A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler

Hiroshi Inoue†, Hiroshige Hayashizaki†, Peng Wu‡ and Toshio Nakatani†
† IBM Research – Tokyo
{inouehrs, hayashiz, nakatani}@jp.ibm.com
‡ IBM Research – T.J. Watson Research Center
pengwu@us.ibm.com

***Abstract*—This paper describes our trace-based JIT compiler (trace-JIT) for Java developed from a production-quality method-based JIT compiler (method-JIT). We first describe the design and implementation of our trace-JIT with emphasis on how we retrofitted a method-JIT as a trace-based compiler. Then we show that the trace-JIT often produces better quality code than the method-JIT by extending the compilation scope. Forming longer traces that span multiple methods turns out to be more powerful than method inlining in extending the compilation scope. It reduces method-invocation overhead and also offers more compiler optimization opportunities. However, the trace-JIT incurs additional runtime overhead compared to the method-JIT that may be offset by gains from the improved code quality. Overall, our trace-JIT achieved performance roughly comparable to the baseline method-JIT. We also discuss the issues in trace-based compilation from the viewpoint of compiler optimizations. Our results show the potentials of trace-based compilation as an alternative or complementary approach to compiling languages with mature method-based compilers.**

Keywords—trace-based compilation, JIT, Java

I. INTRODUCTION

Trace-based compilation uses dynamically-identified frequently-executed code sequences (traces) as units for compilation. This approach was first introduced by binary translators and optimizers [1, 2], where method structures are not available. Recently, trace-based compilation has gained popularity in dynamic scripting languages because it provides more opportunities for type specialization and concretization [3, 4, 5, 6] compared to the traditional method-based compilation. In spite of the success of trace-based compilation in dynamic language runtimes and binary translators, the benefits and drawbacks of trace-based compilation against mature method-based compilation have not yet been studied.

In this work, we explore trace-based compilation in Java as an alternative or complement to method-based compilation. Our motivation is to see if a trace-based JIT compiler (trace-JIT) can address a limitation of traditional method-based JIT compilers (method-JITs): limited compilation scope when dealing with those with largely flat execution profile. Today's method-JITs are very good at handling programs with hot spots, but not programs with flat profiles. This is because method-JITs cannot apply aggressive method inlining to cold methods to avoid excessive code duplication. As a result, programs with flat profile cannot be fully optimized. In contrast, trace-JIT can potentially improve the compiled code quality by forming larger compilation scopes than traditional method inlining even in cold program regions.

To identify the benefits and drawbacks of the trace-based compilation, we developed a trace-based JIT compiler based on the IBM J9/TR Java VM and method-JIT [7]. We extended the JVM to monitor the running Java program and to select hot code sequences as traces to be compiled. The code generator in the method-JIT was enhanced to accept a Java trace, which can start from the middle of a method, span multiple methods, and end at the middle of a method. We adapted most of the optimizers used in the method-JIT for the standard optimization level called *warm* to work in our trace-JIT. We also introduced two new techniques, allowing JNI calls in trace and introducing a shadow array to reduce hash lookup operations, to reduce the runtime overhead in the trace-JIT.

Prior to our work, it has not been demonstrated that a method-based optimization framework can be adapted for trace compilation. A recent publication [8] proved the unsoundness of some traditional optimizations such as dead store elimination in trace compilation. Our technical challenge is to address the unsoundness of method-based optimizations used in the trace-JIT. We identified that such unsoundness comes mainly from the mismatch between the trace compilation scope and the method compilation scope (*scope mismatch* problem). This is because some optimizations implicitly rely on properties that are only true within a method scope. For example, the dead store elimination assumes that local variables are dead after the end of the (method) compilation scope, which is no longer valid in the trace-JIT. We solve this problem by analyzing the bytecode sequence of a method, regardless of the current compilation scope. To achieve good optimizations in the trace-JIT, we need to use information from outside of the current (trace) compilation scope.

Our trace-JIT accelerated the latest DaCapo benchmark suite [9] by about 9 times on POWER6 processors over interpreted execution in the original JVM. It is about 4.5% slower than the warm-opt method-JIT. Although the trace-JIT often sped up the execution of compiled code, it incurred additional overhead due to monitoring and transitioning between compiled traces and the interpreter. The overall effect of improved compiled code quality and additional overhead depends on the application, but the relative performance of the trace-JIT over the method-JIT ranged from 21.5% slower to 26.4% faster. In the best case, our trace-JIT outperformed even the method-JIT with its full optimization capacities by 12.8%. We believe that these results are promising for trace-based

978-1-61284-357-5/11/$26.00 ©2011 IEEE

compilation considering the fact that our trace-JIT still has a huge optimization space to explore.

This paper makes the following contributions. (1) We describe the design and implementation of our trace-JIT for Java with emphasis on how we retrofitted a method-based compiler as a trace-based compiler. We identified that the differences of trace-JIT and method-JIT mainly come from the *scope mismatch*. (2) We evaluate the trace-JIT using the method-JIT as a reference point in steady-state performance, startup performance, compiled code size, and compilation time. Such a comparison sheds light on the effectiveness of the trace-JIT with respect to a mature, state-of-the-art compiler, which has never been done in previous evaluation of trace compilation. Since the two compilers share the same interpreter, libraries, garbage collector, and compilation framework, the evaluation can provide some insights on the potential strength of trace compilation over method compilation as well as highlights its limitations. (3) We present two new techniques to improve the trace-JIT performance by reducing the runtime overhead.

The rest of the paper is organized as follows. Section II discusses previous techniques. Section III gives an overview of our trace-based JIT. Section IV illustrates our proposed techniques to reduce the runtime overhead. Section V describes our experimental results. Section VI considers remaining problems in the trace-JIT and discusses future directions. Finally, Section VII summarizes this work.

II. Related Work

Dynamo [1] was the first trace-based optimizing compiler. The traces are formed out of binaries by a binary interpreter. Dynamo pioneered many early concepts of trace selection and trace runtime management. The optimizer is extremely lightweight and performs basic redundancy elimination optimizations in a forward and a backward pass as well as basic register allocation.

HotpathVM [10], YETI [11] and Maxpath [12] are trace-JITs that target Java. HotpathVM emphasizes its efficiency in domains where resources are quite constrained. It introduces a tree-SSA representation for fast analysis. It also introduces pseudo-instructions to represent information about the surrounding context that is not part of the trace. This is similar to the handling of scope mismatch in our trace-JIT. YETI showed that the trace-based compilation eased the development of a JIT compiler by allowing incremental implementation of the compiler on top of the language VM. Maxpath is a trace-JIT designed for Java environment without an interpreter. It first compiles the Java programs using the non-optimizing compiler with instrumentations to select hot sequences as traces. The traces are then compiled by the optimizing trace compiler. Maxpath reuses the code generator of Maxine method-JIT, and achieved better performance than the method-JIT.

Recently, trace compilation has been explored extensively in compilation for dynamic scripting languages, such as PyPy [5] for Python, SPUR [6] and TraceMonkey [3] for JavaScript, and LuaJIT [4] and [13] for Lua, and Tamarin-Tracing for ActionScript [14]. Such systems tend to recognize more complex traces (such as traces representing nested loops). As the traces become more complex, so does the functionality of the optimizer. TraceMonkey, for example, performs extensive type specialization for nested cyclic traces. SPUR performs redundant guard elimination, indirect store-load forwarding, invariant code motion, and loop unrolling. PyPy uses escape analysis, store-load forwarding, and redundant guard elimination for its traces.

One common trait of previous trace-JITs is that the optimizer is specifically tailored to the simple topology of traces in consideration for compilation speed as well as reducing the development cost of the compiler itself. As a result, the optimizer is simpler but also less powerful than a typical method-based optimizer. In our work, we retrofitted a method-based JIT for trace code generation and optimization. Our approach can leverage existing mature optimization infrastructures. Our trace optimizer is more powerful in the sense that it is a true region-based optimizer that can potentially optimize traces of arbitrary structures. It also opens up opportunities to combine trace- and method-based compilation in one framework with minimal maintenance overhead.

The most relevant theoretical work to ours is [8], where Guo *et al* studied the soundness of traditional method-based optimizations on trace compilation (and vice versa). The work proved that traditional forward data-flow optimizations such as folding and dead branch elimination are sound for trace compilation; whereas backward data-flow optimizations such as dead store elimination are not. These conclusions are consistent with our experiences in retrofitting the method-JIT to the trace-JIT. The unsoundness in backward data-flow, for example, is rooted in the scope mismatch problem we pointed out in the paper. One contribution of our work is that we addressed the unsoundness problem by extending existing method-based optimizations with special consideration of traces.

Based on our results, we conclude that generating long traces is a key to achieving good performance for a trace-JIT because it reduces the transitioning overhead and improves the opportunities for compiler optimizations. Zhao *et al* [15] also pointed out that generating longer traces yielded more optimization opportunities in their binary translator.

III. Design of Our System With Trace-JIT

In this section, we describe the architecture of our trace-JIT for Java, which is based on an IBM J9/TR JVM and method-JIT. Figure 1 is an overview of the entire system. Our system starts its execution using an interpreter until hot traces are identified and compiled. We do not use the method-JIT in our system, instead relying on mixed execution of the interpreter and the trace-JIT.

A. Tracing Runtime and Trace Selection Algorithm

We implemented a new software component, a *tracing runtime*, to monitor the execution of the running program and to select hot code sequences to compile. The tracing runtime is driven by execution events sent from the interpreter. To drive the tracing runtime, we modified the Java interpreter to call the tracing runtime at control-flow events including branches (goto, if, or switch), method invocations, method returns, and

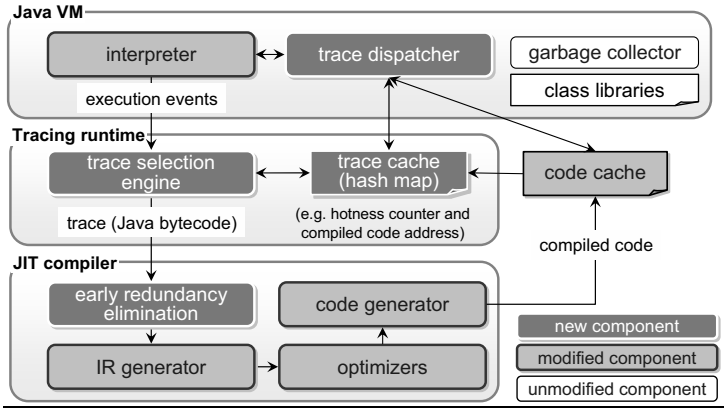

Figure 1. Overview of our trace-JIT system architecture.

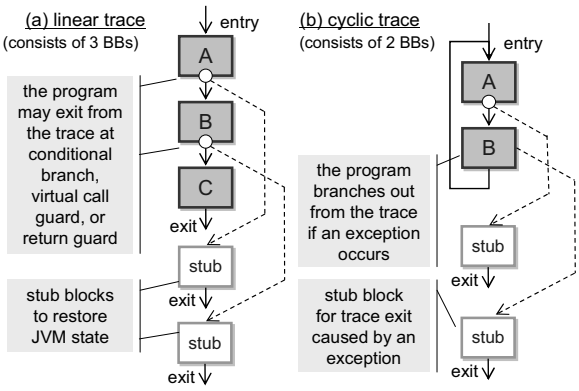

Figure 2. Example of (a) a linear trace and (b) a cyclic trace. Each box shows a basic block.

exception throws or catches. By abstracting at the level of execution events, our tracing runtime supports multiple language runtime systems including the JVM that we describe in this paper.

Our trace selection algorithm first determines a point to start a trace (a *trace head*) and then records the next execution starting from the trace head as a trace, similar to the well-known NET (Next Executing Tail) strategy [1]. To focus on the basic trace-JIT characteristics, we currently collect only linear traces or cyclic traces (where a cyclic trace has a jump to its own trace head at the end). Hence we do not have any join points in our traces except for the trace head of a cyclic trace. Figure 2 shows examples of a linear trace and a cyclic trace.

To identify a hot trace head, we assign a counter called a *hotness counter* for each potential trace head, which includes the target of a taken backward branch and any bytecode that immediately follows the exit point of an already formed trace to achieve sufficiently high coverage for the JIT compiled code. We manage the information associated with the bytecode addresses, such as the hotness counter or the compiled code address, using a globally synchronized hash map called a *trace cache*. The trace selection engine increments the hotness counter when it receives an event for the bytecode address and it starts recording the execution to form a trace when the hotness counter reaches a predefined threshold. We used 500 as the threshold in this paper. For example, a loop head is selected as a trace head after 500 iterations. We picked the threshold based on the thresholds used in the baseline method-JIT to start initial compilation.

In the recording mode, the trace selection engine records all basic blocks (BBs) executed until one of the trace termination conditions is satisfied. We terminate a trace when (1) it forms a cycle in the recording buffer, (2) it executes a backward branch (even it does not form a cycle), (3) it calls a native (JNI) method that we cannot include in a trace, (4) it throws an exception, or (5) the recording buffer becomes full. The default size of the recording buffer is 128 BBs for one recording. As we will describe in Section IV, we allow traces to include calls to a selected set of JNI methods from the Java standard library to maximize the performance. Calls to other JNI methods will terminate the recording of a trace. If a trace forms a cycle by jumping to its trace head, the trace becomes a cyclic trace. We identify a cyclic execution patterns accurately by checking the calling context of each BB in the trace [16]. Otherwise, the trace becomes a linear trace. A trace collected by the selection engine is sent to a shared waiting queue that is processed by the compilation thread.

When the compilation thread compiles the trace, it puts the entry point address of the compiled code in the trace cache. Once the compiled code address becomes available in the trace cache, the interpreter transfers control to the entry point of the compiled code when the execution reaches the head of a trace. At the exit of the compiled trace, it returns control to the interpreter or directly dispatches the next compiled trace using a technique called trace linking [1]. Currently we do not employ a specialization technique and thus there is at most one trace starting from the same bytecode address.

B. *Trace-based JIT Compiler and Scope Mismatch*

We implemented our trace-JIT by enhancing a mature method-JIT instead of implementing it from scratch. Our trace-JIT takes a Java bytecode sequence and the originating location (Java method and bytecode index in the method) for each bytecode in the trace as input.

In trace-based compilation, a compilation scope probably does not match the method scope. Thus we need to assume that local variables and operands in the operand stack may live at the beginning and the end of the compilation scope, while all these values must be dead in the method-based compilation. We call this problem *scope mismatch*. Scope mismatch is a large obstacle when implementing a trace-based compiler from a method-based compiler. For example, the first bytecode in a trace may require operands on the operand stack, but a compiler cannot identify the type of the operands because the value comes from outside the current compilation scope. To handle problems caused by scope mismatch, we implemented a helper function that analyzes the bytecode sequence of a method, regardless of the current compilation scope. The helper function identifies the type and liveness of operands on stack and local variables at the specified program location. The liveness information at compilation scope boundaries obtained from this helper function is critical for both code generation and optimization. For example, the IR (Intermediate

Representation) generator, which translates Java bytecode to the compiler's IR, uses this helper function to identify the type of the operands at the beginning of a trace and inserts the appropriate load instruction before the first instruction.

C. Optimizers

Our trace-JIT uses most of the optimizers from the existing method-JIT in the standard optimization level called *warm*. Most of the issues in reusing these method-JIT optimizers were due to scope mismatch. The problem often appears in optimizations that rely on live range information such as dead store elimination (DSE). Here, we explain how we modified the optimizations of the existing method-based compilers to use in the trace-JIT using DSE as an example. DSE for the method-JIT assumes that all local variables are live only within the current compilation scope, but this assumption is not true for trace-JIT. Local variables may be accessed after exiting the current trace. One naive modification for the trace-JIT is to assume all local variables are live at the end of the compilation scope, but this naive approach drastically reduces the benefits of DSE. To prevent DSE from removing live variables without sacrificing the advantages of DSE, we extend the use-def analyzer and the live range analyzer, which are shared by several optimizations including DSE and GC metadata generation, to identify the live variables at each trace exit by calling the helper function described in Section III-B. In general, to achieve good optimizations in the trace-JIT, we need to use knowledge from outside of the current compilation scope.

In contrast, the DSE in the trace-JIT may miss opportunities if the trace spans multiple methods. The method shown in Figure 3 has an obvious dead store (istore_1 in line 7). The method-JIT can easily remove this dead store without costly global analysis because it is located at the end of the compilation scope and the compilation scope matches the live range of the variable. With trace-JIT, however, the live range of the variable does not match the compilation scope and hence the DSE cannot identify the dead store in the example if this method appears in the middle of a trace. To enhance the optimizations by expressing the live ranges in IR trees, we introduced a new IR opcode to show the end of the live range of a local variable (*end-of-live-range mark*). The IR generator inserts the end-of-live-range mark at the end of method scope. In the example of Figure 4, the IR generator inserts the end-of-live-range mark for the two local variables of the getTotal() method after Line 7. With the end-of-live-range mark, the DSE can easily identify the dead store without costly global analysis as in the method-JIT. We minimize the implementation effort by defining the end-of-live-range IR opcode as a special case of the store opcode, which is ignored in the code generator. Because a store kills the live range of the existing variable, most of the existing optimizers identify the new IR opcode without modification. In general, it is important for trace-JIT optimizers to be aware of the live range of local variables regardless of the current compilation scope.

In addition to DSE, we also modified most of the optimizations used in the warm-opt method-JIT, such as common subexpression elimination, dead code elimination, value propagation, and global register allocation. In the current implementation, we do not enable some of the loop optimizers

```
...
1: aload_0
2: getfield  field1
3: aload_0
4: getfield  field2
5: iadd
6: dup
7: istore_1 (dead store)
8: ireturn
...
```

```
Java code:
public int getTotal() {
    int result = field1 + field2;
    return result;
}
```

We insert end-of-live-range marks for local variables at method boundaries in a trace.

Figure 3. Example of a dead store and end-of-live-range marks.

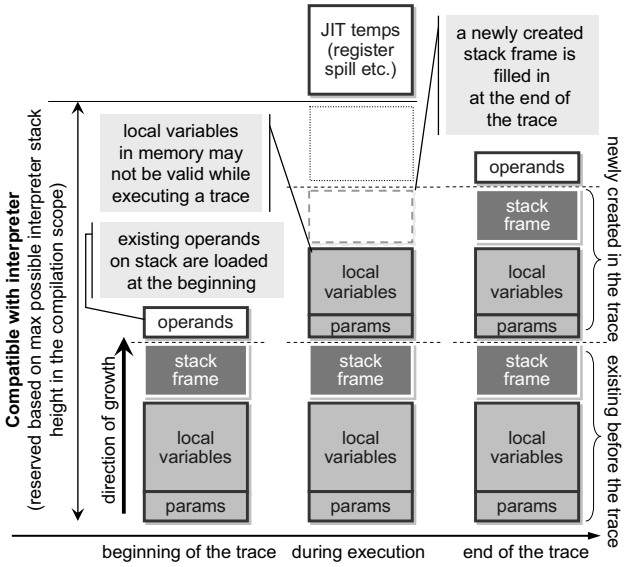

Figure 4. Java stack design of our trace-JIT.

nor the optimizers that are not applicable for the trace compilation, such as method inlining. We will discuss a fundamental problem in loop optimizations with the existing trace selection algorithm in Section VI.

D. Java Stack Design and GC

Because we assume mixed execution of the interpreter and the trace-JIT, our trace-JIT employs a Java stack design compatible with the interpreter stack design to avoid overhead in transition between the interpreter and compiled code caused by on-stack replacement [17]. Figure 4 illustrates our Java stack. We assure that the Java stack, including local variables, stack frames, and operands, is compatible with that of the interpreter at the end of a trace. To speed up the common path of trace execution, we maintain the Java stack lazily. When a method is invoked in the trace, the stack frame for the invoked method is created at the end of the trace only when the frame is still live at the end. The local variables in memory may become invalid during the execution of a trace because we aggressively allocate registers for local variables and also reorder loads and stores to maximize the performance.

To ensure that the JVM states are compatible with the interpreter at the trace exit points, we prepare an exit code sequence for each bytecode that potentially leads to an exit from the trace, including conditional branches, switches,

invocations of a virtual or interface method, and method returns. If an exception occurs during the execution of a compiled trace, we exit from the trace and fall back to the interpreter. Thus we also prepare an exit code sequence for each bytecode that can potentially throw an exception, such as loads and stores for an object's field or numerical division instructions. Figure 2(b) includes an example of an exit caused by a potential exception.

If a stack walk is required while a Java thread is executing a compiled trace, for example when a stop-the-world garbage collection begins, we fill in all of the live stack frames in the Java stack before the stack walker in the JVM is initiated. We prepare special metadata for this construction at the JIT compilation time for each GC-safe point. We also provide a bitmap to show all live references in the Java stack and the registers for each GC-safe point in the same way as the method-JIT. To generate this bitmap, we needed to extend the live range analyzer in the method-JIT by taking into account *scope mismatch* in the trace-JIT. For example, a local variable may be live at the beginning of the compilation scope because it is used before entering the trace that starts from the middle of a method. Likewise, a local variable with no explicit reference in the given scope may be live at the end of the compilation scope because it is accessed after the trace ends in the middle of a method. In extreme cases, there are live variables that were never accessed in the compilation scope. Hence, it is not possible to generate accurate live variable information only from the bytecode sequence of the trace.

The changes in stack design affect only the stack walker in the JVM. We do not need to modify the garbage collectors. Our trace-JIT can run with any of the garbage collectors supported in the original JVM.

E. Interprocedural Traces Support

Our trace-JIT takes a trace that might span multiple (sometimes tens of) Java methods as input. Because there can be multiple targets for a virtual method invocation or a method return and the actual target can be different from the target recorded on a trace, we insert runtime checks at method boundaries. For each virtual method invocation, we insert a class equality check as a runtime guard, which compares the class of the receiver object and the class at the recording time of the trace. If the class equality check fails, we exit from the current trace. We also insert a guard for method return by comparing the targets of the return at runtime and at recording time. We avoid inserting a redundant check at a return instruction if the corresponding invocation is also included in the same trace.

F. Other Performance Optimizations

Our trace-JIT supports trace linking optimizations [1] to reduce the overhead in the tracing runtime. We directly link each trace exit to the entry point of the next trace without going back to the interpreter if the trace exit has only one candidate for the next program counter value. This is accomplished by code patching to the exit code sequence. At linking time, we check the Java stack requirements for both traces. If the preceding trace requires a larger Java stack compared to the following trace, then we skip the stack overflow check at the beginning of the following trace. Such check can be skipped because the following trace, which uses smaller Java stack than the previous trace, cannot cause a stack overflow. When the trace exit has multiple candidates for the next program counter value, such as an exit at method return or virtual method invocation, we do not use the trace linking. Instead, when a trace ends with a method return, we inline the code to find the next trace using the shadow array, as described in Section IV, to avoid a transition to the interpreter.

In the IR generation phase, for each trace exit point we need to prepare an exit code sequence, which restores the JVM state compatible with that of the interpreter. The exit code sequences increase the size of the generated compiler IR tree and thus affect the compilation time. To minimize the increase in the IR tree size, we employ a simple one-path value propagation phase before the IR generation phase. We call it *early redundancy elimination*. This phase identifies the redundant NULL checks, virtual call guards, and conditional branches. The later phase optimizer executes more detailed analysis and redundant check elimination, but the earlier redundancy elimination contributes to reducing the execution time of the optimizers. By exploiting the simple topology of the compilation scope, the execution time of the early redundancy elimination phase is almost negligible. In the current implementation, this phase is integrated with the IR generation phase. We generate only one general code sequence to fill in the stack frames in each trace and all exit points share the same sequence. Because a trace frequently has lots of exit points, this sharing greatly reduced the overall compiled code size.

G. Current Limitations

Currently we only support one optimization level and we do not support upgrade compilation. The method-JIT already supports upgrade compilation to higher optimization levels. At the higher optimization levels, a profiling mechanism plays an important role to generate specialized compiled code. We do not yet support any profiling mechanism in the current implementation of our trace-JIT. The trace selection itself is a kind of profiling and thus more detailed profiling, such as value profiling, can be implemented in the trace selection phase with little additional overhead.

We do not specialize the optimizations for the trace-JIT and so they can accept any IR trees, including trees containing join points in the control flow. Also, we do not carefully tune the optimizations for trace compilation. Hence there are huge opportunities to improve both the optimized code quality and the compilation speed by exploiting a simpler structure of the traces that have no inner join points in the control flow. Only the early redundancy elimination phase executed prior to the IR generator exploits a simple topology of the traces to reduce the compilation time. Though we do not employ the trace-specific optimization techniques after the IR tree is generated, the baseline method-JIT already covers most of the optimizations used by the previous trace-based compilers. For example, the baseline method-JIT has an optimization to move instructions to cold blocks based on execution frequency. In our trace-JIT, we treat an exit code sequence as a cold block and hence redundant computations are moved to the exit code sequence. Even with the same optimizations, the trace-JIT can leverage the benefits of the trace, such as a larger compilation scope and

simpler control flow. For example, basic-block-local optimizations are often more effective for trace-JIT than for method-JIT, because the main path of a trace forms a large extended basic block.

IV. Runtime Overhead Reduction

This section describes two new techniques we introduced to reduce the runtime overhead in the trace-JIT.

A. Hash Lookup Reduction Using a Shadow Array

In our trace-JIT, the overhead to search the global hash table (*trace cache*) is significant, because we frequently search the address of the compiled code or the hotness counter associated with the current bytecode address from the trace cache. To avoid such overhead, we allocate a shadow array for each method to directly find the address of the compiled code or the hotness counter. Because we allocate one word (4 bytes) for each bytecode index in a method, the size of the shadow array entry is four times larger than the bytecode itself. To avoid excessive memory consumption by the shadow array, we allocate the shadow array entry only when a potential trace head is identified in the method for the first time. The shadow array element associated with a bytecode index holds the address for the hotness counter until compiled code starting from that address becomes available. After the compiled code becomes available, we replace the address of the hotness counter with the compiled code address to maximize the steady-state performance. The number of hash table lookups becomes negligible when we use the shadow array. We will describe the performance improvements and the memory overhead for the shadow array in Section V.

B. JNI Inclusion in Trace

In HotpathVM [10], a trace is always terminated when the trace encounters a native (JNI) method call. When a JNI call appears frequently on hot paths of a running program, this stop-at-JNI approach causes frequent trace exits. This incurs significant transition overhead. In our trace selection algorithm, we allow some JNI methods in traces (JNI inclusion): to continue a trace at a method call to certain native methods, and to allow a trace to call such native methods without exiting from the trace. The JNI inclusion can reduce the runtime overhead and improve the compiled code quality by generating longer traces. Also, the baseline method-JIT inlines the frequently used JNI methods, such as the methods of the `java.lang.StrictMath` class, into the compiled code. By allowing these JNI calls in the traces, our trace-JIT can leverage this JNI call inlining capability of the baseline method-JIT.

To simplify the implementation, we allow JNI inclusion only for certain JNI methods in the standard libraries that are known to never cause Java-related events (GC, exceptions, or any other JNI interface calls including Java method callbacks). We do not need to modify the native methods to be called from a trace. JNI method calls in traces are handled in a similar way to those of the method-JIT. The benefits of JNI inclusion are quite significant in some programs and the maximum acceleration we observed solely from this technique was about 2.7 times.

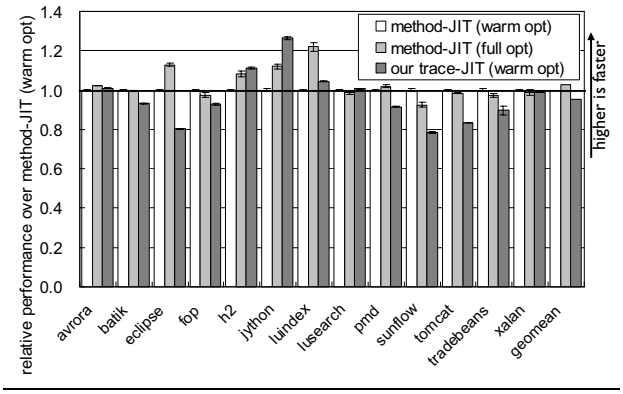

Figure 5. Relative steady-state performance of our trace-JIT over method-JIT for each benchmark.

V. Performance Evaluation

This section evaluates our trace-JIT by comparing it to the method-JIT. We ran the evaluation on an IBM BladeCenter JS22 using 4 cores of 4.0-GHz POWER6 processors with 2 SMT threads per core. The system has 16 GB of system memory and runs AIX 6.1. The size of the Java heap was 512 MB using 16-MB pages and we used the generational garbage collector. To focus on the differences between method-based and trace-based compilations, we compare the performance of our trace-JIT against the method-JIT with the warm optimization level, which uses almost the same set of optimizations as the trace-JIT. For this configuration of the method-JIT, we add a JVM option: `-Xjit:optLevel=warm`. We also compared the trace-JIT against the method-JIT with its full optimization capacities by enabling the upgrade compilation to higher optimization levels (full opt). In our evaluation, we used DaCapo 9.12 [9] running with the default data size in our tests. We did not include the `tradesoap` benchmark because the baseline system with the method-JIT sometimes caused an error for this benchmark. For each result, we report the average of 16 runs along with the 95% confidence interval.

A. Performance of Trace-JIT and Method-JIT

Figure 5 compares the steady-state performance of our trace-JIT to the method-JIT with two optimization levels. We took the average of 5 iterations after 20 iterations of warm up in each run (but 5 iterations of warm up for `eclipse` and 50 iterations for `fop` because they run much longer and shorter, respectively, compared to the other benchmarks). Compared to the warm-opt method-JIT, our trace-JIT outperformed the method-JIT in three benchmarks, while it was slower in seven benchmarks. The overall average performance of our trace-JIT was 95.5% of the warm-opt method-JIT. Compared to the full-opt method-JIT, the performance of our trace-JIT was 92.8% on average. For the `jython` benchmark, which is a python runtime implemented on top of the JVM, our trace-JIT outperformed the warm-opt method-JIT by 26.4% and the full opt by 12.8%. In contrast, our trace-JIT was slower than the warm-opt method-JIT by more than 15% for three benchmarks (`eclipse`, `sunflow`, and `tomcat`).

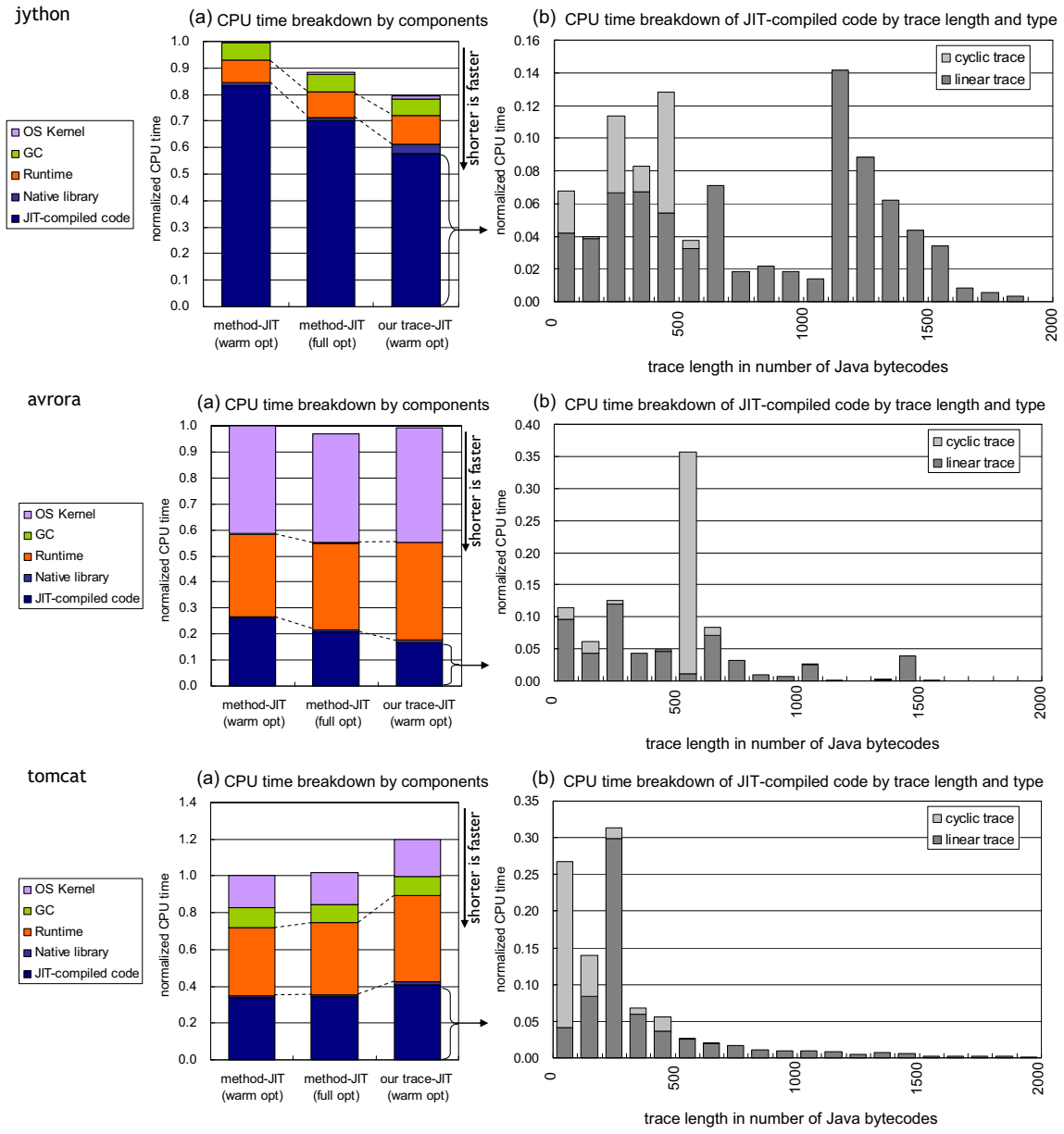

Figure 6. (a) CPU time breakdown into JVM components and (b) detailed breakdown of CPU time spent in JIT-compiled code by trace length and type.

For further insight into the causes of the performance differences between the trace-JIT and the method-JIT, Figure 6(a) shows profiles of the CPU time for three benchmarks broken down into the JVM components based on how much active CPU time was spent in each component as measured by using the hardware performance monitor of the processor. We normalized the profiles based on the peak performance of the warm-opt method-JIT shown in Figure 5. We show the results for jython, in which our trace-JIT was 26.4% faster than the warm-opt method-JIT, for avrora, in which the trace-JIT was roughly tied with the method-JIT, and for tomcat, in which the trace-JIT was 15.6% slower than the method-JIT.

In jython and avrora, the JIT-generated code components for trace-JIT were shorter than those for the warm-opt and full-opt method-JIT. Since in all of the benchmarks more than 99% of the bytecode is executed out of compiled traces, this means that the trace-JIT generated better code for these two benchmarks. However, the JIT-generated code component for tomcat was larger than the method-JIT. Thus, for this benchmark, our trace-JIT failed to generate better code compared to the method-JIT. Note that the longer CPU time in the Native library component for trace-JIT was because some native methods were not inlined into the compiled code in the trace-JIT, while they were inlined into the compiled code in the

method-JIT. Hence we added this component to the JIT-compiled code component for a fair comparison.

In all three benchmarks, there is a net increase of CPU time in the runtime component for the trace-JIT. Most of this increased CPU time in the runtime was spent in transitioning code between the traces and the interpreter and in the hook interface that monitors the control-flow events for the trace selection. In jython, the reduction in the JIT-generated code components was more significant than the increase in the runtime components due to the additional runtime overhead. As a result, the trace-JIT outperformed the method-JIT. In avrora, the improvement in the JIT-compiled code and the increased runtime overhead were almost comparable and hence the trace-JIT achieved similar performance to the method-JIT. For tomcat, the trace-JIT was not able to compete with the method-JIT due to the additional runtime overhead and the inferior JIT-compiled code performance.

To show the causes of the differences among the three benchmarks, we show a detailed breakdown of the CPU time spent in the JIT-compiled code within the trace-JIT by the trace length (in number of Java bytecodes) and the topology (cyclic or linear) in Figure 6(b). In the figure, the x-axis shows the trace length and a larger value means a longer trace. Obviously, jython spent much more CPU time in the long traces compared to the other two benchmarks. For example, the ratio of CPU time spent in traces longer than 1,000 bytecodes was 40.1% for jython, while it was only 11.4% for avrora and 6.4% for tomcat. A longer trace tends to span more methods, and hence it reduces method invocation overheads and offers more optimization opportunities to the JIT compiler by extending the compilation scope. Also, longer traces (or highly iterated cyclic traces) mean less transitioning between traces, and thus less runtime overhead. In contrast, shorter linear traces cause frequent transitioning between compiled traces and the interpreter. In tomcat, about a half of the CPU time was spent in linear traces shorter than 400 bytecodes. Such transitioning overhead is charged in both the JIT-compiled code component and the runtime component in the breakdowns. In the JIT-compiled code, we ensure that the JVM states are compatible with those of the interpreter before exiting the compiled code. Then we find the next trace to execute in the runtime. The trace linking and our shadow array technique greatly reduce this runtime overhead, but the overhead still matters for the overall performance of some benchmarks.

These results show that generating longer traces is quite important for trace-based compilers to achieve superior performance. The method-JIT also extends the compilation scope by method inlining to some extent, but the trace-JIT is more capable to extend the compilation scope. That is the main potential area where the trace-based compilation may beat method-based compilation in certain scenarios.

B. Compiled Code Size and Total Compilation Time

Figure 7 shows the relative compiled code size of our trace-JIT and the method-JIT. The relative size of the generated code from the trace-JIT ranged from 52% smaller (luindex) to 4.9 times larger (pmd) relative to the warm-opt method-JIT. Averaging all of the benchmarks, the trace-JIT generated

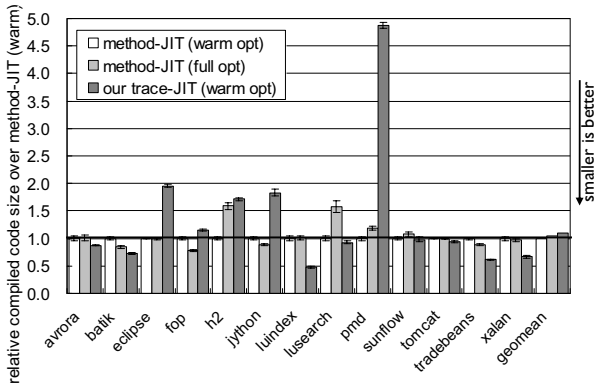

Figure 7. Relative compiled code size of our trace-JIT over method-JIT for each benchmark.

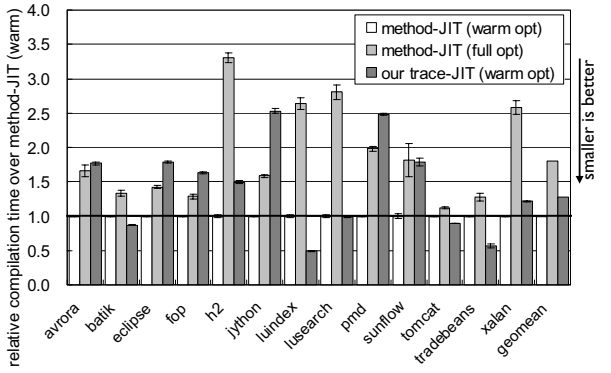

Figure 8. Relative compilation time of our trace-JIT over method-JIT for each benchmark.

10.5% more code. The larger code size for the trace-JIT was caused by the redundant trace selection and the extra code size used for exit stubs.

The trace selection algorithm in our trace-JIT emphasizes the performance over the code size by allowing duplication among traces. For example, the same method called from many different places may appear in many traces because it achieves an effect equivalent to method inlining. The warm-opt and full-opt method-JIT also allow duplication by method inlining, but we found that duplication by method inlining was more limited than duplication by our trace selection. Method inlining in the method-JIT is well tuned to inline only important methods without significant code bloat. We observed that the compiled code size for trace-JIT was smaller than that of the method-JIT when we use the NET [1] selection algorithm, which greatly reduces the duplications, but the performance of the trace-JIT was degraded. How to design selection algorithms that balance between trace length and space efficiency is still future work. For example, adaptively combining the current selection algorithm with another selection algorithm with good space efficiency, such as NET, might be a good approach to reduce the code size in the trace-JIT.

In the compiled traces, the exit stubs occupy on average 44.9% of the total code size. Because we need to prepare an exit code sequence for each potential trace exit, reducing the number of instructions in each exit stub could greatly reduce

the overall code size. We already minimized the exit stubs by omitting unnecessary exit stubs and also by sharing the same code among the stubs, as described in Section III. On the positive side of the trace-based compilation, traces include only frequently executed code sequence and hence rarely executed code blocks such as error handlers are not compiled. Because the method-JIT compiles the entire method including these cold blocks, the trace-JIT can potentially reduce the code size compared to the method-JIT.

Figure 8 shows the total compilation time of our trace-JIT and the method-JIT. On average, our trace-JIT consumed 27.0% more compilation time than the warm-opt method-JIT. The full-opt method-JIT had much longer compilation time due to more aggressive optimizations used in upgrade compilation for hot methods. In general, trace-based compilers sometimes had much shorter compilation time than method-based compilers by exploiting a simpler topology of the compilation scope. However, because our trace-JIT reuses the optimizers originally developed for the method-JIT, the compilation time for our trace-JIT was not shorter than that of the method-JIT. For the worst case, the trace-JIT consumed 2.5 times more compilation time than the warm-opt method-JIT in jython and pmd. In jython our trace-JIT formed very long traces, as shown in Figure 6. The long traces tend to require more compilation time than shorter traces in exchange for more optimization opportunities, even if the total amounts of code to compile were comparable. The long compilation time in pmd was due to a large amount of duplicated code among traces.

C. Startup Performance

Figure 9 compares the startup performance of the trace-JIT and the method-JIT. Here, we use the performance of the first iteration of each benchmark as the startup performance. In Figure 5 we show the startup performance relative to the steady-state performance of the warm-opt method-JIT. Obviously, the startup performance of the trace-JIT is strongly correlated with the total compilation time shown in Figure 8 because the execution time of the first iteration includes much of the total compilation time. Faster startup performance with the full-opt method-JIT was due to the fact that it starts compilation with a lower optimization level (cold) and then gradually upgrades only the important methods.

D. Memory Usage

Figure 10 shows breakdowns of the memory usage into major data structures related to the JIT compilers, averaged for all of the benchmarks. Because we built our trace-JIT on top of the method-JIT, the trace-JIT consumes more memory than the method-JIT due to additional data structures specific to the trace-JIT. In the figure, *trace structure* is a data structure that manages the current status of an already formed trace. It is allocated for each trace. *BB structure* is a data structure that represents each basic block included in a trace. BB structure is not required for execution after the trace including the BB is compiled. In the current implementation, however, we do not delete BB structures after compilation as we collect statistics. As already described, the *trace cache* is a large hash table to manage information associated with each byte code address, and the *shadow array* is used to reduce the number of searches

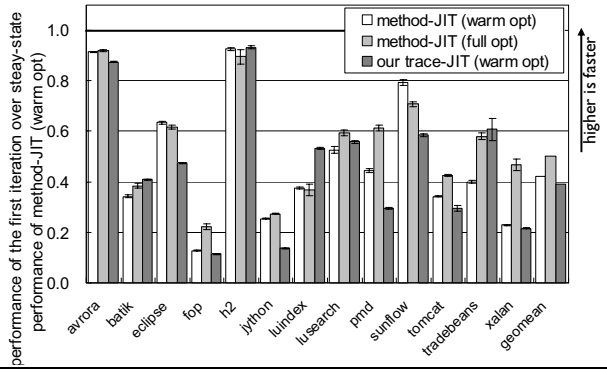

Figure 9. Startup performance (performance of the first iteration) over steady state performance of method-JIT (warm opt) for each benchmark.

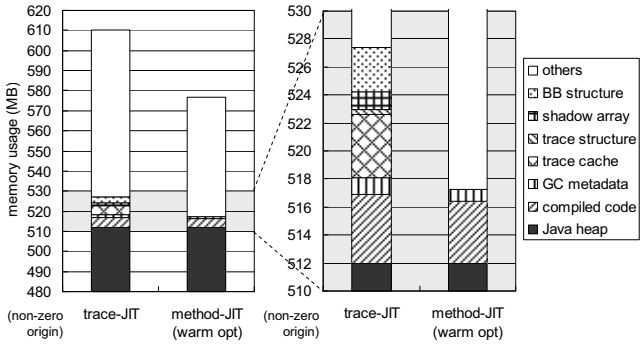

Figure 10. Comparison of the memory usage by our trace-JIT and method-JIT (warm) on average of all benchmarks.

in the trace cache. These trace-specific structures were about twice as large as the total compiled code. We also saw increases in the unaccountable memory consumption (*others* in the figure) for our trace-JIT. We suspect that this was due to the fact that our trace-JIT required more working memory to compile targets having larger compilation scope than the method-JIT. In fact, we found the largest increases for jython, in which the trace-JIT formed long traces.

E. Effect of Maximum Trace Length

In our trace-JIT, the maximum trace length is limited by the size of the trace recording buffer as described in Section III and the default size of the buffer is 128 BBs. We decided on this default value based on our experiments. The trace buffer size involves a trade off between the compiled code size, which often dominates the startup performance, and the steady-state performance. For example, using 256 BBs as the recording buffer size increased the compiled code size by 7.9% while it increased the steady-state performance by 0.4%. Most of the tested benchmarks were not accelerated by using a larger buffer size. Among the benchmarks jython showed the largest speed up of 6.2%. In jython, many hot traces are terminated due to the buffer size limitation and hence the buffer size larger than 128 BBs increased the performance. Because jython benchmark in the DaCapo suite executes rather simple python programs, many long traces that span both translated python

code and jython runtime code are formed without hitting a loop, which terminates the trace. This is one of the best possible cases for the trace-based compilation over method-based compilation. When using 64 BBs as the recording buffer size, we observed about 2.2% performance degradation in exchange for 10.6% smaller compiled code size on average.

F. Effect of Our Runtime Overhead Reduction Techniques in Trace-JIT

Figure 11 compares the performance of our trace-JIT with and without our hash lookup reduction technique. The graph shows that this technique enhanced the performance of the trace-JIT in all of the programs, with an average improvement of 27.4%. This was because the number of hash lookups was quite significant in the trace-JIT without our hash lookup reduction technique. In fact, we observed more than one million lookups per second during our evaluations. With our reduction technique, however, we observed only negligible number of lookups. Our technique used some additional memory for the shadow array for each method to achieve this performance improvement. Figure 12 illustrates the total size of the allocated shadow arrays for each benchmark. The total size of the shadow arrays was 1.3 MB on average, as shown in Figure 10, and up to 6.8 MB. We believe that this memory consumption was reasonable for the achieved performance improvements.

Figure 13 shows how the JNI inclusion affected the overall performance of the trace-JIT. The improvement was quite significant in sunflow (2.7 times improvement), while the improvement was not significant in some other benchmarks. Terminating a trace at each JNI call increased the runtime overhead and also reduced the opportunities for compiler optimizations by shortening the trace lengths. In sunflow, a JNI call appears in a hot loop and hence the effect of including JNI calls was most significant among all of the tested benchmarks. Though the JNI calls in Java programs look relatively rare, our results showed that proper handling of the JNI calls in trace selection is important for overall performance.

VI. Discussion and Future Work

A. Loop Optimization in Trace-JIT

Though our trace-JIT uses many of the optimizers from the warm-opt method-JIT, we did not enable some of the loop optimizers, such as loop unrolling. This is because the loop optimizers for the method-JIT did not work well for the cyclic traces selected by our current selection algorithm. Figure 14 shows an example of a loop selected as a cyclic trace by the current selection algorithm. Our trace selection algorithm captures hot loops by counting the number of taken backward branches as in the earlier selection algorithms. After identifying a hot backward branch, we record the next execution starting from the target address of the backward branch as a trace. This selection algorithm can capture the loop body of the hot loop, but the selected trace does not include loop preheader, such as the initialization of the loop induction variable. Because the loop preheader is not included in the selected cyclic trace, the loop optimizer cannot find the number of iterations to execute and hence it may miss an opportunity to optimize the loop. As

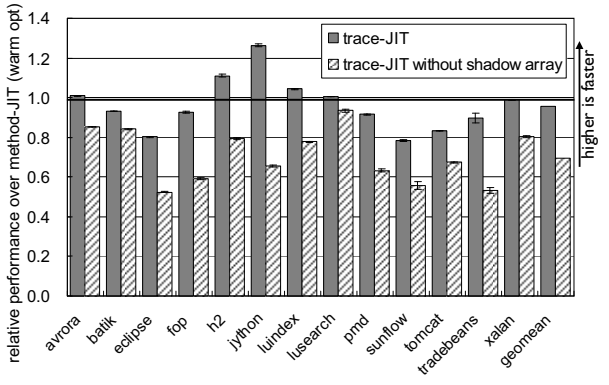

Figure 11. Relative performance of our trace-JIT over the warm-opt method-JIT with and without hash lookup reduction by using shadow arrays.

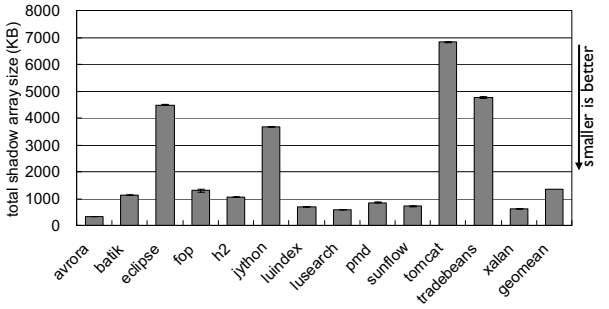

Figure 12. Total shadow array size in KB for each benchmark.

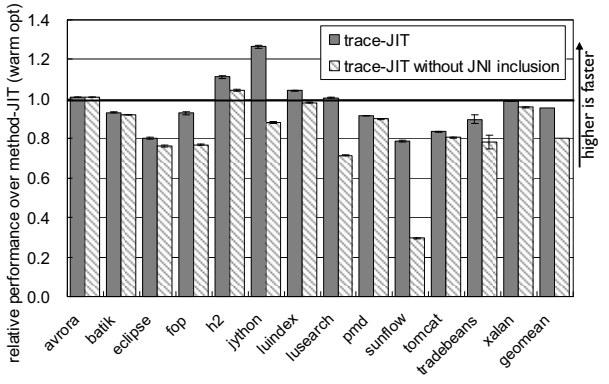

Figure 13. Relative performance of trace-JIT over warm-opt method-JIT with and without JNI inclusion.

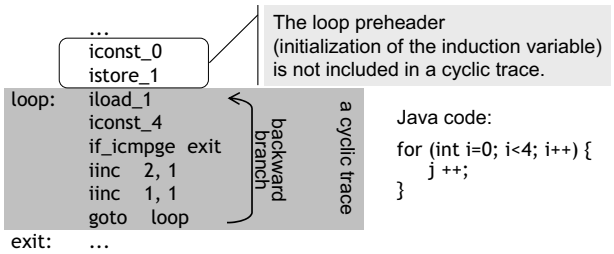

Figure 14. Example of a cyclic trace selection.

shown in Figure 6(b), cyclic traces consumed much of the total CPU time so that optimizing the cyclic traces is important to achieve higher performance. We also missed optimization opportunities for other optimizations, such as value propagation, within the hot loop because a cyclic trace always starts from the target address of the taken backward branch and it cannot include the code sequence before the loop.

To overcome this limitation, we are planning to enhance the trace selection algorithm to include the loop preheader in a cyclic trace. Compared to a binary translation and optimization system, a JVM can tell the structure of the running program and hence it can select better traces to maximize the benefits of the later optimizations in the JIT compiler. Unfortunately, including the loop preheader in a trace may offset some of the advantage of the simple structure of traces. For example, the trace may include join points in the control flow. Although it has received little study to date, enhancing the trace selection algorithm to maximize optimization opportunities in the compiler would be important and interesting future work.

B. Mixed Execution of Trace-JIT and Method-JIT

Our trace-JIT is currently only designed for mixed execution with the interpreter. We showed that the trace-JIT can potentially generate better compiled code by extending the compilation scope if we could select long traces. The trace-JIT, however, may suffer from larger runtime overhead if it selects lots of short traces, especially in the hot spots of the running program. In such programs, combining the trace-JIT with the method-JIT can be a good way to improve the overall performance by taking advantage of both approaches. To identify the best methods for cooperation between the trace-JIT and the method-JIT is another important area for future work.

VII. SUMMARY

In this paper, we reported on the design and implementation of our trace-JIT we developed from a production-quality method-JIT in Java. We showed that our trace-JIT achieved on average about 95.5% of the method-JIT with a set of comparable optimizations. The trace-JIT achieved better compiled code performance than the method-JIT by extending the compilation scope. Hence selecting a longer trace is a key to achieve superior performance with trace-JIT. We also showed that our techniques to reduce runtime overhead significantly improved the performance of the trace-JIT.

Our results showed that 1) retrofitting an existing method-JIT for a trace-JIT is a practical approach to implement a trace-JIT. We showed that, in some benchmarks, our trace-JIT developed from a method-JIT achieved better performance than the method-JIT without adding trace-specific optimization techniques. 2) Although the trace-based compilation has benefits and drawbacks over the method-based compilation, it is practical not only for binary translation systems or dynamic scripting languages, but also for the languages with mature method-based compilers. We believe that neither a method-based nor a trace-based approach is a one-size-fits-all solution, and the best solution might be a mixture of trace-based and method-based compilation to take advantage of both approaches.

REFERENCES

[1] Bala, V., Duesterwald, E., and Banerjia, S., "Dynamo: A Transparent Runtime Optimization System," In *Proceedings of the ACM Programming Language Design and Implementation*, pp. 1–12, 2000.

[2] Bruening, D., Garnett, T., and Amarasinghe, S., "An infrastructure for adaptive dynamic optimization," In *Proceedings of the ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 465–478, 2003.

[3] Gal, A., *et al.*, "Trace-based Just-In-Time Type Specialization for Dynamic Languages," In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pp. 465–478, 2009.

[4] LuaJIT design note. http://lua-users.org/lists/lua-l/2009-11/msg00089.html

[5] Bolz, C., Cuni, A., Fijalkowski, M., and Rigo, A., "Tracing the Meta-Level: PyPy's Tracing JIT Compiler," In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pp. 18–25, 2009.

[6] Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., and Venter, H., "SPUR: A trace-based JIT compiler for CIL," In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 708–725, 2010.

[7] Grcevski, N., Kielstra, A., Stoodley, K., Stoodley, M., and Sundaresan, V., "Java just-in-time compiler and virtual machine improvements for server and middleware applications," In *Proceedings of the USENIX Virtual Machine Research and Technology Symposium*, pp. 151–162, 2004.

[8] Guo, S. and Palserg, J., "The Essence of Compiling with Traces," in Proceedings of *38th Symposium on Principles of Programming Languages*, 2011. (to be published)

[9] The DaCapo benchmark suite. http://dacapobench.org/

[10] Gal, A., Probst, C., and Franz, M., "HotPathVM: An Effective JIT Compiler for Resource-constrained Devices," In *Proceedings of the International Conference on Virtual Execution Environments*, pp. 144–153, 2006.

[11] Zaleski, M., Demke-Brown, A., and Stoodley, K., "YETI: a gradually Extensible Trace Interpreter," In *Proceedings of the 3rd ACM/USENIX international conference on Virtual Execution Environments*, pp. 83–93, 2007.

[12] Bebenita, M., Chang, M., Wagner, G., Gal, A., Wimmer, C., and Franz, M., "Trace-based compilation in execution environments without interpreters," In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pp. 59–68, 2010.

[13] Yermolovich. A, Wimmer, C., and Franz, M, "Optimization of dynamic languages using hierarchical layering of virtual machines," In *Proceedings of the 5th Symposium on Dynamic Languages*, pp. 79–88, 2009.

[14] Mandelin, D., "Tamarin Tracing Internals, Part I to V," 2008. http://blog.mozilla.com/dmandelin/2008/05/.

[15] Zhao, C., Wu, Y., Steffan, J., and Amza, C., "Lengthening Traces to Improve Opportunities for Dynamic Optimization," In *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*, 2008.

[16] Hayashizaki, H., Wu, P., Inoue, H., Serrano M., and Nakatani, T., "Improving the Performance of Trace-based Systems by False Loop Filtering," In *Proceedings of Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011. (to be published)

[17] Suganuma, T., Yasue, T., and Nakatani, T., "A region-based compilation technique for dynamic compilers," *ACM Trans. Program. Lang. Syst.*, Vol. 28 (1), pp. 134–174, 2006.