Unpicking The Knot: Teasing Apart VM/Application Interdependencies

Yi Lin
Australian National University
Yi.Lin@anu.edu.au

Stephen M. Blackburn
Australian National University
Steve.Blackburn@anu.edu.au

Daniel Frampton
Australian National University
Daniel.Frampton@anu.edu.au

Abstract

Flexible and efficient runtime design requires an understanding of the dependencies among the components internal to the runtime and those between the application and the runtime. These dependencies are frequently unclear. This problem exists in all runtime design, and is most vivid in a metacircular runtime — one that is implemented in terms of itself. Metacircularity blurs boundaries between application and runtime implementation, making it harder to understand and make guarantees about overall system behavior, affecting isolation, security, and resource management, as well as reducing opportunities for optimization. Our goal is to shed new light on VM interdependencies, helping all VM designers understand these dependencies and thereby engineer better runtimes.

We explore these issues in the context of a high-performance Java-in-Java virtual machine. Our approach is to identify and instrument transition points into and within the runtime, which allows us to establish a dynamic execution context. Our contributions are: 1) implementing and measuring a system that dynamically maintains execution context with very low overhead, 2) demonstrating that such a framework can be used to improve the software engineering of an existing runtime, and 3) analyzing the behavior and runtime characteristics of our runtime across a wide range of benchmarks. Our solution provides clarity about execution state and allowable transitions, making it easier to develop, debug, and understand managed runtimes.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Languages, Design, Measurement

Keywords Metacircular, Virtual Machine, Isolation, Dependency

1. Introduction

The engineering of a high performance managed runtime is a major undertaking. The competing goals of correctness, completeness, performance, robustness, agility and flexibility sit in tension. In this paper we explore the interdependencies among the components internal to a managed runtime and those between the application and the runtime. We do so in a metacircular runtime — a context where these interdependencies are particularly important and particularly hard to tease out. The challenge of cleanly and correctly identifying and designing for these dependencies is key to runtime design and perhaps the greatest impediment to wider use of metacircularity.

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.

VEE'12, March 3–4, 2012, London, England, UK.

Copyright © 2012 ACM 978-1-4503-1175-5/12/03…$10.00

Metacircular runtimes are implemented in the same language that they target, an approach which can bring software engineering and performance benefits, intriguing technical challenges, and a satisfying sense of closure. The intellectual appeal of metacircular managed runtimes is therefore unsurprising. We don't see metacircularity as a meaningful end in runtime design and there are few, if any examples of the approach being commercially successful. However, we are convinced of the value of using a high-level language to build a high performance runtime [6, 10]. When the target language is also a good candidate for runtime implementation, metacircularity is the logical outcome. One outcome of this paper is a teasing out of the benefits of metacircularity from its shortcomings. Another outcome is that our findings allow us to make significant improvements to an existing metacircular JVM and shed light on an element of runtime design relevant to implementers of future runtimes, metacircular or not. Because it acutely demonstrates the problem of identifying runtime interdependencies, the context throughout the remainder of this paper is a metacircular runtime.

Metacircularity can have three obvious benefits. First, the strengths of the target language are brought to bear on the implementation. In the case where the target is a strongly typed high level language, the software engineering benefits may be clear, particularly where the high level language is capable of efficiently expressing low-level semantics [10]. Second, the impedance mismatch between implementation and target languages is removed. This problem is most clear at performance-critical boundaries between the runtime and the application, such as in the hot paths of barriers and allocation sequences [3]. Finally, a metacircular runtime 'eats its own dogfood'. This means that the implementers are particularly motivated towards correctness and performance because their implementation has a circular dependency on those properties. Successful cases of metacircular runtimes include Singularity in C# [11], Jikes RVM/Squawk/Maxine/JNode in Java [1, 14, 16, 18], PyPy in Python [15], Klein VM in Self [19], etc.

Despite this academic success, we know of no commercially successful metacircular runtimes. One explanation might be the performance overheads that one associates with high level languages, due to garbage collection, dynamic compilation, etc. However, Alpern et al. note that Jikes RVM achieved 95% the performance of its then commercial counterpart, the IBM 1.3.0 DK JVM, on the SPECjvm98 benchmarks on Linux/IA32 [2]. Indeed, metacircularity has the attraction of potential performance benefits:

- *Aggressive inlining and optimization:* Runtime code can be directly inlined into the application context, and optimizations applied across the resulting integrated code.
- *Seamless library downcalls into runtime:* No cross-language bridge is needed for library downcalls, which provides more inlining and optimization opportunities.

However, while these benefits flow from addressing the target-host impedance mismatch, they also blur the relationship between the application and runtime. The application, language library and run-

time components do not have a crisply defined boundary within the runtime, and are heavily mixed together. So it is generally not possible to tell, dynamically, whether a thread is executing code on behalf of application or the runtime. This absence of runtime/application isolation is the metacircular runtime's dirty laundry, manifest in issues such as security and resource management.

Resource management is one conspicuous manifestation of the lack of isolation that comes with metacircularity. In general it is impossible to know whether an object is allocated by the runtime or by the application because they both allocate to the same heap. A single allocation site (within a standard library, for example), may allocate objects on behalf of the application and the runtime without distinction. Thus class objects, class metadata, code, compiler detritus, and other generic objects used by the runtime are intermixed with application allocations. As a consequence, it is hard to know what the real footprint of the application is, and it is hard to know what the adverse locality effect is of this intermingling. Ogata et al. reported that in the IBM J9 JVM, a conventional JVM written in C/C++, non-Java memory usage can account for more than the Java heap in over half of the DaCapo benchmarks [12]. We use our framework to conduct a similar experiment with Jikes RVM, and find that the runtime/application breakdown in our system is similar. As an average of ten DaCapo benchmarks, 56.6% of memory is used by runtime allocated objects, with only 43.4% of memory consumed by application objects. Thus the Java heap footprint of a benchmark running on a runtime such as Jikes RVM is significantly overstated.

Security can also suffer when isolation is diminished and boundaries are blurred. Consider, by contrast, the boundary between the system and user spaces in an operating system, which are crisply and rigorously defined. Generally within a runtime there are many instances of code that should be considered as privileged. When the execution context of such privileged code is ill-defined security is compromised. This downside to metacircular runtimes is a significant inhibitor to commercial uptake. During the development of the Moxie project, leading developers were polled, and the issue of application/runtime isolation was rated as one of the most pressing issues confronting metacircular runtime design [8].

In this paper we first propose an efficient framework for creating efficient and clean runtime/application isolation and extend it to distinguish among multiple runtime sub-components such as the compiler, garbage collector, classloader, etc. We examine the Jikes RVM code base and identify 41 calls into the runtime. We take the model further by introducing a well-defined concept of *runtime services* code; statically identifiable code that is executed by the application on behalf of the runtime, and is typically inlined and optimized by the compiler. This approach minimizes points where execution context cannot be inferred statically. We have implemented this framework in Jikes RVM, and present preliminary performance numbers that show that it can be implemented at virtually no overhead. We give examples of the mechanism we use to analyze allocation patterns across the application and runtime components. Finally, we argue that the isolation and clarity from using this framework carries with it software engineering benefits, including enhanced modularity within the runtime.

The contributions of this paper are: 1) a critique of metacircularity that identifies lack of isolation and blurred boundaries as blockers to more widespread uptake of metacircularity, 2) a concrete implementation of a new very low overhead framework that reintroduces isolation without forsaking the benefits of metacircularity, 3) a quantitative analysis of Jikes RVM's behavior that was not possible without our framework, and 4) insight into how these lessons can be applied to existing and future runtime implementations, whether they are metacircular or not.

2. Tracking Execution Context

Our goal is to better understand the dependencies and dynamic execution behavior of a managed runtime. In particular we want to remove contextual ambiguity, which arises because of the interactions and dependencies between components. This problem is particularly acute in a metacircular setting, because metacircular runtimes are *reentrant*: while servicing a request from the application the runtime may need to re-enter itself. For example, while performing a classloading operation, the runtime may need support from the memory manager to allocate objects. In that example, what the memory manager sees as the 'application' is in fact the classloader, another part of the runtime. In the absence of dynamic context to inform it, the memory manager is not aware of the fact that it is being called by another part of the runtime.

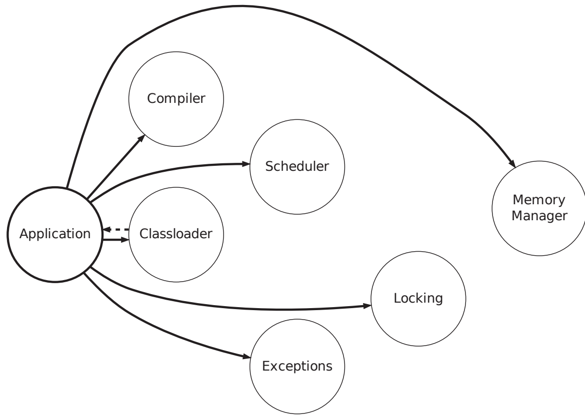
We attack the issue of contextual ambiguity directly by dynamically maintaining an explicit *execution context* for each thread in the runtime. We define execution contexts as holding the current runtime request being executed (e.g., classloading or compilation). This state allows us to:

1. Realize context-specific policies, such as using different heaps according to the *dynamic* context of the allocation. For example, the compiler could use a separate, dedicated, memory management policy.
2. Explicitly assert that certain code may only be executed from within certain dynamic context(s), improving isolation and security, and making it easier to understand the code.
3. Relate behavior to contexts. For example, establishing how often a given library function is dynamically called from within different runtime components or by the user application. Such information could guide optimizations or structural improvements to the runtime design.

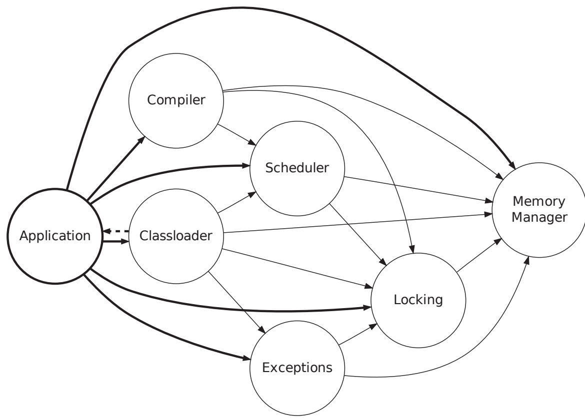
Transition Points In order to track execution context, we identify and annotate *transition* points. These points are comprised of *runtime downcalls* (DCs): transitions from the 'application' to the runtime; and *runtime service calls* (SCs): intra-runtime requests where the responsibility for execution is transferred to another service. In a metacircular runtime, the 'application' may in fact be the runtime itself, so intra-runtime transitions may be due to either downcalls or service calls. We analyzed the Jikes RVM code base and annotated all calls that constitute a transition. A simplified depiction of the context transition graph for Jikes RVM is illustrated in Figure 1(a).

Limiting Interdependencies While dependencies between runtime components are an essential property of a managed runtime (in particular in the metacircular case), infinite regress is not. It follows that certain transitions are permissible whilst others are not. As an example, it is fine for the compiler to trigger a garbage collection, but it is not correct for the garbage collector to recursively trigger a garbage collection. So while intra-runtime transitions are permissible, we require that the graph of these context transitions be acyclic. This helps to curtail dependencies within the runtime, and is of particular value in dealing with metacircularity. In general, this corresponds to disallowing individual features or services from (transitively) requiring access to the feature or service they implement.

Selecting Contexts We broaden the definition of execution contexts from the binary *application/runtime* divide, to include a number of key components of the runtime as discrete contexts, such as *memory management*, *class loading*, etc. This refinement allows us to differentiate an acyclic downcall from the classloader to the memory manager from a cyclic call from the classloader to itself or from the memory manager to itself. Figure 1(b) exposes the intra-runtime dependencies that were elided from Figure 1(a). Only sub-



(a) Transitions from the application



(b) All context transitions, including metacircular edges. Moving from left to right, subsystems exhibit fewer and fewer dependencies.

Figure 1. A simplified depiction of dynamic context transitions in Jikes RVM.

stantive transitions are tracked. There also exist 'lightweight', superficial, transitions which are not tracked for pragmatic reasons and therefore do not register as transitions, as we discuss below.

Upcalls Managed runtimes often require upcalls from the runtime back to the application. For example, a downcall to the classloader may eventually lead to an upcall to the application to execute a class initializer (illustrated in Figure 1 as a dashed line). The upcall may in turn lead to a downcall, creating a cycle. The possibility of generating cycles this way is a property of the runtime specification, independent of the runtime implementation, and is therefore outside the scope of this paper. However, our approach must accommodate upcalls. To achieve this, we include an explicit transition of execution context when an upcall is made, and we discount upcall edges when checking for cycles in the graph of context transitions. Upcalls in effect amount to a break in the graph of metacircular context transitions.

Lightweight Transitions In addition to the 'substantive' transitions discussed above, there also exist lightweight transitions which we pragmatically do not track. Examples of these may include SCs such as accessor methods within the classloader which are used by the compiler or garbage collector to identify field offsets, or DCs such as the fast path of an allocation sequence or write barrier. In the absence of a context transition, the dynamic context remains that of the calling context (e.g. the compiler) although the code falls within the static domain of the callee context (e.g. the classloader). In a non-metacircular case, the code for a lightweight DC is easy to identify because it will be written in the target language (e.g. Java) or perhaps in inlined assembler. However in a metacircular case there is no language transition so the transition point for the lightweight DC is ambiguous. To remove this ambiguity, we explicitly identify and annotate such code with a `@RuntimeService` annotation, which facilitates static checking whilst still supporting this common optimization.

Precise Transition Placement In practice, a transition between components, in particular from the application to the runtime, may manifest as a relatively lengthy call chain, inviting the question of where to draw the line and declare the transition point. For example, a request for memory when allocating a new object will typically start with a frequently taken *fast path*, and only fall through to a *slow path* in the uncommon case. In both metacircular and non-metacircular runtimes, the fast path is typically inlined into the application code by the compiler, while the call to the slow path is kept out of line. In this case we can argue that the fast-path constitutes a lightweight DC with the substantive DC only beginning at the call to the slow path. In a non-metacircular runtime, a natural choice may be the inevitable language boundary between the supported language (Java) and implementation language (C++), which most likely occurs at the call to the slow path. In a metacircular runtime the language transition does not exist, forcing the implementer to make a decision in each case. Our approach, when given a choice, was to select the first out-of-line call as the point of the DC. We reason that this allows us to trigger transitions as close to the application as possible whilst ensuring transitions do not occur too frequently, which would lead to unacceptable overheads.

3. Implementation

We now describe an implementation of our framework for efficiently tracking execution context. Concretely, the implementation involves two steps: 1) additions to the runtime that provide the mechanisms for efficiently identifying and tracking context transitions, and 2) modifications to the runtime to identify all transition points (DCs and SCs).

3.1 Supporting Mechanisms

We create two annotations, `@RuntimeDownCall` and `@RuntimeServiceCall`, to allow us to cleanly identify and differentiate DCs and SCs within the runtime. Figure 2(a) illustrates how `@RuntimeDownCall` annotation is used to identify `Object.wait()` as a downcall. The compiler then expands the annotated method to include context switching calls, as illustrated in Figure 2(b). Figure 3 illustrates context switch methods that track the execution context for each thread, and may optionally implement assertions to ensure only legal context transitions occur.

Assertion of execution context. We use two basic techniques to assert the current execution state, helping with both finding transition points and making runtime development easier. First, we implement an `@AssertExecutionState` annotation that may be applied at the granularity of a class or method, which asserts that the code covered by the scope of the assertion is only executed within a specified set of execution contexts. Thus, if there exists a class that may only ever be called within the context of garbage collection, such an assertion will ensure that this restriction is enforced. Second, we automatically inject assertions into user code at run time to ensure that such code is only executed in an 'application' context.

```
1 @RuntimeDownCall(Context.Scheduler)
2 public void Object.wait() {
3
4     /* Original code */
5
6 }
```

(a) Annotating `Object.wait()` as a DC

```
1 public void Object.wait() {
2   int old = switchContextTo(Context.Scheduler);
3   try {
4
5     /* Original code */
6
7   } finally {
8     switchContextBack(old);
9   }
10 }
```

(b) The same code after expansion by the compiler

Figure 2. An example transition point, Java's `Object.wait()`.

```
1 class VMThread {
2
3   /* Current execution context */
4   int executionContext;
5
6   /* Change context and return previous context */
7   int switchContextTo(int newContext) {
8     int oldContext = executionContext;
9
10    /* Assert that transition is valid */
11
12    executionContext = newContext;
13    return oldContext;
14  }
15
16  /* Change back the context */
17  void switchContextBack(int oldContext) {
18    executionContext = oldContext;
19  }
20
21  ...
```

Figure 3. `VMThread` support for dynamic execution context.

Newly created threads are assigned an initial execution context based on the type of thread being created. For threads that exist within the runtime, such as the compiler thread, we set their initial execution context appropriately.

3.2 Finding Transition Points

We identified transition points iteratively. First we identified major runtime contexts such as the garbage collector, scheduler, etc, and found all transitions into each context. We resolved whether a transition was a DC (from the 'application' to the runtime) or an SC (intra-runtime requests), and marked it with corresponding annotation. For example, Figure 2(a) illustrates a downcall to the `Scheduler` context. We then enforced the acyclic limitation on interdependencies, and further refined context selection. We chose the set of contexts to reflect the structure of the runtime and to maintain the invariant that no transition point is called from within the context it targets.

We also made extensive use of assertions, which we added to code whose context we knew unambiguously. For example, an assertion of garbage collection state that is placed unambiguously within the garbage collector would fail if a transition point into the garbage collector had been overlooked. By following the stack trace from the failed assertion we were able to identify the missing transition point. We repeated this process until we found all transition points. During this process we identified small sections of code within the gray area where execution context is ambiguous. Our identification of these ambiguous areas led to us refactoring the runtime to remove such ambiguities, as we discuss in Section 4.

Transition Point Class	Count
MemoryManager	8
Scheduler	8
Classloader	6
StackTrace	6
Compiler	4
Exception	3
Locking	2
Reflection	2
YieldPoint	1
NewArrayArray	1
Total	**41**

Table 1. Transition Point classes and counts for Jikes RVM.

Table 1 summarizes the transition points we found in Jikes RVM, and their groupings. In all, we identified 41 transition points grouped into 10 classes. We implemented a simple tool that generates a state transition diagram, which allows us to visualize all state transitions. This tool was used to automatically generate Figure 1. Note from this figure that the memory manager can be reached directly from the application, and also via a chain through the classloader, exceptions, and locking, but never from itself. This graph highlights the fact that the memory manager is written in a restricted style which ensures that it does not generate downcalls to any of the other subsystems of the runtime.

4. Design Implications and Lessons

The act of making context and context transitions crisp and explicit encouraged us to rethink elements of the runtime design. In our case this resulted in some minor refactoring and explicit annotations with the consequence that the static code context more closely matches the dynamic execution context. Such changes improved the VM we target and we imagine that the patterns we applied will be relevant to other VMs, whether or not they are metacircular.

In Section 2 we introduced an annotation, `@RuntimeService`, to statically identify contexts where lightweight service calls are executed by the caller on behalf of the callee's context. When we systematically applied this annotation throughout Jikes RVM, it highlighted the fact that ambiguity between contexts, including those between the application and runtime code are re-enforced by the structure and software engineering of the runtime. We now describe how we refactored code to better reflect execution context boundaries.

Inter-class refactoring. Where possible, we moved all lightweight SCs out of their original context into distinctly named packages. By better reflecting the dynamic contexts in the structure of the runtime, this restructuring avoided a large number of unnecessary transition points. It also became straightforward to assert the correct context in the majority of the codebase, since it was largely cleansed of ambiguous code, making the static to dynamic context mapping straightforward in many cases.

Figure 4 illustrates our refactoring with the example of a bump pointer allocator. Figure 4(a) shows the fast path of the allocator, which is executed by the application on behalf of the memory manager. The code is in a distinct package for runtime services and

```
1  package org.jikesrvm.runtime.services.mm
2
3  @RuntimeService
4  public class BumpPointer {
5    Address cursor;
6    Address limit;
7    BumpPointerSpace global;
8
9    @Inline
10   public Address alloc(int bytes) {
11     Address start = cursor;
12     Address end = start.plus(bytes);
13     if (end.GT(limit))
14       return global.allocSlow(this, bytes);
15     cursor = end;
16     return start;
17   }
18 }
```

(a) Fast path of bump pointer allocator; an example of a runtime service.

```
1  package org.mmtk.util.alloc
2
3  public class BumpPointerSpace {
4
5    @NoInline
6    @RuntimeDownCall(Context.MemoryManager)
7    public Address allocSlow(BumpPointer allocator,
8                             int bytes) {
9      Extent blockSize = getBlockSize(bytes);
10     Address start = acquire(blocksize);
11     if (start.isZero()) {
12       return Address.zero();
13     }
14     allocator.cursor = start;
15     allocator.limit = start.plus(blockSize);
16     return allocator.alloc(bytes);
17   }
18
19    ...
20 }
```

(b) Slow path of bump pointer allocator in the memory manager.

Figure 4. Refactored bump pointer allocation code.

the class is annotated with `@RuntimeService`. When a local pool of memory is exhausted (in line 14), the fast path falls through to a slow path. This call is a transition point, a fact reflected by the call into the memory manager proper to a method annotated with a `@RuntimeDownCall` annotation (Figure 4(b)).

Our refactoring also brings the metacircular runtime closer to a non-metacircular runtime, where the impedance matching service code is explicitly identified and exposed. The change also makes it easier to consider hosting another language.

Intra-class refactoring. We found that it was not always practical or desirable to factor runtime services into separate classes and packages. Figure 5 contains an example of the intra-class refactoring that was applied to the classloader. The classloader we use represents classes with `RVMClass` instances. These contain metadata and simple methods to inspect that data in addition to heavy-weight operations such as those to support loading and resolving classes. A number of services, such as `isResolved()`, are typically inlined by the compiler into the application code. These cases are lightweight DCs because they are executed by the application on behalf of the runtime. In this circumstance, insisting on separating them into separate classes would not achieve our software engineering goals, and would only serve to complicate the structure of the runtime. Instead we place `@RuntimeService` annotations on these inspection methods. This solution does not provide all the benefits of separate classes, but allows stronger isolation and clarity.

```
1  public class RVMClass {
2    private byte state;  //current class-loading stage
3
4    ...
5
6    @RuntimeService
7    @Inline
8    public boolean isResolved() {
9      return state >= CLASS_RESOLVED;
10   }
11
12   @RuntimeService
13   @Inline
14   public synchronized void resolve() {
15     if (isResolved()) return;
16     if (superClass != null) superClass.resolve();
17     for (RVMClass interface : declaredInterfaces)
18       interface.resolve();
19
20     resolveInternal();
21   }
22
23   @RuntimeDownCall(Context.Classloader)
24   public synchronized void resolveInternal() {
25     /* heavyweight resolving code */
26
27     state = CLASS_RESOLVED;
28   }
29 }
```

Figure 5. Intra-class refactoring of runtime services: `RVMClass`.

5. Results

We now present results for several experiments we conducted using our framework. In the first experiment we measure the performance impact of our refactoring and our framework for tracking dynamic execution context, showing that the overhead of each is negligible. In the second experiment we use the framework to analyze the behavior of Jikes RVM, measuring the distribution of transitions, as well as a break-down of the time spent in each execution context. In the third experiment we analyze Jikes RVM's memory management behavior, showing that different contexts exhibit different behavior, and that on average around half the memory footprint is due to the runtime.

5.1 Methodology

Our implementation is based on Jikes RVM [1] release 3.1.1 +hg r10393. All experiments use a *production* build: the default high-performance configuration which has debugging assertions turned off, builds code with an aggressive optimizing compiler, and utilizes a high-performance generational Immix garbage collector [5]. For performance measurements we run each benchmark 10 times (10 invocations) and report the average time for the 5th iteration (steady state performance). We also report 95% confidence intervals for the average using Student's t-distribution.

We draw our benchmarks from the DaCapo [7] and SPECjvm98 [17] benchmark suites. We use benchmarks from both the 2006-10-MR2 and 9.12 Bach releases of DaCapo to enlarge our suite and because some 9.12 benchmarks do not run on Jikes RVM.

Performance measurements were conducted using a six-core AMD Phenom II X6 1055T with 3MB of L2 cache running at 2.8GHz, and 4GB of RAM. The machine was running the Ubuntu 10.04.01 LTS server distribution with a 64-bit (x86_64) 2.6.32-24 Linux kernel.

5.2 Framework and Refactoring Overhead

We start by examining the performance overhead of the design changes we made to support the framework (as described in Section 4) and the overhead of dynamically tracking execution con-

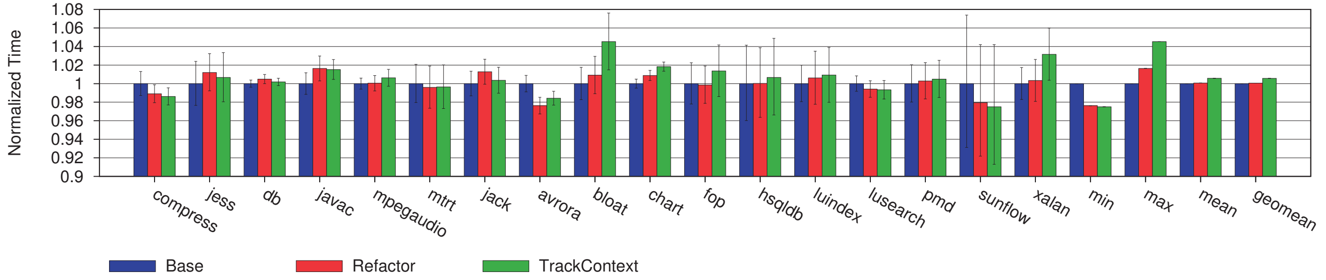


Figure 6. Performance overhead of our refactoring and the mechanism for tracking dynamic execution context.

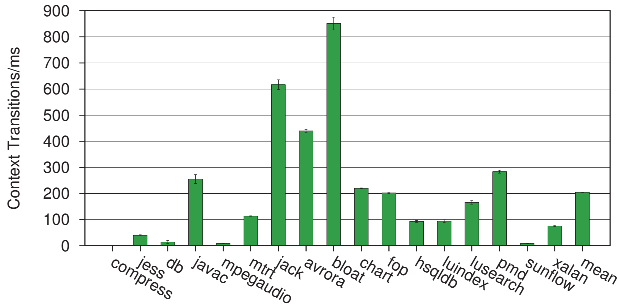


Figure 7. Dynamic transition frequency (transitions per millisecond).

text in a moderate (4× minimum) heap. Figure 6 shows the performance impact due to the refactoring (red) and then the refactoring with dynamic context tracing (green), both relative to an unmodified Jikes RVM configuration (blue).

RuntimeServices refactoring. We did not anticipate any significant overhead due to our refactoring, and Figure 6 shows that this is generally true, with an average overhead of 0.1%, which is well within the noise. Individual benchmarks range from 2.4% speedup to a 1.6% slowdown.

Tracking execution context. Our TrackContext configuration performs on average 0.6% worse than the Base system. Again, these values are very small compared to the noise, suggesting that on average tracking context incurs a negligible overhead. There are two benchmarks, bloat and xalan, that exhibit slightly higher overheads (4.5 and 3.2% respectively) although it is worth noting that the confidence intervals for these benchmarks are also among the largest. In summary, these results show that across a wide selection of benchmarks our execution context tracking incurs minimal overhead.

5.3 Context Transition Analysis

We now use our framework to analyze dynamic context transitions using instrumented versions of our framework. We count both the overall frequency of transitions, and the distribution across individual transition points for each benchmark. We also count the CPU time executing each context to help understand where time is spent during execution.

Execution frequency The frequency of transitions will have a direct impact on the overhead of our system. Figure 7 shows the transition frequency for each benchmark, measured in invocations per millisecond. We see a wide range of values, from 850 calls per millisecond for bloat, to less than 10 per millisecond for compress, mpegaudio and sunflow. Note that these low-frequency benchmarks are kernel-based benchmarks, and that two of the three are from the older SPECjvm98 benchmark suite. On average, the frequency is 205 transitions/ms.

In general, we expect benchmarks with more frequent transitions to incur a greater overhead. However, comparing Figures 7 and 6 shows no clear relationship. As an example, avrora has an invocation frequency more than 30× that of db, but avrora speeds up, while db slows down. This indicates that the overhead of changing execution state is small, and being obscured by other factors. We believe these effects are due to interactions with the optimizing compiler. Injecting code may affect inlining decisions, and may more significantly affect methods which have greater register pressure.

Distribution of executed transition points. Table 2 shows the most frequently executed transition points, both on average and for individual benchmarks. By far the most frequently executed transitions occur at `Lock.lock` and `Lock.unlock`, which support the implementation of Java's **`synchronized`** keyword. Note that the frequency of lock and unlock calls in Table 2 do not match. This is primarily due to the use of a biased locking protocol. When an object is first locked, a DC is required for the first thread to obtain the bias for the lock, which then allows that thread to continue to lock and unlock that object within a lightweight DC, assuming no contention occurs.

Although allocations and generational write barrier invocations are very frequent, heavyweight DCs related to these operations are uncommon because they only occur during (rarer) slow path invocations. This shows the benefit of using lightweight DCs for the fast path. If heavyweight DCs were used on the write barrier fast path, they would account for over a third of all transitions on average.

Time spent in execution contexts. Table 3 shows the CPU cycles spent in each execution context when running a single iteration for each benchmark (i.e. startup, rather than steady state). Each result is the fraction of total CPU time spent in each context (including the application context and runtime contexts). We measure cycles for all threads in the system, which includes applications threads as well as runtime threads (e.g, garbage collection and adaptive compilation threads) which never directly run application code but still contribute to overall execution time.

Table 3 shows that for most benchmarks the vast majority of time is spent performing application work. However, for some benchmarks, time spent in the runtime can be significant, in particular for avrora which spends more than half of all of its cycles executing slow paths for synchronization primitives (e.g., wait, notify, lock, and unlock). On average, nearly 13% of cycles are spent performing compilation, and nearly 10% of cycles can be attributed to synchronization/scheduling operations. Garbage collection and

rank	transition point	mean	cumulative mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	chart	fop	hsqldb	luindex	lusearch	pmd	sunflow	xalan
1	`Lock.lock`	44.09	44.09	37.20	42.31	55.02	61.29	37.86	15.50	64.38	44.96	84.59	7.41	69.03	50.35	42.31	38.51	30.33	18.56	50.02
2	`Lock.unlock`	21.06	65.15	21.89	21.32	30.97	29.39	22.14	8.16	6.79	26.74	0.65	3.97	12.22	44.04	33.32	21.07	27.94	12.01	35.39
3	`Scheduler.yieldpoint`	5.75	70.90	18.45	6.18	4.26	1.14	20.75	6.67	0.67	2.16	0.68	2.04	1.29	1.64	3.08	1.92	1.90	22.38	2.56
4	`MM.addFinalizer`	5.74	76.64	2.32	3.33	0.34	0.62	2.02	28.56	0.13	0.22	4.28	41.19	3.35	0.06	7.01	0.25	1.35	0.16	2.42
5	`UPCALL.invokeFinalize`	5.64	82.28	1.73	3.25	0.25	0.46	1.87	28.49	0.12	0.21	4.27	41.14	3.11	0.04	6.90	0.20	1.33	0.11	2.38
6	`MM.allocSlowInline`	4.27	86.55	2.33	13.40	2.73	1.04	3.69	6.39	1.09	0.51	0.93	1.95	1.14	1.39	1.74	5.99	2.10	23.15	3.02
7	`Exception.deliverException`	3.95	90.50	0.28	0.39	0.06	2.51	0.30	0.11	26.09	0.01	0.01	0.10	1.92	0.05	0.06	15.20	19.87	0.07	0.15
8	`MM.addReferenceCandidate`	2.18	92.68	8.03	4.41	2.41	0.43	5.06	2.48	0.34	0.28	0.12	1.16	3.91	0.89	2.89	0.36	1.88	1.67	0.71
9	`StackTrace.<init>`	1.73	94.40	0.14	0.20	0.03	2.49	0.15	0.05	0.02		0.01	0.09	1.70	0.03	0.04	15.44	8.26	0.05	0.64
10	`Runtime.newArrayArray`	1.10	95.50		0.17			0.04					0.02						18.40	
11	`Scheduler.wait`	1.08	96.58	1.00	0.14	0.17	0.03	0.58	0.16	0.01	15.53	0.01	0.02	0.02	0.04	0.07	0.05	0.04	0.51	0.06
12	`Reflection.invoke`	1.07	97.65	2.37	1.73	2.50	0.15	1.77	1.69	0.08	0.10	4.29	0.20	0.47	0.22	0.33	0.29	0.26	0.48	1.19
13	`Scheduler.notifyAll`	0.82	98.47	1.17	0.26	0.19	0.13	1.13	0.31	0.02	8.77	0.01	0.04	0.03	0.07	0.55	0.07	0.07	1.02	0.11
14	`Compiler.compile`	0.39	98.86	0.98	0.84	0.32	0.11	0.93	0.44	0.06	0.08	0.04	0.16	0.78	0.18	0.54	0.06	0.43	0.50	0.22
15	`StackTrace.getClassFromStack`	0.20	99.06	0.01	0.06			0.01					0.04				0.01	2.96		0.35
16	`MM.allocSlowOOL`	0.17	99.24	0.69	0.32	0.21	0.04	0.48	0.32	0.03	0.02	0.01	0.04	0.14	0.08	0.19	0.03	0.11	0.17	0.06
17	`WriteBarrier.DequeAlloc`	0.17	99.41	0.20	0.38	0.09	0.05	0.13	0.15	0.06	0.16	0.04	0.10	0.05	0.38	0.14	0.22	0.38	0.18	0.20
18	`WriteBarrier.DequeEnqueue`	0.17	99.58	0.20	0.38	0.08	0.05	0.12	0.14	0.06	0.16	0.04	0.10	0.05	0.38	0.14	0.22	0.38	0.18	0.20
19	`Classloader.resolve`	0.11	99.69	0.26	0.27	0.10	0.02	0.26	0.10	0.02	0.02	0.01	0.06	0.24	0.04	0.19	0.03	0.13	0.10	0.05
20	`Classloader.instantiate`	0.10	99.79	0.26	0.27	0.10	0.02	0.26	0.10	0.02	0.02	0.01	0.04	0.23	0.04	0.11	0.01	0.10	0.10	0.05

Table 2. Distribution across individual transition points, showing the 20 most frequently executed on average. Each number indicates the percentage of total transitions for that benchmark that were of the given type.

Benchmark	Application (%)	Runtime (%)					
		Classloader	Compiler	Garbage Collector	Memory Manager	Scheduler /Lock	Other
compress	91.78	0.15	0.99	2.14	1.29	2.39	1.27
jess	58.51	0.93	18.73	8.34	6.31	4.64	2.53
db	85.65	0.17	5.23	3.56	1.21	2.98	1.19
javac	56.75	0.62	15.49	10.86	2.66	9.81	3.27
mpegaudio	84.16	0.32	8.28	1.76	0.74	3.30	1.43
mtrt	74.56	0.38	8.31	6.60	3.67	3.79	2.68
jack	56.01	0.41	12.78	7.62	4.05	10.90	1.88
avrora	30.78	0.27	8.83	1.76	0.85	57.08	0.43
bloat	43.93	0.34	30.51	7.75	3.37	11.93	2.16
chart	63.31	0.90	14.84	8.82	4.96	3.99	3.17
fop	65.46	3.76	10.10	7.09	3.12	7.41	2.82
hsqldb	41.94	1.14	26.54	13.29	3.04	12.65	1.40
luindex	69.17	1.50	14.09	5.32	2.59	5.56	1.75
lusearch	55.42	0.31	10.54	7.02	12.81	7.66	4.90
pmd	56.93	2.65	9.86	6.63	3.56	9.62	7.93
sunflow	88.53	0.25	2.35	1.34	2.73	3.88	0.91
xalan	63.99	0.87	17.49	5.74	3.67	6.59	1.62
min	30.78	0.15	0.99	1.34	0.74	2.39	0.43
max	91.78	3.76	30.51	13.29	12.81	57.08	7.93
mean	**63.93**	**0.88**	**12.65**	**6.21**	**3.57**	**9.66**	**2.43**

Table 3. CPU cycles spent in each execution context.

other memory management operations (such as slow path allocation) account for a further 10% of execution on average.

5.4 Memory Management Behavior

One of the motivations for this work was isolating application and runtime behaviors, allowing new analyses and optimizations. The behavior of different contexts are typically conflated, in particular when analyzing a metacircular runtime such as Jikes RVM. In this experiment we examine the memory management behavior of the runtime and applications separately. Specifically, we explore allocation volume, nursery survival rate, and heap footprint. These experiments were run using a 2.5× minimum heap size.

Object Allocation To understand object allocation patterns, we collect allocation information by instrumenting the allocation fast path and tagging each object as it is allocated. Because many execution contexts allocate very little (or not at all) we use a *logarithmic* scale, and only show those contexts that make a significant contribution, aggregating the rest into an 'Other Runtime' category.

Figure 8 shows the fraction of total allocation attributed to each execution context. It is clear that application allocation dominates, although it does vary significantly between benchmarks (not shown in the figure). The most active context is the *compiler*, followed by *booting* and *classloader*. In all the benchmarks, the compiler allocates more than all other runtime contexts combined, and in 13 of the 17 benchmarks, more than 10× that of the rest of the runtime.

Survival Ratios Object lifetime patterns help guide memory management policies. If different runtime contexts exhibit significantly different lifetime patterns, there may be an opportunity to apply context-specific memory management optimizations. Here we measure the fraction of data that survives the first nursery collection after allocation. To measure survival rate for each context, we tag the current execution context in the header of each object at allocation time and count the total number of bytes allocated in each context. Then during collection we count the bytes that survive the collection, and express this as a ratio of the total bytes allocated in that context. The experiment is imperfect because a nursery col-

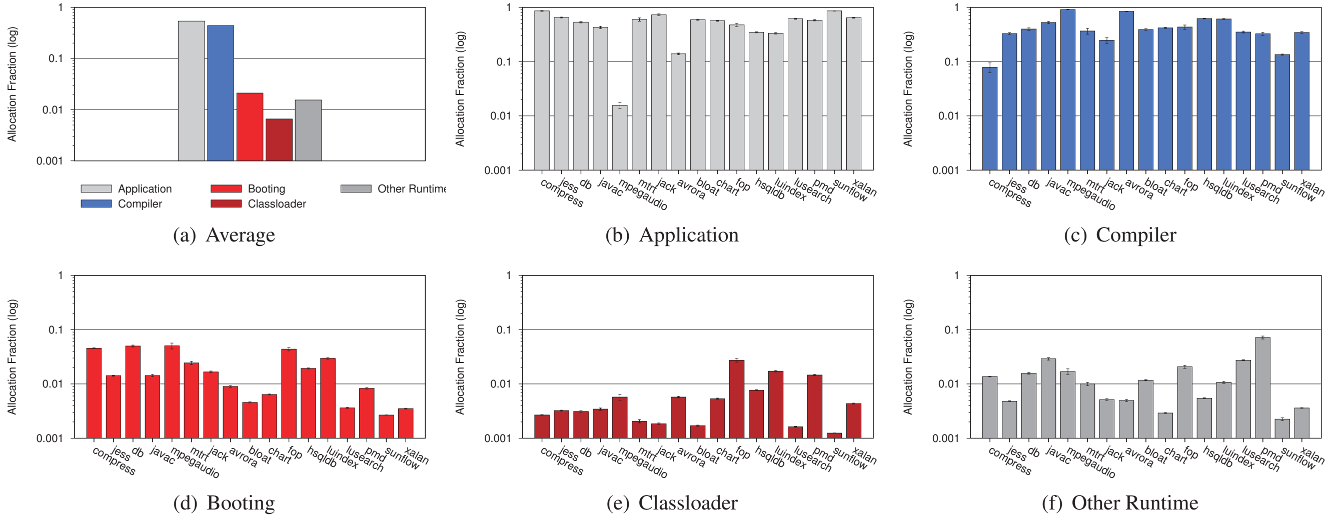


Figure 8. Fraction of total allocation attributed to execution contexts.

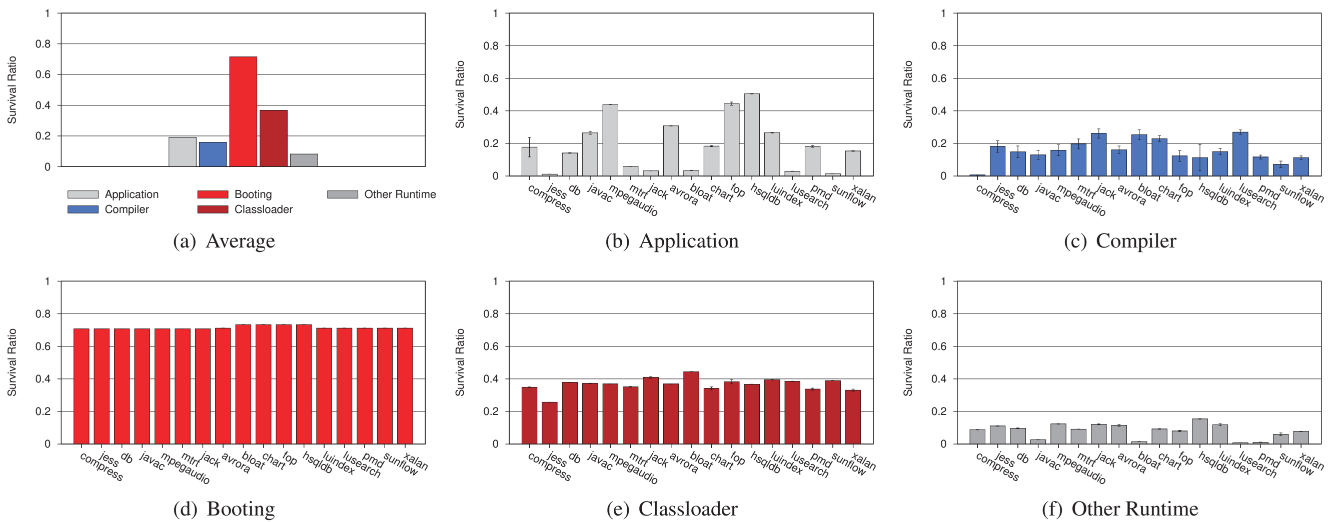


Figure 9. Nursery survival characteristics for objects allocated by execution contexts.

lection is triggered based on total allocation, not allocation per context, but the results still provide interesting insights. Figure 9 shows the result of this experiment. The *booting* and *classloader* contexts show consistently high survival rates. In comparsion, the survival rate for the *compiler* is consistently low, with less than 20% of data surviving the first nursery collection in 12 of the 17 benchmarks.

These experiments show that while allocation behavior between applications varies, allocation in runtime contexts follows a more regular pattern. The compiler context allocates more memory than all other runtime contexts combined for every benchmark, and also shows a consistently low survival rate, averaging just 16.4%. The other allocation-intensive contexts are the classloader and runtime booting procedure, which have stable and high survival ratios.

Heap Footprint Our framework allows us to easily determine the objects allocated by the runtime. This is generally difficult in a metacircular runtime because application and runtime objects are mixed together in a single heap. In our preliminary study we allocate runtime objects into a different region of memory based on the dynamic execution state. Then at garbage collection time, we can measure the live size of the respective heaps to determine the memory footprint for the application and the runtime separately.

Figure 10 shows that in more than half of the benchmarks the memory footprint of the runtime is larger than that of the application. Ogata et al. make a similar observation for the IBM J9 runtime, which is not metacircular [12].

5.5 Summary

These results show that a framework such as ours can disambiguate dynamic execution context at very low overhead, and can also be used to both understand and improve the implementation of managed runtimes.

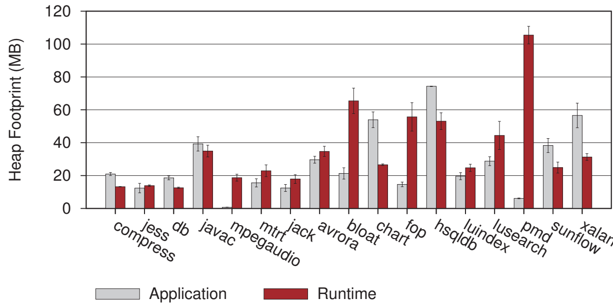


Figure 10. Memory footprint for application and runtime allocated objects.

6. Related work

We categorize related work in two parts: 1) the state of the art for metacircular runtimes, for which ambiguous context is one of the major problems, and 2) approaches to improve isolation within runtimes more broadly.

6.1 Metacircular VMs

There is a long history of metacircular runtimes, tracing back to LISP in the 1960s. In some languages, metacircular implementations are in the mainstream (e.g., LISP and Smalltalk). For others, in particular Java, metacircular implementations have been more of a sideshow to commonly used production runtimes, despite a long thread of interest: the first Java-in-Java runtime was created not long after Java's initial introduction.

Jikes RVM (formerly known as Jalapeño) was the first high-performance metacircular runtime written in Java [1]. Besides a small amount of C code used to boot the runtime and act as an OS interface layer, Jikes RVM is entirely implemented in Java, and once built can execute without the support of a host JVM. Jikes RVM largely ignores matters of application/runtime isolation in its design. While Jikes RVM generally identifies methods that have calls injected by the compiler (one type of downcall from the application into runtime) this convention is not enforced by the runtime. Our implementation is based on the most recent release of Jikes RVM.

Moxie was designed to be a platform for developing production-quality JVMs and performing research into new JVM designs and technologies [8]. Moxie is metacircular, and targets portability, clean bootstrap, debugging and systems programming. The importance of VM/application isolation was recognized early in the Moxie design process. Moxie set a goal of having all transitions from application to JVM services explicit (similar to an operating system trap). As part of this process, Moxie developed a quite different bootstrap model to that used in Jikes RVM, and its *hosted-target* execution model was an important step towards strong VM/Application isolation [8]. Unfortunately, the Moxie VM was never publicly released.

Maxine is, like Moxie, a research platform for next generation Java runtimes [18]. Maxine is metacircular, and is built on principles of modularity and configurability. Maxine provides dedicated replacement mechanisms for important components, including the compiler, garbage collector, and object model. To our knowledge, Maxine does not particularly address the issue of application/runtime ambiguity that arises due to metacircularity and which is the focus of our work.

Ovm is a research JVM that particularly targets realtime applications. [13]. Ovm provides the basic components of a managed runtime, each of which is written almost entirely in Java. Ovm is designed as a general framework to support different object models. Ovm's intermediate intermediate representation, OvmIR reduces some of the ambiguity between the runtime and target language. However, this approach does not serve as a common solution to the context ambiguity and isolation problem that arises from metacircularity, which is our focus in this paper.

6.2 Prior Work in VM Isolation

Other work has focused on the issue of isolation in the context of managed runtimes more broadly, motivated by other concerns such as security and inter-application isolation.

KaffeOS [4] is a Java runtime system designed to implement isolation and resource management boundaries between different applications in a single runtime based on the Kaffe VM. KaffeOS is not metacircular, but is closely related to our work because isolation between different contexts is a primary goal of KaffeOS. In KaffeOS, to support application isolation, limited VM/application isolation is also required. KaffeOS uses a *red line* metaphor [9] to divide 'user mode' and 'kernel mode' contexts. This 'red line' metaphor is similar to our dynamic switching, but there are also some important differences, due to the non-metacircular nature of KaffeOS. First, the kernel in KaffeOS is written in C while the other parts are written in Java, thus a clear cross-language boundary indicates parts of the 'red line' between the runtime and application. Also, the metaphor does not translate well to the metacircular setting, because the metacircular runtime is *reentrant*: during execution a 'red line' could be crossed multiple times. We analyze this problem in detail and propose a clean approach for maintaining a more appropriate runtime state for a metacircular runtime. In addition, our approach to maintaining context has a low overhead, while the isolation used by KaffeOS is more heavyweight, with an 11% overhead.

7. Conclusion

The dependencies between the application and the runtime that supports it–and between the components of the runtime itself–are critical elements of runtime design, yet are often unclear. We examine the problem of these dependencies in the context of a metacircular runtime, where the boundaries between the application and individual runtime components are particularly unclear. We demonstrate a low overhead framework that allows us to clearly identify the dynamic execution context within the runtime, and show how this has led to software engineering improvements in the JVM we use.

Our approach is to restructure and annotate the runtime to transparently reflect context transitions. To do this, we: 1) introduce transition point annotations that allow the runtime to maintain a clear dynamic execution context within the metacircular runtime with very low overhead, 2) use the explicit identification of transition points to drive a restructuring of an existing runtime to substantially reduce structural context ambiguity, and 3) use our framework to analyze behavior and runtime characteristics of a metacircular runtime across a wide range of benchmarks.

Although our implementation is developed in the context of a particular runtime, the principles we apply are more general. In particular, we hope that our insights and analysis may help runtime developers think more clearly about the various contexts that constitute a VM and their relationship. We hope that by substantially reducing ambiguity, this work will make metacircular designs more tenable for the next generation of managed runtimes.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN*

Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '99, pages 314–324. ACM, 1999.

[2] B. Alpern, M. Butrico, A. Cocchi, J. Dolby, S. Fink, D. Grove, and T. Ngo. Experiences Porting the Jikes RVM to Linux/IA32. In *Proceedings of the 2nd Java(TM) Virtual Machine Research and Technology Symposium*, JVM '02, pages 51–64. USENIX Association, 2002.

[3] Apache. DRLVM – Dynamic Runtime Layer Virtual Machine. `http://harmony.apache.org/subcomponents/drlvm/`.

[4] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *In Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, OSDI 2000, pages 333–346, 2000.

[5] S. M. Blackburn and K. S. McKinley. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '08, pages 22–32. ACM, 2008.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146. IEEE Computer Society, 2004.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190. ACM, 2006.

[8] S. M. Blackburn, S. I. Salishev, M. Danilov, O. A. Mokhovikov, A. A. Nashatyrev, P. A. Novodvorsky, V. I. Bogdanov, X. F. Li, and D. Ushakov. The Moxie JVM experience. Technical Report TR-CS-08-01, Australian National University, Department of Computer Science, May 2008.

[9] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *In Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–193. USENIX Association, 1994.

[10] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying Magic: High-level Low-level Programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 81–90. ACM, 2009.

[11] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[12] K. Ogata, D. Mikurube, K. Kawachiya, S. Trent, and T. Onodera. A Study of Java's non-Java Memory. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 191–204. ACM, 2010.

[13] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a Common Intermediate Representation for the Ovm Framework. *Science of Computer Programming*, 57:357–378, September 2005.

[14] E. Prangsma. Why Java is practical for modern operating systems, 2005. Presentation only. See `http://www.jnode.org`.

[15] A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953. ACM, 2006.

[16] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the Bare Metal of Wireless Sensor Devices: The Squawk Java Virtual Machine. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 78–88. ACM, 2006.

[17] Standard Performance Evaluation Corporation. Specjvm98. `http://www.spec.org/jvm98/`.

[18] Sun Microsystems. Maxine Research Project. `http://research.sun.com/projects/maxine/`.

[19] D. Ungar, A. Spitz, and A. Ausch. Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment. In *Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 11–20. ACM, 2005.