Real Time Java on resource-constrained platforms with Fiji VM

Filip Pizlo Lukasz Ziarek Jan Vitek
Department of Computer Science, Purdue University, W. Lafayette, IN, 47909
Fiji Systems, LLC, Indianapolis, IN 46202.
E-mail: {fil,luke,jan}@fiji-systems.com

Abstract

Real-time Java is quickly emerging as a platform for building safety-critical embedded systems. The real-time variants of Java, including [8, 15], are attractive alternatives to Ada and C since they provide a cleaner, simpler, and safer programming model. Unfortunately, current real-time Java implementations have trouble scaling down to very hard real-time embedded settings, where memory is scarce and processing power is limited. In this paper, we describe the architecture of the Fiji VM, which enables vanilla Java applications to run in very hard environments, including booting on bare hardware with only very rudimentary operating system support. We also show that our minimalistic approach delivers comparable performance to that of server-class production Java Virtual Machine implementations.

1. Introduction

Scaling modern, mainstream languages and their associated runtimes down to power-constrained embedded settings is difficult, especially if hard real-time guarantees must be ensured. While Java is emerging in a wide variety of real-time application domains, the availability of standards-compliant Java implementations for embedded systems is limited, especially if automatic memory management is desired. In this paper, we explore the areas where existing Java implementations are lacking, and report on the work completed so far on a brand new, clean-room implementation of Java – the Fiji VM. In this paper we present the following five contributions:

1. To our knowledge, the **first high-performance real-time Java implementation that is small yet robust enough to boot from bare metal** through integration with the RTEMS [18] real-time kernel, while retaining the ability to run vanilla Java code. In particular, we show performance results of our VM running the SPECjvm98 benchmark suite, with real-time garbage collection, on bare hardware.

2. To our knowledge, the **first high-performance real-time Java implementation that can run bottom-half interrupt handling routines written in Java**, *while allowing those routines to use heap allocation*. We show that the performance of these routines is unaffected by garbage collection.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JTRES'09, September 23-25, 2009 Madrid, Spain
Copyright 2009 ACM 978-1-60558-732-5/09/9 ......$10.00.

3. A **survey of limitations Java virtual machines face when targeting hard real-time systems**. We discuss our previous experience embedding Java virtual machines and identify six key goals state-of-the-art virtual machines should address to be better suited for hard real-time and mission critical application domains.

4. A **concise description of the Fiji VM** compiler and runtime system. We describe the unique features of the Fiji VM compiler and runtime system and their implementation. We address each of the six goals in the implementation and design of the Fiji VM.

5. A **performance comparison** between the Fiji VM and state-of-the-art, server-class Java VMs. We also include performance number of the SPECjvm98 benchmark suite running on bare hardware.

1.1 Background

Over the past seven years, our group at Purdue has been developing the Ovm [2, 22] – a Java-in-Java metacircular virtual machine that aims to provide hard real-time guarantees, while additionally complying with a large subset of the Java language. Development of Ovm has resulted in the first UAV flight using avionics written in Java [2, 23] and the first *open-source* real-time garbage collector with high throughput and good mutator utilization [21]. But, much like other Real Time Java implementations, Ovm still lacks functionality in key areas that are essential for deeply embedded safety-critical systems. Application domains that are of particular interest include *space flight*, which requires running on very minimal radiation-hardened hardware with a small real-time OS kernel; *avionics*, which utilize extensive mixed-criticality support and adhere to guidance given in DO-178B; *medical devices*, especially implanted ones, which often have a fraction of a megabyte of memory and stringent certification requirements; as well as type-safe *operating system support* – the ability to write operating system components with the safety and security guarantees that come with type soundness of Java but with the expressive power of C. Within these application domains, we identified the following key areas where existing Real Time Java implementations need improvement.

Broad architecture support. Ovm is portable – we have demonstrated it running on ARM, SPARC, LEON2 and LEON3, x86, and PowerPC. However, porting to each of these architectures requires a significant time investment. Even though the Ovm compiler generates mostly-portable C code, its optimizations make heavy use of platform-specific assumptions. For a VM to be truly portable, we would like to see it make *no* platform-specific assumptions, and consist entirely of pure, portable ANSI C code.

Concurrency and parallelism. Ovm is uniprocessor-only and other Real Time Java implementations (such as [1, 3, 24, 27]) have varying levels of multiprocessing support. We are interested in a VM that comes with multiprocessor support out of the box, and achieves excellent performance and scalability no matter how many

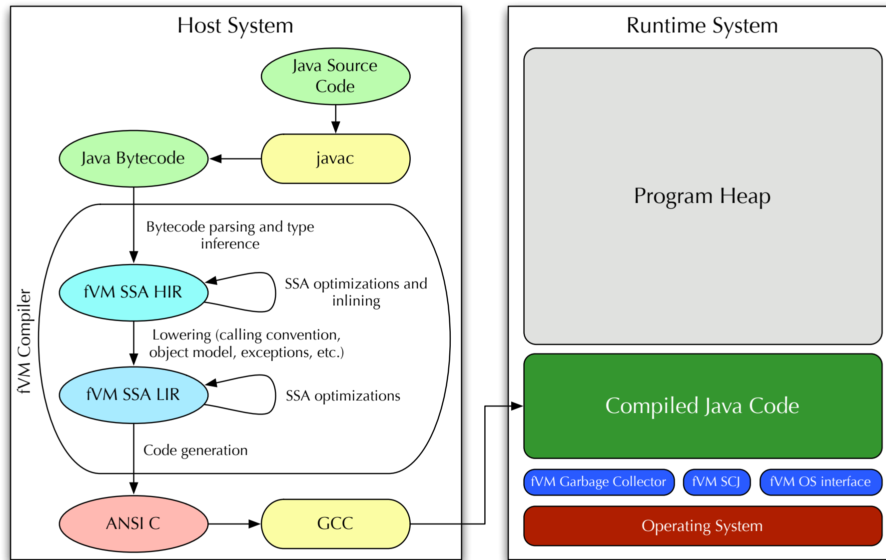


Figure 1. Fiji VM architecture.

processors or threads are in use. This is particularly important with the current hardware trends.

Library independence. Current Real Time Java virtual machines tend to be locked to a particular class library, making the selection of VM harder. For example, product X may have the best performance on a developer's target platform, but may require a library that is too large for successful deployment. We would like a VM that comes with multiple library choices, enabling developers to choose a footprint target without having to change infrastructures.

Control only that, which needs to be controlled. Ovm, much like [24], expects to be able to override scheduling decisions made by the operating system. This has at least three undesirable side-effects. First, it impedes multiprocessing support. While it does not prevent multiprocessors from being utilized, it makes the job much harder. Second, it causes an increase in complexity that can lead to difficulty in achieving certification, for example under DO-178B. Since supporting VM scheduling requires duplicating much of the scheduling logic in the operating system, the resulting runtime has difficult-to-characterize complexities that may lead to undesirable behavior. Proving otherwise to a certification authority will be an unwelcome burden. Third, and perhaps most importantly, having a VM make assumptions about scheduling behavior makes it more difficult, and in Ovm's case, impossible, to run Java code in the bottom half of an operating system.

Bottom-half support. Few managed-code systems allow for type-safe code to run in the bottom-half of an operating system. Yet, such a feature would not only be useful but seems on the surface to be a good fit for embedded and real-time systems. Why not leverage type safety when writing interrupt handlers? The Singularity operating system [17] supports such a notion, but unfortunately, Singularity is built around C#, not Java, and thus lacks the hard real-time support found in existing Real Time Java implementations.

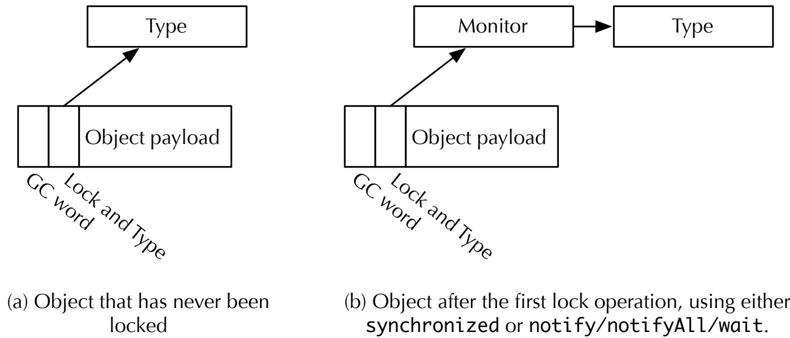
Simplicity. We strive to have a Java implementation that can be DO-178B certified. Certification is more easily achieved if the implementation is simple and easily understood by a third party. Even if certification is not the ultimate development goal, simplicity has many benefits such as increases in performance and stability, but also confidence: developers like to use tools they can understand.

Speed. Last but certainly not least, Java code *must* run fast. Predictability is paramount – but speed is no less important. Consider that a predictably slow implementation is not much better than an unpredictably fast one, if the latter's worst-case performance is no worse than that of the former. While a Real Time Java implementation will always take a performance hit compared to those VMs that are strictly throughput-optimized, we would like the gap to be as small as possible.

With these requirements as our core development principles, we set out to design and implement a new Java implementation from scratch. We call it the Fiji VM, and the remainder of this paper discusses its design, notable aspects of our implementation strategy, and performance.

2. Fiji VM Design

Fiji VM, or fVM for short, is an ahead-of-time compiler for Java bytecode programs. Our goal with fVM is to provide developers with an automated mechanism for converting high-level Java applications into small and compact real-time executables. Fiji VM consists of a compiler, a runtime library, and a choice of open-source class libraries. In addition to supporting full Java 5 (sans class loading) and JNI 1.4, Fiji VM also has some unique features: on-the-fly concurrent real-time garbage collection based on Immix [7], and stack allocation. Our stack allocation support, described in detail in Sec. 2.2.2, allows for a high-performance easy-to-use alternative to scoped memory. The sections that follow describe the Fiji compiler, runtime, and libraries in detail.



(a) Object that has never been locked

(b) Object after the first lock operation, using either `synchronized` or `notify/notifyAll/wait`.

Figure 2. Fiji VM object model. Objects have two headers: a GC data word, and a Lock and Type word. Both words are sized according to the pointer size on the given architecture (we support both 32-bit and 64-bit architectures). The GC data word's format is shown in Fig. 3. The Lock and Type word may either point to a Type record or to a Monitor. The Type lives in the `data` section of the executable and contains the vtable, interface table, all data required for type inclusion tests, and everything needed to support reflection. When an object is locked for the first time, a Monitor is allocated in the heap; a pointer is installed in the Monitor to point to the object's Type; and the Lock and Type word is compare-and-swapped to point to the Monitor. Because the Type record has a pointer to itself at the same offset where the Monitor has a pointer to the Type, this means that retrieving the Type of an object can always be done with just two indirections and without branching or further computation.

2.1 Fiji VM Compiler

Fiji VM parses bytecodes in the Java 1.6 or earlier format from which it generates high-performance ANSI C[1]. The generated C code is then automatically passed to the chosen C compiler for the target platform. Typically, we use a recent variant of GCC as our C compiler of choice. Using C as a target language carries both benefits and dangers. Among the biggest benefits are portability and the ability to leverage already existing high-performance compiler backends; among the biggest dangers is the mismatch between bytecode and C semantics. Additionally, the C compiler cannot perform some high-level optimizations as effectively on C code generated from Java as a Java-optimized compiler could. Such optimizations include control flow optimizations, such as exception optimizations; locking and memory optimizations; and inlining. Thus, the Fiji VM compiler does extensive high-level optimizations prior to generating C.

2.1.1 Dangers of generating C from Java bytecode

Since the C language semantics are more loosely defined than those of Java bytecode, the fVM compiler must bridge this semantic mismatch. Consider whether the expression $x < x + 5$, where x is an integer, is always true. A C compiler is free to assume that this is the case, because the result of overflowing on $x + 5$ is undefined. However, in Java the compiler cannot make that assumption, because $x + 5$ is strictly defined as being equal to $(x + 5) \bmod 2^{32}$ – i.e. the wrap-around on overflow is mandatory. Of course, there are more subtle semantic differences, even just for integer arithmetic. For example, what does $(-2^{31}) \div (-1)$ yield under signed 32-bit integer arithmetic? In Java, the result is guaranteed to be -2^{31}, while in C it may yield -2^{31} or an arithmetic exception, depending on the optimizations that the C compiler performs. The differences between Java and C semantics become even more subtle and complicated with floating point arithmetic. Therefore, any bytecode-to-C compiler cannot simply generate the naïve C equivalent for the input bytecode; it must exercise additional caution. In the case of Fiji VM, this includes a combination of out-of-line helper functions for tricky arithmetic operations, which perform additional checks, and the careful use of C compiler flags to turn off optimizations known to cause problems.

[1] We have also successfully tested Java 1.7 bytecode, but since Java 1.7 is still a moving target, the final release may include bytecode features or pathologies that require unforeseen changes in our system.

2.1.2 Garbage collection and scoped memory support

The Fiji VM compiler has extensive support for garbage collection and scope check barriers, including those barriers necessary to implement common anti-fragmentation techniques such as [24] or [20]. Additionally, we use a combination of techniques from [6] and [11] to enable accurate stack scanning of generated C functions. In Fiji VM, these techniques have been both enhanced with more aggressive compiler optimizations, and extended to allow for portable and accurate stack traces for exception handling.

2.1.3 High-level optimization

As shown in Fig. 1, the Fiji VM compiler performs a variety of optimizations, at multiple stages of compilation, using a static-single-assignment (SSA) intermediate representation. The optimizations currently performed include:

- *Inlining*
- *Devirtualization* – turning virtual calls into direct calls
- *Virtualization* – turning interface calls into virtual calls
- *Copy propagation*
- *Constant folding*
- *Tail duplication*
- *Lock elision* – if a lock is known to already be held, don't generate code to recurse on it; additionally, don't lock objects that are known to be thread-local
- Intra-procedural and whole-program *type propagation*, with 0CFA
- *Null check elimination*
- *Array bounds check removal*

Unsurprisingly, inlining is the most profitable of these optimizations. In particular, we have found it to be most profitable when applied only to the smallest methods. For non-recursive methods whose bodies are smaller than a callsite, we inline in a fixpoint; for other methods we inline based on a variety of factors, 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 is inlining only non-recursive

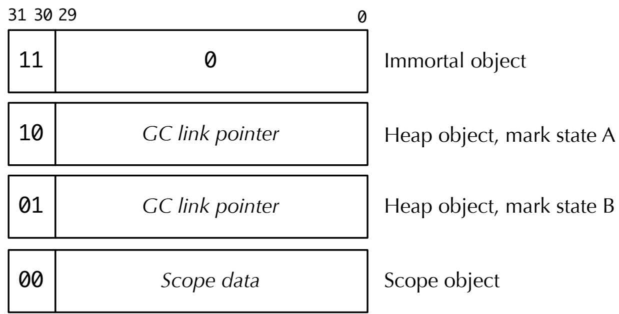


Figure 3. The GC data word. As shown in Fig. 2, this is the first word of the object header. In this figure, we show the GC word as it looks on 32-bit platforms. The highest-order two bits indicate the state, and the remaining bits contain the payload information. It may be in any of four states: one state for immortal objects, such as those allocated in the `data` section of the executable (these include string and class literals), or objects manually allocated in the Immortal space. For these, the payload is always zero. Two states are reserved for heap objects, and represent the two mark states of objects; in those states, the payload will contain a link pointer to the next object on the current GC queue. Finally, one state is reserved for scope objects, in which case the payload contains information about the scope.

methods whose bodies are smaller than the code we would generate for a callsite, we reap most of the advantage of our more complicated approach.

In our implementation of fVM optimizations, we have noted 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 0CFA, in which all points-to sets are represented using a single machine word. The points-to set uniquely identifies an allocation site, an exact type, or in the worst case, a type upper bound. This simple analysis converges very rapidly (20 seconds for all of SPECjvm98 with the full GNU Classpath 0.97.2 [9]), and appears to be quite precise, based on our current performance statistics. However, a thorough comparison of this algorithm's performance relative to other scalable 0CFA algorithms has not yet been made.

2.1.4 Object model

The Fiji VM uses a two pointer-word header and a contiguous object model. All object pointers point to the first array element, or to four bytes past the first field if the object is not an array; this approach, also used in Jikes RVM [12] results in simplified array accesses. As shown in Fig. 2, the two words in the header are the GC data word, and the Lock and Type word. The GC data word, shown in detail in Fig. 3, holds all information necessary to support the Real Time Specification for Java's memory areas - one state is reserved for immortal memory, two states for heap memory (representing the two mark states in the garbage collector), and one state for scoped objects. We use the two high-order bits to represent the state. This has some interesting properties. Given a `gcword`, without loss of generality, we outline the properties for 32-bit systems:

- If the `gcword` comes from an object not in scoped memory, given a `markedState` where the `markedState` is either the bits `01` or `10` and indicates the expected state of currently marked objects,

  ```
  ((gcword >> 30) & markedState) != 0
  ```

  implies that the object should be kept alive, and the converse implies that the object is dead. Note that since the garbage collector never sees scoped objects except in a specialized scan (i.e. it will never stumble upon a scoped object by following a pointer from a heap or immortal object), this test is used extensively in the collector to quickly identify whether the object is known to be alive or suspected to be dead.
- `gcword < (1<<30)` implies that the object resides in scoped memory.
- `((gcword >> 30) & 3) == 3` implies that the object resides in immortal memory.
- `((gcword >> 30)-1)` $<_{\text{unsigned}}$ `2` implies that the object resides in the heap.
- For scoped memory systems in which cactus scope stacks are not allowed (i.e. the scope hierarchy is linear), such as should be the case in Reflexes, Safety Critical Java, and when performing stack allocation, the payload for the scope state simply contains a scope ID such that

  ```
  scopeID1 >= scopeID2
  ```

  implies that the object corresponding to scopeID1 has a longer lifetime than the object corresponding to scopeID2. If GC is disabled, then the store check is just

  ```
  gcword1 >= gcword2
  ```

  since `gcword` $= 3 * 2^{30}$ implies that the object is immortal, and this value is greater than the GC words for scopes. If the GC is enabled, the scope check is augmented with arithmetic that sign-extends the two high-order bits to fill the entire word; i.e. any GC word for which

  ```
  ((gcword >> 30) & 3) != 0
  ```

  is replaced with $2^{32} - 1$.

The GC data word is followed by a Lock and Type word, which may either point to a Monitor object (the Java equivalent of a lock), or a Type record. The first word of both a Monitor object and a Type record is a pointer to the Type record. The Type record thus points to itself. This means that retrieving the Type from an object can always be accomplished by first loading from the Lock and Type word, and then immediately loading from the pointer. This approach allows us to closely mimic the performance of thin locks [5], while ensuring predictability: the Monitor is inflated on first use; this inflation step is very cheap (it's just a pointer-bump allocation in the Monitor space). The Monitor contains OS-specific locking data structures. On platforms that support it, the compiler inlines the locking fast path such that the lock operation becomes a cheap compare-and-swap. This gives us performance that is close to what throughput-optimized systems achieve.

2.1.5 Type inclusion and interface dispatch

The Fiji VM Type 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 object-oriented systems; the Type 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 unpredictable behavior. In Fiji VM, both operations are guaranteed constant-time. In the case of type inclusion,

we use type displays [28] 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.

2.2 Fiji VM Runtime

The Fiji VM runtime is light-weight; it contains two components: the memory manager (scopes and optional 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 library, discussed in Section 2.3, has its own OS abstractions for I/O and whatever other facilities the library may choose to support.

2.2.1 Garbage Collection

Fiji VM currently supports an optional Immix-style on-the-fly concurrent real-time garbage collector. This collector can be run in either a purely concurrent, or a purely slack-based mode. In this system, objects are allocated in either a bump-pointer fashion or first-fit fashion; in most cases, bump-pointer is used. Garbage collections fall into two categories: minor and major. In minor garbage collections, which are performed opportunistically at high frequency, only entirely empty pages are freed. Partially full pages are ignored. This style of collection is very fast, and for typical Java programs, frees large amounts of space very rapidly. A major collection places the empty holes in partially free pages on a freelist, which is then used up in a first-fit or bump-pointer fashion, depending on the size of the hole. While this approach does not support defragmentation, it is 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. 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 are working on an alternative collector, which will add concurrent copying; however this collector is not yet stable and we leave its evaluation to future work.

2.2.2 Stack Allocation

Attempts at supporting manual memory management, such as those in the Real Time Specification for Java, are often maligned for being too complex. In Fiji VM, we offer a simple alternative: stack allocation. Any method may be marked with the `@StackAllocation` attribute; as a result, any uses of the `new` keyword will result in the object being allocated in that stack frame. Additionally, we have a `@AllocateAsCaller` attribute, which indicates that the method should allocate the same way as its caller; i.e. either allocate in the same stack frame as the caller if the caller was marked `@StackAllocation`, or in whatever memory area its caller was allocating in if it was not marked `@StackAllocation`. In particular, if a method without either annotation is called from a method marked `@StackAllocation`, then this method will allocate in the memory area determined by the Real Time Java library.

Stack allocation is not statically checked; instead, we have run-time store checks (cannot store a stack-allocated object into the heap, into a stack-allocated object at a lower stack frame, or into a static field), run-time return checks (cannot return an object allocated in the current stack frame), and run-time throw checks (can only throw heap- or immortal-allocated objects).

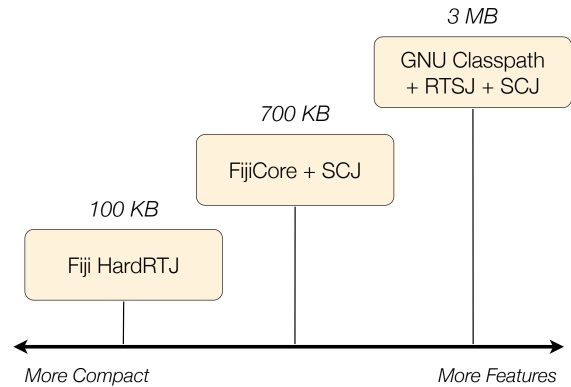


Figure 4. Fiji VM libraries. Three libraries are in the works; the most feature-rich option, GNU Classpath, and the middle option, FijiCore, are already operational.

```
final Timer t=new Timer();
t.fireAfter(10 /* ticks */,new Runnable(){
   public void run() {
      ...
      t.fireAfter(10, this); // next iteration
   }
});
```

Figure 5. A timer tick interrupt handler written in Java. In this code snippet, the body of the `run()` method runs in the bottom half, in response to a hardware timer interrupt. This method may arbitrarily allocate memory, and use all Java features except for locking (which raises an exception to protect against deadlock). Additional APIs are provided to disable interrupts, which provides for the traditional approach for synchronizing between interrupt handlers and code executing in threads.

Just as we have an option to disable garbage collection, we can also disable RTSJ memory area support – in that case, stack allocation is still enabled, but the runtime checks are significantly optimized since the stack frame hierarchy is linear. Though we have not experimented with stack allocation in applications, much of our runtime and libraries is written using stack allocation. We find it convenient in a number of respects. First, stack-allocated objects cannot move, no matter the garbage collection style. Thus, stack-allocated byte array, or example, may be passed to system calls directly. Second, because stack-allocation can be enabled adhoc with just an annotation, we found it easy to simply place these annotations wherever we needed them – it required very little planning and in most cases, the code just worked. By contrast, with scoped memory, one has to first identify when and were to create a scope, how to share it between threads, and how to allocate the `Runnable` instance when entering it. None of these concerns exist when using stack allocation.

2.2.3 Threading and Locking

Fiji VM's threading and locking implementations are simple: we pass all of the hard work to the underlying operating system. 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.

2.3 Library Support

We currently support two different class libraries, with plans to add a third. The three tier approach looks as shown in Fig. 4. The largest library, with the most feature support is GNU Classpath. Currently we use Classpath 0.97.2. With this library, we plan to add the open-source Purdue RTSJ and SCJ (Safety Critical Java) implementations [22]. For applications that still desire most of Java's features but want a smaller footprint, we offer FijiCore along with the SCJ implementation. FijiCore is a library that we are developing, and plan to release as open-source using the same license as Classpath. It includes code from both Classpath and OpenJDK [26], but is stripped down to be a slight superset of the Java ME Connected Device Configuration. A further down-sized library, called Fiji HardRTJ, will also be made available, which will follow the Java ME Connected Limited Device Configuration with some custom add-ons for real-time support. Currently, our FijiCore library is large enough to run benchmarks like SPECjvm98, and we use it as our main testing library. With this library, the footprint of typical small programs ends up being approximately 700KB when running on bare hardware; this includes the OS kernel, OS interface libraries, FijiCore library, the user's code, and global data.

In addition to supporting standard APIs, all Fiji VM libraries support additional functionality such as stack allocation (discussed in Sec. 2.2.2) as well as a low-level OS interface layer for writing interrupt handlers in Java. This API allows, among other things, pure heap-allocating Java code to run in bottom-half interrupt context inside the operating system. See Fig. 5 for sample code. When running in interrupt context, Java code may do any thing supported by Java except:

- Locking. Attempting to lock throws an exception to prevent deadlock.
- System calls. System calls are not allowed, and also throw an exception. However, native calls that do not result in system calls are allowed.

Additional APIs are provided for disabling and re-enabling interrupts in a block-structured fashion, to allow for synchronization between interrupt handlers and non-interrupt code. Note that on multicore systems, these APIs would have to be used in conjunction with a either a spin-lock or the use of thread-processor affinity; our APIs do not currently provide for either, but spin-locks can be trivially implemented using compare-and-swap, which our APIs do provide.

2.3.1 Portability

Fiji VM was designed with portability in mind, with support for many flavors of operating system and a wide array of architectures. Currently fVM can be executed on Linux, Darwin, NetBSD, and RTEMS. We have also successfully deployed the fVM on a range of architectures, including: x86, x86_64, ERC32, ARM, PowerPC, SPARC, and LEON. RTEMS offers a wide selection of BSP support which we hope to leverage and deploy fVM on a larger set of architectures. In particular, when running on RTEMS we leverage its already existing portability APIs, which enable us to run on exotic architectures with little effort. For example, our SPARC, ERC32, and LEON ports worked on the first try - we simply told RTEMS to target the respective BSPs.

3. Evaluation

Fiji VM has a number of unique features, which we evaluate in this section. Our evaluation covers four areas:

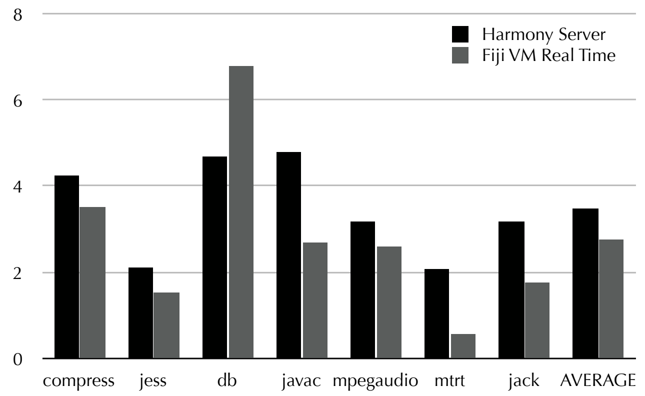


Figure 6. Execution time in seconds of Fiji VM versus Apache Harmony Server, after one iteration of execution. Lower numbers are better. Without warm-up, Harmony is 24% slower than Fiji VM.

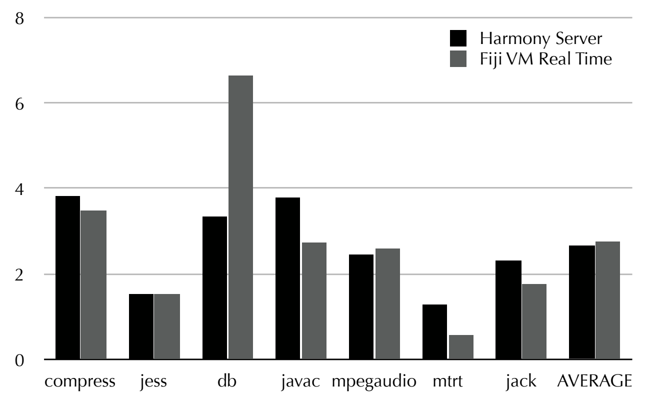


Figure 7. Execution time in seconds of Fiji VM versus Apache Harmony Server after two warm-up iterations. Lower numbers are better. With warm-up, Harmony performance improves by 30%; Fiji VM's performance improves by less than 1%. The lack of performance increase in Fiji VM is due to our concern for determinism rather than raw throughput. After warm-up, Harmony is 4% faster than Fiji VM.

1. Fiji VM's performance versus an open-source server-class virtual machine, to demonstrate the throughput of the Fiji VM. We utilize the standard SpecJVM98 benchmark suite comparing both cold and hot runs versus Apache Harmony Server.
2. Footprint. Fiji VM is an ahead-of-time compiler; this is sometimes thought to be an issue for systems in which memory is limited since bytecode is often smaller than machine code. Thus, we compare the size of the Fiji VM SPEC executable to the size of the fastest embedded VM we know of: phoneME Advanced MR2-b97 with JIT enabled.
3. Fiji VM's ability to boot Java from bare hardware. In this test, we use Fiji VM to create a boot CD out of the SPECjvm98 benchmark suite, and report performance results of running SPECjvm98 on bare metal.
4. Interrupt handling. We show the results from a test in which an interrupt handler that is written in Java allocates memory, thus stressing the GC. We show the performance of the interrupt handler with or without GC running in the background. Additionally, we run this in an ERC32 simulator with 4MB of RAM as an additional test of Fiji VM robustness.

	No GC	With GC
Max iteration duration	$915\mu s$	$913\mu s$
Average iteration duration	$648\mu s$	$605\mu s$
Number of iterations	932	5856

Table 1. Performance of Fiji VM when running an interrupt handler that allocates memory. Note that there is virtually no difference between the worst-case performance with or without GC running. In fact, the average performance is a bit better when the GC is running; this is due to a slightly different memory structure when collecting, which leads to faster allocation.

For (1), (2), and (3), we perform experiments on an 8-way Xeon 2.33 GHz with 8GB of RAM, running Fedora Core 10. All experiments were done with no other applications running. For (4), we run on the ERC32 emulator that comes with RTEMS 4.9; the emulator was running on the same machine as (1), (2), and (3).

3.1 Throughput

To evaluate the throughput of Fiji VM, we compare against Apache Harmony Server VM revision 761593 using the SPECjvm98 benchmark suite. We choose Harmony because it is a well-known, and reasonably well-understood, high-performance open source Java virtual machine. Though we do not compare to other JVMs (notably, we do not compare to other RT JVMs), we feel that including a comparison to a well-known, publicly available VM allows others to extrapolate a performance comparison to any other JVM. We use the SPECjvm98 benchmark suite because of its popularity, and because the full suite is amenable to ahead-of-time compilation (unlike Dacapo, in which only some benchmarks can be run in an ahead-of-time system).

All experiments were run with all 8 cores enabled; in the case of Fiji VM at most 3 cores were used (1 for the collector, and 2 for the raytrace threads in the `mtrt` benchmark). In both VMs, we limit the heap to 50MB, and as an additional test of Fiji VM's capabilities, we set the concurrent collector's trigger at 10MB. This means that in Fiji VM, *the collector is always running*.

The first experiment involved no warm-up. This is not an interesting case for traditional VM throughput experiments, but it is interesting for real-time applications, where cold-start times are important. The results are shown in Fig. 6. Notice that Harmony is 24% slower than Fiji VM in this case. The one benchmark where Harmony is faster is `db`. This is due to two artifacts of Fiji VM's real-time support: heavy barriers and a simpler, more conservative approach to locking. We have heavier barriers than Harmony since fVM is running in our full real-time configuration, which includes both GC store barriers and scope check barriers. Since `db` spends most of its time manipulating complex data structures, this takes its toll on performance. Additionally, `db` is a benchmark that makes heavy use of the Java `synchronized` statement. While our implementation of `synchronized` is quite fast, it is not as fast as what typically would be found in a non-real-time VM. This occurs because non-real-time VMs do not have to worry about the potentially catastrophic performance of unfair locks and its impacts on predictability. Thus, they are free to perform optimizations which are unacceptable in a hard real-time setting. It is interesting to note that Fiji VM significantly outperforms Harmony on `mtrt`, which is the only parallel benchmark in the SPECjvm98 suite. We are not sure what limitations in Harmony are preventing it from performing as well as Fiji VM, but we feel this result does suggest that Fiji VM itself does not introduce impediments to scalability.

The second experiment involves two warm-up iterations and is more typical of traditional throughput benchmarking. The first warm-up iteration is a full one, the second one involves a 10% execution of the benchmark. This gives Harmony the opportunity to optimize the benchmarks more thoroughly. In this case, it beats fVM by 5%. The fVM performance after warm-up is virtually unchanged: the difference is less than 1%, while Harmony speeds up by 30%. Again, `db` is the benchmark on which our performance is the worst. These benchmark indicates that fVM is a stable, consistent, real-time system capable of performance comparable to that of a server-class production VM with no real-time support.

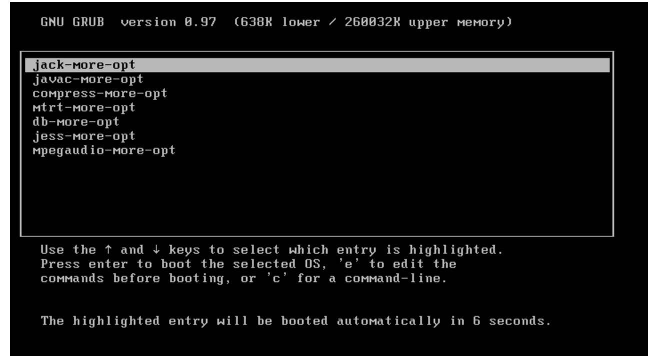


Figure 8. The GRUB menu used for booting SPECjvm98 from bare hardware.

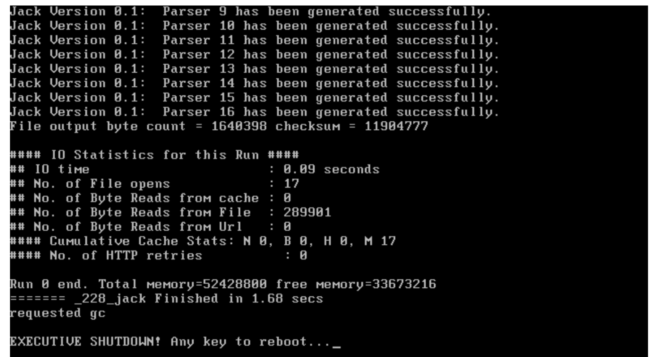


Figure 9. The SPECjvm98 `jack` benchmark running to completion on bare hardware.

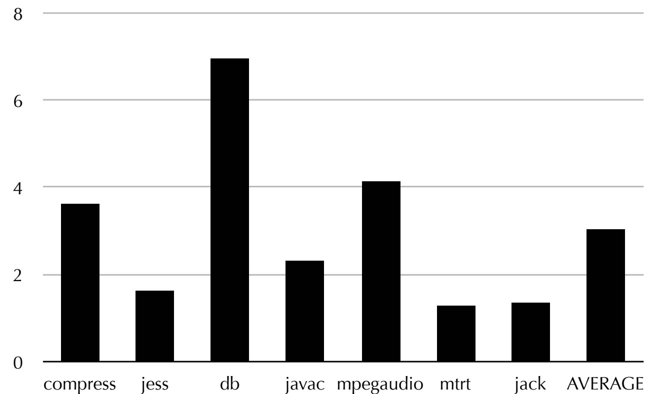


Figure 10. Execution time in seconds of Fiji VM running on bare hardware with the help of RTEMS 4.9. Lower numbers are better. Note that our performance is slightly worse on bare hardware, for two reasons: first, RTEMS 4.9 does not currently support SSE registers or instructions, and second, RTEMS is uniprocessor, thus making it harder for multithreaded programs to scale; in particular, the garbage collector ends up time-slicing with the application as opposed to running on a separate processor. As a result, performance on RTEMS is 10% worse than when running on Linux.

```java
import com.fiji.hardrtj.*;
public class testhardrtj {
    static Integer[] array;
    static double cyclesPerMS;
    static class Stats {
        int iterations;
        long maxDuration;
        long[] durations=new long[1000];
        int durationI;
        void add(long duration) {
            iterations++;
            if (duration>maxDuration) {
                maxDuration=duration;
            }
            durations[durationI++]=duration;
            if (durationI==durations.length) {
                durationI=0;
            }
        }
        double max() {
            return maxDuration/cyclesPerMS;
        }
        double avg() {
            long sum=0;
            for (int i=0;
                 i<Math.min(iterations,durations.length);
                 ++i) {
                sum+=durations[i];
            }
            return sum
                / ((double)Math.min(iterations,durations.length))
                / cyclesPerMS;
        }
        void report() {
            System.out.println("   Number of iterations: "+iterations);
            System.out.println("   Max duration:         "+max()+" ms");
            System.out.println("   Average duration:     "+avg()+" ms");
        }
    }
    static Stats withoutGC = new Stats();
    static Stats withGC = new Stats();
    public static void main(String[] v) throws Throwable {
        System.out.println("Figuring out processor speed...");
        long beforeCPU=HardRT.readCPUTimestamp();
        long beforeOS=System.currentTimeMillis();
        Thread.sleep(1000);
        long afterCPU=HardRT.readCPUTimestamp();
        long afterOS=System.currentTimeMillis();
        System.out.println("Slept for "+(afterOS-beforeOS)+
                           " ms according to OS.");
        System.out.println("Slept for "+(afterCPU-beforeCPU)+
                           " cycles according to CPU");
        cyclesPerMS=((double)(afterCPU-beforeCPU))/(afterOS-beforeOS);
        System.out.println("Cycles per millisecond: "+cyclesPerMS);
        final Timer t=new Timer();
        t.fireAfter(10,new Runnable(){
                public void run() {
                    long before=HardRT.readCPUTimestamp();
                    if (array==null) {
                        array=new Integer[2000];
                        for (int i=0;i<array.length;++i) {
                            array[i]=new Integer(i);
                        }
                    } else {
                        for (int i=0;i<array.length;++i) {
                            if (!array[i].equals(new Integer(i))) {
                                throw new Error(
                                    "verification failed at i = "+i+".");
                            }
                        }
                        array=null;
                    }
                    t.fireAfter(10,this);
                    long after=HardRT.readCPUTimestamp();
                    if (GC.inProgress()) {
                        withGC.add(after-before);
                    } else {
                        withoutGC.add(after-before);
                    }
                }
            });
        for (;;) {
            Thread.sleep(5000);
            System.out.println();
            System.out.println();
            System.out.println("ITERATIONS // NO GC:");
            withoutGC.report();
            System.out.println("ITERATIONS // WITH GC:");
            withGC.report();
            System.out.println("Total iterations:    "+
                               (withGC.iterations+withoutGC.iterations));
            System.out.println("Average WithGC/NoGC: "+
                               (withGC.avg()/withoutGC.avg()));
            System.out.println("Average WithGC-NoGC: "+
                               (withGC.avg()-withoutGC.avg())+" ms");
        }
    }
}
```

Figure 11. Source code for interrupt handling benchmark, the results of which are shown in Tbl. 1.

3.2 Footprint

Code size is important for embedded systems. Thus, we compared the size of the Fiji VM SPECjvm98 executable to the total code size required to run SPECjvm98 on top of Sun phoneME Advanced MR2-b97 with JIT enabled. The comparison is done with 32-bit x86 Linux ELF executables.

For Fiji VM, the executable is standalone in the sense that it only requires libraries that are standard on Linux (libc, libpthread, and libm). The total size of this executable is 3,775,888 bytes. Note that only about 200 KB of this is the Fiji VM runtime; the rest is those parts of the standard library that are required to run SPEC, and the code for SPEC itself.

For phoneME, we include the size of the `cvm` executable, the sizes of the necessary shared libraries, and the size of the standard library Jar file. Then we add the size of the SPECjvm98 bytecode. Note that we exclude those class files that are obviously not needed to run SPEC. The base footprint, without the SPEC bytecode, is 2,865,771 bytes. With SPEC, the footprint is 3,714,564 bytes. Thus, it is true that phoneME has a 1% size advantage – but we feel that this is quite small when we consider that:

- phoneME can only run four of the SPEC benchmarks (`compress`, `mpegaudio`, `mtrt`, `jack`) – and to run even those four benchmarks, significant modifications to their code were required due to missing features in phoneME. This suggests that Fiji VM would be even smaller if we cut out those same features that are excluded from phoneME.
- For those four benchmarks, phoneME is more than $2\times$ slower than Fiji VM on average.
- The full SPECjvm98 application is large. It is larger, in terms of bytecode size, than the whole code of Fiji VM (compiler and runtime) combined. The smaller the application, the smaller the executable Fiji VM generates. For example, the SciMark 2.0a benchmark only requires 824,396 bytes on Fiji VM, but nearly 3MB on phoneME.

That being said, bytecode *is* more compact than machine code – so it is possible that for truly large applications (in the over 1 MLoC range), Fiji VM will generate code that is significantly larger than the size of phoneME plus the bytecode size. For those cases where size matters and the application is very large, it is likely that Fiji VM will have to include a bytecode interpreter as well as an ahead-of-time compiler in order to be practical. As well, such an interpreter would enable class loading, which is a useful feature for many other reasons. We are working on adding such features to Fiji VM, however for the purpose of this report these features do not exist yet.

3.3 SPECjvm98 on Bare Hardware

In this set of experiments, we told the Fiji VM compiler to compile the Java code against RTEMS 4.9 for the pc386 BSP target, and generate a boot CD ISO9660 image using GRUB [10] and `mkisofs`. We burned a CD, and ran the tests by simply rebooting the test machine. All of the SPEC benchmarks are selectable via a GRUB menu. After running each benchmark, we rebooted the machine, and then selected the next benchmark from the menu.

See Figs. 8 and 9 for screenshots that we took when repeating this exercise in VMware. The GRUB menu is visible in Fig. 8. Fig. 9 shows the output after the `jack` benchmark finishes running.

The performance of SPECjvm98 running on bare hardware is shown in Fig. 10. Note that the performance in these tests is about 10% worse than when running on Linux, for two reasons. First, RTEMS 4.9 does not have the kind of SMP support that we need; in particular, it assumes that each processor is separate from the others and communication only happens through a controlled fashion.

Though there is nothing preventing us from running Java on a multiprocessor in this way, it is not compatible with the way that benchmarks such as `mtrt` work – they expect full shared-memory SMP. Thus, we run RTEMS 4.9 in a purely uniprocessor mode. Second, the RTEMS 4.9 pc386 BSP does not yet support SSE2, which we typically use on x86 to optimize floating point. The inability to use SSE2 significantly slows down the `mpegaudio` and `mtrt` benchmarks. Even so, this shows the ease with which vanilla Java code can be run on bare hardware using Fiji VM.

3.4 Interrupt Handling in Java

For our third and final experiment, we wrote an interrupt handler in Java. See Fig. 11 for the full source code listing. Once again we compiled against RTEMS 4.9, but this time for the ERC32 target. The test was structured as a garbage collection pathology to stress test our concurrent real-time GC. We registered an interrupt handler written in Java to trigger on receipt of a timer interrupt. Note that this is a true bottom-half interrupt handler; when `testhardrtj` in Fig. 11 calls `Timer.fireAfter()`, this bottoms out in the RTEMS `rtems_timer_fire_after()` function. In the interrupt handler, we executed our pathological GC stress test which consists of a allocating, populating, and verifying an array of `Integer` objects. In the body of the loop we tested equality of each element of the array with a newly allocated Java integer object. This test allocates both large, in the case of the array, and very small, in the case of the Integers, Java objects with varying lifetimes. Results for this test are presented in Table 1 with the left most column depicting performance without GC intervention and the right most when the interrupt preempts the GC, and thus is running effectively concurrent to the GC. The time depicted in the table is measured over the whole execution of the interrupt handler. Because of the high load on the GC, the number of iterations when the GC was not run is a fraction of those which the GC was triggered. However, the worst case performance of both is almost identical, illustrating that our concurrent real-time garbage collector introduces negligible overheads. The result is that fVM retains predictable behavior even in the presence of programs with pathological allocation and memory usage.

4. Related Work

There are currently six commercial Real Time Java implementations, and a number of open-source ones available. [3] and [27] are perhaps most well-known of the bunch, and are widely regarded as offering full compliance to the RTSJ specification. Two smaller implementations also exist, [1] and [24], both of which claim to offer RTSJ compliance. The aforementioned implementations all have real-time garbage collection, though implemented in different idioms (ranging from time-based to work-based, with varying degrees of support for concurrency). Additionally, [19] and [4] offer real-time support in their products, though this mostly implies real-time garbage collection rather than other real-time features. Of these systems, only [1] has support for deeply-embedded platforms. To our knowledge none of the existing Real Time Java implementations have been shown to run vanilla Java code in the bottom half of an operating system. Open source Real Time Java implementations include [22] and [14], though neither include the level of Java compliance found in Fiji VM, or any of the other commercial implementations.

Running Java, or other forms of managed code, in deeply embedded environments is not new, if neither real-time support nor performance are critical. [17] and [13] are both operating systems written entirely in managed languages; C# and Java, respectively. Unfortunately, neither offers real-time support. Also, [16], an interpreter based virtual machine, can run in deeply embedded environments, but not nearly at the same level of performance as Fiji VM, or any of the other real-time VMs mentioned above.

5. Steps to Certification

Fiji VM is designed for use in systems that are certified to DO-178B level A. We feel that the simple design of the VM (notably, no class loading or dynamic compilation) make this possible. In particular, the manner in which code is compiled by Fiji VM, as shown in Fig. 1, is significantly simpler than what would be found in any other high-throughput system. The compiler is only roughly 40 KLoC, which includes complete self-verification phases; code is not emitted if these phases detect that any optimization has violated Java's safety properties. Furthermore, we believe that in the future we will be able to use the type system of Fiji VM's internal representation, as well as the flow analyses currently used for certification, to autogenerate certification artifacts. That being said, the compiler itself would need to be certified to be used in true safety-critical systems. Whether or not this would require certain optimization phases to be disabled is yet to be seen. The modular architecture of the Fiji VM compiler makes this easy to do – thus, we do not believe that any architectural decisions we have made will impede the certification process.

The harder part of certification will be the Fiji VM runtime. Whether or not a garbage collector will ever be used in a certified system is still unclear. However, in Fiji VM it is possible to turn the collector off; in that case, the runtime is nothing more than a thin glue layer that connects the Java library's notions of threading, and Java bytecode's notions of locking, to the facilities provided by the operating system kernel.

6. Conclusion

We have developed a new Java virtual machine, fVM, targeted at hard real-time and safety-critical systems. The Fiji VM is a full fledged Java virtual machine capable of running on bare hardware and expressing interrupt handlers in pure Java. It supports a concurrent real-time garbage collector and a host of standard libraries. The fVM has been successfully deployed on an array of architectures, ranging from ERC32 to x86. Even with full real-time support fVM performs comparably to server-class production VMs on standard Java benchmarks.

References

[1] AONIX, *PERC Products,* `http://www.aonix.com/perc.html`, 2009.

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

[3] J. AUERBACH, D. F. BACON, B. BLAINEY, P. CHENG, M. DAWSON, M. FULTON, D. GROVE, D. HART, AND M. STOODLEY, *Design and implementation of a comprehensive real-time Java virtual machine*, in Conference on Embedded software (EMSOFT), 2007.

[4] AZUL SYSTEMS, `http://www.azulsystems.com/`

[5] D. F. BACON, R. KONURU, C. MURTHY, *Thin Locks: Featherweight Synchronization for Java*, in PLDI 1998.

[6] J. BAKER, A. CUNEI, T. KALIBERA, F. PIZLO, J. VITEK, *Accurate garbage collection in uncooperative environments revisited.* In Concurrency: Practice and Experience 2009.

[7] S. BLACKBURN, K. MCKINLEY, *Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance*, in PLDI 2008.

[8] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[9] Free Software Foundation, *GNU Classpath*, http://www.gnu.org/software/classpath/

[10] Free Software Foundation, *GNU GRUB*, http://www.gnu.org/software/grub/

[11] F. Henderson, *Accurate Garbage Collection in an Uncooperative Environment*, in ISMM 2002.

[12] Jikes RVM, http://jikesrvm.org/

[13] JNode.org, *JNode: Java New Operating System Design Effort*, http://www.jnode.org/

[14] jRate, http://jrate.sourceforge.net/

[15] JSR 302, *Safety critical Java technology*, 2007.

[16] R. Lougher, *JamVM – A compact Java Virtual Machine*, http://jamvm.sourceforge.net/

[17] Microsoft Corporation, *Singularity*, http://research.microsoft.com/en-us/projects/singularity/

[18] OAR Corporation, *RTEMS Home Page*, http://www.rtems.org/

[19] Oracle Corporation, *Overview of WebLogic Real Time 2.0*, edocs.bea.com/wlrt/docs20/intro_wlrt/intro.html

[20] F. Pizlo, D. Frampton, E. Petrank, B. Steensgaard., *Stopless: a real-time garbage collector for multiprocessors.* in ISMM 2007.

[21] F. Pizlo, J. Vitek, *An empirical evalutation of memory management alternatives for Real-time Java*, in Real-Time Systems Symposium (RTSS), Dec. 2006.

[22] Purdue, *The Ovm virtual machine,* www.ovmj.org.

[23] Purdue, *Purdue Researchers Participate in Development of ScanEagle UAV,* cs.purdue.edu/news/1-12-06scaneagle.htm

[24] F. Siebert, *Realtime Garbage Collection in the JamaicaVM 3.0*, in JTRes 2007.

[25] F. Siebert, *Real-time garbage collection in multi-threaded systems on a single processor*, in Real-Time Systems Symposium (RTSS), 1999.

[26] Sun Microsystems, *OpenJDK*, http://openjdk.java.net/

[27] Sun Microsystems, *Sun java real-time system,* java.sun.com/javase/technologies/realtime/, 2008.

[28] J. Vitek, R. Horspool, A. Krall., *Efficient type inclusion tests*, in OOPSLA 1997.