Trace Fragment Selection within Method-based JVMs

Duane Merrill   Kim Hazelwood

University of Virginia

Abstract

Java virtual machines have historically employed either a "whole-method" or a "trace" methodology for selecting regions of code for optimization. Adaptive whole-method optimization primarily leverages intra-procedural optimizations derived from classic static compilation techniques whereas trace optimization utilizes an interpreter to select, manage, and dispatch inter-procedural fragments of frequently executed code.

In this paper we present our hybrid approach for supplementing the comprehensive strengths of a whole-method JIT compiler with the inter-procedural refinement of trace fragment selection and show that that the two techniques would be mutually beneficial. Using the "interpreterless" Jikes RVM as a foundation, we use our trace profiling subsystem to identify an application's working set as a collection of hot traces and show that there is a significant margin for improvement in instruction ordering that can be addressed by trace execution. Our benchmark hot-trace profiles indicate that 20% of transitions between machine-code basic blocks as laid out by the JIT compiler are non-contiguous, many of which are transfers of control flow to locations outside of the current virtual memory page. Additionally, the analyses performed by the adaptive whole-method JIT compiler allow for better identification of trace starting and stopping locations, an improvement over the popular next-executing-tail (NET) trace selection scheme. We show minimal overhead for trace selection indicating that inter-procedural trace execution provides an opportunity to improve both instruction locality as well as compiler-directed branch prediction without significant run-time cost.

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

General Terms   Design, Experimentation, Performance

Keywords   JVM, JIT compilation, Region selection, Traces

1. Introduction

Today, more and more software is written in languages whose code is designed to run in virtual execution environments (VEEs). Software written in such high-level languages (e.g., Java and C#) are compiled into bytecode representations that target virtual machine abstractions. A virtual machine implementation then dynamically interprets or compiles this bytecode for a target processor using just-in-time (JIT) compilation. These virtual machines belong to a class of VEEs known as software dynamic translators (SDTs). In addition to the usual programming benefits of this genre of managed code (e.g., portability, strong type-safety, memory protection, garbage collection, etc.), virtual machines have the ability to exploit

runtime information to perform dynamic optimizations that are not feasible in the static compilation model.

The nature of the translation process in a virtual machine is one of dynamic and adaptive optimization: patterns of execution are profiled and then used to drive optimizations of frequently executed code in order to improve program performance. Because profiling and optimization is performed online, care must be taken in order to ensure that the resulting performance benefits outweigh the associated overhead and cost. Additionally, such profiling and optimization must be done swiftly; overly lengthy profiling and optimization will be marginalized by missed opportunity.

We use the term *region* to denote the portion of frequently executed code that a dynamic optimizer uses for optimization [14]. Proper region management is critical for improving performance and reducing overhead. Java virtual machines have historically used the *whole method* approach to region selection. JVMs such as the Sun Hotspot compiler(s) [26] and the IBM JVM [24] operate in *mixed mode*, meaning that a method's bytecode is initially executed using an interpreter until the JVM determines its suitability for further optimization, at which point the method is compiled into machine code by a JIT compiler. This approach affords the JVM a quick response time, yet the ability to perform advanced JIT compiler optimizations (e.g., constant and copy propagation, common sub-expression elimination, array bounds check elimination, etc.) in the static, traditional manner based on procedures and other programmer-defined units of code [2].

An alternative to whole-method region selection is *trace* region selection. A trace history (or simply just trace) is a specific instruction sequence that is observed at run time. Subject to selection policy, traces can cross procedure boundaries, branches, and can span an arbitrary number of instructions. Like superblocks, traces generally contain a single entry point and multiple exit points. A control module (usually an interpreter) identifies frequently executed hot traces and translates them into optimized machine-code *trace fragments* which can be cached for subsequent dispatch.

Traces are used as the regions of analysis in many lightweight JVMs [6, 12], e.g., those designed for embedded use where simple interpretation is too slow, yet the inclusion of a full-featured JIT compiler is not viable. Traces are also used as the regions of analysis in many other SDT genres, particularly those that do not translate structured source/byte code: transparent optimizers (Dynamo [4], DynamoRIO [9]), binary translators (Strata [21], Transmeta's CMS [10]), and dynamic instrumentation tools (Pin [18]). The successes enjoyed by these projects are indicative of the power of traces to efficiently provide excellent instruction locality as well as inter-procedural optimizations of the (relatively few [5]) hot paths contained within most programs.

Comprehensive whole-method compilation and inter-procedural trace fragment optimization need not be mutually exclusive. This work argues the case for trace fragment execution as a feedback-directed optimization to be performed on whole-method code produced by a JIT compiler. By supplementing the Jikes Research Virtual Machine (Jikes RVM) [1] with a trace fragment selection subsystem, we demonstrate that the hot paths traced from machine code generated by the JVM's whole-method JIT compiler exhibit

significant opportunities for further refinement. We used our trace fragment selection subsystem to identify hot path profiles of the DaCapo [7] and SpecJVM [23] benchmarks as collections of trace fragments. By recording the method, compilation level, and location of each basic block sampled, we were able to use the resulting data to paint a picture of how the average benchmark application's working set matches the code layout produced by the Jikes RVM optimizing compiler. We found that 20% of transitions between basic blocks within our profiled traces exhibit a discontinuity. One fourth of these discontinuities transfer control flow to a location within a different virtual memory page (usually as a result of a call to a subroutine). Eliminating these discontinuities with inter-procedural trace execution will improve instruction locality. Additionally, trace-execution helps compiler-directed branch prediction by eliminating, on average, 15% of mispredicted fall-though blocks on hot execution paths.

The design for a trace fragment selection subsystem that further refines code generated by a whole-method JIT compiler presents unique challenges and opportunities. Current trace fragment selection approaches have employed an interpreter as the locus of control: the interpreter profiles and assembles traces fragments, dispatches execution to them, and reacquires control upon fragment exit. In an "interpreterless", compile-only JIT environment such as Jikes RVM, the locus of control is the compiled binary method. As such, the JIT compiler must be augmented to generate application machine code that also directly performs much of the work required for trace selection. On the other hand, our scheme benefits from the work already performed by the adaptive whole-method JIT compiler: the "free" control-flow graph analysis allows for the simple identification of trace starting and stopping locations, resulting in an improvement for selection of inter-procedural and nested-loop paths over the popular next-executing-tail (NET) trace selection scheme.

We isolate the performance overhead of our trace fragment selection subsystem and show that it is minimal. To do so, we devised a scheme in which the optimizing compiler assembles two regions of identical functionality for each method: a low-fidelity region to profile hot loop headers and a high-fidelity region to sample each code block touched by a trace. The low-fidelity region transfers execution to the high-fidelity region for trace formation when a hot loop header threshold has been reached. Execution is returned to the low-fidelity region upon trace conclusion. The result is that the majority of execution within our evaluation system will occur within uninstrumented code once the application has reached steady-state, allowing us to evaluate the overhead of the trace profiling subsystem in isolation.

The remainder of this paper is organized as follows. In Section 2 we review other approaches for improving instruction locality within Jikes RVM as well as provide more background on trace selection. Section 3 describes the general overview of our trace fragment selection scheme. In Section 4, we present our testing methodologies as well the empirical results gathered to support our case for the inclusion of trace execution. Section 5 identifies areas for follow-up work based upon the ideas and results discussed within this paper, and Section 6 summarizes.

2. Background and Related Work

2.1 Improving Code Locality

Code reordering is a well-known approach that has been investigated for improving instruction locality of reference. The Pettis-Hansen "Algorithm 2" (HP2) [20] for basic block positioning is based upon profiled/predicted execution counts of basic-block edges, and has become a de-facto scheme for code layout. In fact, the Jikes RVM implements HP2 based upon statically determined edge estimates. More specific to virtual machines, schemes for reordering the placement of code within the heap with consideration for execution frequency, temporal locality, and alignment have been shown to yield minor performance improvements [17].

Procedure inlining is an important optimization for most object-oriented programming languages because those programs are decomposed into large numbers of relatively small methods. Inlining reduces the overhead of invoking and returning from methods and allows standard intra-procedural analysis to be performed on what would have originally been inter-procedural code. By copying a helper procedure into a parent procedure, inlining improves code locality at the cost of code expansion. Dynamic inlining uses profile information regarding frequently-executed call edges to drive inlining decisions [2, 19, 25].

More advanced inlining decisions can be made based upon context-sensitive trace profiling [8, 13]: e.g., decisions to inline nested procedures can be made based upon *how* a program arrived at a particular point in the execution. Traces constructed for context-sensitive inlining are generally represented as histories of method or block identifiers, as opposed to binary code fragments that are executed natively. Context-sensitive inlining is similar to trace execution in that both schemes attempt to compensate for the drawbacks of method-based compilation by facilitating inter-procedural optimization.

The use of traces for execution within Java virtual machines has been explored in other related work. Berndl et al. [6] explored the potential of selecting and optimizing bytecode-level traces obtained from their Sable JVM interpreter. Their approach for trace selection involves using branch correlation graphs to locate highly correlated sequences of instructions that should be placed within the same trace. The lightweight HotpathVM [12] project similarly uses an interpreter to drive trace selection at the byte-code level; their bytecode traces are identified using a NET-like scheme and then compiled using a novel single static assignment (SSA) construction. The work done by Shankar et al. [22] implements *specialized* intra-procedural traces at the bytecode-level within the Jikes RVM. Specialized traces are instruction sequences that have frequently occurring values "hard-coded" within them.

Contrary to previous approaches, we neither use an interpreter to manage trace selection nor do we perform trace fragment selection at the bytecode level. We choose to address trace fragment management within the Jikes RVM at the *machine code level* for several reasons:

- Our work explores trace fragment selection as a feedback-directed optimization to be performed on whole-method code produced by a JIT compiler. The final output from the JIT compiler is an array of machine code (and several annotation data structures).

- The Jikes RVM JIT compiler can potentially translate contiguous bytecode instructions into fragmented machine code. For example, the `idiv` integer division bytecode causes Jikes RVM to emit several non-contiguous machine code blocks in order to handle the divide-by-zero trap. Additionally, the Jikes RVM optimizing compiler inserts non-contiguous machine-code blocks to perform JVM administrative tasks, such as the insertion of "yieldpoints" to the VM thread scheduler within method prologues and loop backedges.

- Constructing runnable trace fragments from bytecode traces would incur the overhead of partial compilation to translate these traces into machine code. This overhead is not necessary if traces are profiled at the native level: trace fragments can be assembled by scavenging already-compiled native code blocks. Additionally, such a partial compilation facility does not presently exist for the Jikes RVM.

2.2 Selecting Traces

As mentioned in Section 1, most SDTs use trace fragments as the unit of optimization, translation, and execution. The success of any trace management scheme depends on three factors:

- The ability to mitigate overhead (which may occur in the form of increased runtime or excessive memory usage). The general path-profiling problem can require storage equal to the number of dynamic paths, which can be exponential in the size of the program. The online path profiling problem is also computationally expensive: extra instructions must be executed at the site of each conditional branch, regardless of the profiling scheme employed.

- The ability to minimize the amount of prediction delay required before enough dynamic information is available to make an optimization decision (e.g., establishing a hot trace fragment). The work done by Duesterwald and Bala [11] indicates that prediction delay is a significant factor in evaluating the quality of a prediction scheme.

- The code quality and dynamic path coverage of the trace fragments selected. For example, the Dynamo project [4] performs a variety of trace fragment optimizations, including several types of redundancy removal, propagation, and peephole optimizations. The quality of path coverage is typically associated with a trace selection scheme's "X% cover set" [3]. (For a given program, the X% cover set is defined as the smallest set of trace fragments whose combined execution percentage exceeds X.) The ability to achieve a 90% cover set – and the storage required to do so – are important evaluation points for a trace profiling scheme.

The standard reference for trace fragment construction is the next-executing-tail (NET) policy introduced in Dynamo [4]. NET attempts to select traces that begin at loop headers by considering only instructions that are the targets of backward branches or exits from existing traces. This policy of concluding trace profiling when a backward branch is taken has several drawbacks: e.g., traces cannot span inter-procedural cycles and nested loops must be covered with more than the minimum number of traces.

To address these shortcomings, Hiniker et al. [14] propose an alternate trace selection algorithm known as last-executed-iteration (LEI). They show that LEI suffers less trace separation (a condition in which related execution paths are placed far apart in the trace fragment cache) and code duplication than NET. LEI selects traces that correspond directly to cycles though the use of a history buffer of taken branches. (Similarly to NET, LEI also only considers trace sources that are the targets of backward branches or exits from existing traces.) Our solution provides similar improvements over NET without the need for profiling all taken branches.

The work done by Hiser et al. [15] highlights the potential impacts of many other design decisions concerning trace formation. Specifically, they address conditions for terminating trace formation, eliding basic blocks by removing direct jumps, partial inlining of call instructions, fragment alignment, and trampoline/exit-stub placement.

Much attention has been given to the efficient handling of indirect branches profiled within the application code. The work done by Hiser et al. [16] shows a significant correspondence between the number of indirect branches executed and the overhead of an SDT system. They evaluate the effectiveness of different schemes for handling indirect branches, e.g., indirect branch translation caches (IBTCs), open hashing *sieves* for mapping target addresses to code cache targets, and inlining indirect branch targets. Handling of indirect branches within these SDT systems is complicated by the fact that the SDT may lose control of the application if an indirect

branch target is not properly translated. In our design for the Jikes RVM, the system does not fail if execution returns to the translated code contained within the JVM's whole-method code arrays.

3. Design and Implementation

This section provides a general overview of our proposed JIT-based trace management system for Jikes RVM as well as a description of the trace fragment selection implementation we used to evaluate the viability of trace execution.

3.1 Trace Management Overview

Trace management can be logically decomposed into three different tasks:

- **Trace fragment selection**. Trace fragment selection employs a trace profiling scheme to identify hot paths of dynamic execution. Traditionally such schemes must strike a three-way balance between producing traces that accurately predict future program behavior, producing such traces quickly to avoid the hidden cost of missed optimization opportunity, and maintaining an overhead that can be compensated by the speedup derived from trace execution. As mentioned earlier, we have decided to pursue a trace selection scheme similar to the NET approach.

- **Trace optimization and assembly**. In accordance with the "make the common case fast" mantra, inexpensive code improvements (e.g., various forms of redundancy removal, propagation, and peephole optimizations) are performed on hot trace fragments. Assembly includes "linking in" existing traces where appropriate as well as generating any necessary exit stubs needed for returning control to an interpreter (or the native application code). Again, the speedup derived from trace execution must compensate for the overhead of such processing.

- **Trace execution**. Assembled traces must be managed within a "code cache". The primary code cache management activity is to invalidate/evict trace fragments due to (a) changes in the application's current working set, (b) self-modifying code (e.g., recompilation of a method by the adaptive JIT compiler), and (c) unloading code modules, such as dynamically-linked libraries.

The Jikes RVM presents unique challenges for a trace management system (TMS). The lack of an interpreter completely alters the landscape of design alternatives regarding trace selection. Assuming a NET-like trace selection scheme, how do we identify loop header candidates? Once a hot loop header is found, how do we profile the rest of the trace body? How can we transfer execution to a trace fragment within the code cache? When execution exits the trace cache, where does it go? In typical SDT implementations, the interpreter is responsible for handling these profiling tasks, and control flow is passed between the interpreter and trace fragments via expensive "context switches".

We do not have an interpreter at our disposal, but we do have handy the JIT compiler responsible for generating the application code. We can use the JIT compiler to create "code-driven" trace profiling by embedding instrumentation instructions within the application code. The general idea is that the application code profiles itself, establishing trace paths that can be assembled later into trace fragments during thread scheduling yieldpoints.

Trace assembly involves using the various block-offset metadata associated with each compiled method to locate and copy the blocks indicated within the trace path into a contiguous trace fragment. Care is taken to translate exit branch addresses to jump back to the corresponding locations in the JIT-compiled method.

(a) Inter-procedural Loop
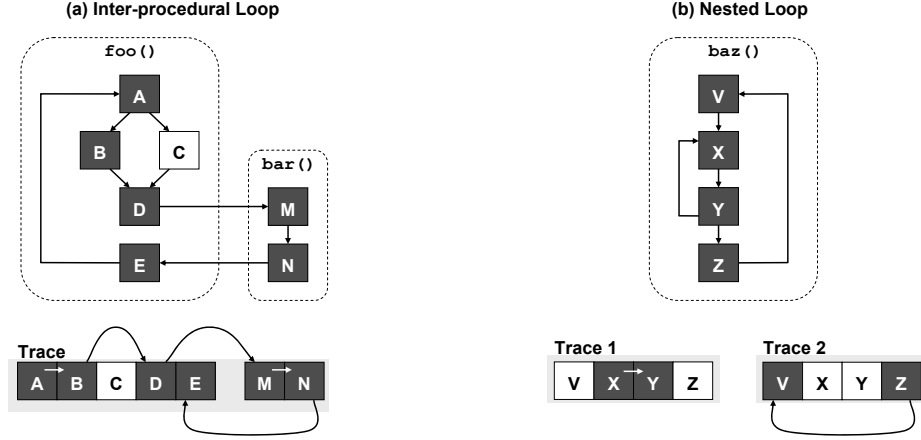
(b) Nested Loop

Figure 1. Traces selected for (a) an inter-procedural loop and (b) nested loops. The control flow graph in (a) contains a loop with a function call on its dominant path ($ABDMNE$). Below the CFG, we illustrate the execution flow through the JIT compiler's whole-method native code arrays. Of the five transitions between basic blocks recorded by the trace, three are flow discontinuities ("gaps"). The CFG in (b) contains a simple nested loop. When both paths become "hot", the method is ideally covered in two traces.

Once a trace fragment has been assembled for a given loop header, we can "patch" the original application code with an absolute jump to the location of the new fragment. This patching process is facilitated by the JIT compiler, which ensures that there is enough room to overwrite the hot counter instrumentation with a near jump to the new fragment.

As with most SDT implementations, the TMS scheme is complicated by the presence of indirect branches within the application code. In our scenario, we have the advantage of knowing the exact circumstances in which indirect control flow is generated by the JIT compiler, including:

- **Virtual method dispatch**. Transfer of execution to an overloaded object method requires jumping to the result of a table lookup. However, the layout and structure of these tables are well-known, allowing the trace assembly implementation to replace an entry pointing to a JIT-compiled method with the location of a corresponding trace fragment. Compensation instructions can also be inserted into an inter-procedural trace to verify with the dispatch table that execution should stay on-trace.

- **Returns from procedure calls**. The method prologue and epilogue (including the `call` and `iret` return instructions) can be elided from the trace path during trace assembly.

- **Switch statement language constructs**. Currently, these constructs are implemented using a creative scheme in which the `iret` return instruction is used to jump to a mangled location placed on the stack. A simple mitigating solution is to have the JIT compiler translate this functionality as a large "if..then..else if.." control-flow pattern.

3.2 Trace Fragment Selection Scheme

The trace fragment selection scheme that we implement for our trace profiling subsystem is primarily derived from the next-executing-tail (NET) approach, which has been shown to be quite efficient at producing quality hot execution trace paths [4, 11, 14]. The scheme is based on the premise that the ideal trace comprises a straight-line path though a loop iteration. As such, NET uses a "target-of-backward-branch" heuristic to identify locations that likely serve as loop headers. After a loop header becomes "hot", that location is used as the start of a new trace profile.

Rather than determine the locations of possible loop headers at runtime, we can leverage the JIT compiler's static loop analysis phase (which identifies loop backedges) to identify loop header locations at compile time. The other likely candidate for a looping trace header is the method's first instruction: traces beginning at method headers provide effective coverage of recursive execution. The compile-time identification of these potential trace header locations can be achieved at almost zero extra cost.
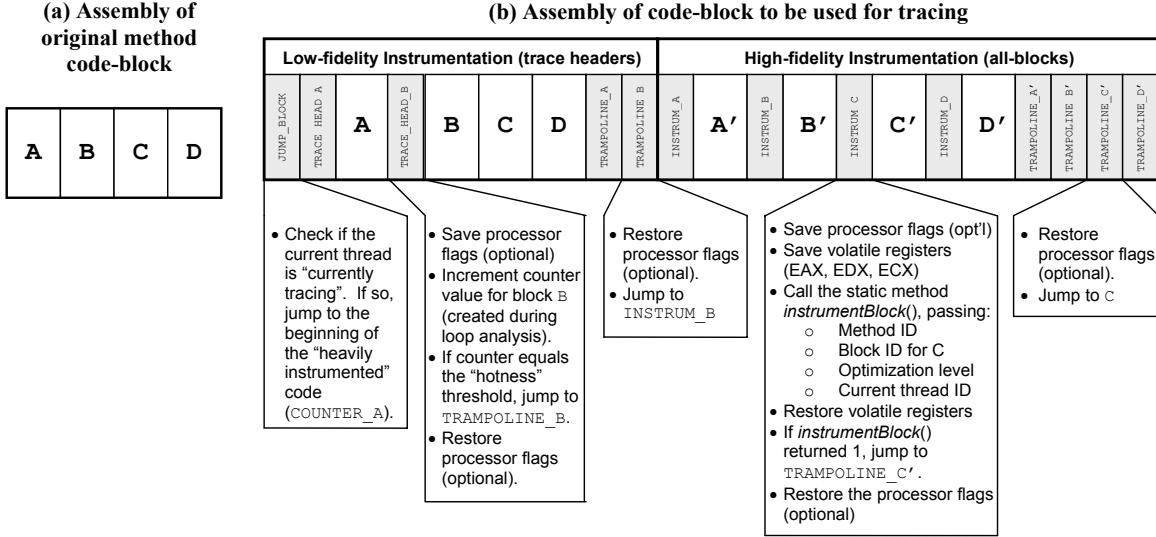
The NET scheme also considers any block that follows the exit from a trace as a candidate for starting a new trace. Unfortunately, such locations are not statically determinable at method compile-time. The targets of any given conditional branch have the potential for being incorporated as trace exit points within some trace fragment. Our design for the full TMS accommodates trace-starting criteria during the assembly of the trace fragments themselves: exit points are assembled with instructions to enable "trace-formation" when execution leaves the trace fragment.

We illustrate two simple examples of trace profiling in Figure 1. Figure 1(a) depicts a loop containing a function call on its dominant path. The JIT compiler statically identifies blocks A (because it is the target of a loop backedge) and M (because it is the method prologue of $bar()$) as potential trace headers. Block A will become hot before M, causing the subsequent iteration to profile the cyclic trace history $ABDMNE$. In Figure 1(b), the trace header candidates are blocks V and X. Block X will become hot first, resulting in the cyclic trace XY. Upon exit, block Z will be identified as a trace header, resulting in the abutting trace ZV.

Our trace selection scheme also differs from NET in its trace conclusion criteria. Our scheme concludes trace path profiling when one of the following conditions are met:

- *Cycle*: Execution has returned to the loop header basic block. The trace in Figure 1(a) is concluded by this criteria. This is the likely fate of traces profiled from inner loops.

- *Abutted*: Execution has reached a basic block that starts another trace. The trace ZV in Figure 1(b) is terminated by this criteria. The majority of all traces are concluded by this condition, as only the first hot path profiled through an inner loop can be concluded as a cycle.

- *Length Limited*: The limit of 128 basic blocks was encountered.

(a) Assembly of original method code-block

| A | B | C | D |

(b) Assembly of code-block to be used for tracing

Low-fidelity Instrumentation (trace headers) | High-fidelity Instrumentation (all-blocks)

JUMP_BLOCK | TRACE_HEAD_A | A | TRACE_HEAD_B | B | C | D | TRAMPOLINE_A | TRAMPOLINE_B | INSTRUM_A | A' | INSTRUM_B | B' | INSTRUM_C | C' | INSTRUM_D | D' | TRAMPOLINE_A' | TRAMPOLINE_B' | TRAMPOLINE_C' | TRAMPOLINE_D'

- Check if the current thread is "currently tracing". If so, jump to the beginning of the "heavily instrumented" code (COUNTER_A).

- Save processor flags (optional)
- Increment counter value for block B (created during loop analysis).
- If counter equals the "hotness" threshold, jump to TRAMPOLINE_B.
- Restore processor flags (optional).

- Restore processor flags (optional).
- Jump to INSTRUM_B.

- Save processor flags (opt'l)
- Save volatile registers (EAX, EDX, ECX)
- Call the static method instrumentBlock(), passing:
  - Method ID
  - Block ID for C
  - Optimization level
  - Current thread ID
- Restore volatile registers
- If instrumentBlock() returned 1, jump to TRAMPOLINE_C'.
- Restore the processor flags (optional)

- Restore processor flags (optional).
- Jump to C.

Figure 2. Overview of code layout for methods that have been assembled to facilitate trace selection.

- *Rejoined*: Execution has returned to a basic block already in the trace. This criteria is indicative of unusual control flow: something about the profiling iteration is different from the previous iterations, e.g., the looping of a subroutine or nested loop that does not normally recurse.

- *Exited*: Execution has exited the method in which the trace header was recorded without having met any of the above conditions. This can be detected with a comparison to the stack pointer location recorded when the trace header was profiled. This criteria is also indicative of unusual control flow: at some point, a portion of the method exhibited more visits than its parent, yet at profile time, no looping behavior was demonstrated.

One of the primary weaknesses of NET is that it will conclude trace formation when a backward branch is taken. This prevents it from using a single trace to form an inter-procedural cycle. To illustrate, NET would cover the hot path in Figure 1(a) with two traces: either $ABDMN$ and E or ABD and MNE, depending on whether or not bar() is located at a higher address than foo(). The backward-branch criteria also prevents NET from covering nested loops with a minimum number of traces. For example, the control flow graph in Figure 1(b) would be covered by NET with three traces: XY, Z, and V. By construction, our traces provide the same coverage as NET, use the same amount of counter storage for detecting when trace headers have become "hot", and do not exhibit these "trace separation" deficiencies.

3.3 Trace Profiling Implementation

Two of the primary evaluation goals for this work were to determine (a) the opportunities for code improvement that can be addressed by trace fragments, and (b) the runtime overhead that we should expect from a trace fragment selection scheme. We chose to address both agendas with the implementation of a single trace profiling subsystem for the Jikes RVM. In this subsection, we describe the details of our implementation.

Our modified implementation of the optimizing compiler must assemble code that performs trace profiling, yet exhibits virtually no overhead during steady-state execution. This is accomplished by assembling two regions of identical functionality for each method as depicted in Figure 2: a low-fidelity region to pro-
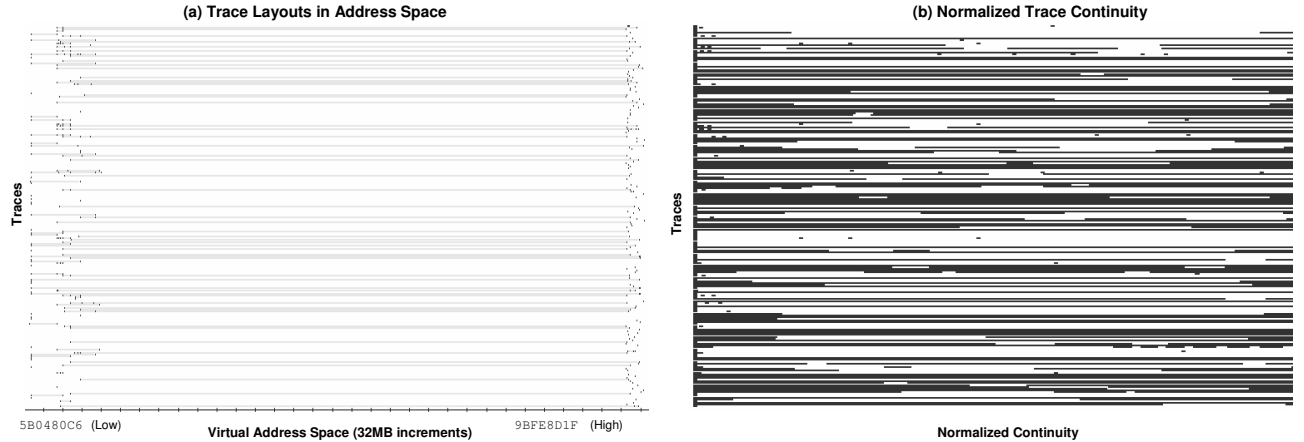
file hot loop headers and a high-fidelity region to sample each code block touched by a trace. (Although technically not necessary for trace profiling, we sample each basic block encountered during trace formation in order to analyze any dynamic discontinuities.)

A method that has been selected for (re)compilation by the optimizing compiler passes though four primary compiler transformations: bytecode $\Rightarrow$ HIR $\Rightarrow$ LIR $\Rightarrow$ MIR $\Rightarrow$ machine-code. During the HIR optimization, we augment the yieldpoint phase (which inserts yieldpoint transitions to the JVM's thread scheduler within, among other things, loop backedges) to insert dummy *INSTRUMENT_LOOP_HEADER* instructions into the control flow graph. When a *INSTRUMENT_LOOP_HEADER* is created, an integer counter is reserved for it. The location of this counter is stored with the dummy instruction.

When the compilation process reaches the final phase of MIR processing, the majority of the MIR instructions have already been converted to IA-32 machine-code instructions. At this point the final MIR expansion phase expands any dummy instructions remaining in the control flow graph. Each *INSTRUMENT_LOOP_HEADER* is expanded into roughly nine IA-32 machine instructions (e.g., the *TRACE_HEAD_B* region in Figure 2 (b)). These instructions save the processor status flags, increment the counter location specified in the dummy instruction, and test to see if it meets the "hot loop header" threshold. If so, execution is transferred to a trampoline block (e.g., *TRAMPOLINE_B*) which serves to restore the processor flags if necessary and to forward control to the corresponding block in the high-fidelity region of instrumentation (e.g., B').

The resulting control flow graph is then passed to the IA-32 assembler, which encodes all of the instructions in code-order into a contiguous byte array, forming the low-fidelity portion of the method. We also store the start/end offsets for each basic block in the *VM_OptCompiledMethod* data structure for later inspection.

The high-fidelity region of the method's code array records the identity of each basic block as execution passes through it. Because all intra-procedural control flow within a compiled method is handled by PC-relative jumps, creating the high-fidelity region is almost as simple as running the instructions through the assembler a second time. Two things must happen first: we must remove any loop-header instrumentation from the control flow graph and we must insert the per-block profiling instrumentation. The per-

(a) Trace Layouts in Address Space	(b) Normalized Trace Continuity

5B0480C6 (Low)   **Virtual Address Space (32MB increments)**   9BFE8D1F (High)
                Normalized Continuity

Figure 3. Visualization of trace histories selected for the _227_MTRT raytracing benchmark. Traces are represented horizontally: the locations of instruction blocks recorded within a trace history are indicated by horizontal black bars. Part (a) shows the blocks per trace in the process's virtual address space with horizontal gray shading to indicate groups of blocks belonging to the same trace. Part (b) is scaled such that each trace is normalized to the width of the figure; horizontal white areas indicate gaps in the traces.

block instrumentation consists of roughly 14 instructions (e.g., the *INSTRUM_B* region in Figure 2(b)). These instructions save the volatile processor context (3 of 6 general purpose registers) and call the static *instrumentBlock()* method, passing the method and block id as parameters. (The prologue and epilogue of the *instrumentBlock()* method save the non-volatile registers, making this region of instrumentation relatively expensive.) The return value of the *instrumentBlock()* method indicates whether or not a trace-termination condition was met. If so, execution is transferred to a trampoline block (e.g., *TRAMPOLINE_B'*) which restores the processor flags if necessary and to forward control to the corresponding block in the low-fidelity region of instrumentation (e.g., *B*).

We implement the *instrumentBlock()* procedure as a high-level Java method; a mostly-convenient luxury stemming from the fact that the Jikes RVM is itself implemented in Java. Unfortunately, we are not allowed to allocate new Java objects on the heap or call any methods not compiled with the `@uninterruptible` pragma because we have not set up the appropriate garbage collection maps indicating what the stack should look like at this point of execution within our method. In spite of these hindrances, the *instrumentBlock()* method manages to append the block sample to the current trace (which is maintained as a data structure of the current thread) and detect any conditions indicating that trace formation should be concluded. The *instrumentBlock()* performs one additional task when it is called to instrument the first block in a new trace: it "paves over" the corresponding loop header instrumentation in the low-fidelity with a short jump, preventing us from wasting time executing it in the future.

Inter-procedural trace profiling is facilitated with an *isTracing* bit maintained on each *VM_Thread*. The final MIR expansion emits a *JUMP_BLOCK* at the beginning of each method to inspect this bit and forward execution to the heavily-instrumented section if set.

4. Evaluation

We used the DaCapo [7] and SPECjvm98 [23] benchmark suites to evaluate our trace fragment selection scheme. These benchmark suites are collections of Java benchmark applications that can be used to gauge the performance of Java virtual machine (JVM) client platforms. We evaluated trace selection on all eleven DaCapo benchmarks and nine of the ten SPECjvm98 benchmarks, excluding _200_check because its of short execution duration (4ms). The benchmark tests are intended to be representative of a variety of application pathologies, with functionalities ranging from verification of language features to raytracing. Our test platform is an Intel Intel64 dual Xeon 3.2 GHz DualCore with 4 GB of RAM running Linux 2.6.9. We note that the performance results obtained from our benchmark evaluations do not meet the criteria for official SPECjvm98 reporting, as we used the SPECjvm98 command-line interface rather than the applet interface.

4.1 Improvement Opportunity

The performance benefits derived from trace execution have historically been closely related to instruction cache locality and branch optimization. As such, the argument for trace fragment optimization is predicated on demonstrating that there is room for improvement regarding the branch accuracy and the dynamic locality of the code produced by the Jikes RVM optimizing compiler.

The JIT compiler within Jikes RVM assembles the non-linear control-flow graph for each method into a linear array of machine code bytes. It strives to statically arrange the basic blocks in such a way that dynamic execution will *transition* directly from one basic block to its subsequent *fall-through* block. Figure 1 showed the flow of execution through assembled code arrays for the simple inter-procedural and nested-loop examples. As we saw in part (a), only two of the five block transitions recorded by the trace history flow directly into their fall-through blocks. The other three transitions are *discontinuities* that result in *gaps* in the range of instruction addresses recorded in the trace profile. We can judge the opportunity for improvement by the number of traces containing discontinuities and the proportion of gap vs. fall-through transitions.

We used our profiling scheme as outlined in the previous section to create trace histories for the "hot" code paths within the evaluation benchmarks. By storing the start and end addresses for
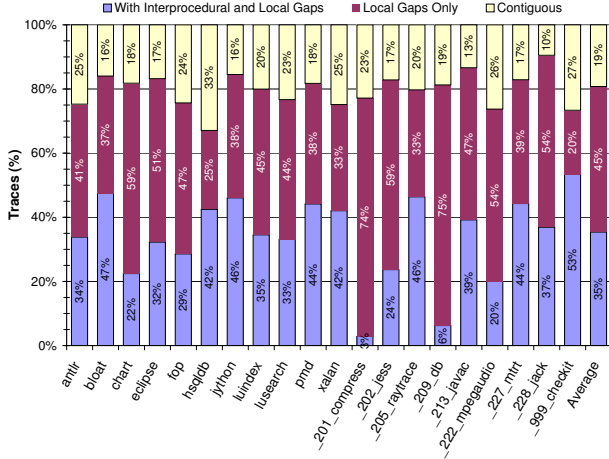
Figure 4. Trace continuity for the DaCapo and SPECjvm98 benchmarks indicating the proportions of traces that are contiguous, that contain only intra-procedural gaps, and those containing inter-procedural gaps.

each basic block sample, we can use traces to "paint a picture" of an application's working set as laid out in its address space. For example, Figure 3(a) illustrates the traces recorded for a run of the _227_mtrt raytracing benchmark test. Each trace is represented as a horizontal gray region grouping a collection of dark blocks. Each dark block represents the address range of a code block sampled in that trace. (This is similar to the way we depicted trace flow through the compiled code arrays in Figure 1, although on a much larger, aggregate scale.) For the contiguous (or very localized) traces, no light-gray shading is visible. As we can see by the amount of visible gray shading, there are a fair number of non-contiguous traces within the _227_mtrt trace set. (Also evident is the pathology of the two-generation-with-nurseries heap/collection memory management subsystem which is used to allocate the machine code byte arrays on the heap.)

One problem with this type of visualization, however, is that the instruction blocks are located sparsely within the address space. For example, the instruction blocks recorded within the _227_mtrt trace set are located within a 1GB region of the virtual address space. At this scale (where x-axis hashes represent 32-MB intervals in the address space), even trace "gaps" of two or three page frames (4KB each on our test platform) appear contiguous. Figure 3 (b) presents the same traces such that the width of each trace is normalized to the width of the figure. Now we can see that the majority of traces truly are fragmented (a contiguous trace would be displayed as a solid bar across the figure).

Plotting the traces for the other benchmarks produces similar results. We present a more analytical summary of trace continuity for the benchmarks in Figure 4. A third of the traces are necessarily fragmented (as they span multiple procedures). Of the intra-procedural traces, the majority are non-contiguous.

Although Figure 4 indicates that a majority (81%) of traces profiled are non-contiguous, it is important to consider the degree to which they are fragmented. Figure 5 presents a more detailed breakdown of the types of transitions that occur between code blocks within traces. We see that the fragmentation situation is not quite as bad as Figure 4 might indicate; the assembler has laid out the code blocks such that control flow passes to the fall-though block 80% of the time. The implication is that 20% of *all* code block transitions on "hot" execution paths are candidates
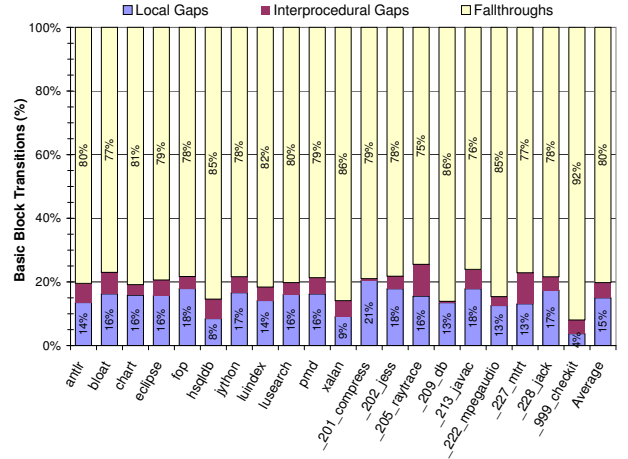
Figure 5. Categorization of the code-layout transitions between basic blocks recorded within traces.

Distance	Transition Gaps
0 - 64B (cacheline)	34.74%
65B - 4KB	40.71%
4KB+	24.55%

Figure 6. Categorization of transition gap distances as seen in traces recorded across the SPECjvm98 and Dacapo benchmarks.

for improvement by trace execution (and one fourth of these are transitions to subprocedures).

Figure 5 is also an interesting performance review for the static branch analysis performed by the optimizing compiler: it shows an average misprediction rate of 15% for choosing the appropriate fall-though block for the "hot" execution paths profiled by the traces. (The inter-procedural transitions are unavoidable discontinuities for static compilation.) This indicates that trace execution may be able to significantly lower compiler-directed branch prediction misses for "hot" execution paths.

We also consider the lengths of the block-transition discontinuities. Figure 6 indicates that 38% of transition discontinuities transfer control to a location within 64B. The impact of these discontinuities is likely negligible because control-flow is being transfered to a location within the width of a cache-line on our target platform. However, 25% of discontinuities involve transferring control flow to a location in a different virtual memory page (more than 4KB away). This proportion corresponds well with the data from Figure 5, which indicates that procedure-call discontinuities represent one fourth of all gaps.

We note that the proportions of noncontiguous traces and block transitions in Figures 4 and 5 remain fairly consistent across all of our benchmarks, indicating that a trace execution subsystem would be welcome for a wide variety of application classes.

4.2 Trace Fragment Selection

Two of the most important decisions a trace profiler must make are (a) when and where to start profiling a trace and (b) when and where to stop profiling a trace. As mentioned in Section 3, we attempt to form cyclic trace fragments that begin with "hot" loop header blocks. We used a hot loop header threshold of 20 samples for the results presented in this paper. (A detailed investigation of the impact, if any, of tuning this parameter is left for future work.)
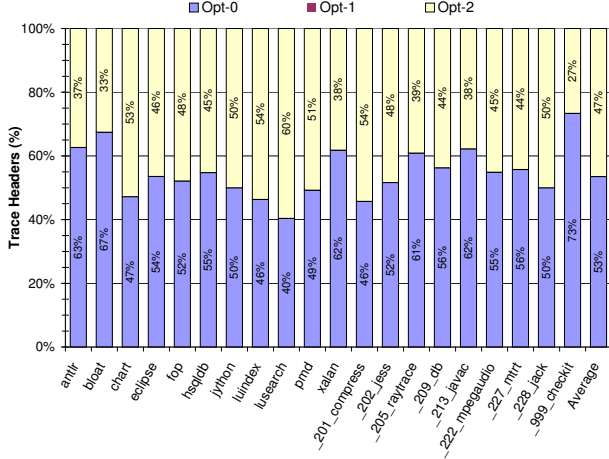
Figure 7. Proportions of traces starting in methods compiled in Opt-0, Opt-1, and Opt-2 optimization levels.

Figure 8. Trace-completion criteria used to terminate traces selected for the SPECjvm98 benchmarks.

Jikes RVM gives us the ability to specify the optimization levels at which we might want to allow compiled code to form traces (baseline, opt-0, opt-1, and opt-2). In order to survey the number and types of traces formed at each level, our optimizing compiler instruments every method it (re)compiles to perform trace fragment selection, regardless of optimization level. We chose not to evaluate trace profiling within the baseline compiler for two reasons: (a) the quality of code it produces is very poor, and (b) it is assumed that baseline-compiled methods will be promoted to opt-0 or higher very quickly if they are executed at any reasonable frequency.

Figure 7 shows the proportions of traces that were started at each optimization level. For these data, we gathered traces over five iterations of each benchmark in order to represent a composition that is valid for steady state and unlikely to change rapidly during further execution. We recorded almost equal amounts of opt-0 and opt-2 traces across the benchmarks, yet no traces starting in opt-1 methods. The lack of opt-1 methods is due to a restructuring of the Jikes RVM adaptive optimization subsystem that occurred in version 2.9.1. We noted that many methods are escalated from baseline directly to opt-2: only 28% of methods seen at opt-0 were seen again at opt-2. This indicates that many methods are promoted to opt-2 before their loop headers can become hot and start any opt-0 traces, which saves the implementation the overhead of profiling them at a lower level. It also indicates that the adaptive optimization system chose not to promote many of the opt-0 traces. We surmise that the threshold for trace formation should be opt-2: setting an optimization threshold below opt-2 would result in a significant loss of trace coverage, yet a threshold of opt-0 results in wasted effort due to escalation and stagnation.

As mentioned in Section 3, our implementation also differs from the traditional NET [4] scheme with respect to the criteria used to conclude trace formation. Like NET, we terminate traces when execution abuts the start of another trace header or when a size limit is reached (128 basic blocks, in our implementation). However, we replace the terminate-on-backward-branch criteria with three others: we terminate traces when a cycle is detected (i.e., cycling back to the loop header or rejoining a code block interior to the trace), or when execution exits the method in which the loop header was sampled (i.e., our stack pointer is higher than it was when we recorded the loop header block).

Figure 8 shows the proportions of traces concluded by each criteria. For most of the benchmarks, the *cycle* and *abutted* criteria
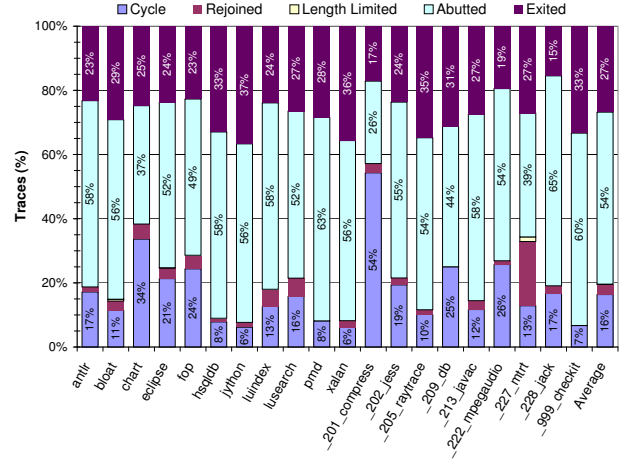
together conclude the majority of traces profiled. This is a desirable quality for trace coverage: a core of repeatable computation regions upon which ancillary hot paths can be layered.

We had anticipated that the other three types of traces (*length-limited*, *rejoined*, and *exited*) would be very infrequent because they indicate non-repetitive execution where it had been detected earlier. As we anticipated, less than 3% of the traces profiled exceeded the length limit or rejoined at a location recorded mid-trace.

Somewhat surprising is the number of *exited* traces profiled. Execution within these traces has left the scope of the trace header's method without revisiting any of the instruction blocks sampled along the way. Further inspection of the methods that spawned *exited* traces reveals that they contain looping control flow structures whose loop termination conditions are usually (but not always) met upon first entry. This allows such loop headers to become hot faster than their predecessors (e.g., the method prologue), yet the chances of the "next iteration" achieving a cycle are slim.

4.3 Performance

The value of a path profiling scheme directly corresponds to the amount of dynamic overhead required for its operation. In addition to minimizing slowdowns in application throughput, a low overhead profiling scheme is important for minimizing prediction delay (opportunity cost). As mentioned in Section 3, our trace profiling subsystem implementation is designed expressly to reflect the overhead it would impart on the system assuming no speedup/slowdown from a corresponding trace optimization/execution subsystem. The goal of our implementation is to profile hot traces as quickly and efficiently as possible and then to return execution to regions of code that closely resemble what would have been produced by the unmodified optimizing compiler.

When executing benchmark tests on a virtual machine, the first test iteration is usually the slowest due to additional overhead that does not typically occur for subsequent executions, including:

- Loading the benchmark classes
- Verification of classes (including security checks)
- Compilation of bytecode to native code (JIT)
- Initialization of static class variables
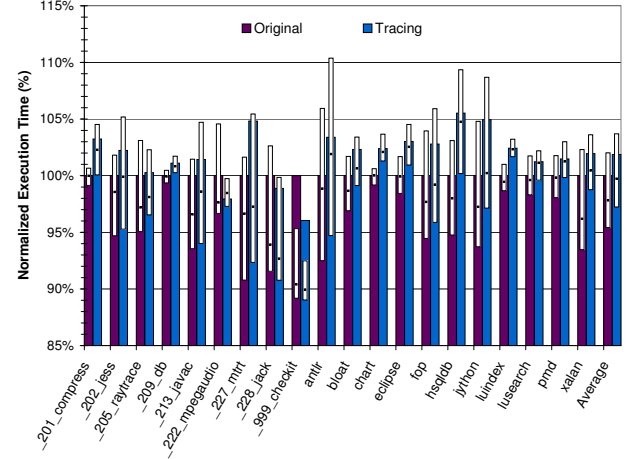- Loading the benchmark's input data
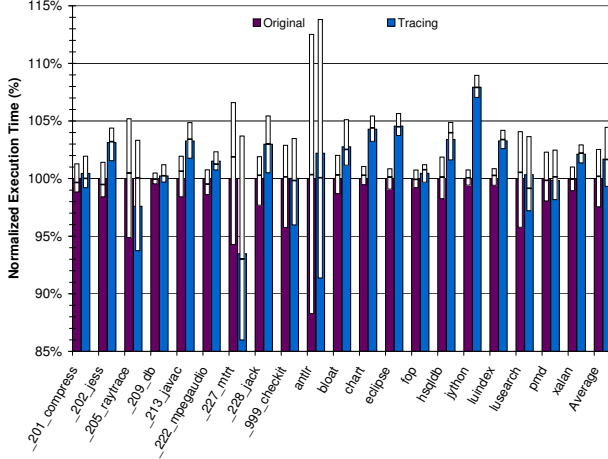
Figure 9. First-time execution times for the SPECjvm98 benchmarks using the *production* Jikes RVM build configuration (normalized to the original unmodified implementation). The colored vertical bars indicate average execution time, the white boxes indicate the interquartile ranges, and the black hashes indicate the median execution times.

In addition to these "one-time" tasks, our Jikes RVM implementation also experiences the initial overhead of profiling hot loop headers and their corresponding traces. In order to evaluate the extra overhead of trace profiling during this warm-up phase, we report the execution times of the first test-iterations for each SPECjvm98 benchmark in Figure 9. For each benchmark, we performed 40 one-iteration tests (starting the JVM from scratch each time). Because of the relatively short test durations, the variability of benchmark finish times was quite high (and is indicated with the interquartile ranges presented as white bars). We see mixed results: average overhead ranges from 7.4% (jython) to -6.5% (_227_mtrt). The overhead across the benchmark suite is 1.7%. We conclude that such a small slowdown is minimal considering with the relatively high variability of timing results and the overhead incurred by the other startup activities.

In order to ensure that we were not over-reporting the overhead of our trace profiling subsystem, we show the execution times for the SPECJvm98 benchmarks under steady-state operation in Figure 10. The results from Figure 10 were taken from 280 iterations per benchmark. (We performed 8 invocations of Jikes RVM each benchmark with each invocation specified to run for 40 iterations, and dropped the first five iterations of each invocation to discount any warm-up overhead.) As these results show, the implementation of our code-patching scheme works as intended by allowing execution to revert back to (mostly) original code once the application has reached steady-state. As with the startup measurements, the variability of the steady-state benchmark finish times was substantial (and is again indicated with the interquartile ranges presented as white bars). Average steady-state overhead is a minimal 1.8%, with benchmarks ranging from 6.3% (jython) to -4.0% (_999_checkit).

5. Future Work

Many of the design decisions made for the trace fragment selection subsystem described in this paper would need to be revisited in order to integrate it with a fragment execution subsystem. Such integration would obviate the need for a low-fidelity portion of the compiled method. Other aspects of our scheme are deserving of a much more detailed investigation, particularly:

Figure 10. Steady-state execution times for the SPECjvm98 benchmarks using the *production* Jikes RVM build configuration (normalized to the original unmodified implementation). The colored vertical bars indicate average execution time, the white boxes indicate the interquartile ranges, and the black hashes indicate the median execution times.

- The co-location of the JIT compiler allows the trace fragment generator to exploit knowledge of runtime data structures (e.g., virtual method dispatch tables).

- Optimizations regarding JIT-compiled methods. The JIT compiler can leverage the branch histories recorded during trace profiling to guide layout during opt-1 or opt-2 recompilation. If a method's dynamic paths can be shown to be covered by traces reachable from the method prologue, the JIT-compiled method can be discarded and space reclaimed. Space can also be reclaimed by resizing the compiled code arrays after trace header "hotness" instrumentation is no longer needed for a particular block.

- Evaluation of the 90% cover set. We did not evaluate this metric for our scheme as the results may change with the addition of a trace fragment assembler that instigates trace formation upon trace exit.

6. Conclusion

In this paper, we presented an overview of our trace fragment selection system for the Jikes RVM. Our system is novel in that it envisions trace fragment execution as a feedback-directed optimization to be performed on whole-method code produced by a JIT compiler. In particular, the design space for pairing trace management with a JIT compiler is different than that of other SDT genres that rely on interpretation to establish runnable trace fragments within a code cache.

The two major issues impacting the suitability of this new type of optimization are: (a) the opportunities remaining for code improvement and (b) the potential runtime overhead caused by self-profiling JIT code. We evaluated these considerations by implementing a trace fragment selection scheme within the Jikes RVM that selects hot paths using a approach similar to the NET technique. We used the traces reported by our implementation to profile the average DaCapo/SpecJVM benchmark application's working set, allowing us to evaluate the continuity of hot dynamic program paths. We discovered that 20% of observed transitions between basic blocks were non-contiguous and that a majority of these were

control-flow transfers outside of the current virtual memory page. Additionally, our implementation did not impart significant runtime overhead. We conclude that inter-procedural trace fragment selection, optimization, and execution would provide an opportunity to further improve both instruction locality as well as compiler-directed branch prediction.

The trace fragment selection scheme employed by our trace profiling implementation leverages both the JIT compiler and the JVM runtime to assist in determining trace header candidates as well as trace conclusion conditions. Our scheme improves upon the NET scheme by statically selecting trace headers during loop analysis. Replacing the target-of-backward-branch heuristic with statically-determined loop headers allows for traces to span inter-procedural loops as well as for optimal trace coverage of nested loops. The JVM runtime affords us the additional ability to inspect the application's stack in order to conclude trace formation if execution has exited the method scope in which the trace was started.

Acknowledgments

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, 2000.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization. Technical Report HPL-1999-77, Hewlett Packard, June 1999.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Programming Language Design and Implementation*, pages 1–12, Vancouver, BC, 2000.

[5] T. Ball and J. Larus. Programs follow paths. Technical Report MSR-TR-99-01, Microsoft Research, Jan. 1999.

[6] M. Berndl and L. Hendren. Dynamic profiling and trace cache generation. In *First Int'l Symposium on Code Generation and Optimization*, pages 276–285, San Francisco, CA, 2003.

[7] 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. In *21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, Portland, OR, Oct. 2006.

[8] B. J. Bradel and T. S. Abdelrahman. The use of traces for inlining in Java programs. In *Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 179–193, West Lafayette, IN, Sept. 2004.

[9] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *First Int'l Symposium on Code Generation and Optimization*, pages 265–275, Mar. 2003.

[10] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *First Int'l Symposium on Code Generation and Optimization*, pages 15–24, San Francisco, CA, Mar. 2003.

[11] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *12th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, Cambridge, Massachusetts, Oct. 2000.

[12] A. Gal, C. W. Probst, and M. Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *2nd Conference on Virtual Execution Environments*, pages 144–153, Ottawa, ON, 2006.

[13] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *First Int'l Symposium on Code Generation and Optimization*, pages 253–264, San Francisco, CA, Mar. 2003.

[14] D. J. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *38th Int'l Symposium on Microarchitecture*, pages 141–154, Barcelona, Spain, Nov. 2005.

[15] J. D. Hiser, D. Williams, A. Filipi, J. W. Davidson, and B. R. Childers. Evaluating fragment construction policies for SDT systems. In *Virtual Execution Environments*, pages 122–132, Ottawa, ON, 2006.

[16] J. D. Hiser, D. Williams, J. Mars, B. Childers, and J. W. Davidson. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *5th Int'l Symposium on Code Generation and Optimization*, 2007.

[17] X. Huang, S. M. Blackburn, D. Grove, and K. S. McKinley. Fast and efficient partial code reordering: taking advantage of dynamic recompilation. In *International Symposium on Memory Management*, pages 184–192, Ottawa, ON, 2006.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[19] Paleczny, C. Vick, and C. Click. The java hotspot server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[20] K. Pettis and R. C. Hansen. Profile guided code positioning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, NY, 1990.

[21] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *First Int'l Symposium on Code Generation and Optimization*, pages 36–47, Mar. 2003.

[22] A. Shankar, S. S. Sastry, R. Bodik, and J. E. Smith. Runtime specialization with optimistic heap analysis. *SIGPLAN Notices*, 40(10):327–343, 2005.

[23] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/jvm98.

[24] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatini. Overview of the ibm java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[25] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method in-lining for a java just-in-time compiler. In *2nd Java Virtual Machine Research and Technology Symposium*, pages 91–104, Berkeley, CA, 2002. USENIX Association.

[26] Sun Microsystems. The java hotspot performance engine architecture. http://java.sun.com/products/hotspot/whitepaper.html, Apr. 1999.