Techniques and Applications for Guest-Language Safepoints

Benoit Daloze
Johannes Kepler University Linz, Austria
benoit.daloze@jku.at

Chris Seaton Daniele Bonetta
Oracle Labs
first.last@oracle.com

Hanspeter Mössenböck
Johannes Kepler University Linz, Austria
moessenboeck@ssw.jku.at

Abstract

Safepoints are a virtual machine mechanism that allows one thread to suspend other threads in a known state so that runtime actions can be performed without interruption and with data structures in a consistent state. Many virtual machines use safepoints as a mechanism to provide services such as stop-the-world garbage collection, debugging, and modification to running code such as installing or replacing classes. Languages implemented on these virtual machines may have access to these services, but not directly to the safepoint mechanism itself. We show that safepoints have many useful applications for the implementation of guest languages running on a virtual machine. We describe an API for using safepoints in languages that were implemented under the Truffle language implementation framework on the Java Virtual Machine and show several applications of the API to implement useful guest-language functionality. We present an efficient implementation of this API, when running in combination with the Graal dynamic compiler. We also demonstrate that our safepoints cause zero overhead with respect to peak performance and statistically insignificant overhead with respect to compilation time. We compare this to other techniques that could be used to implement the same functionality and demonstrate the large overhead that they incur.

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

Keywords Virtual Machine, Safepoints, Java, Truffle, Graal, Ruby, JavaScript

1. Introduction

A virtual machine (VM) is a collection of services that support a running program at a higher level of abstraction than a physical architecture provides. This may include services such as automatic memory management through garbage collection, dynamic optimization through just-in-time compilation, debugging and instrumentation, dynamic code loading and reloading.

To provide many of these services the VM needs to be able to pause the running program to inspect and modify its state. Even in a single-threaded environment this can require coordination between the running application and the VM, as the trigger for the VM's action may be an external source such as attaching a remote debugger. However the problem becomes significantly more complex when there are multiple running application threads that all need to be coordinated with the VM. One example of a service that needs to coordinate application threads with the VM is the garbage collector. With the exception of some highly specialized collectors, a garbage collector will at some point need to pause all application threads to update the heap when objects are moved and collected. Conventional architectures and system libraries for threading such as `pthreads` generally do not provide a native mechanism to pause a running thread, so the VM must provide this itself by adding code to the application thread. Any addition must have very low overhead — re-using existing synchronization primitives such as locks would mean that application threads would continually be using these locks even though they are infrequently required. The implementation must also have low latency — in that when threads are requested to pause for garbage collection there should not be a long delay until the collection can go ahead. Additionally, as well as just being paused those threads must be paused in some kind of consistent state where modifications to the heap will not conflict with the work of the application thread.

1.1 Safepoints and Terminology

The conventional solution to the problem of VM-level synchronization and coordination is a technique called *safepoints*. A ***safepoint*** is a point in an application thread where it is safe to *pause* its execution. The VM and its services have special knowledge of safepoints, and can use them to perform several VM-level operations.

The word ***safepoint*** refers to several components of the system, therefore we disambiguate the meaning by qualifying the term. We define ***VM safepoint*** as the lapse of time during which host threads are paused by the VM in their respective safepoints. The alternative — our contribution — is a ***guest-language safepoint*** or simply *guest safepoint*, the lapse of time during which guest threads are paused by the guest-language implementation. For clarity, we will refer to the location within an instruction stream where any of these safepoints can be entered as a ***safepoint check*** rather than just *safepoint*.

1.2 Guest-Language Safepoints

What we contribute is not a mere transposition of VM safepoints to guest languages. This already exists to some extent with Truffle Assumption [19] or JSR 292 SwitchPoint [14], which we detail later as possible underlying mechanisms.

What we propose is an API which allows to interrupt *any guest thread* to run *arbitrary code*, with low latency and zero overhead on peak performance.

Applications for this API are numerous and very powerful: efficient intra-thread communication, signal handling in an existing thread, enumeration of all live objects, call stack examination, always-available debugging, and even deterministic parallel execution.

ICOOOLPS '15, July 6, 2015, Prague, Czech Republic.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1.3 Implementation Context

Our implementation has been developed as part of the open source JRuby+Truffle project [19]. JRuby [12] is an existing effort to implement the Ruby language on top of the JVM using conventional techniques such as bytecode generation and the invokedynamic instruction. Truffle is a framework for writing self-optimizing AST interpreters on top of the JVM [23], and can be seen as an alternative to the conventional approach used in JRuby. JRuby+Truffle brings the Truffle approach as an optional backend to JRuby. The Truffle framework is normally used in combination with the GraalVM [22] — a modification of OpenJDK that includes the Graal dynamic compiler. As Graal is written in Java, Truffle can use it like a library, creating, manipulating and compiling the compiler's intermediate representation. Truffle's Assumption is used to implement guest-language safepoints, meaning all guest compiled code is invalidated when a guest safepoint is triggered and needs recompilation later. For our current use-cases, this is not critical as guest safepoints are triggered only on exceptional and rare events.

1.4 Contributions

The contributions of this paper are:

- A simple but powerful API for guest safepoints that can be conveniently used in languages implemented on top of a virtual machine.
- Example applications of this API to implement existing and new features in the Ruby language.
- An efficient implementation of this API that has provably zero overhead in peak performance and statistically insignificant overhead in compilation time.

2. An API for Guest-Language Safepoints

This section describes our novel API for using safepoints in a guest-language implementation. What we propose is a mechanism that pauses guest-language threads at guest safepoint checks and allows us to interrupt *any thread* to run any *safepoint action*, that is arbitrary code written in the host language. This is an extension to the functionality of VM safepoints as they are restricted to run predefined code in only a subset of the threads. An API for such a mechanism has many application opportunities. Our API is composed of two main parts, illustrated in Figure 1.

The first part of the API is used to respond to safepoint requests from other threads. To this end, application threads make a call to the `poll()` method at every location that is a *guest safepoint check*. These calls are inserted in the guest-language implementation code and not in the application code. For example, a guest-language method will call `poll()` once at the start of each method, and again within every loop[1]. As will be described in Section 4 and later evaluated, the number of calls to `poll()` does not affect peak performance. Only threads controlled by the guest-language implementation should use `poll()`. The polling threads should not perform blocking operations that cannot be interrupted through normal JVM thread interruption. We will discuss these limitations in Section 7.

```
// Guest-language threads call
poll();

pauseThreadsAndExecute(threads, -> {
    // Action to run in every thread.
    // All threads wait for others
    // to complete their actions.
});

pauseThreadsAndExecuteLater(threads, -> {
    // Deferred action to run in every
    // thread, after the thread has
    // exited the guest safepoint.
});
```

Figure 1. API for guest-language safepoints.

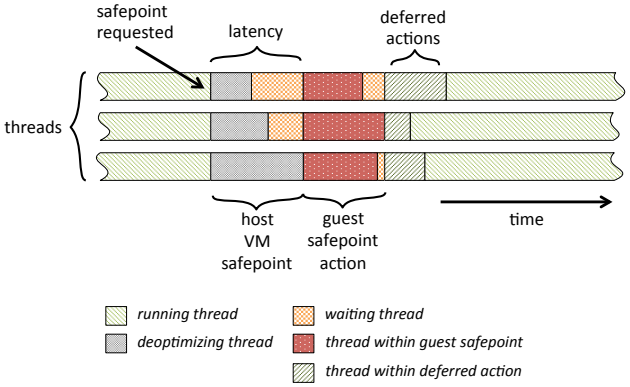

Figure 2. Phases of a guest-language safepoint

The second part of our API is for triggering a guest safepoint and for asking the threads to run the given *safepoint action*. We call the thread requesting the guest-language safepoint the *initiator* thread since it initiates the whole process (activating the safepoint guard, interrupting other threads, sending them the action, etc).

The list of threads affected by a safepoint action is a subset of the threads running guest-language code. However, for all use cases we present, we either want to interrupt all guest-language threads or just a single one. Therefore, usages of our API will use either a single thread or `allThreads` to mean all guest-language threads.

Different safepoint actions require different forms of synchronization between the affected threads. In any case the *initiator* thread has to wait for all affected threads to reach the guest safepoint checks. We distinguish whether the safepoint *action* is run inside the guest safepoint or immediately after the application thread leaves the guest safepoint and resumes normal execution — in which case we call it a *deferred action*. When we want to run the *action* within the safepoint we call `pauseThreadsAndExecute()`. This method accepts a list of threads to be paused and a lambda expression to execute in the guest safepoint by each thread. All affected threads wait that all actions are run to completion. Running the action in the guest safepoint ensures no guest-language thread runs anything else than the *safepoint action*, which can be used to prevent uncontrolled modifications of the heap, similarly to a VM safepoint.

When the initiator does not need these properties, it calls `pauseThreadsAndExecuteLater()` for a deferred action. The action runs after the guest safepoint has finished and the application has left the safepoint, resuming normal execution concurrently with other threads. This is useful for actions that run guest-language code such as callbacks which may themselves use safepoints or run for a longer time. We still have the guarantee that the action will be executed by the other threads before they return from `poll()` and before they continue executing their original code.

[1] As a concrete example, the JRuby+Truffle interpreter performs a call to `poll()` only at seven locations across the whole JRuby+Truffle code base which implements the majority of a large, complex existing language. These locations were the common nodes for method preludes and loop constructs.

It is worth noting that `pauseThreadsAndExecute()` can be expressed in terms of `pauseThreadsAndExecuteLater()` by adding a barrier synchronization at the end of the provided action. Having a separate method reveals better the intent and the desired properties.

These higher level concepts are illustrated in Figure 2. The actions run in each thread are shown with other threads waiting until all are ready to run their action, and until all are finished. Deferred actions are shown running after the main action, concurrently with other threads running normal code.

3. Applications

In this section we describe how we have applied our API in JRuby+Truffle. We also discuss another application currently under development in our Truffle-based JavaScript engine [15].

3.1 Intra-Thread Communication

Some languages allow a thread to cause some action to run on another thread. This is quite a good fit for our API as we can run any action in any thread. For example, a recurrent requirement is to be able to stop a thread because, e.g., it is blocked or the task is no longer needed. In Ruby this functionality is provided by the `Thread.kill` method. To implement this in JRuby+Truffle, the first thread initiates the safepoint:

```
pauseThreadsAndExecuteLater(targetThread, -> {
    throw new KillException();
});
```

The action (the lambda in the call) will be run on the target thread in the next call to `poll()` where it will throw the exception.

We use an exception as the target thread should not just immediately die — the language semantics are that any cleanup actions such as `ensure` (Ruby's equivalent of Java's `finally`) blocks should be run first. This means that even if there was a primitive to just kill the underlying thread it would still not be appropriate.

Any `ensure` clauses will be run to free resources, until the exception terminates the thread execution. We use the deferred version of the API call because throwing an exception may cause arbitrary Ruby code to be executed, which is not designed to run inside a guest safepoint. In Ruby, it is also possible to raise an arbitrary exception in another thread. This is just a variant of the previous example, in which the exception is provided by the user.

3.2 Guest-Language Signal Handlers

Signal handling on a traditional JVM typically spawns a new thread per signal, for example when using the `sun.misc.Signal` package. This is rather inefficient as it needs to spawn a new thread for every signal and incurs some latency as the thread needs to start up, be scheduled and the threads to interrupt need to react to the notification. Furthermore, the newly spawned thread has no generic way to pause or interrupt other threads. With our API we implement a mechanism for signals that is closer to how C handles them, i.e., by running the signal handler on top of a thread call stack (in `poll()`) and returning to normal execution when finished. This allows an application to respond in a more timely manner to signals and to run the handler in a well-known thread. This is in turn very useful in Ruby to implement interruptions such as the SIGINT signal. The signal handler is defined by Ruby code and runs in the main thread, performing the appropriate action to interrupt it such as stopping the running server or raising an `Interrupt` exception.

3.3 Enumerating Live Objects

Heap walking, that is enumerating all live objects in a VM has always been a challenge to implement efficiently. While its utility for general applications might be questioned, it is often useful for debugging, especially when searching for memory leaks. It is one of the most powerful features a language might have and can, for example, help in upgrading a live system by migrating its objects to the new version. In Ruby, access to heap walking is provided by the `each_object` method. Implementing such a feature requires exclusive access to the heap so that objects are not created or destroyed while the list of live objects is created, and it also requires access to the stack of all running threads. Our API provides the functionality to pause all threads and by running an action on each thread we can access all of their stacks. The existing Truffle API, unrelated to safepoints, already provides a mechanism to access values on the guest-language stack.

```
Set<Object> liveObjects;
pauseThreadsAndExecute(allThreads, -> {
    synchronized (liveObjects) {
        visitCallStack(liveObjects);
    }
});
```

The safepoint action then consists of walking the stacks of all threads as well as the thread-local memory in order to look for root pointers, similar to what the marking phase of a GC would do. Each thread adds the objects it can reach to a common set, which is given to the caller once all threads have completed the action.

3.4 Examining Call Stacks

One simple way of finding out what a program is doing is to inspect its threads' call stacks. `jstack` does this for Java programs, but its implementation requires VM support and so it is normally not possible to implement the same functionality for a guest language.

Using our API we implemented a version of `jstack` for JRuby+Truffle. We added a VM service thread that listens on a network socket. Our equivalent of the `jstack` command sends a message to this socket, and the service thread uses our safepoint API to run an action on all threads that prints the current guest-language stack trace.

```
pauseThreadsAndExecute(allThreads, -> {
    printRubyBacktrace();
});
```

3.5 Debugging

Proper debugging support is probably amongst the most useful tools a programming language can provide. In almost all other platforms, including HotSpot, debugging comes with some overhead. This may be significant enough that a production system cannot be debugged.

In previous work [18] it was demonstrated that by re-using VM safepoints a debugger can attach and remove breakpoints in a running program with zero overhead until the breakpoints are triggered. In that work the debugger was in-process and could only be used by writing application code to enter the debugger where commands could be entered. Using our guest-language safepoints we can extend that functionality to allow a debugger to be attached remotely. We reused the same VM service thread as before that listens on a network socket. When a message is received the service thread runs a safepoint action on the main thread telling it to enter the debugger, from where breakpoints can be added and removed and the program inspected.

```
pauseThreadsAndExecuteLater(mainThread, -> {
    enterDebugger();
});
```

3.6 Deterministic parallel execution

A less conventional usage of safepoints can be found in the context of deterministic parallel programming models, i.e., programming models that guarantee deterministic parallel execution. One example of such models is the RiverTrail [6] parallel API for JavaScript, an array-based data-parallel programming model providing implicit parallelism for array-based operations (e.g., via the `Array` built-in object). To enforce deterministic parallel execution, the RiverTrail runtime has to make sure that functions are not performing any "unsafe" operation during parallel execution. This is enforced via the following model of execution:

- Functions can read data from every object that is in their scope at the moment the parallel computation is started. Functions can also freely write to local variables, but they are not allowed to write to objects that are potentially in the scope of other functions as well.
- Parallel functions that attempt to modify an object potentially shared with other functions will cause the parallel execution to *bailout* to sequential mode. In this way, deterministic execution is enforced via sequential execution. In this case, parallel execution is aborted and a single function (usually the first function trying to modify the shared object) is responsible for completing the rest of the computation.

This model of execution is called *temporal immutability* [11], and is implemented in the Truffle JavaScript engine [15] using our safepoint API. In particular, every time a function performs a read operation that could potentially lead to a bailout (e.g., reading from a field that has been modified by another function), the `poll` operation is invoked to ensure that no other thread is attempting to write to some shared object (that is, to ensure that parallel execution is safe). Once a thread attempts to perform an unsafe access that will lead to a bailout (e.g., writing to a shared object), it will request a safepoint. In the safepoint action, the thread will kill all the other threads, and complete the computation sequentially, taking over the work of the aborted threads, if any. A similar deterministic parallel programming model called Deterministic Parallel Ruby [10] has also been proposed in the context of Ruby. Differently from RiverTrail, the Deterministic Parallel Ruby model does not enforce deterministic parallel execution using a bailout protocol, but via a "debug" mode that dynamically checks for non-deterministic execution paths. An alternative implementation of the model could use our safepoints API to check for deterministic execution at very low overhead.

4. Implementation

We now describe the implementation of guest-language safepoints, first from a conceptual point of view and then gradually into more details. Figure 2 shows how the implementation details work together.

4.1 A flag check

Conceptually, to implement the proposed API, all threads need to regularly perform a check — for instance by reading a flag — in `poll()` which tells if the thread should enter a guest-language safepoint. A thread must be able to change the result of that check when it calls `pauseThreadsAndExecute()` or its variant.

A simple but high-overhead implementation could use a global volatile boolean variable, as shown in Figure 3. The variable needs to be volatile as the JVM memory model allows non-volatile fields to be read once and the value re-used unless there is a synchronization point. In practice this may mean that a method containing an infinite loop may only actually read the field once, and the check within the loop could just use a cached value. A thread running the infinite loop would never detect the guest safepoint.

For simplicity when describing the implementation, we remove the `threads` parameter and instead assume there is a condition in the action to decide if it should be executed by the current thread.

```
volatile boolean guestSafepoint = false;

void poll() {
    if (guestSafepoint) {
        // enter guest safepoint
        // execute action
        // wait for other threads
    }
}

void pauseThreadsAndExecute(Action action) {
    // notify the safepoint action
    guestSafepoint = true;
    // wait for threads to enter the safepoint
    guestSafepoint = false;
    // execute action
    // wait for threads to complete the action
}
```

Figure 3. Simple implementation of the API using a volatile flag.

We reset the flag as soon as all affected threads have reached the guest safepoint because the action could potentially call `poll()` and we would not want to keep entering the guest safepoint if there is no need.

One key limitation of a volatile flag is that the whole point of the volatile modifier is that it prevents the compiler from performing some optimizations such as caching the value instead of performing multiple reads. When our Ruby code is compiled and optimized by Graal we generally inline a very large number of methods, as in Ruby all operators are method calls. This means that we are likely to end up with a large number of volatile reads in each method — at least one for each operator application — and the compiler will not attempt to consolidate them, as this is exactly what the volatile modifier is for. Therefore, we would rather optimize the guest safepoint polls by reusing the existing VM safepoints.

4.2 Truffle and Graal

The Truffle language implementation framework provides an abstraction named `Assumption` to model these checks reusing VM safepoints. An `Assumption` provides the `isValid()` method to check its validity and an `invalidate()` method to invalidate the assumption.

The Graal dynamic compiler has special knowledge of calls to `isValid()`. First of all, it requires the receiver assumption to be a compile-time constant (annotated with `@CompilationFinal`). This allows the compiler to inspect the actual `Assumption` object and know whether it is valid. In our case, the assumption is always valid when compiled by Truffle and Graal as it is replaced with a new and valid assumption before recompilation. The calls to `isValid()` are then simply replaced by their compile-time value (`true` in our case). This means that a condition such as in Figure 4 is omitted *from the compiled code* as the branch is known to never be taken. Finally, the compiler registers the assumption with the generated code.

When `invalidate()` is called on an assumption, the compiled code depending on the assumption will be deoptimized (or marked as invalid) and all threads executing that code will transfer to the interpreter. The dependent code is known thanks to the previous registration. The VM will then trigger a VM safepoint, pausing all threads to deoptimize the dependent code. This is how guest safe-

```
// The assumption to implement the guest safepoint
@CompilationFinal Assumption assumption =
    Truffle.getRuntime().createAssumption();

void poll() {
    if (!assumption.isValid()) {
        enterGuestSafepoint();
    }
}
```

Figure 4. Implementation of `poll()` with an `Assumption`.

points effectively reuse VM safepoints. Threads depending on the assumption (all guest-language threads in our case) will run in the interpreter when they execute code dependent on the assumption. When running in the interpreter, calls to `isValid()` are actually performed and not omitted, leading to the invalidated assumption behavior. In our case, threads will enter the guest safepoint.

We show in Figure 5 the core of `pauseThreadsAndExecute()` with an assumption. `barrier()` is a barrier to synchronize all threads in the guest safepoint. `interruptOtherThreads()` calls `Thread.interrupt()` on each thread in the guest safepoint except the current thread.

The code is simplified in that the `synchronized` modifier is not enough to invoke the `pauseThreadsAndExecute()` method concurrently. Instead, exclusive access to trigger a guest safepoint needs to be obtained in an interruptible way, calling `poll()` when interrupted.

```
volatile Action safepointAction;
volatile Thread initiator;

synchronized
void pauseThreadsAndExecute(Action action) {
    safepointAction = action;
    initiator = Thread.currentThread();
    assumption.invalidate();
    interruptOtherThreads();
    enterGuestSafepoint();
}

void enterGuestSafepoint() {
    // wait for all to reach the safepoint
    barrier();

    // renew the assumption
    if (Thread.currentThread() == initiator) {
        assumption = Truffle.getRuntime()
                            .createAssumption();
    }
    // wait for all to see the new assumption
    barrier();

    try {
        safepointAction.run();
    } finally {
        // wait for all to finish the action,
        // unnecessary for deferred actions
        barrier();
    }
}
```

Figure 5. Implementation of `pauseThreadsAndExecute()` with an `Assumption`.

Our implementation in JRuby+Truffle, in the class `SafepointManager`, is slightly more complex as we need to handle a few additional arguments to pass the execution context correctly. Our implementation, as the examples shown above, also does not accept a list of threads for the `pause` methods but rather lets threads register and unregister with guest safepoints. When we only want to affect a subset of the threads, we use a condition in the action. This is an intended restriction as currently the HotSpot JVM only provides "global" VM safepoints affecting all threads and therefore we need to pause all threads anyway.

4.3 Existing implementation of VM Safepoints

In this section we describe how VM safepoints are implemented in our host VM, OpenJDK. A necessary property for VM safepoints is that they can be reached by all threads in a small amount of time (the latency). Once a VM safepoint is reached, the initiator thread is guaranteed to have exclusive access to all heap memory and VM data structures. In OpenJDK [13], VM safepoints are implemented using different techniques based on the kind of code each thread is running — Java bytecode in the interpreter, just-in-time compiled code or native library code.

OpenJDK has two bytecode interpreters, the simple C interpreter and the standard template interpreter. The former one is only used on architectures where the template interpreter is not supported. The C interpreter checks for safepoint requests by reading a volatile variable at method entries, return instructions and loop *back edges*. These locations are chosen for safepoint checks as that is where the code might *go back* to execute the same instructions again and therefore where it spends most of the time. This is in fact similar to our flag check approach in Section 4.1 but at the VM level.

When a safepoint is requested in the template interpreter, the templated code of the affected threads is patched to contain an additional indirection checking for safepoints before executing normal behavior. This is achieved by replacing the normal dispatch table — the mapping between bytecodes and instructions — with a dispatch table calling back to the VM for checking safepoints and then resuming execution in the normal dispatch table. Once the safepoint is reached, the code is patched again to use the normal dispatch table to remove the overhead of this indirection. In the end, the volatile variable is read before entering the safepoint to verify if there was a request as it is not atomic to restore the normal dispatch table (a memory copy of a large region).

In the machine code generated by one of the just-in-time compilers (client, server and also Graal for the GraalVM), checking for safepoint requests is achieved by performing a read on a special safepoint polling memory page. It is not the value of that read which is important but its side effects that are significant. To cause the thread to enter a safepoint, the permissions on the page are changed so to disallow access to that page. This provokes an access violation on the next read, which gets reported as a SEGV signal (Linux/BSD) or an Exception (Windows). The corresponding handler then enters the safepoint if the faulty read location was the polling memory page.

To describe the specific instruction used for a safepoint, we take the concrete example of the common AMD64 architecture. Instructions for other architectures are similar. The chosen instruction is `test`, which reads an address in memory, performs a bitwise conjunction and then sets flags. As this is not a branch instruction, no prediction is needed and the instruction does not modify any general purpose registers, meaning it can be pipelined efficiently by the processor. The locations to add VM safepoints is decided by the compiler and is entirely independent from the number and location of calls to `poll()` using our API. VM safepoints are normally inserted once in each generated machine code function (which could include multiple guest-language methods, or less than a guest-language method such as just a loop body), and once inside each loop. However, optimizations can remove some of these. For ex-

ample, in a loop of a known number of iterations safepoints may be removed because a finite number of instructions can execute there before the loop is exited. Finally, when a thread is performing a blocking operation or similarly running native code (e.g., via JNI), it is considered as already in the safepoint as it may not modify the VM internals or the Java heap. On exit of such a blocking operation, the thread must wait until the safepoint action is complete. This is problematic for guest-language safepoints and we discuss it in the next section.

4.4 Running code in any thread

The existing VM safepoints alone are not expressive enough for our purpose because they do not allow paused threads to execute user-defined code before returning to normal execution. Having regular calls to `poll()` solves that issue.

While reusing VM safepoints provide an efficient implementation, they also do not pause *all* threads. Instead they consider some blocked threads to have reached the VM safepoint because they can not manipulate the VM internals or the Java heap. This is insufficient for guest-language safepoints as we want to run arbitrary code in *any* of the threads participating in the guest-language safepoint. We discuss in this section how to circumvent this problem.

We identify three major cases in which threads might be blocked. The first is blocking calls, such as sleeping for some time, trying to acquire a lock, waiting on a condition or waiting for another thread or process. Most of these blocking calls already provide a way to be interrupted, for instance via `Thread.interrupt` throwing an exception in the blocked thread. These calls are idempotent such that restarting them later behaves as if they were not interrupted. We can therefore interrupt blocked guest-language threads when initiating a guest safepoint. As we do not know which threads are blocked in such a case, we call `Thread.interrupt` on all guest-language threads. When the blocking call is interrupted, we `poll` and then restart the operation.

Another case is blocking IO. Many of these operations are not idempotent, so we need to be careful when restarting them. There are a few alternatives to avoid blocking the guest-language thread. Some IO calls can already be interrupted like blocked calls above. If they are idempotent, the solution described above just works. One solution is to simulate blocking IO with asynchronous IO. Asynchronous IO typically allows interruption and may be restarted. In Java this can be achieved via select-able channels. Finally, as blocking IO operations typically do not depend on the thread running them, we can spawn some host-language threads, invisible to the guest language, to actually perform the blocking IO. Guest-language threads send their requests and wait for the completion synchronously. This waiting operation is much easier to interrupt.

The third case is executing native code in which it might take an arbitrary amount of time for it to finish, if ever. This becomes a serious problem if native code is used for long operations. In practice, many usages of native code might prove non problematic to interrupt if they do not perform long operations before calling back to the VM, which is typical for JNI and guest-language native extensions as they will likely interact often with the guest-language entities. However, this would impact latency.

In the context of Truffle, some of this "native code" (e.g., Ruby C extensions running on top of TruffleC [5]) is actually also executed by the same VM. In this case, that execution is not considered to be native code by the VM and it is possible to add calls to `poll()` in that program like in the main guest-language implementation. If none of that works, running native calls in separate threads remains a possibility.

5. Evaluation

To evaluate our implementation we used a total of 55 benchmarks, both common synthetic benchmarks such as *fannkuch* and *mandelbrot*, as well as a suite of benchmarks produced by taking key methods from a pair of Ruby libraries for manipulating images, `chunky_png` and `PSD.rb` [21]. These benchmarks do represent an extreme of computational intensity — not all Ruby programs are CPU bound — but they are also real code being run in production today. All experiments were run on a system with 2 Intel Xeon E5345 processors with 4 cores each at 2.33 GHz and 64 GB of RAM, running 64-bit Ubuntu Linux 14.04. Benchmarks are run until they reach a steady state, determined by looking at the range of values over a moving window, and then 10 sample iterations are measured. Problem sizes were configured so that their fastest iteration takes at least 10ms and timing calls do not dominate. An arithmetic mean of the samples gives us the reported time. Reported errors are the standard deviation. When summarizing across multiple benchmarks we report a geometric mean of both the sample and the error. Our experiments use version 0.6 of Graal and are made available in a fork of the JRuby+Truffle repository [20].

5.1 Overhead on Peak Performance

We were interested in the overhead that JRuby+Truffle causes when our safepoint API is available but not used. This scenario is relevant because it is how we anticipate our safepoint mechanism to be used in most cases — ready to be used but not a common operation. Furthermore, we wanted to compare the overhead of our safepoint API implementation with that of alternative implementations.

We first measured the performance of JRuby+Truffle including references to the safepoint API (the *api* configuration). Then we removed these references and measured again (the *removed* configuration). As the API is simple and compact, we only had to modify 74 lines of code. We finally measured the performance of JRuby+Truffle when using alternative safepoint implementation techniques. For example, we tried an implementation that explicitly checks a volatile flag (the *volatile* configuration). We also tried an implementation of our API that uses a JSR 292 SwitchPoint object (the *switchpoint* configuration) [16].

Figure 6 shows our results. There is no statistically significant difference in performance between *removed*, *api* and *switchpoint*. This is because both *api* and *switchpoint* are reusing the existing lower level VM safepoints, whose overhead is already part of the VM. Thus, our API adds no penalty to peak performance, and so we refer to it as *zero-overhead*.

There is, however, a statistically significant difference between the performance of *api* and *volatile*. The geometric mean of the overhead for the *volatile* configuration is $25\% \pm 1.5\%$. Of course, this is the overhead on the whole benchmarks and if we could compare the overhead of just the safepoint `poll()` operations it would be much greater.

5.2 Parallel JavaScript

To confirm the validity of our approach we also performed an initial evaluation of the usage of safepoints in the context of the RiverTrail parallel programming model for JavaScript, as discussed in Section 3. Our implementation is based on the GraalJS JavaScript engine [15], and is currently under active development.

Safepoints are used in our RiverTrail prototype implementation to ensure that every read on potentially shared objects is consistent with other reads during parallel execution. This is needed to enforce the RiverTrail parallel model of execution, which permits reads to shared objects and writes to scope-local ones. The model also supports writes to shared objects, at the cost of parallel execution: when write operations are performed on objects that are po-

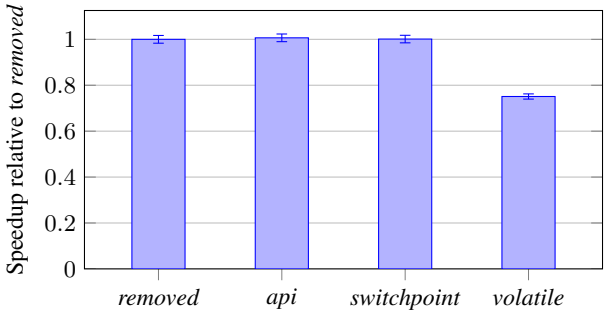

Figure 6. Geometric mean of peak performance over all benchmarks, normalized to the *removed* configuration. Higher is better.

tentially shared, parallel execution is aborted and the computation is completed in a single-threaded mode. Our API is used by every parallel thread to assert that parallel execution has not been aborted. As long as this is true, the threads will never enter the safepoint (i.e., they will never suspend). In the unlikely case of execution bailout, the aborting thread (i.e., the thread that is attempting to write to a potentially shared object) will initiate the safepoint operation, and will complete the parallel computation within the safepoint action, sequentially.

As done in the previous section, we have implemented safepoints by using a volatile check and a Truffle compiler assumption. The performance comparison of the two implementations is depicted in Figure 7, where we present the performance of five JavaScript benchmarks from the Ostrich benchmark suite [9], adapted to use the RiverTrail model. As expected, the overhead for the volatile flag check is considerably high. Since the volatile check is performed on read operations on potentially shared objects (i.e., on objects in the scope of multiple functions running in parallel), benchmarks that perform only thread-local computations are not affected by the volatile check. This is the case for the *Primes* and the *Mandelbrot* benchmarks in the figure. Conversely, benchmarks that rely on read-only shared state (such as *CRC*, *FFT* and *NBody*) benefit from our API.

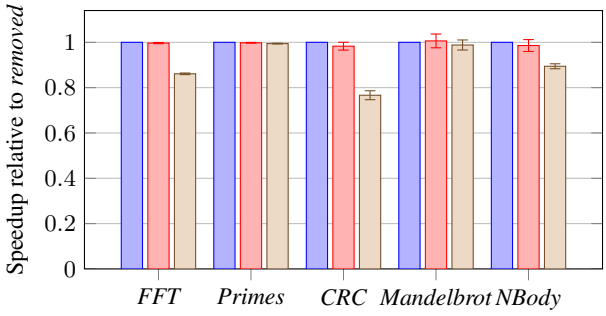

Figure 7. Geometric mean of peak performance for RiverTrail in JavaScript. The *removed* configuration ( ) is compared against our *safepoint API* ( ) and *volatile* ( ). Higher is better.

5.3 Detailed Analysis

To further explain our results, we examined the machine code produced by the different configurations. We wrote a simple method that was run in an infinite loop to conveniently trigger the dynamic compiler. Our example code, written in Ruby and executed by JRuby+Truffle, is shown in Figure 8. It is intentionally kept small to improve the readability of the machine code and just contains a few arithmetic instructions in the loop body to better show the effect of the different configurations. Every arithmetic operator in Ruby (in this case, $+$, $\times$ and $<$) is a method call. Therefore, any of these operations is a call site and conceptually does the check for guest-language safepoints. The body of the loop simply adds 7 to the counter i at each iteration until i becomes greater or equal to n. The method is called with different arguments to prevent *argument value profiling*, which would eliminate the whole loop entirely in compiled code as it has no side-effects.

```
def test(i, n)
  while i < n
    i += 1 + 2 * 3
  end
end

while true
  test(100, 200)
  test(200, 300)
end
```

Figure 8. Example code for detailed analysis of the generated machine code.

We present the machine code with *symbolic names* for absolute addresses and rename the specific registers to the uppercase *name* of the variable they contain. We use the Intel syntax, in which the destination is the first operand.

The machine code produced by the *removed*, *api* and *switchpoint* configurations is identical if we abstract from absolute addresses and specific registers, and is shown in Figure 9. This supports our measurements in that our API really has zero overhead, rather than just a low or difficult-to-measure overhead.

We can observe that the produced code is very close to the optimal code for such a loop. The operations in the body of the loop are reduced to a single addition thanks to constant propagation. There are only two redundant *move* instructions, which copy the variable i between registers I and I'. The value of i is copied in I' to perform the addition because, if the `add` overflows, the original value of i needs to be accessed by the code performing the promotion to a larger integer type. In theory, the promotion code could subtract the second operand from the overflowed i, but this is a fairly complex optimization to implement. The second *move* reunifies the registers.

The loop begins with a read on the safepoint polling page as described in Section 4.3, which checks for VM safepoints[2]. In the *api* and *switchpoint* configurations, this check is also used for guest-language safepoints at no extra cost. After the `mov`, we `add` 7 to i and then check for overflow with `jo`, an instruction that jumps to the given address if there was an overflow in the last operation. We then have the second `mov`, followed by the loop condition $i < n$. The order of the operands in the machine code is reversed, so we must jump to the beginning of the loop if n is greater than i.

We now look at the machine code produced by the *volatile* configuration (Figure 10). The generated code is much larger. The loop starts by testing the condition $i < n$, again with reversed operands. The condition is negated, $n \leq i$, as the test is to break out of the loop. Otherwise we enter the loop body. The body begins with 4 reads of the volatile flag from memory, and if it is found to be 0, the code jumps to a deoptimization handler with `je`. Of these 4 checks, the first is for the loop itself and the other 3 are for the different calls to $+$, $+$ and $\times$ in the loop body. We then have the read on the safepoint polling page checking for VM safepoints.

[2] Actually, Graal moves this check out of the loop as it notices this is a bounded loop. We disabled that optimization for clarity.

```
loop:
  test   safepoint polling page, eax # VM safepoint
  mov    I', I
  add    I', 0x7
  jo     overflow
  mov    I, I'
  cmp    N, I # n > i ?
  jg     loop
```

Figure 9. Generated machine code for the *api*, *removed* and *switchpoint* configurations.

The remaining code is identical to Figure 9, except for the last two instructions. They perform a read on the volatile flag to check for guest-language safepoints at the call site of $<$, in the loop condition. If the flag is found to be valid, the control goes back to the beginning of the loop.

The 5 extra reads produced by the volatile flag are clearly redundant in the presence of the existing lower-level VM safepoints. They increase the number of instructions for the loop from 7 to 17, incurring a significant overhead as shown in Figure 6.

```
loop:
  cmp    N, I # n ≤ i ?
  jle    break out of loop
  cmp    VOLATILE FLAG, 0x0 # while loop safepoint
  je     deopt
  cmp    VOLATILE FLAG, 0x0 # i += 1
  je     deopt
  cmp    VOLATILE FLAG, 0x0 #      1 + 2
  je     deopt
  cmp    VOLATILE FLAG, 0x0 #          2 * 3
  je     deopt
  test   safepoint polling page, eax # VM safepoint
  mov    I', I
  add    I', 0x7
  jo     overflow
  mov    I, I'
  cmp    VOLATILE FLAG, 0x0 # i < n
  jne    loop
```

Figure 10. Generated machine code for the *volatile* configuration.

5.4 Overhead for Compilation Time

We also considered the time taken for dynamic compilation for benchmarks in different configurations by measuring the time taken to compile the main method from the *mandelbrot* benchmark. This is a relatively large method with a high number of method calls which need to be inlined and several nested loops, all of which add guest safepoints checks. We ran the benchmark 20 times for each configuration.

Figure 11 shows our results, with the columns showing mean compilation and the error bars showing one standard deviation. We found no significant difference in compilation time between *removed*, *api* and *switchpoint*. Compilation time for *volatile flag* appeared to be only slightly higher. All of the techniques explored require extra work in the compiler due to extra `poll()` calls, but this appears to be insignificant compared to the rest of the work being done by the compiler. The *volatile flag* is different to the other implementations in that the code is not removed in early phases and adds extra work to later phases of the compiler.

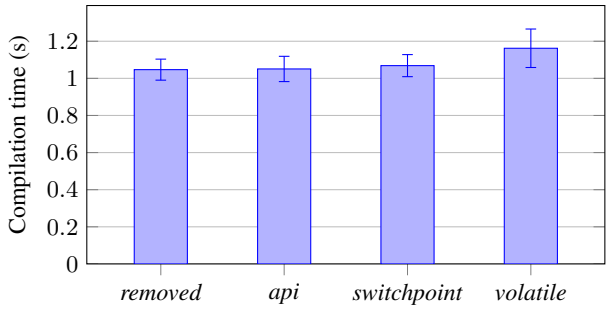

Figure 11. Mean compilation time for the *mandelbrot* method across different configurations. Lower is better.

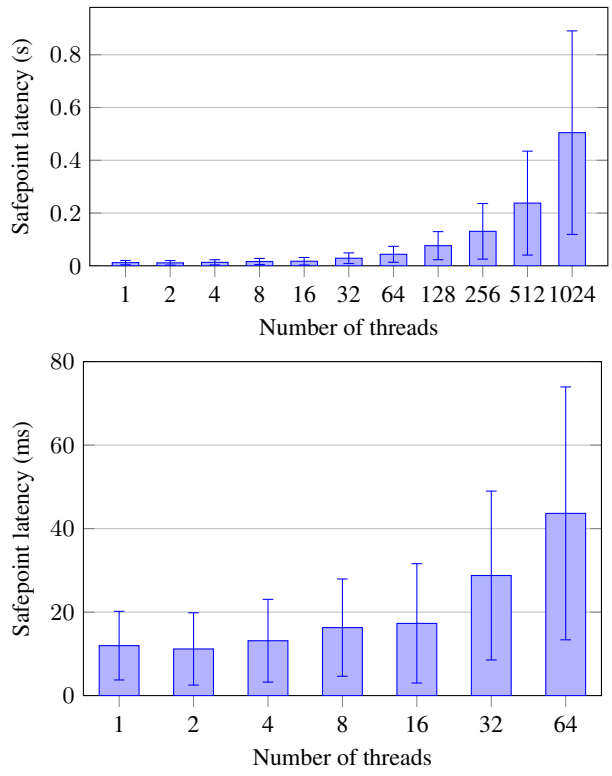

Figure 12. Safepoint latency for the *mandelbrot* for our implementation. Lower is better.

5.5 Latency

Finally, we considered the time it takes for all threads to reach a guest-language safepoint after one thread requested it — the *latency*. Comparing the different configurations is not relevant here, as the costs can be primarily attributed to the VM safepoint latency and the necessary deoptimizations to run the omitted Java code, replaced in the compiled code by a VM safepoint check.

We ran the *mandelbrot* benchmark with a variable number of threads. After steady state was reached, a separate thread requested all others to enter a guest safepoint.

Figure 12 shows our results, with the columns showing mean latency and the error bars showing one standard deviation. Latency was reasonable for most applications, responding to requests for a guest safepoint within about 1/100th of a second when running 8 threads. Variance was surprisingly high, which would make it hard to provide performance guarantees. Latency increases with

the number of threads running concurrently, however the trend is sublinear. Deoptimization for multiple threads is a parallel task, so although multiple threads add extra work, they also add extra potential parallelism. For 1024 threads, a number well beyond typical for normal modern Ruby applications, latency was half a second. It should also be noted that due to deoptimization, the use of a guest safepoint brings peak performance down for some time before code can be reoptimized. Very frequent use of guest safepoints could cause an application to run entirely in the interpreter.

6. Related Work

6.1 Safepoints

Host VM safepoints are a common technique found in many VMs. The major JVMs implement safepoints using the optimized techniques we described in Section 4. Microsoft's CLR uses a volatile flag. Simpler VMs such as the reference implementation of Ruby and the Rubinius implementation also use the flag implementation technique. Rubinius has multiple flags for separate applications such as GC, passing exceptions to threads, and debugging, which means that each safepoint can consist of multiple loads, and as the value of the flag is being explicitly checked it also includes a branch instruction.

In the introduction we said that threading libraries do not generally provide a mechanism to interrupt a running thread which could be used instead of each thread polling. In fact, in some systems such FreeBSD and HP-UNIX the `pthread_suspend` and `pthread_continue` calls are available, which will pause and restart a running thread. The Boehm-Demers-Weiser conservative GC [1] uses these calls when available and sends signals to threads when they are not. However this only addresses one application — pausing threads so they do not modify the heap while collection phases run. They do not allow an action to be passed to another thread.

6.2 Deoptimization

A key part of our technique is that we use deoptimization to jump from code that does not actively poll for safepoints to code that does. Deoptimization is closely associated with development of the Smalltalk language, where it was described as a technique for implementing a debugger [7]. For this application safepoints were also required, which were originally called "interrupt points" [3], as those were the locations where the debugger could interrupt the running program. The emphasis was on making the program at that point able to be inspected by having debug information available.

Our system re-uses the VM's deoptimization mechanism, which is implemented at the level of directly manipulating native call stacks. Others have attempted to re-implement deoptimization at the level of the guest VM [8] using high level language features such as exceptions and a threaded interpreter that allows a program to be resumed by running the correct method for a particular instruction. However, without a similar re-implementation of safepoints this would not be sufficient for our applications.

6.3 SwitchPoint

An instance of class SwitchPoint [14] is "an object which can publish state transitions to other threads". SwitchPoint is part of the `java.lang.invoke` package, which contains various dynamic language support classes, that are part of JSR 292 [16]. The implementation of SwitchPoint uses VM safepoints and adds no overhead to peak performance. In that regard, it is very similar to Truffle's `Assumption` class and can be used as an alternative implementation to access the VM safepoint functionality. Contrary to our API, it provides no way to run arbitrary code in other threads. As far as we know, current use cases of SwitchPoint are limited to rare invalidation, which is a small subset of the applications that we presented in our paper and uses safepoints as a cheap check but not as an opportunity to run code in other threads. One existing use case of rare invalidation is an "almost constant value" [4]. JRuby [12], the implementation of Ruby on top of the JVM and an heavy user JSR 292, uses them for constant invalidation and for modifications of methods in existing classes. Invalidation of a SwitchPoint always triggers deoptimization as it must run non-compiled Java code in the interpreter.

6.4 Biased locking

The biased locking technique described in [17] uses VM safepoints to revoke the bias from an individual lock. This happens when a thread tries to access a lock biased towards another thread. The biased lock is then replaced with some other locking implementation which can handle multiple owners over time more efficiently, i.e., without needing safepoints. Safepoints are also used in bulk re-biasing and revocation to ensure a consistent view of the object headers and their lock states. These bulk operations operate on all objects of a given type to amortize the cost of individual revocation. Bulk re-biasing supports re-initializing the lock state to *unbiased*, allowing other threads to take the bias of one of these objects.

Biased locking is not always easy to apply to guest languages as these might provide unstructured locking (`lock`/`unlock` instead of only `synchronized` blocks) or they might require the `lock()` operation to be interruptible (we need it for guest-language safepoints). However the techniques presented in the paper might be applicable to concurrency primitives in the guest language and their expression would be rather elegant our API.

6.5 Biased Reader-Writer locks

LarkTM [24], a Software Transactional Memory implementation uses VM safepoints for coordinating the resolution of lock conflicts with their biased reader-writer locks [2]. They partition the threads into two categories: threads executing a blocking operation (waiting, I/O, running native code, etc) and all others threads. An *explicit* protocol is used to handle the threads that are *not* executing a blocking operation, in which case the responding thread resolves the conflict. In the *implicit* protocol, used for the blocked threads, the requesting thread does the operation on behalf of the blocked thread, which must wait for that operation to complete before returning from its blocking operation. This approach only works if the code does not depend on the thread in which it is run (call stack, thread-local state, etc) and needs careful memory barriers to ensure visibility of the changes. Our API avoids these limitations by providing the ability to run code in any participating thread. We also discussed how to avoid blocked threads such that an *implicit* protocol is not needed.

7. Future Work

Our current implementation of the safepoint API provides the functionality that we need to implement existing Ruby functionality and other useful new Ruby features. Our API imposes zero overhead on peak performance and insignificant compilation overhead, which means that it causes no overhead for programs which do not actually use a safepoint. Latency is reasonable, but could probably be lowered. Our current implementation uses deoptimization of all application threads which is both time consuming and also damages peak performance after the safepoint is finished until the compiler recompiles the methods. Future work will focus on mitigating these costs so that safepoints can be used for high performance operations such as intra-thread message passing. This can be achieved by moving the API down into the Truffle framework and

by making Graal aware of how to optimize these safepoints. When this is done we aim to measure the costs for various use cases of the safepoint API (e.g., the time needed to attach or detach a debugger), rather than just the costs of the API primitives themselves.

Currently, the VM safepoints that we use are global safepoints pausing all host-language threads in the system. We intend to explore optimizing the case of pausing just a single thread, which is common in many applications of guest-language safepoints and could significantly reduce the latency. It would also improve global throughput as other threads would not be interrupted.

When our safepoint API has been moved down into the Truffle framework, another interesting area of research will be cross-language safepoints. Threads running code written in multiple languages will then be able to cause each other to enter safepoints by using the common Truffle safepoint API. We are planning to use this technique to allow safepoints to work within Ruby C extensions by applying the same API both within the Ruby interpreter and our C interpreter.

Safepoints are primarily useful in multithreaded virtual machines, but our implementation of Ruby currently has a global interpreter lock so that although there is concurrency between threads there is no parallelism. This does not invalidate our work as safepoints have been useful since long before multiprocessors were available. They are still useful for concurrent threads, but it means we are not exploiting the full potential of safepoints. We are currently working to remove this limitation.

Specifically in the implementation of JRuby+Truffle we see safepoints as potentially the key primitive to provide an alternative to locking VM data structures. Structures that are frequently read but infrequently modified can be written to in a safepoint, meaning that no lock is required for readers.

Finally, our system can be generalized as a high performance mechanism to send code from one thread to another. When looked at in this way, we see further applications in areas such as parallelism — where code could be passed between cores that own data, rather than data being passed between cores that own code.

8. Conclusion

We have given the motivation for having a way to use safepoints as a guest-language implementation on a virtual machine, by showing how they can be used to implement several existing features in the Ruby language, and how they allow useful new features to implemented. We described a design for this API that is high level, and showed how we can implement it by re-using the underlying VM safepoint mechanism. We evaluated our implementation and found that it meets our requirements for low overhead on application threads — in fact it has zero overhead, which can be proved by inspection of the machine code. We found the overhead on compilation time to be statistically insignificant, and the latency to be very low for a realistic number of threads. The design and implementation therefore meet the requirements that we had. Future work will look to move this API down into the language implementation framework and to continue to improve latency through improvements at the compiler level.

Acknowledgments

We gratefully acknowledge the contributions of the JRuby+Truffle team including Kevin Menard, the wider Virtual Machine Research Group at Oracle Labs, the Institute for System Software at JKU Linz and everyone else who has contributed to Graal and Truffle.

Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] H.-J. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Softw., Pract. Exper.*, 18(9):807–820, 1988.

[2] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and controlling cross-thread dependences efficiently. In *ACM SIGPLAN Notices*, volume 48, pages 693–712, 2013.

[3] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System, 1984.

[4] R. Forax. JSR 292 Goodness: Almost static final field., 2011. `https://weblogs.java.net/blog/forax/archive/2011/12/17/jsr-292-goodness-almost-static-final-field`.

[5] M. Grimmer, C. Seaton, T. Würthinger, and H. Mössenböck. Dynamically composing languages in a modular way: supporting C extensions for dynamic languages. In *Proc. of Modularity*, pages 1–13, 2015.

[6] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River Trail: A path to parallelism in JavaScript. In *ACM SIGPLAN Notices*, volume 48, pages 729–744, 2013.

[7] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proc. of PLDI*, pages 32–43, 1992.

[8] M. N. Kedlaya, B. Robatmili, C. Caşcaval, and B. Hardekopf. Deoptimization for dynamic language JITs on typed, stack-based virtual machines. In *Proc. of VEE*, pages 103–114, 2014.

[9] F. Khan, V. Foley-Bourgon, S. Kathrotia, E. Lavoie, and L. Hendren. Using JavaScript and WebCL for Numerical Computations: A Comparative Study of Native and Web Technologies. In *Proc. of DLS*, pages 91–102, 2014.

[10] L. Lu, W. Ji, and M. L. Scott. Dynamic enforcement of determinism in a parallel scripting language. In *Proc. of PLDI*, page 53, 2014.

[11] N. D. Matsakis. Parallel Closures: A New Twist on an Old Idea. In *Proc. of USENIX HotPar*, 2012.

[12] C. Nutter, T. Enebo, O. Bini, N. Sieger, et al. JRuby, 2015. `http://jruby.org/`.

[13] Oracle. OpenJDK, 2015. `http://openjdk.java.net/`.

[14] Oracle. Class SwitchPoint, 2015. `http://docs.oracle.com/javase/8/docs/api/java/lang/invoke/SwitchPoint.html`.

[15] Oracle Labs. GraalJS - High-Performance JavaScript Engine, 2015. `http://www.oracle.com/technetwork/oracle-labs/program-languages`.

[16] J. Rose et al. JSR 292: Supporting Dynamically Typed Languages on the Java Platform, 2011. `https://jcp.org/en/jsr/detail?id=292`.

[17] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *ACM SIGPLAN Notices*, 41(10):263–272, 2006.

[18] C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at Full Speed. In *Proc. of Dynamic Languages and Applications (DYLA)*, 2014.

[19] C. Seaton, B. Daloze, K. Menard, T. Würthinger, et al. JRuby+Truffle - a High-Performance Truffle Backend for JRuby, 2015. `https://github.com/jruby/jruby/wiki/Truffle`.

[20] C. Seaton, B. Daloze, K. Menard, T. Würthinger, et al. A JRuby+Truffle fork with the branches used for evaluation, 2015. `https://github.com/eregon/jruby/tree/safepoint`.

[21] C. Seaton et al. Bench9000, 2014. `https://github.com/jruby/bench9000`.

[22] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proc. of Onward!*, pages 187–204, 2013.

[23] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proc. of DLS*, page 73, 2013.

[24] M. Zhang, J. Huang, M. Cao, and M. D. Bond. Low-overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In *Proc. of PPoPP*, pages 97–108, 2015.