Trace-Based Compilation in Execution Environments without Interpreters

Michael Bebenita Mason Chang Gregor Wagner Andreas Gal
Christian Wimmer Michael Franz

Department of Computer Science
University of California, Irvine
{mbebenit, changm, wagnerg, gal, cwimmer, franz}@uci.edu

Abstract

Trace-based compilation is a technique used in managed language runtimes to detect and compile frequently executed program paths. The goal is to reduce compilation time and improve code quality by only considering "hot" parts of methods for compilation. Trace compilation is well suited for interpreter-based execution environments because the control flow of an application program is highly visible and recordable. In this paper, we show that trace compilation is also feasible and beneficial in runtime environments without interpreters where it is more difficult to monitor the control flow of an application.

We present the implementation of Maxpath, a trace-based Java just-in-time compiler for the meta-circular Maxine virtual machine. Maxine uses a tiered compilation strategy where methods are first compiled with a non-optimizing just-in-time compiler in order to collect profiling information, and then recompiled with an optimizing compiler for long-term efficient execution. We record traces by dynamically inserting instrumentation code in non-optimized methods. Execution traces are first collected into trace regions, after which they are compiled, optimized and linked to non-optimized methods for efficient execution. We show that trace-based compilation is an effective way to focus scarce compilation resources on performance critical application regions.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Algorithms, Languages, Performance

Keywords Java, just-in-time compilation, trace-based compilation, SSA form, optimization, trace regions

1. Introduction

The popularity of virtual execution environments has risen dramatically with the advent of the Java virtual machine. Benefits such as automatic memory management, platform independence and memory safety have helped to make the virtual machine (VM) the execution model of choice. Java, C#, JavaScript, Python, and PHP are among the most popular programming languages in use today, all of which rely on VM support.

Early Java virtual machines (JVMs) relied on interpretation for the execution of bytecodes. This is due to the fact that interpreters are relatively easier to write and debug than compilers. However, even the most sophisticated interpreters lag in performance compared to natively compiled code. As JVMs matured, just-in-time compilation was introduced to boost the performance of Java applications. Just-in-time compilers invest some compilation time upfront, hoping to recuperate that cost by generating more efficient code, and thus saving time in the long run. Many state of the art VMs, such as the Java HotSpot™ VM, use a hybrid execution strategy. Code that is assumed to execute rarely is interpreted. After some time, the optimizing just-in-time compiler kicks in and compiles performance critical regions in order to boost performance.

One of the main advantages of a virtual execution environment is introspection. By inspecting the hosted application, a Java virtual machine can adapt to its runtime behavior. This allows a just-in-time compiler to perform many feedback directed optimizations which are not possible in a static setting. Feedback directed optimizations [1] such as deep inlining and the de-virtualization of dynamic methods are crucial to Java performance.

Unfortunately, just-in-time compilers used in VMs are often quite similar in structure to their static counterparts. In case of static compilation, the compiler processes the program code one method at a time, constructing a control-flow graph (CFG) for each method, and performing a series of optimization steps based on this graph. Just-in-time compilers behave similarly, with the distinction that they only process methods that are considered important for performance.

In a static compiler, using methods as compilation units is a natural choice. In static compilation there is usually no profiling information available that could reveal whether any particular part of a method is "hotter" and thus more "compilation worthy" than another. In a static compiler it actually makes perfect sense to always compile entire methods and all possible paths through them, since compilation time is not very important. In contrast to its static counterpart, a just-in-time compiler has access to runtime profile information that can be collected by the VM during interpretation. With this profiling information, the just-in-time compiler can decide which parts of a method actually contribute to the overall runtime, and which parts are rarely taken and are in fact irrelevant from a global perspective as far as optimization potential is concerned. In short, a just-in-time compiler can optimize only frequently taken execution paths (*trace-based compilation*).

In this work, we present the design and implementation of Maxpath, a complete trace-based just-in-time compiler for the Max-

PPPJ '10, September 15–17, 2010, Vienna, Austria.

ine VM [12]. Unlike other VMs where trace-based compilers have been implemented, Maxine does not use an interpreter at all. Instead, it relies on a lightweight non-optimizing just-in-time compiler for the initial execution of Java bytecode. The execution performance of this type of just-in-time compiler is relatively good, much faster than that of a traditional interpreter. The design of Maxine presents two challenges to trace-based compilation: 1) recording and controlling execution of a compiled code is significantly more difficult than in an interpreter, and 2) since the performance difference between the baseline non-optimizing just-in-time and the optimizing compiler is smaller than it is in a mixed-mode interpreted environment, the trace-based compiler needs to be more aggressive in finding compilation-worthy regions in order to pay back compilation cost or any incurred overhead.

This paper makes the following contributions:

- Identification and recording of frequently executed code traces using instrumentation in execution environments without interpreters.
- An efficient control transfer mechanism for executing and returning from traces.
- A design strategy for retrofitting a Java virtual machine with a traced based compiler.

The remainder of this paper is organized as follows: Section 2 provides a general overview of trace-based compilation and introduces the concept of trace regions which we use to capture and compile frequently executed code regions. In Section 3 we describe Maxpath, the implementation of our trace-based compiler for the Maxine VM. In Section 4 we evaluate Maxpath on a set of benchmarks. Related work is discussed in Section 5. The paper ends with an outlook on future work and conclusions in Section 6.

2. Trace-Based Compilation

Trace-based compilers rely on profiling information collected by the VM to detect and compile performance critical application fragments; the most primitive of which is a single linear code path, or an execution trace. Unlike most just-in-time compilers that operate at the method level, trace-based compilers delve in deeper into the control-flow of a method, they profile the execution of program paths. The ultimate goal is to capture the smallest set of execution traces that are representative of the dynamic behavior of the application. Doing so, a trace-based compiler can focus all of its optimization budget on a tiny, yet very important part of an application. Because a trace-based compiler is only concerned with execution traces, it is free to ignore the method as the principal compilation unit. Methods are purely language constructs, and are often not the optimal way of partitioning an application. By profiling and completely disregarding method boundaries, trace-based compilers can effectively infer compilation-worthy regions that can sometimes even span multiple methods.

2.1 Discovering Hot Program Regions

Bala et al. [2] proposed a very effective, yet simple, way of detecting hot program regions in their Dynamo system. Execution counters are used to count the number of backward branches taken at runtime. If this number exceeds a certain threshold, the target of the branch instruction is considered a frequently executed loop header, and naturally all instructions within the loop are presumed to be hot as well. Certainly this is not always the case, often times cold code, or even warm code that deals with corner cases appears in hot loop regions. In order to exclude such code from compilation, dynamic path profiling is used to detect which program paths are more frequently executed than others.

Previous work on path profiling by Ball [3] investigated ways in which information about path execution frequency can be collected and used in tuning performance. In that work, path profiling was accumulated over complete runs of an application. Therefore, any performance benefits would only apply to subsequent compilations and executions of an application.

Collecting accurate path profiling information in the context of a just-in-time compiler is complicated by the fact that only a small fraction of the application behavior is observed. Precise path profiling information would be worthless to a just-in-time compiler if it were only made available when the application terminated. Therefore, for path profiling information to be useful, it must be made available early on, when the just-in-time compiler can actually make use of it.

Our Maxpath trace-based compiler performs path profiling by sampling the execution of program paths within hot loop regions. Hot loop regions are detected by instrumenting the execution of the targets of backward branches, much in the same way Dynamo detects hot loop regions. During trace sampling, Maxpath records program traces and assembles them into larger structures called *trace regions*. The growth of trace regions is guided by various heuristics. Once a trace region has matured, no further traces are accumulated. The trace region is then optimized and linked in for future execution.

2.2 Detecting Loop Headers (Anchors)

Previous work on trace-based compilation in the Hotpath VM [6] used dynamic loop header discovery. Hotpath detected loop headers by monitoring the execution frequency of backward branches using a Java interpreter. Once the execution frequency exceeded a given threshold, the branch target would be considered a hot loop header. Hotpath would then begin recording traces starting at the loop header, and continue recording until the loop header was reached again.

We found that this approach does not scale well. First of all, not all backward branch targets are loop headers. The loop detection method used in Hotpath may incorrectly classify some branch targets as loop headers. Secondly, and most importantly, not having a clear view of the loop structure of a method may lead to the construction of sub-optimal trace regions.

Consider the behavior of Hotpath in the following example:

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        // inner loop body
    }
    // outer loop body
}
```

In this example, the inner loop header is discovered first since it is executed more frequently than the outer loop header. Traces that cover the inner loop body are then recorded. As additional traces are accumulated, some may escape into the outer loop scope and connect back to the inner loop header. This would have the negative effect of polluting the recorded trace region with the cold code that appears in the outer loop header, leading to poor code quality.

The approach we are taking in Maxpath is to perform a static loop analysis phase as a preprocessing step during class loading. Loop header information can be collected essentially for free during bytecode verification. This allows us to guide the growth of trace regions so that they do not escape block scopes. An additional reason for this design choice is that unlike other interpreter-based trace compilers, Maxpath relies on code instrumentation for profiling and trace recording. Instrumenting branch instructions, rather than branch targets, is more difficult and requires more instrumen-

tation code since branch instructions more frequent than branch targets.

2.3 Trace Formations

Trace A *trace* is a sequence of successive instructions or basic blocks observed during the execution of a program. Since a trace contains only one observed execution path, off-trace basic blocks are considered to be trace exit points. During the execution of a trace, if an exit point is reached, control falls back on the interpreter, or in our case, control is transferred to the non-optimizing just-in-time compiler. Figure 1 shows the control-flow graph of a method with two nested loops. The inner loop contains a call to a method named `bar()`. A possible trace through this control-flow graph starting at basic block B_1 is $\{B_1, B_2, B_4, B_9, B_{11}, B_{12}, B'_4, B_5, B_6\}$ as shown in Figure 2. The trace includes 9 basic blocks and partially inlines the `bar()` method. (Block B'_4 represents the code region following the call instruction in the original B_4, shown here for clarity as it would normally be appended to block B_{12}). Blocks B_7, B_3, B_{10}, B_2, B_8 are off-trace basic blocks, or exit points.

Trace Trees A *trace tree* is a collection of connected traces, covering multiple program paths. Each trace in a trace tree starts and ends at the same program location, namely the trace tree anchor, or loop header. Therefore, a trace tree is essentially a collection of cyclic traces that are all rooted at the same point. Figure 3 shows the growth of a trace tree anchored at basic block B_2, the inner loop header. The left-most trace is the first trace to be added to the trace tree. The second trace includes basic block B_{10}, which appears in the inlined method, and tail duplicates blocks $\{B_{12}, B'_4, B_5\}$. Two additional traces are attached to cover more and more of the original control-flow. The primary advantage of trace trees is that they do not contain basic blocks that join control-flow (with the exception of the first basic block). The lack of control-flow join nodes simplifies many compiler optimizations, making trace trees attractive to just-in-time compilers that have a limited compilation budget.

Trace trees were first introduced in the Hotpath VM, and later used in the Spur project by Bebenita et al. [4]. These projects have focused primarily on small kernel-based, numerically intensive benchmarks, and have not attempted to run large-scale benchmarks. Our goal with Maxpath is to provide a full, and complete implementation of a trace-based compiler, where we can measure trace compilation in much larger applications.

Unfortunately, in large programs, excessive tail duplication can lead to code explosion, making trace trees unviable. Consider the following pathological example where all 256 possible control-flow paths are taken, causing a trace tree to grow to have 256 branches.

```
for (int i = 0; i < 256; i++) {
    if ( (i &   1) != 0 ) { ... }
    if ( (i &   2) != 0 ) { ... }
    if ( (i &   4) != 0 ) { ... }
    if ( (i &   8) != 0 ) { ... }
    if ( (i &  16) != 0 ) { ... }
    if ( (i &  32) != 0 ) { ... }
    if ( (i &  64) != 0 ) { ... }
    if ( (i & 128) != 0 ) { ... }
}
```

Trace Region In Maxpath we introduce a more flexible data structure, namely the trace region. A *trace region* is similar to a trace tree in that all traces are rooted at one program point, the trace region anchor. Unlike trace trees, trace regions can include join nodes, making trace-based compilation viable for large-scale applications. Figure 4 shows a trace region following the same growth path as the trace tree shown in Figure 3. As new traces are added to the region, basic blocks are immediately linked to previously recorded blocks, and no tail duplication occurs.

To our knowledge, Maxpath is the first trace-based compiler to use trace regions as its primary compilation data structure.

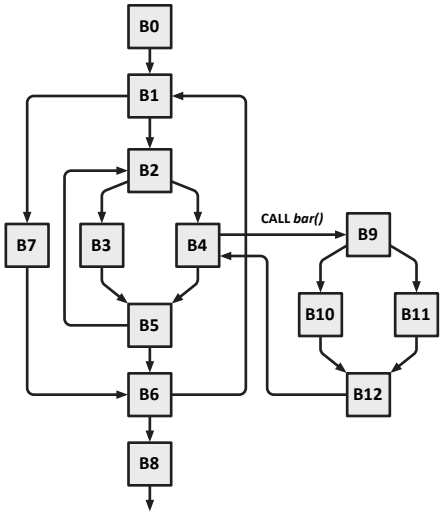

Figure 1. Control-flow graph of a method with two nested loops where the inner loop invokes the `bar()` method.

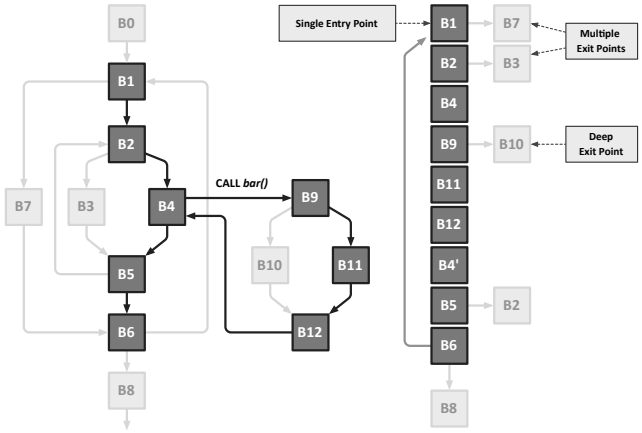

Figure 2. A recorded program execution trace through the control flow graph shown in Figure 1. The trace has a single entry point at basic block B_1, and many exit points at each of the off-trace basic blocks. The trace partially inlines basic blocks in the `bar()` method and therefore contains deep exit point at basic block B_{10}.

3. Maxpath

Maxpath, our trace-based compiler and trace collection framework, is built on top of the Maxine VM. Maxine is a meta-circular research VM under development at Sun Labs. It is written entirely in the Java programming language, leveraging the benefits of meta-circular design and the state-of-the-art development tools. Maxine uses a tiered compilation strategy where methods are first compiled with a non-optimizing just-in-time compiler in order to collect profiling information, and then recompiled with an optimizing compiler for long-term efficient execution.

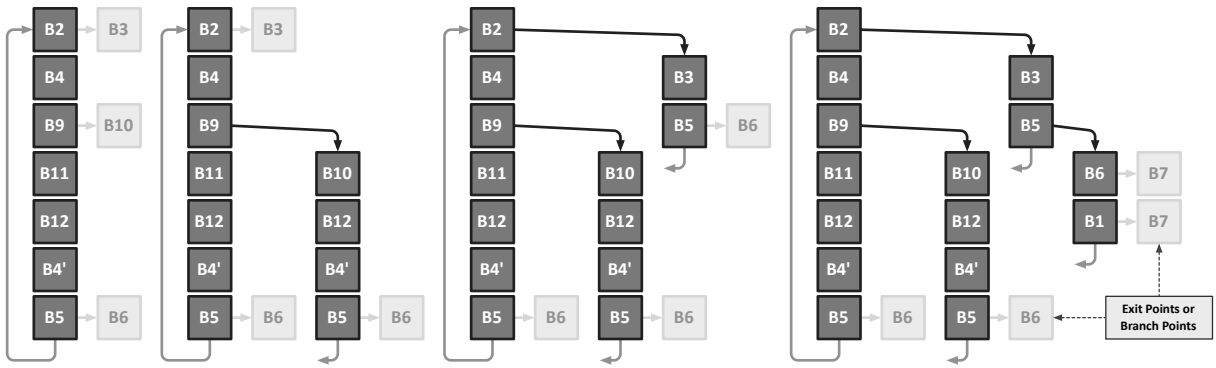

Figure 3. A possible growth phase of a trace tree with excessive tail duplication anchored at basic block B_2 in Figure 1. This trace tree contains four traces, all starting at and branching back to B_2.

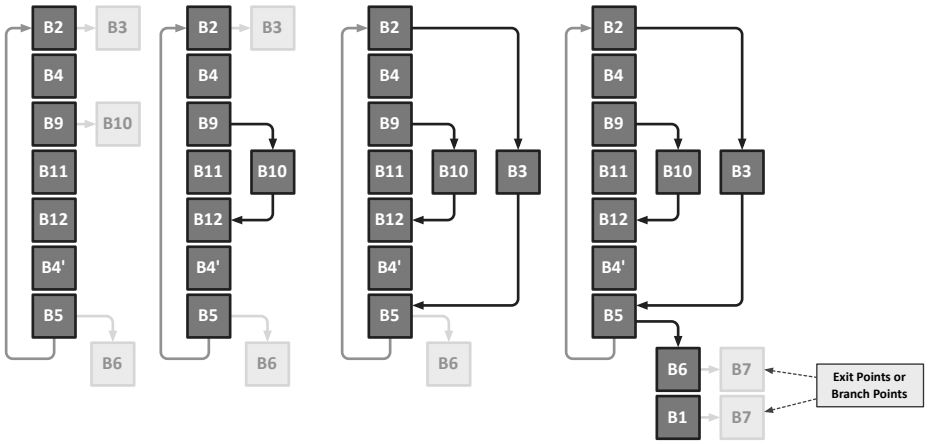

Figure 4. The growth phase of a trace region following the same growth path as the trace tree shown in figure Figure 3. The final trace region contains a lot fewer basic blocks than the trace tree. (For illustration purposes, in this example, the trace region is allowed to include basic blocks outside of the loop, namely B_6 and B_1, whereas Maxpath would not normally include these basic blocks in trace regions).

The non-optimizing just-in-time compiler in Maxine is essentially an inline-threaded interpreter. Each Java bytecode is translated to a snippet of pre-assembled machine instructions. The just-in-time compiler generates code quickly, in a single forward pass, by concatenating and linking these snippets together. An important aspect of the Maxine non-optimizing just-in-time compiler is that it preserves the Java bytecode stack semantics. Figure 5 shows an example output of the Maxine non-optimizing just-in-time compiler for a small sequence of Java bytecode instructions (`ILOAD_0`, `ICONST_1`, `IADD`, `ISTORE_0`).

Maxine also ships with an optimizing compiler. The optimizing compiler is much slower than the non-optimizing just-in-time compiler, but produces better code, and is used for bootstrapping. It performs various standard optimizations, such as method inlining, and whole-method register allocation. A second optimizing compiler is currently under development [13]. The optimizing compiler uses an optimized stack frame layout and calling convention which is incompatible with the stack frame layout used by the non-optimizing just-in-time compiler. This requires the use of adapter frames to adapt parameters from one calling convention to another.

```
ILOAD_0:
  push [RBP + 0]
ICONST_1:
  push 1
IADD:
  pop  RBX
  pop  RAX
  add  RAX, RBX
  push RAX
ISTORE_0:
  pop  [RBP + 0]
```

Figure 5. Example AMD64 generated code by the Maxine non-optimizing just-in-time compiler. Java bytecode stack semantics are preserved by the just-in-time compiler. (`RBP` is used as a frame pointer to access frame locals).

3.1 Trace Recording

The main prerequisite for building a trace-based compiler is the capability to record program execution traces. Building a trace recorder in an interpreter-based execution environment is fairly easy.

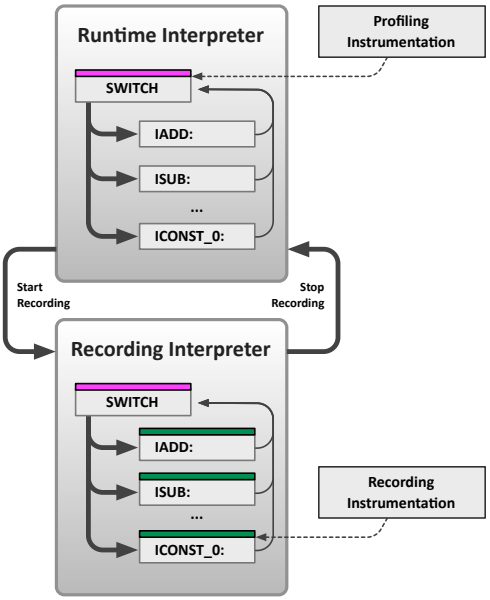

Figure 6. Design of a simple interpreter based trace recorder. In order to minimize trace recording overhead, two interpreters are used. A runtime interpreter is used to profile and execute instructions, while a recording interpreter is used to record instructions. The VM can switch between the two interpreters at any time.

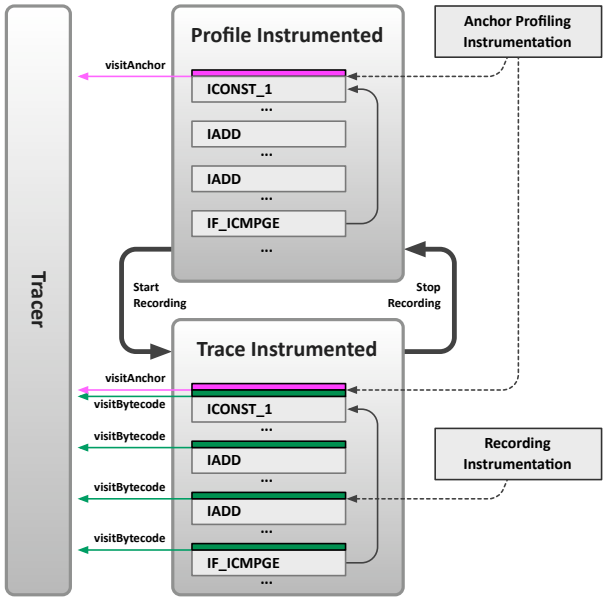

Figure 7. Design of a non-optimized just-in-time based trace recorder. In order to minimize instrumentation overhead, two versions of a method are compiled. The first, profile instrumented, version contains *anchors* that monitor the execution frequency of select program locations and trigger trace recording and compilation. The second, trace instrumented, version contains recording instrumentation that is used to record traces. The two method versions can be used interchangeably because they share a common stack frame layout.

3.1.1 Trace Recording using Interpreters

To record traces, recording code can be inserted in the interpreter loop before or after each Java bytecode instruction handler. This code can record the execution of each bytecode as well as inspect results produced by the execution of the instruction. Unfortunately, this code impacts the performance of the interpreter. Since trace recording is a relatively infrequent process, a two-interpreter trace recording system can be used to minimize any trace recording overhead (Figure 6). One runtime interpreter is used to execute Java bytecode instructions while a functionally equivalent recording interpreter is used to record traces. Once the runtime interpreter decides to record a trace, it can switch to the recording interpreter. Similarly, once trace recording is complete, the recording interpreter can switch back to the non-instrumented interpreter.

3.1.2 Trace Recording in Non-Optimizing Just-in-Time Compilers

Since the performance of the non-optimizing just-in-time compiler is much higher than that of a traditional interpreter, it is more difficult to pay back the overhead incurred by trace recording instrumentation. The more efficient the baseline execution environment is, the more important it becomes to reduce instrumentation overhead. In order to reduce the overhead of tracing we use a technique similar to the one used in the dual interpreter approach. Maxpath maintains two compiled versions for each method that is subject to trace recording, a *profile instrumented* version and a *trace instrumented* version (Figure 7). The profile instrumented version is sprinkled with profiling instrumentation or *anchors* at select program locations. These locations are generally loop headers and are discovered using an inexpensive static analysis performed during Java bytecode verification. During program execution these anchors profile program behavior and trigger the recording of program traces and the compilation of trace regions. Trace recording is performed by hot swapping the profile instrumented version of the method with the trace instrumented version. The trace instrumented version contains tracing instrumentation at each Java bytecode location. This is used to signal the execution of each Java bytecode as it is executed. Once trace recording is complete, execution is resumed in the profile instrumented version of the method. Switching between the two versions is possible because they share the same stack frame layout, namely the Java bytecode stack frame layout.

3.1.3 Tracer

The *tracer* is responsible for the interaction between profile and trace instrumented methods, as well as controlling the growth and selection of trace regions. The tracer is a thread-local runtime component that receives messages from instrumented methods and triggers the recording and the execution of trace regions. Instrumented methods interact with the tracer by sending two types of messages `visitAnchor` and `visitBytecode`. The tracer in turn responds with a *resumption address*. This address indicates where the instrumented code should resume execution. If the resumption address is *zero*, the execution falls through to the next Java bytecode instruction.

The pseudo code presented below is the instrumentation code that is inserted at each anchor or bytecode location.

In profile instrumented methods, the `visitAnchor` message tells the tracer which anchor is about to be executed and what the current frame pointer is. The tracer uses this information to profile the anchor's execution behavior and trigger trace recording. If the tracer wants to start recording, it replies to the `visitAnchor` message with the program address of the bytecode to be recorded in the trace instrumented version of the method. Effectively, execution is transferred from the profile instrumented version of the method to

the trace instrumented version. If the tracer wants to keep profiling the execution of the anchor, it replies with the *zero* address.

```
resumeAddr = visitAnchor(anchor, RBP);
if (resumeAddr != 0) {
    jump(resumeAddr);
}
```

Tracing Bytecode Execution Once execution is transferred to the trace instrumented method, the `visitBytecode` message tells the tracer which bytecode is about to be executed. It does this by passing along the current instruction pointer, as well as the stack and the frame pointer. The tracer can infer the executed bytecode from the instruction pointer by using metadata produced during non-optimizing just-in-time compilation. (Using the stack pointer, the tracer can inspect the current values on top of the Java stack. This is necessary for recording branch conditions or speculating on the receiver types of virtual calls.) After processing the message, if the tracer wants to continue recording, it replies to the message with the *zero* address which resumes the execution of the current bytecode. Otherwise, if the tracer wants to stop recording it can reply with the program address of the equivalent current bytecode in the profile instrumented version of the method. Effectively, switching back to the more efficient profile instrumented version of the method.

```
resumeAddr = visitBytecode(RIP, RSP, RBP);
if (resumeAddr != 0) {
    jump(resumeAddr);
}
```

This technique allows Maxpath to record traces with minimal runtime overhead, at the expense of maintaining duplicate method versions (profile and trace instrumented). However, since trace instrumented methods are only ever needed if trace recording actually occurs, they are createad on demand and freed at will. Moreover, any residual anchor profiling instrumentation overhead can be minimized through code patching. In Maxpath, we simply overwrite the anchor instrumentation with a `JUMP` instruction that effectively jumps over the instrumentation if it's no longer needed.

Tracing Method Invocations Method invocations present a challenge to trace recording. Maxpath uses an inlining policy that dictates which methods should be partially inlined, or traced through. Should the method be inlined, the trace instrumented version of the method is invoked. If the method does not exist yet, it is compiled with trace instrumentation on demand. The invoked method will continue to send `visitBytecode` messages. If the method is not inlined, the tracer is placed on hold and is only resumed once the callee method returns. While the tracer is on hold, waiting for the callee to return, the callee method, or a method further down the execution stack may trigger yet other traces to be recorded and executed. Putting the tracer on hold, may prohibit these traces from being recorded. For this reason, we use a stack of tracers. Once one tracer is on hold, waiting for the callee method to return, it is pushed on a tracer stack, and a new tracer is created that is ready to record and execute additional traces. Once the callee method returns the old tracer is popped off the tracer stack and recording is continued. At any one point, any number of tracers can be on the stack for any given thread, but only one tracer is active per thread, while the remaining ones are on hold.

Threading presents yet another problem. What if two threads are executing the same exact piece of code, and therefore competing for growing the same trace region? In order to avoid lock contention on a shared trace region data structure, we instead grow multiple regions in parallel. The tracers commit the recorded trace regions in completing order, and only the last one to be committed is kept, the rest are discarded.

Trace Aborting Tracing can be aborted for many reasons. The most common case is when the recorded trace exits the recording scope, leaving the loop, or the method containing the original trace anchor. Another common scenario is when the trace length exceeds a certain limit. In these cases, trace recording can abort gracefully, since the tracer knows exactly where to resume execution. The topmost tracer is popped off the tracer stack, and the new tracer that is now on top of the tracer stack is resumed. A more problematic case is when an exception is thrown as a result of an invoked method. In this case, it would be difficult to track how the VM performs stack unwinding and where execution is ultimately resumed. In this case, we abort all active tracers belonging to the thread throwing the exception, by clearing the tracer stack.

3.2 Trace Region Compilation

In order to cooperate with Maxine's runtime infrastructure, we model trace regions as methods. Stack walking and garbage collection rely on the method as being the sole computational element. Modeling trace regions as methods is the most elegant way to fit into Maxine's runtime environment. An additional benefit of this approach is that we are able to use Maxine's back-end register allocator and code generator.

One crucial distinction between trace regions and methods is that trace regions have many exit points, while methods have only one. In a method, all returning control-flow can be joined into one exit point. Moreover, unlike methods, trace regions have many live-out parameters, while Java methods have at most one.

Efficient transfer of control into, and out of trace regions, is very important. Trace regions are generally anchored at loop headers where many local variables could be live. Many of these live variables must flow into trace regions, and therefore an efficient calling convention is critical for performance.

We have tried various approaches, ranging from passing live variables by value or by reference, or a combination of both. These approaches incur a significant overhead since they require quite a bit of parameter shuffling to occur on the stack.

In Maxpath we take a more efficient approach. We statically link the trace region at anchor call sites in methods compiled with the non-optimizing just-in-time compiler. Since we have complete information about stack frame layouts, we can adapt the parameters of trace regions (that are modeled as methods) to the exact stack locations of live-in variables located in non-optimized method frames. Compiled regions have efficient direct read or write access to live-in variables. To do this we modify the register allocator so that it maps input parameters to stack locations in the caller's frame. Live-out variables can be handled similarly, they are written back directly into the caller's method frame. This is possible only because a trace region is only ever called from one call site.

3.3 Transfers

One of the challenging aspects of trace-based compilation is building an efficient control-transfer mechanism. Although trace regions attempt to capture as much control flow as possible, a significant number of trace region exits still occur, requiring execution to be resumed in non-optimized methods.

3.3.1 Fast Transfers

In order to implement transfers, we have introduced a new control-flow instruction to the Maxine VM named `Transfer`. This instruction carries with it enough information to reconstruct a sequence of stack frames, write back appropriate values, and then resume execution at a different program location. This information is essentially the symbolic Java stack state that was recorded during trace recording.

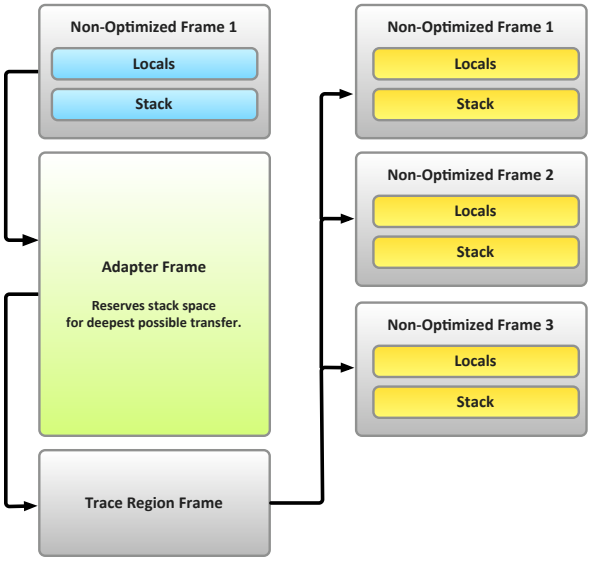

Figure 8. Invocation of a trace region from a non-optimized method. An adapter frame reserves space on the stack for the deepest possible transfer. The trace region frame has direct access to the live-in variables in the original non-optimized method frame. Upon exiting, the trace region writes live-out variables and reconstructs additional non-optimized method frames, before resuming execution.

Machine code that writes back each individual live-out variable, and additional bookkeeping information to link non-optimized frames is emitted for the transfer instruction during code generation. The last machine code instruction generated for the transfer is an unconditional branch to Java bytecode address where execution should resume.

Because transfers may restore multiple non-optimized frames, as is the case if a region exit occurs in a deeply inlined method, we need to ensure that enough space is reserved on the stack for the deepest possible region exit. If we don't reserve enough space, the restored methods may overwrite the currently executing trace region frame. To accomplish this, we use a trace region adapter frame that simply allocates some space on the stack before calling the trace region (Figure 8).

3.3.2 Slow Transfers

The code generated for each transfer instruction could lead to an increase in compilation time, and code cache size. To optimize the size of the generated code we statically predict which transfers are more likely than others. Surely, array index checks, and other exceptional cases are less likely to cause transfers than explicit control-flow instructions. For these cases, we use an additional transferring mechanism. Instead of writing back each value using custom generated machine code, we instead invoke a `transfer()` method within the Maxpath runtime. This method uses a special calling convention where it first saves the state of all machine registers before executing (similar to the way a trap handler would work). The transfer method then inspects metadata about the transfer which indicates the target location of each live-out variable (e.g. whether the variable was stored in a register or spilled to a stack location). Using this data, it can reconstruct non-optimized frames in the reserved stack region and resume execution. This technique is not as efficient as the fast transfer method, but can be used to dramatically reduce code size.

4. Evaluation

To evaluate the performance and runtime characteristics of Maxpath we have chosen a wide variety of benchmarks. Ranging from small computational kernels, to large benchmarks representative of real world applications. All experiments were performed on a Mac Pro with 2 Quad-Core Intel Xeon Processors clocked at 2.8 GHz and 10 GB RAM running MacOS 10.5. Each benchmark has a self timing mechanism, and these are the results that are reported here. Each of the benchmarks was executed with two warmup runs, and an additional timed run, for a total of three iterations at the default problem size.

DaCapo Benchmark Suite We have chosen four of the larger benchmarks in the DaCapo 2006 benchmark suite. Bloat performs a series of optimizations on Java bytecode files. Jython is a Python interpreter written in Java. Luindex is a text indexing tool. Eclipse executes non-GUI performance tests for the Eclipse IDE and is the largest of the benchmarks, with over 12,400 methods.

SPECjvm2008 Benchmark Suite Compress encodes data using a Lempel-Ziv method. Aes, rsa, and signverify are crypto benchmarks. Compiler is the OpenJDK Java front end and compiles itself. Mpeg performs mp3 decoding. Fft, lu, sor, sparse and monte carlo are Java ports of the Scimark benchmark.

Experimental Setup The recording threshold for Maxpath was set to 1000. Every trace region was extended with traces a maximum of 4 times. The maximum trace length was configured to 32 basic blocks. We compare the performance results of three different VM configurations:

- *Baseline*: Maxine is executed only with the non-optimizing just-in-time compiler. Under this configuration, no adaptive optimizations occur.
- *Method*: Maxine is executed with adaptive optimizations. The non-optimizing just-in-time compiler profiles method execution count and performs re-compilation with the optimizing compiler.
- *Trace Region*: Maxine is executed with Maxpath enabled. Adaptive method re-compilation still occurs, but only for methods that contain straight-line code. All other program regions are handled by the trace-based compiler.

Results On the four DaCapo benchmarks the Maxpath trace-based compiler lags behind the Maxine optimizing compiler (Figure 9). We speculate the reason for this is that there are very few hot program regions in these benchmarks. An analysis of the benchmarks [5] indicates in eclipse only about 0.5% of the methods are hot. Bloat, jython, and luindex have more hot methods, with 4.9%, 9.2%, 17% respectively. This trend appears to be consistent with the benchmark results, in that both the Maxine optimizing compiler and the Maxpath trace-based compiler are both able to detect and optimize hot loop regions. The interesting result here is that both the Maxine optimizing compiler and Maxpath perform worse than the baseline configuration. One possible explanation for this behaviour is that profiling and compilation overhead never pays off in this case. On average, trace region compilation is 16% faster than the baseline for the four selected DaCapo benchmarks. The Maxine optimizing compiler is 27% faster on average.

On the SPECjvm2008 benchmarks Maxpath performs much better, up to 100% faster than the baseline, and 60% faster than the Maxine optimizing compiler. The biggest performance difference appears in the Scimark suite of benchmarks: fft, lu, sor and sparse. A possible explanation for this is that the Maxine optimizing compiler fails to identify hot program regions due to its inability to perform on stack replacement. Maxpath identifies these regions and

Benchmark	visitAnchors	visitBytecodes	Blocks	Traces	Anchors	Regions	% Grown	Transfers	Tran/Reg
DaCapo Benchmark Suite [5]									
bloat	306,490	20,946	16,323	488	753	191	25%	529	2.77
eclipse	881,106	57,167	67,232	1,062	2,645	458	17%	1,397	3.05
jython	380,012	163,358	27,691	564	995	221	22%	658	2.98
luindex	151,729	14,314	10,504	236	429	93	22%	368	3.96
SPECjvm2008 Benchmark Suite [10]									
compiler	555,852	39,032	22,044	783	933	358	38%	944	2.64
compress	66,771	6,797	8,716	105	349	37	11%	71	1.92
aes	112,147	10,600	11,745	165	459	60	13%	110	1.83
rsa	151,604	15,045	11,594	259	474	98	21%	193	1.97
signverify	148,145	13,757	10,297	280	434	106	24%	186	1.75
mpeg	160,646	27,798	9,905	288	465	110	24%	178	1.62
fft	88,295	13,945	8,818	137	366	51	14%	74	1.45
lu	95,324	12,966	8,832	144	373	55	15%	78	1.42
sor	66,950	9,256	8,635	96	343	34	10%	51	1.50
sparse	66,908	6,047	8,638	100	344	35	10%	53	1.51
monte carlo	59,056	4,801	8,602	83	338	29	9%	42	1.45

Table 1. Various statistics collected during program execution under the Trace Region configuration. The first two columns indicate the number of `visitAnchor` and `visitBytecode` messages recieved by the tracer. Column 3 indicates the number of basic blocks in the benchmark. Column 4 indicates the number of times a trace was recorded and added to a trace region. Column 5 shows the number of anchors detected and inserted as part of our loop analysis phase. Column 6 shows the number of trace regions that were grown and compiled during the execution of the benchmark. Column 7 indicates the percentage of anchors for which a trace region was ever grown. Column 8 indicates the number of transfer instructions that were inserted in the compiled regions. The last column indicates the average number of transfer instructions that were inserted per region.

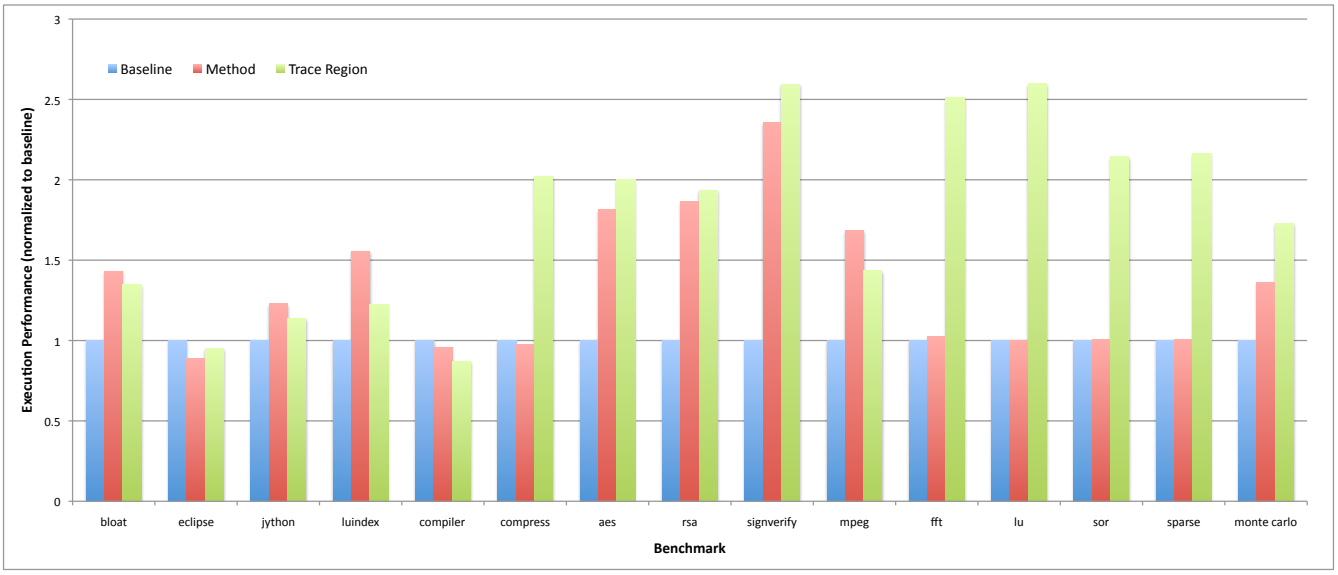

Figure 9. Benchmark results are normalized to the baseline Maxine non-optimizing just-in-time compiler (higher is better).

is able to pull ahead. Maxpath outperforms the Maxine optimizing compiler on all but the compiler and mpeg benchmarks. The compiler benchmark is the largest of the SPECjvm2008 benchmarks we selected, and Maxpath likely suffers in the same way that it did on the DaCapo benchmarks.

Overall the results show that Maxpath performs well and optimizes small computational kernels, but has trouble optimizing larger-scale benchmarks.

5. Related Work

Tracing is a well established technique for dynamic profile-guided optimization of native binaries. Bala et al. [2] introduced tracing as method for runtime optimizing native program binaries in their Dynamo system. They used backward branch targets as candidates for start of a trace, but did not attempt to capture traces of loops. Zaleski et al. [15] used Dynamo-like tracing in order to achieve inlining, indirect jump elimination, and other optimizations for Java. Their primary goal was to build an interpreter that could be extended to a tracing VM.

Gal et al. [7] proposed to build dynamic compilers in which no CFG is ever constructed, and no source code level compilation units such as methods are used. Instead, runtime profiling is used to detect frequently executed cyclic code paths in the program. The compiler then records and generates code from dynamically recorded *code traces* along these paths. It assembles these traces dynamically into a tree-like data-structure that covers frequently executed (and thus compilation worthy) code paths through hot code regions. A major benefit of this approach is that the trace tree data structure only contains actually relevant code areas. Edges that are not executed at runtime (but appear in the static CFG) are not considered in the trace representation, and are delegated to an interpreter in the rare cases they are taken. The absence of control flow merge points in this tree-based representation greatly simplifies optimization algorithms and this results in optimization passes being quicker than compilers that use traditional CFG-based analysis. The system relies on an interpreter to collect traces, while we utilize a just-in-time compiler.

Gal et al. [8] extended the previous work on trace-based compilation for Java and built a production-level traced-based VM (TraceMonkey) for JavaScript, currently shipping in the Mozilla Firefox Browser.

Whaley [14] uses *partial-method compilation* to reduce the granularity of compilation to the sub-method level. His system uses profile information to detect never or rarely executed parts of a method and to ignore them during compilation. If such a part gets executed later, execution continues in the interpreter. Compilation still starts at the beginning of a method.

Similarly, Suganuma et al. [11] propose *region-based compilation* to overcome the limitations of method-based compilation. They use heuristics and profiles to identify and eliminate rarely executed sections of code. In combination with method inlining, they try to group all frequently executed code in one compilation unit, but to exclude infrequently executed code. If an excluded code part has to be executed, they rely on recompilation and on-stack-replacement (OSR). Our trace-based compilation reaches this goal without requiring complex heuristics. They observed not only a reduction in compilation time, but also achieved better code quality due to rarely executed code being excluded from analysis and optimization.

Merrill et al. [9] presented a solution, implemented on the Jikes RVM, for selecting trace fragments within an non-interpreter based JVM. Their system compiles each method into two equivalent binary representations: a low-fidelity region with counters to profile hot loops and a high-fidelity region that has instrumentation to sample every code block reached by a trace. When a hot loop has been identified, the low-fidelity code transfers control to the high-fidelity region for trace formation. Once a trace has been formed, execution jumps back to the appropriate low-fidelity region. In their system, profiling and trace formation happens at the machine code level, not at the bytecode level as is the case in Maxpath. Unlike Maxpath, their system assembles traces by stiching together machine level basic blocks that were previously compiled with the whole method just-in-time compiler. Therefore higher-level compiler optimizations like CSE and loop invariant code motion are not performed along traces in their system. The Maxpath trace optimizing compiler on the other hand operates on CFGs and is able to perform optimizations accross many basic blocks and methods.

6. Conclusions

In conclusion, we have shown that trace-based compilation is feasible and beneficial in runtime environments without interpreters. We have introduced trace regions as the primary compilation unit and have shown how they can be recorded and grown using dynamic instrumentation. In order to execute trace regions, and return from them efficiently, we have presented a technique based on frame linking with the help of the register allocator. Lastly, we have described how a Java virtual machine can be retrofitted to support trace-based compilation with relatively little effort.

For this work we have built a new SSA-based optimizing compiler. Our compiler performs a series of optimization passes, the most interesting of which are alias analysis and invariant code motion. In future work we plan to focus more on trace specific optimizations, such as escape analysis, guard strength reduction and speculative execution.

Finally, our experimental results are promising, showing that compilation based on trace regions is comparable with method compilation in large-scale benchmarks, and performs better in smaller computationally heavy benchmarks.

Acknowledgments

Parts of this effort have been sponsored by the California MICRO Program and industrial sponsor Sun Microsystems under Project No. 07-127, as well as by the National Science Foundation (NSF) under grants CNS-0615443 and CNS-0627747. Further support has come from generous unrestricted gifts from Sun Microsystems, Google, and Mozilla, for which the authors are immensely grateful. The authors would also like to thank Bernd Mathiske, Doug Simon the rest of the Maxine team at Sun Labs for their guidance and support.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and should not be interpreted as necessarily representing the official views, policies, or endorsements, either expressed or implied, of the NSF, any other agency of the U.S. Government, or any of the companies mentioned above.

References

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005. doi: 10.1109/JPROC.2004.840305.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2000. doi: 10.1145/349299.349303.

[3] T. Ball and J. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Paris, France, December 1996.

[4] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. Technical Report MSR-TR-2010-27, Microsoft Research.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. pages 169–190. ACM Press, 2006. doi: 10.1145/1167473.1167488.

[6] A. Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine*. PhD thesis, University of California, Irvine, 2006.

[7] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 144–153. ACM Press, 2006. doi: 10.1145/1134760.1134780.

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, B. Kaplan, G. Hoare, D. Mandelin, B. Zbarsky, J. Orendorff, M. Bebenita, M. Chang, M. Franz, E. Smith, R. Reitmaier, and M. Haghighat. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 2009.

[9] D. Merrill and K. Hazelwood. Trace fragment selection within method-based JVMs. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 41–50. ACM Press, 2008.

[10] *SPECjvm2008*. Standard Performance Evaluation Corporation, 2008. http://www.spec.org/jvm2008/.

[11] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems*, 28(1):134–174, 2006. doi: 10.1145/1111596.1111600.

[12] Sun Microsystems. Maxine Virtual Machine. http://research.sun.com/projects/maxine/, 2008.

[13] B. L. Titzer, T. Würthinger, D. Simon, and M. Cintra. Improving compiler-runtime separation with XIR. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 39–50. ACM Press, 2010. doi: 10.1145/1735997.1736005.

[14] J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 166–179. ACM Press, 2001.

[15] M. Zaleski, A. D. Brown, and K. Stoodley. Yeti: a gradually extensible trace interpreter. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 83–93. ACM Press, 2007. doi: 10.1145/1254810.1254823.