. Note that C++ has a similar feature to Java's static initializers: a code module may declare global variables of object type, and the respective classes may have constructors with arbitrary code. However, unlike our implementation that uses injected checks, C++ uses sophisticated linker techniques to reduce the performance penalty. We have considered this, but have not implemented it because we have not found static initialization to lead to any significant overhead in the programs we have thus far tested.

Field accesses. Most field accesses are compiled to simple load-from-offset or store-to-offset operations in C, thus making them as fast as their C equivalents. As mentioned above stores to reference fields may include a barrier. Fortunately, stores are much less common than loads, and stores of references are less common than stores of primitives. Hence these checks end up having a small effect on throughput, as documented in [6].

Array accesses. Array accesses can be a large source of overheads in Java programs due to the combination of null-checks, bounds check, array store check, and store barrier. Array store checks only apply to stores on object arrays and are used to ensure that the object being stored is a subtype of the array element type. This check is needed to maintain type safety in the presence of covariant arrays. Put together, these checks may incur a substantial amount of overhead in certain programs. However, programs which primarily access arrays in a sequential fashion, and primarily use arrays of primitives, will often have all of these checks eliminated by the compiler.

Allocation. In Fiji VM, the Java new statement is efficient. It is roughly:

```
result = threadState->bumpPtr + size;
if (result > threadState->limitPtr) result = allocSlow();
else threadState->bumpPtr = result;
initObject(result);
```

The code implements bump-pointer allocation, where the thread-local bumpPtr indicates the location where the next object may be placed, and the limitPtr indicates the limit up to which objects may be safely allocated. If memory is plentiful, threads will be given page-size regions to allocate in; if memory is scarce or fragmentation has occurred, the memory regions will be small and allocSlow will attempt

first-fit allocation. The initObject routine is a simple inline function that establishes the object's header, and typically involves two stores.

In the common case, this code is much faster than calling malloc, since no synchronization is required, and the relevant fast path code is inlined. In the slow path, this code is no worse than most malloc implementations – unless memory is exhausted. The Fiji VM garbage collector is designed to run concurrently to the program, and thus fresh memory will be freed all the time. However, if the collector is not paced appropriately, memory may be exhausted requiring allocating threads to pause until the collector completes.

Synchronization. The synchronized statement is the primary locking facility provided in Java. Like most Java lock implementations, the Fiji VM has its own locking code. In our case, locks must also support the priority inheritance protocol [27] to avoid cases of priority inversion due to locking. The fast path, for uncontended lock acquisition, is basically a compare-and-set and the slow path is mostly functionally identical to a POSIX mutex lock in PRIO_INHERIT mode. However, unlike POSIX locks, the Fiji locking code also implements full debugging, profiling, and lock recursion support by default. Thus Fiji locks can be viewed as implementing a superset of POSIX locking functionality. Nonetheless, Fiji locks are faster than POSIX locks in the common case, and asymptotically no worse in the worst case. The main source of speedups is that our fast path is inlined, and that it only has one mode thus removing the need to perform mode checks on every lock operation. To demonstrate the speed of locking, we wrote a simple test program that makes 10,000,000 lock acquisitions and releases in both Java and C. The Java version is using synchronized while the C version is using `pthread_mutex` using the default configuration (i.e. the C lock is configured for strictly less functionality, as it will lack recursion and priority inheritance; we do this to give the C code a "head start"). Both versions were run on a Xeon 2.33GHz running Linux; the Linux version is 2.6.27 and includes the optimized futex-based locking. The performance of an average lock acquisition and release pair from this test is as follows.

`C version: 54.3` ns `Java version: 31.1` ns

The C locking implementation is 74% slower than Fiji VM's locks. The Linux lock implementation is one of the fastest available, relying on Fast User level Mutexes to allow fast-path locking to occur entirely in user-space, using a single atomic operation. However, even which futexes, POSIX locks still require out-of-line calls and additional checks that are neither mandatory nor desirable in Java. This demonstrates that although Java maybe slower on some programs, the reverse case – pathological programs for which Java is substantially faster – also exists.

3.2 Object layout

Java objects are laid out contiguously by our VM – i.e. a Java object will have a similar memory structure to a C++ object or C struct. This allows for the fastest possible array accesses and field accesses, as described above. As for C++ objects that are not POD (plain-old-data; C++ objects qualify as POD if they have no virtual methods), Java objects require a "vtable" header for virtual dispatch and run-time type identification. The Fiji VM "vtable" pointer refers to either a TypeData structure, which contains the vtable as well as type identification information used for type checks, or a Monitor structure. The Monitor is used for storing the lock, in case the object was used for locking. Thus, for Java objects that never flow into a synchronized statement, there is no space overhead for locking. The Monitor points to the TypeData of the object; the latter contains a pointer at the same offset pointing to itself.[1] Thus, getting the TypeData for an object requires a double-indirection. In addition, every object has a GC header, which includes everything necessary for garbage collection (such as the mark state of the object, which indicates whether the object is marked in this collection cycle, and a pointer to the next-marked object, used by the GC to create a linked list of marked objects) as well as everything needed to implement scoped memory in the RTSJ. Thus, every Java object requires two words of overhead. Additionally, Java arrays have an additional word devoted to the array length. Thus, a zero-length array will require a total of three words of memory, while an object with no fields will require two words of memory. In summary, Fiji VM's object overheads are no worse than those found in a typical C++ runtime: consider that a C++ object will have one word for the vtable, and at least one other word used for bookkeeping by the malloc implementation.

3.3 High-level optimizations

The Fiji compiler performs a variety of optimizations, at multiple stages of compilation, using a static-single-assignment (SSA) intermediate representation. The optimizations currently performed include:

- *Virtualization* – turning interface calls into virtual calls
- *Devirtualization* – turning virtual calls into direct calls
- *Inlining*
- *Copy propagation*
- *Sparse conditional constant propagation*
- *Tail duplication*
- *Type propagation* (0CFA) both intra- and inter-procedural.
- *Null check elimination*
- *Array bounds check elimination*

[1] Strictly speaking there is one word of overhead per TypeData (i.e. one word per class). In our case this is a compile-time constant as we do not consider dynamic loading.

- *Global value numbering*
- *Load-load optimization*
- *Loop peeling and unrolling*

The majority of these optimizations – all but virtualization, whole-program control flow analysis (0CFA from [28]), and array bounds check elimination – would also be found in a C++ compiler, or even a C compiler. Devirtualization in Java or C++ is equivalent to function pointer alias analysis, something that GCC does. The reason why we include these analyses in Fiji VM even though GCC includes them as well is two-fold: first, we have the benefit of richer type information; and second, we are inserting additional code into the program as shown in Fig. 1. Java type information allows us to observe optimization opportunities that GCC would have been obligated to conservatively overlook. Doing optimizations like loop unrolling and peeling early – before emitting C code – allows us to reduce the overheads of code we later insert, such as sync-points. However, though these optimizations all provide some speed-ups, the most profitable optimization remains inlining. In particular, we have found it to be most profitable when applied only to the smallest methods. We always inline non-recursive methods whose bodies are smaller than a callsite. For other methods we inline based on a variety of heuristics, such as frequency of execution of the call based on static estimates, and further size estimates of the callee as well as caller. However, it should be noted that even if our compiler inlined only methods with bodies smaller than the native code for a call, we would reap most of the benefits of our more complicated approach.

In our implementation, we have made the following three observations: (*i*) inlining works best with devirtualization, (*ii*) devirtualization works best with virtualization, and (*iii*) both devirtualization and virtualization work best in the presence of an analysis that can accurately identify the points-to set of the receiver at a callsite. For this, we employ a scalable control flow analysis (0CFA), in which all points-to sets are represented using a single machine word. 0CFA is short for monomorphic (zero context sensitivity, hence the '0') control flow analysis; all compilers include some form of this analysis at the intra-procedural level. Fiji VM does it over the whole program in order to determine, for each variable, what kinds of objects it might point to. Our points-to sets are tuned so that they either represent null (i.e. the variable in question never points to any valid object), uniquely identify a type (for example: an instance of a HashMap), identify any types that are subtypes of a type (for example: HashMap or any subtype of HashMap), or the set of all possible types. Care is taken to ensure that program variables accessed reflectively or via native code are presumed to point to any object; hence this analysis rewards programs that do not use reflection, which is typically the case for embedded Java programs as the reflective APIs are severely restricted in Java Micro Edition. The results of the analysis are used to more aggressively virtualize, devirtualize, and inline method calls, as well as to remove unnecessary type checks. Because of its simplicity, our 0CFA analysis converges very rapidly (20 seconds for all of SpecJVM98 with the full GNU Classpath 0.97.2 [2], and less than a second for either smaller programs or programs that use our FijiCore library that targets embedded systems). Although the analysis is crucial to our performance, giving us a 50% to 2× performance win on most programs, a thorough comparison of this algorithm's performance relative to other scalable 0CFA algorithms has not yet been made.

3.4 Type inclusion and interface dispatch

The TypeData record contains all information necessary to perform virtual method dispatch, interface method dispatch, reflective invocations, reflective field accesses, and type inclusion tests (instanceof and checked casts). Virtual method dispatch is done as in other statically typed single-subclassing object-oriented languages; the TypeData record has a vtable appended to it, through which virtual calls are made. Interface dispatch and type inclusion are more interesting since these operations often have implementations that are not constant-time. In Fiji both operations are guaranteed constant-time. In the case of type inclusion, we use type displays [18] generated using graph coloring. For interface method dispatch, we use graph coloring to allow interface methods that are never implemented in the same classes to share the same interface table entry. For all of SpecJVM98, this approach leads to 12 buckets (12 bytes per type) for type inclusion and 10 interface table entries. The interface tables are further compressed by only including an interface table in types that implement interface methods, and then stripping interface tables that have leading or trailing NULL entries.

3.5 The Runtime System

The runtime is light-weight; it contains two components: the memory manager (for scoped-memory and garbage collection) and an OS abstraction layer for threading and locking. The runtime currently runs on POSIX-like platforms like Linux, NetBSD, and Mac OS X, and on top of the RTEMS classic API. In this section, we discuss our memory management, locking, and threading in some detail. Note that the Java libraries have its own OS abstractions for I/O and whatever other facilities the libraries may choose to support.

3.6 Garbage Collection

Fiji VM currently supports an Immix-style [7] on-the-fly concurrent real-time garbage collector. This collector can be run in either a purely concurrent, or a purely slack-based mode [16]. Immix is a "mark-region" collector that segregates free memory into *regions* and *lines*. The collector proceeds by first marking those objects that are reachable by

[2] http://www.gnu.org/software/classpath

the program, and then identifying large contiguous chunks of free memory (regions) that can be reclaimed rapidly, as well as small chunks of memory (lines), which constitute fragmentation. In this system, objects are allocated in either a bump-pointer fashion (for regions) or first-fit fashion (for lines); first-fit allocation over lines is preferred and bump-pointer is only used when all lines have been exhausted. Free lines occur rarely in practice, and filling them quickly as soon as they are found minimizes fragmentation. Due to the sparsity of lines, bump-pointer allocation is used most of the time in typical programs. While this approach does not support defragmentation, it is mostly lock-free, on-the-fly (i.e. no global stop-the-world phase), and has very short pauses. In fact, the only "pauses" are due to stack scanning, which only affects low-priority threads. This collector can be thought of as similar to the WebSphere Metronome [3], in that like that collector, it lacks defragmentation, but is otherwise well suited for real-time applications. We have separately measured the throughput of this collector to be comparable to non-real-time collectors [23]. We believe this to be largely thanks to its use of Immix-style allocation and highly concurrent design.

The Fiji collector runs either concurrently to the program (when running on a multiprocessor machine) or at a predefined priority. In either mode, the collector *never* offloads work into application actions. In particular, an allocation will never perform collection work. When running at a priority that is lower than some application thread, that thread is only suspended if it makes a request for memory that cannot be immediately satisfied. Proving that this does not happen is an analogous problem to proving the schedulability of a real-time system. In fact, a fixed-priority collector that can run at priorities lower than the application (the so called "slack-based" strategy) has very well defined schedulability tests as shown in Kalibera et al's work on the Minuteman garbage collector [16]. Of course, just as with any schedulability analysis, one that includes a collector is not an easy task and is always OS-, hardware-, runtime-, and application-dependent. We do not include such an analysis in this paper though we are aware of empiric approaches [16] that make it possible.

Compiling Java bytecode to C has traditionally been seen as a challenge for accurate garbage collectors such as ours, as C compilers do not support the generation of stack maps. However, a number of well-known approaches exist for circumventing this [5]. We use the Henderson-style linked list approach in which every pointer local variable is stored in a thread-local data structure accessible to the collector. We further optimize this approach for the fact that in Fiji VM objects are never moved by the collector. Since object pointers only need to be stored into this data structure when (*i*) they are known to not already be there and (*ii*) they will be live after the next sync-point or method call, we can optimize away a large number of these stores. This allows for object pointers to be register-allocated by the C compiler in most cases allowing very good performance. Though this solves the accurate GC problem, it does not address the need to scan every thread's stack at the beginning of a collection cycle. To do this we leverage the observation that (*i*) real-time threads only yield the CPU at shallow stacks [25], and (*ii*) a slack-based collector can only commence stack scanning if it was yielded to by the real-time threads. Thus, if the collector is scanning stacks it is guaranteed that it will be able to do so very rapidly for real-time threads (due to shallow stacks) and will never have to wait for those threads to converge to a safepoint where stack scanning is possible since those threads must already be at such a point. Stack scanning thus causes real-time threads to not be able to run during the very short (microsecond-level) interval during which their shallow stacks are scanned. But we have never seen this affect the predictability or performance of those threads.

3.6.1 Threading and Locking

The threading and locking implementations are simple: we pass all of the hard work to the underlying operating system. Java threads become native threads. This leads to a simple implementation; our entire threading and locking implementation for both POSIX and RTEMS is under 6,000 lines of C code. As well, the use of the operating system's threading implementation allows for natural multiprocessing support; Fiji VM simply had multiprocessor support from the start. Java I/O calls result in calls to POSIX I/O functions. Thus code that relies heavily on threading and I/O will not see any performance degradation under Fiji VM.

4. Evaluation

The goal of this section is to demonstrate that Java is suitable for use in hard real-time settings. To this end, we strive to set a up representative workload on a realistic evaluation platform and compare the costs of idiomatic Java to C. To this end we select the CDx benchmark which models an air traffic Collision Detector algorithm. The Java version of the code, `CDj`, was originally implemented by the first author, while the C version, `CDc`, was created by Gaith Haddad at UCF and modified by us [15].[3] The benchmark is array intensive and performs significant mathematical computations, making it well suited to low-level C programming idioms and a good challenge for a Java implementation. Furthermore, the benchmark has interesting allocation patterns, which do not permit a successful completion of the benchmark without some form of memory management.[4] The platform we selected for our experiment is the LEON3 – a SPARC-based architecture that is used both by the NASA

[3] Sources are available from `http://www.ovmj.net/cdx`. The version used for this paper is tagged as "eurosys".

[4] We observe that the `CDc` version of the code is using `malloc/free` whereas many real-time programmers would rather use object pooling to prevent fragmentation and have more predictable allocation times.

and the European Space Agency [17] – and the RTEMS real-time operating system.

We also compare Fiji VM to other commercial virtual machines to elucidate the importance of our design choices on predictability and to verify how our performance stacks up to that of other real-time virtual machine and to mature throughput oriented VMs. As most Java implementations do not run on the LEON3 platform, we switch to a desktop setting and evaluate IBM's WebSphere SRT and SUN's Hotspot Client and Server. As WebSphere is optimized for multicores, we use the multicore version of the Fiji VM.

Our real-time experiments were run on a GR-XC3S-1500 LEON development board.[5] 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. We used the Rapita Systems RTBx Data Logger[6] for on-device profiling. The desktop were run on an 8-core Intel Xeon X5460 3.16GHz machine with 8GB of RAM, running Ubuntu 7.10 with the 2.6.22-14-server 64-bit SMP kernel. The version of Hotspot used is jdk1.6.0_12 and WebSphere reports "IBM J9 VM (build 2.5, J2RE 1.6.0 IBM J9 2.5 Linux x86-32 jvmxi3260srt-20081016_24573 (JIT enabled, AOT enabled)".

4.1 CDx Overview

The CDx benchmark suite is open source family of benchmarks with identical algorithmic behavior that target different hard and soft real-time platforms. While a complete description is given in [15], we will present enough information for readers to understand the overall nature of the computation performed in CDx.

The benchmark is structured around a periodic real-time 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. The need for detection of potential collisions prior to their occurrence makes CDx a real-time benchmark. Each frame must be processed in a timely fashion.

The algorithm 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 locations are only known at times when the radar frames are generated, they have to be approximated for the times in between. The approximated trajectory is the shortest path between the known locations. A constant speed is assumed between two consecutive radar frames. For this assumption to be realistic, the frequency of the radar frames should be high and the detection has to be fast. This is achieved by splitting detection into two steps. First, the set of all aircraft is reduced into multiple smaller sets. This step allows to quickly rule out aircrafts that are far from each other. Second, for each cluster, every two aircraft are checked for collisions. Both the reduction and the checking operate on pairs 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). A frame also contains a call sign which identifies the aircraft. A motion vector $\vec{m}$ is then defined as $\vec{m} = \vec{f} - \vec{i}$.

The code size of the Java version, CDj, is 3859 lines of code while the C version is 3371. CDc is somewhat simpler since it does not have hooks for the multiple configurations supported by the Java version. CDc is written in an idiomatic C style that tries to follow the algorithmic behavior of the Java code with some small differences. For instance, the hash table used by the C code does not require as much allocation and have constant time traversal.

In a departure from the CDx benchmark we modified the collision detection code to compute the radar frames. One configuration of CDx is using another thread for this, we did not use it to avoid the interference for our experiment. The other configuration uses pre-computed frames, but due to memory constraints it was not possible to load 10,000 pre-computed frames on our embedded board. The main impact of this change is that it forced us to have longer periods.

4.2 Comparing C to Java

The version of CDx used in our benchmark has a single real-time periodic tasks (CD) configured to run every 300 milliseconds. We configured the benchmark with 6 airplanes and executed the algorithm for 10,000 iterations. The Java version ran with a GC thread enabled. The GC is set to run at a lower priority than the CD task. As the CD thread takes between 147 and 275 milliseconds (see Fig. 2), this leaves the collector substantially less than 50% of the schedule to perform memory reclamation. Still, the collector keeps up, and no outliers are ever produced due to memory exhaustion.

The raw runtime performance of CDc compared to CDj is presented in Fig. 2. For real-time developers the key metric of performance is the worst observed time, in our benchmarks Java is only 10% slower than C in the worst-case. On average CDc is 30% faster than CDj. In both executions no deadlines were missed. The computation took roughly 45 minutes to complete on the LEON3 platform. A more detailed view of the performance of CDc and CDj for a subset of the iterations is presented in Fig. 3. The graph clearly indicates that there is a strong correlation between the peaks observed in CDc and CDj. Notice, however, that the peaks in CDj are on average smaller than those in CDc relative to baseline performance (i.e. the distance from average to peak is greater for C than for Java). Overall this data suggests that while Java is somewhat slower, there are no sources of unpredictability in our implementation.

[5] Further specifications can be found at `http://www.gaisler.com`.

[6] For more information see `http://www.rapitasystems.com`.

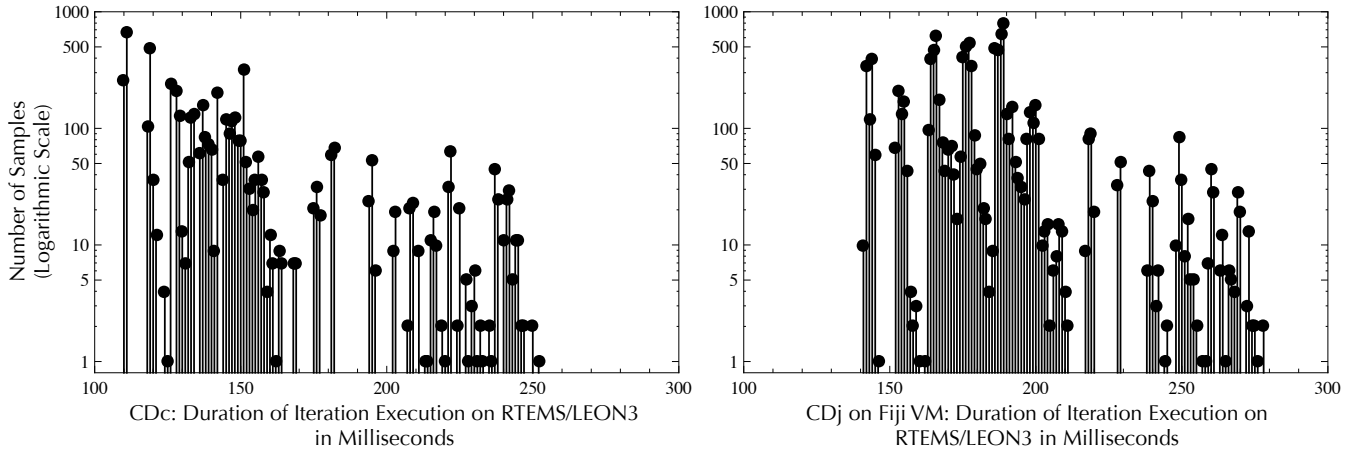

Figure 2. Histograms of iteration execution times for `CDc` and `CDj` on RTEMS/LEON3. Java's worst observed case is 10% slower than C, and the median is 30% slower.

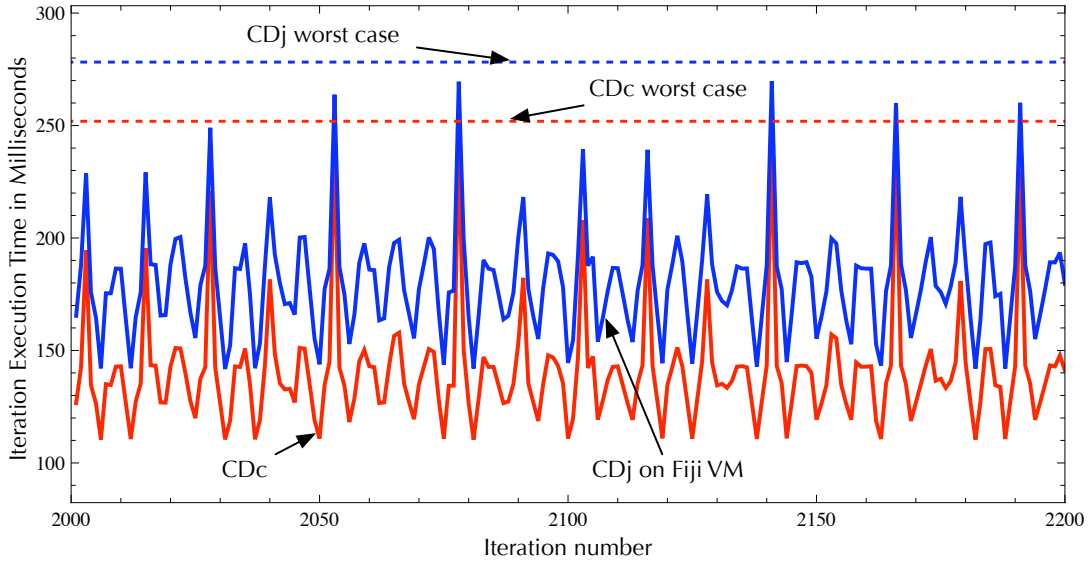

Figure 3. A detailed runtime comparison of `CDc` and `CDj` for 200 iterations. Java and C execution times are closely correlated. `CDj` is run with GC enabled. In 10,000 iterations there are 15 collections, but they never impact execution time.

Figure 4 shows correlations between the execution time of each iteration in Java to that of the same iteration in C. Iterations that take longer are ones that have either more suspected collisions, or more actual collisions. A "suspected collision" is a condition in which the aircraft are sufficiently close that the algorithm must perform more work in order to determine if an actual collision occurred. An actual collision requires more work because a collision report must be produced. The former requires more data structure accesses, more arithmetic, and more allocation, while the latter requires more allocation. The C code does not have any overhead penalties for data structure accesses: no null checks, no array bounds checks, and no type checks. The Java code, on the other hand, makes heavy use of Java container classes. Container class accesses will introduce at a minimum a nullcheck, and sometimes an array bounds check or a type check. Thus we expect that the amount of extra work during longer iterations will result in greater penalties for Java. Yet this correlation shows this not to be the case: longer iterations do not penalize Java any more than they penalize C, indicating that either many of the checks can be eliminated by the Fiji VM compiler or else that they simply do not cost much. For a more detailed look, we plotted the Java iteration execution time versus the number of various checks in Figures 5, 6, and 7. The number of checks per iteration were measured using RTBx instrumentation. These figures show that the benchmark's iterations have six distinct "modes" – i.e. the number of checks executed is fairly discretized. This likely corresponds to the number of suspected collisions (since there are six planes, we would expect six such modes). Two additional conclusions can be drawn from these figures. First, even though the code is dominated

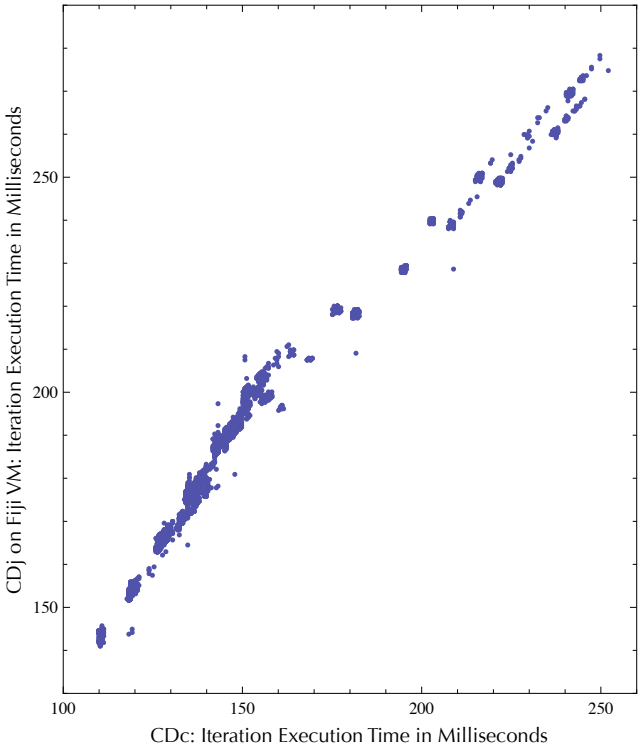

Figure 4. Execution time correlation between CDc and CDj.

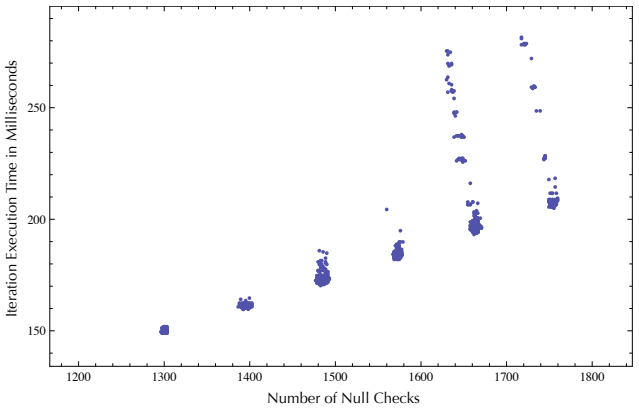

Figure 5. Correlation between number of null checks in an iteration and that iteration's execution time in CDj.

by operations that would appear to cause safety checks, the rate at which these checks are executed is quite small. Second, for longer iterations the execution time can vary by a lot even though the number of checks remain the same, indicating that safety checks are not much of a factor in execution time. For example: null checks, which are the most frequent, are only executed at most at a rate of about ten per millisecond. A null check that does not fail results in nothing but a branch at the machine code level (and indeed, in CDj none of the safety checks ever fail). This means that the presence of Java's null checks only adds about one additional branch

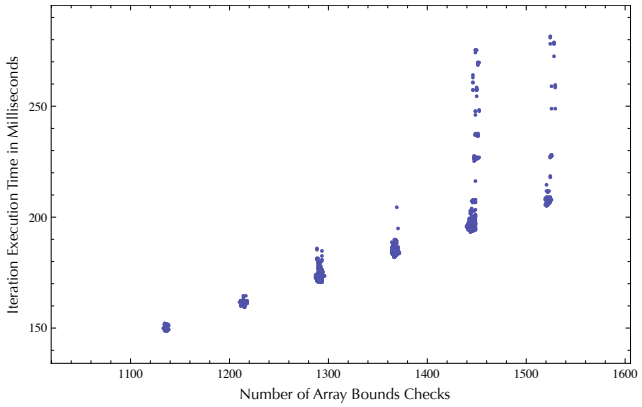

Figure 6. Correlation between number of array bounds checks in an iteration and that iteration's execution time in CDj.

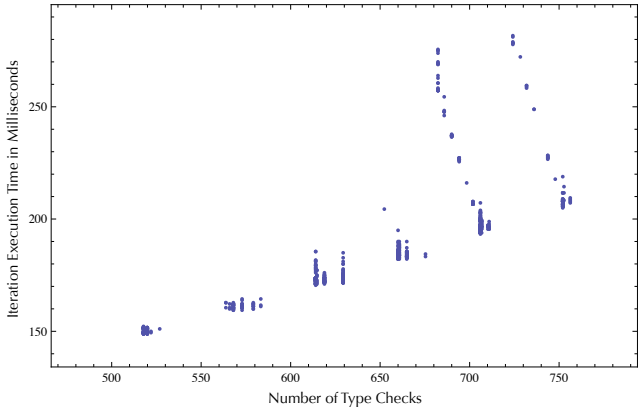

Figure 7. Correlation between number of type checks in an iteration and that iteration's execution time in CDj.

per 100 microseconds of execution on a 40MHz processor, which adds up to about one branch every 4,000 cycles. This implies that Java's safety guarantees are not a major factor in performance for this program. However, we do see that Java is indeed slower than C and suspect that this is due to the overall performance degradation from the sum total of all of Java's overheads, including ones we were unable to study in greater detail: for example the Henderson frames used for accurate stack scanning, sync-points, procedure call overheads, etc.

4.3 Java VM Comparison

In this experiment we compare WebSphere SRT, Hotspot Client, Hotspot Server and Fiji VM using the same benchmark but running on a multicore. CDj 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 CDj and were run with default options. The goal of the experiment is to have a rough idea of the difference in predictability between WebSphere and Fiji, and in per-

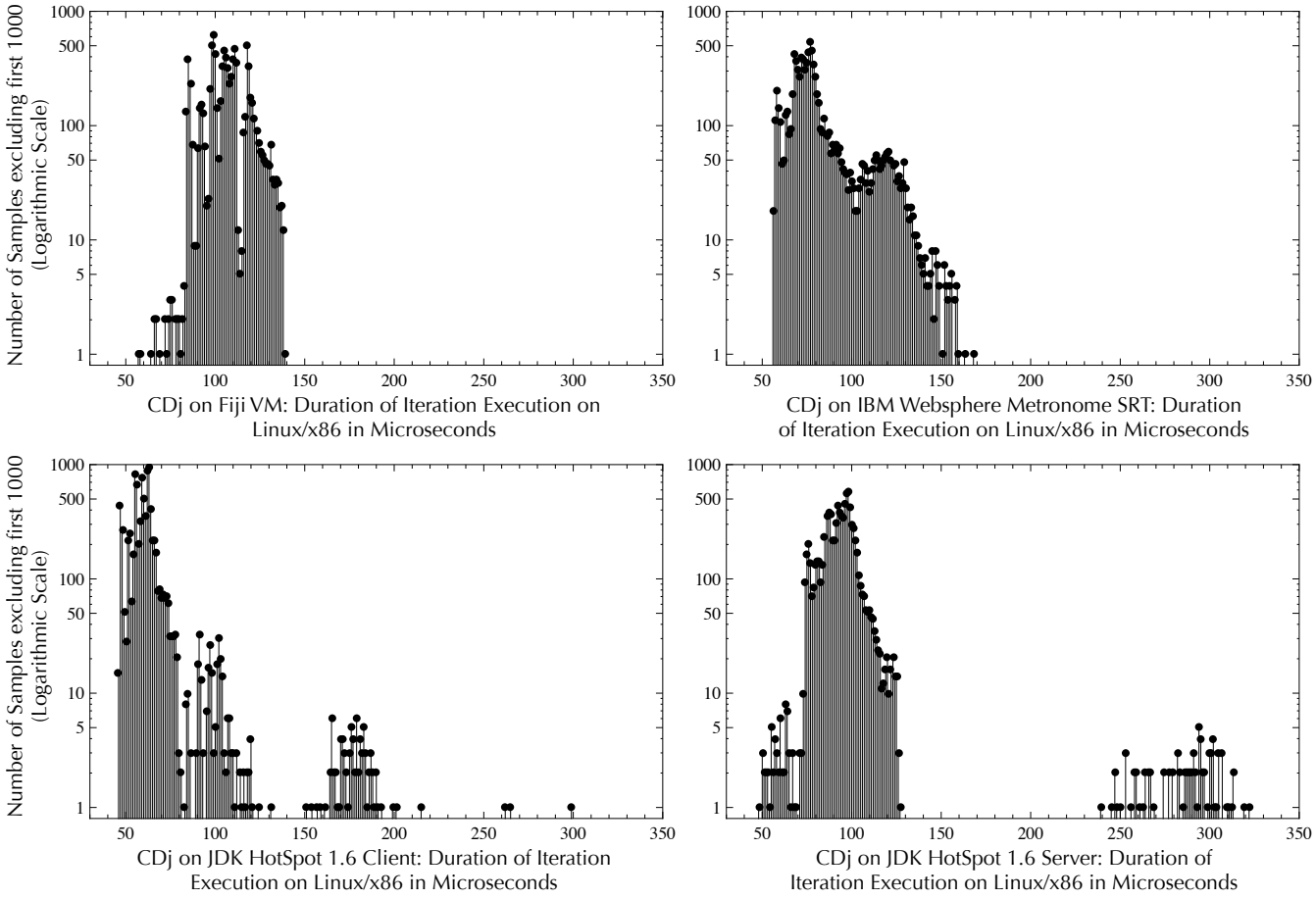

Figure 8. Histograms of iteration execution times for CDj on Linux/x86. Fiji VM achieves the best observed worst-case and has the tightest distribution of execution times – representing the best predictability. WebSphere SRT has a slightly better average performance but with a larger variance. Hotspot client and server have the best average-case performance but exhibit poor worst-case performance with outliers above 300 microseconds. We exclude the first 1,000 iterations from the measurements to avoid initialization bias.

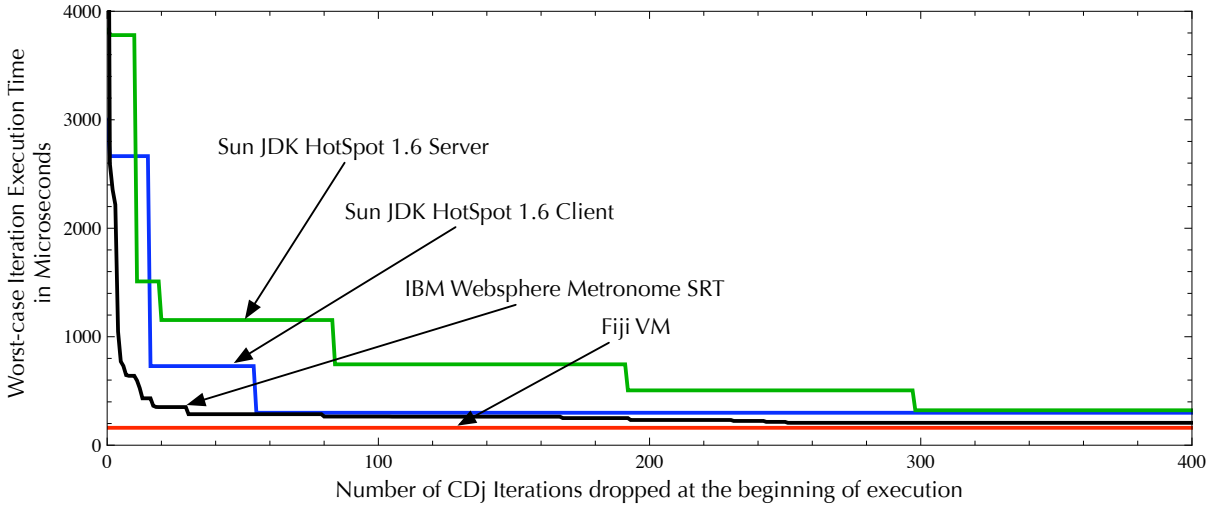

Figure 9. Start-up costs. The Y-axis shows the worst-case observed execution time, while the X-axis shows iterations dropped from the 10,000 performed for each VM. The initial iterations are dominated by the just-in-time compiler. This is particularly true for Hotspot server. WebSphere also has a JIT but it is tuned to stabilize faster. In comparison, the Fiji VM does not suffer from start-up jitter. If roughly 300 or more iterations are dropped, the JIT-based systems have a worst-case that approaches Fiji VM's. At that point the worst-case is dominated by garbage collection where the Fiji VM performs well due to its fully concurrent and slack-based collection strategy.

formance between real-time VMs (WebSphere and Fiji) and production throughput optimized VMs (Hotspot).

The histogram of Fig. 8 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 Fiji and client is 4.7% faster. This is to be expected as it does not emphasizes predictability. The worst observed case is more important for real-time developers. There Hotspot performs between 185% and 200% worse than Fiji, these difference are caused by garbage collection pauses. As for the comparison with WebSphere, Fiji has an observed worst-case that is 4% better than WebSphere but run 15% slower on average. Fiji VM has the tightest distribution (i.e. least deviation from peaks to valleys) of any virtual machine for this benchmark.

In many real-time applications start-up behavior is important. Fig. 9 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 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 has no start up jitters as it is an ahead-of-time compiler.

5. Conclusion

Programming hard real-time embedded devices is particularly demanding as code must be both predictable and efficient. As the size of embedded code bases rises, ensuring the correctness and safety of critical programs written in low-level languages such as C is becoming increasingly difficult. While high-level languages such Java and C# have been advocated as being better suited for developing large software systems, their suitability for embedded development is still being debated.

This paper highlights the overheads of using Java for hard real-time development by contrasting the performance of a representative benchmark written in C and Java on a LEON3 processor with the RTEMS operating system. The Java program is executed by a new Java virtual machine, the Fiji VM, which compiles Java ahead-of-time to C and which has runtime system that includes a real-time garbage collector. The results are encouraging as the throughput overhead of Java is about 30% and, more importantly, the worst observed execution time of the Java program is only 10% higher than that of the corresponding C program. The implication of these results is that Java, even with the presence of a garbage collector, can be used in many hard real-time contexts.

Acknowledgments. The authors thank Tomas Kalibera and Gaith Haddad for their work on the CDx benchmark, the anonymous reviewers and Gilles Muller for their helpful comments.

References

[1] ARMBUSTER, A., BAKER, J., CUNEI, A., HOLMES, D., FLACK, C., PIZLO, F., PLA, E., PROCHAZKA, M., AND VITEK, J. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS) 7*, 1 (2007), 1–49.

[2] AUERBACH, J. S., BACON, D. F., BLAINEY, B., CHENG, P., DAWSON, M., FULTON, M., GROVE, D., HART, D., AND STOODLEY, M. G. Design and implementation of a comprehensive real-time Java virtual machine. In *International conference on Embedded software (EMSOFT)* (2007), pp. 249–258.

[3] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collecor with low overhead and consistent utilization. In *Symposium on Principles of Programming Languages (POPL)* (Jan. 2003).

[4] BAKER, H. G. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Notices 27*, 3 (1992), 66–70.

[5] BAKER, J., CUNEI, A., KABILERA, T., PIZLO, F., AND VITEK, J. Accurate garbage collection in uncooperative environments revisited. *Concurrency and Computation: Practice and Experience* (2009).

[6] BLACKBURN, S., AND HOSKING, A. Barriers: friend or foe? In *International Symposium on Memory Management (ISMM)* (2004), pp. 143–151.

[7] BLACKBURN, S. M., AND MCKINLEY, K. S. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Conference on Programming Language Design and Implementation (PLDI)* (2008), pp. 22–32.

[8] BOLLELLA, G., DELSART, B., GUIDER, R., LIZZI, C., AND PARAIN, F. Mackinac: Making HotSpot real-time. In *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)* (2005), pp. 45–54.

[9] BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., AND TURNBULL, M. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[10] BRUNO, E., AND BOLLELLA, G. *Real-Time Java Programming: With Java RTS*. Addison-Wesley, 2009.

[11] CLICK, C., TENE, G., AND WOLF, M. The pauseless GC algorithm. In *International Conference on Virtual Execution Environments (VEE)* (2005), pp. 46–56.

[12] CORSARO, A., AND SCHMIDT, D. The design and performace of the jRate Real-Time Java implementation. In *The 4th International Symposium on Distributed Objects and Applications (DOA'02)* (2002).

[13] HENDERSON, F. Accurate garbage collection in an uncooperative environment. In *Proceedings of the ACM International Symposium on Memory Management* (Feb. 2002), vol. 38, ACM, pp. 256–263.

[14] HENRIKSSON, R. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University, July 1998.

[15] KALIBERA, T., HAGELBERG, J., PIZLO, F., PLSEK, A., AND BEN TITZER AND, J. V. Cdx: A family of real-time Java benchmarks. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)* (September 2009).

[16] KALIBERA, T., PIZLO, F., HOSKING, A., AND VITEK, J. Scheduling hard real-time garbage collection. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)* (Dec. 2009).

[17] KALIBERA, T., PROCHAZKA, M., PIZLO, F., VITEK, J., ZULIANELLO, M., AND DECKY, M. Real-time Java in space: Potential benefits and open challenges. In *Proceedings of DAta Systems In Aerospace (DASIA)* (2009).

[18] KRALL, A., VITEK, J., AND HORSPOOL, N. R. Near optimal hierarchical encoding of types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (Jyvaskyla, Finland, June 1997).

[19] MULLER, G., AND SCHULTZ, U. P. Harissa: A hybrid approach to Java execution. *IEEE Software 16*, 2 (1999).

[20] NILSEN, K. Garbage collection of strings and linked data structured in real time. *Software, Practice & Experience 18*, 7 (1988), 613–640.

[21] PIZLO, F., AND VITEK, J. An empirical evalutation of memory management alternatives for Real-time Java. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)* (Dec. 2006).

[22] PIZLO, F., AND VITEK, J. Memory management for real-time Java: State of the art. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)* (Orlando, FL, May 2008).

[23] PIZLO, F., ZIAREK, L., AND VITEK, J. Towards Java on bare metal with the Fiji VM. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)* (September 2009).

[24] PROEBSTING, T., TOWNSEND, G., BRIDGES, P., HARTMAN, J., NEWSHAM, T., AND WATTERSON, S. Toba: Java for applications – A way ahead of time (WAT) compiler. In *Conference on Object-Oriented Technologies and Systems* (1997).

[25] PUFFITSCH, W., AND SCHOEBERL, M. Non-blocking root scanning for real-time garbage collection. In *International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)* (2008), pp. 68–76.

[26] SCHOOFS, T., JENN, E., LERICHE, S., NILSEN, K., GAUTHIER, L., AND RICHARD-FOY, M. Use of PERC pico in the AIDA avionics platform. In *International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)* (2009), pp. 169–178.

[27] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers 29*, 9 (Sept. 1990), 1175–1185.

[28] SHIVERS, O. Control flow analysis in scheme. In *Conference on Programming Language design and Implementation (PLDI)* (1988), pp. 164–174.

[29] SIEBERT, F. The impact of realtime garbage collection on realtime Java programming. In *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)* (2004), pp. 33–40.

[30] SIEBERT, F., AND WALTER, A. Deterministic execution of Java's primitive bytecode operations. In *Java Virtual Machine Research and Technology Symposium (JVM)* (2001), pp. 141–152.