Adaptive Multi-Level Compilation in a Trace-based Java JIT 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 multi-level compilation techniques implemented in a trace-based Java JIT compiler (trace-JIT). Like existing multi-level compilation for method-based compilers, we start JIT compilation with a small compilation scope and a low optimization level so the program can start running quickly. Then we identify hot paths with a timer-based sampling profiler, generate long traces that capture the hot paths, and recompile them with a high optimization level to improve the peak performance. A key to high performance is selecting long traces that effectively capture the entire hot paths for upgrade recompilations. To do this, we introduce a new technique to generate a directed graph representing the control flow, a *TTgraph*, and use the TTgraph in the trace selection engine to efficiently select long traces. We show that our multi-level compilation improves the peak performance of programs by up to 58.5% and 22.2% on average compared to compiling all of the traces only at a low optimization level. Comparing the performance with our multi-level compilation to the performance when compiling all of the traces at a high optimization level, our technique can reduce the startup times of programs by up to 61.1% and 31.3% on average without significant reduction in the peak performance. Our results show that our adaptive multi-level compilation can balance the peak performance and startup time by taking advantage of different optimization levels.

Categories and Subject Descriptors D.3.4 [**Processors**]: Compilers, Optimization, Run-time environments

General Terms *Algorithms, Experimentation, Performance*

http://doi.acm.org/10.1145/2384616.2384630

Keywords *Trace selection and compilation, JIT compiler, profiling, Java*

1. Introduction

Trace-based compilation uses dynamically-identified frequently-executed code sequences (*traces*) as the basic units for compilation and execution as opposed to methods for traditional method-based compilers (method-JIT). At the heart of a trace-based JIT compiler (trace-JIT) lies trace selection, the component that forms traces out of executed instructions. Trace selection is a challenging problem because a trace-based system needs to balance two conflicting requirements, code quality and code size. In general, generating longer traces is the key to achieving better performance for a trace-JIT because it increases the opportunities for compiler optimizations and reduces the overhead in transitioning between compiled traces. However, generating long traces often causes more duplication among traces. The duplication significantly increases code size and compilation times and results in poor startup performance [1].

When contrasting a trace-JIT against a method-JIT, generating longer traces has a similar effect to applying more aggressive method inlining in a method-JIT. To achieve both fast startup performance and high peak performance in a method-JIT, adaptive multi-level compilation is widely used in JVMs [2-5]. With adaptive multi-level compilation, a JVM first compiles most of the methods with a low optimization level and less aggressive method inlining to yield fast startup performance. Then the JVM identifies hot methods by profiling and recompiles them with a higher optimization level and more aggressive method inlining to achieve high performance in the steady state.

This paper presents the design and implementation of our adaptive multi-level compilation scheme implemented in a trace-based Java JIT compiler. In a trace-JIT, the compiler is responsible for selecting the new compilation scope for upgrade recompilations. Each trace compilation

can start and end at any point without regard to the method boundaries. Due to this flexibility of trace selection, identifying hot code paths and selecting long traces that efficiently capture the hot code paths for recompilation is not a trivial task. We describe how the trace selection engine, the timer-based sampling profiler, and the code generator work together for effective recompilation.

Although we confirmed that our basic adaptive multi-level compilation scheme significantly improved the peak performance over compiling all of the traces at a low optimization level, we discovered that the basic trace selection technique sometimes generates a sub-optimal compilation scope for recompilation. For additional performance gains from recompilations, we introduced new techniques in the timer-based profiler and the trace selection engine. In the trace selection engine, we build a weighted directed graph representing the control flow among the traces to decide on a better program location to start a long trace for upgrade recompilation. We call this a *trace-transition graph* (*TTgraph*). Each node in the TTgraph represents a trace and an edge represents the transfer of control between two traces. We show how we build the TTgraph based on a timer-based low-overhead sampling mechanism, and then explain how we use the TTgraph to select good traces. We determine the location to start a new trace by traversing the edges in the TTgraph backward to effectively capture the entirety of each candidate hot path.

We implemented our multi-level compilation scheme with the TTgraph-based trace selection algorithm in a trace-based Java JIT compiler [6]. Our results showed that our multi-level compilation improved the peak performance by up to 58.5% and 22.2% on average over the base condition that does not employ upgrade recompilations from the initial optimization level. In these improvements, we showed that the advanced technique using TTgraph gave additional improvements in the peak performance by up to 6.5% and 1.8% on average. When comparing the performance with our multi-level compilation against the case compiling all of the traces with a high optimization level, the peak performance was still 3.3% slower on average but the startup time was significantly reduced by 31.3% on average and up to 61.1%.

This paper makes the following contributions. (1) We describe the design and implementation of our adaptive multi-level compilation scheme for a trace-based JIT compiler. As far as we know, this is the first full-fledged working implementation of adaptive multi-level compilation in a trace-based system. (2) We evaluate the performance of trace-JIT with and without adaptive multi-level compilation and show that the multi-level compilation is a practical way to balance the startup time and peak performance. (3) We show that our new technique based on the TTgraph effectively improves the performance over the basic recompilation scheme. In summary, selecting a better compilation scope for upgrade recompilation is an important problem for overall performance in addition to just using the different optimization level in the JIT compiler.

The rest of the paper is organized as follows. Section 2 gives an overview of our adaptive multi-revel compilation scheme. Section 3 describes our advanced trace selection technique for upgrade recompilation using TTgraph. Section 4 describes our experimental results. Section 5 covers previous techniques. Section 6 summarizes this work.

2. Basic Trace Recompilation

In this section, we give an overview of our adaptive multi-level compilation scheme with emphasis on how we select a new compilation scope for recompilation, which is a problem unique to trace-based systems. Currently, our scheme employs two optimization levels, the *cold level* used in the initial compilation and the *hot level* used in the upgrade recompilation. Here the optimization level includes both the trace selection parameters and the optimization strategy in the code generation. The cold level compilation focuses on the startup time of the JVM, whereas the hot level compilation focuses on the peak performance. Hence the hot level compilation seeks to use much larger compilation scopes than the cold level and also uses more time-consuming optimizations in the code generation.

Our system starts executing the program by using the interpreter with the runtime system monitoring the program execution to find frequently executed code paths and compiles them as traces at the cold level. Then the runtime identifies the especially hot paths by using the timer-based sampling and generates new traces for upgrade recompilation at the hot level.

The upgrade recompilation in our trace-JIT has 3 steps:

1. Find the hot traces that consume large amounts of CPU time by using timer-based sampling,
2. Record a new, longer trace that spans the identified hot traces for upgrade recompilation, and
3. Compile the selected trace using the hot level.

Step 2 is a key component in our multi-level compilation scheme to achieve high performance without a significant increase in code size or compilation time.

We use timer-based sampling to identify the hot traces. In our system, each trace has a *recompilation counter* to track the number of timer ticks in the trace. When the timer-based profiler samples a compiled trace, it increments the counter for the trace and checks the value. If the counter value reaches a *recompilation threshold* within a certain time interval, the trace is identified as a hot trace.

After a hot trace is identified, we select a compilation scope that spans the identified hot trace for upgrade

recompilation. Because the cold-level compilation focuses on quick startup of the program, the compilation scope of each trace compiled at the cold level is typically quite small. This means recompiling a hot trace using a higher optimization level without expanding the compilation scope is inadequate for sufficiently high peak performance.

To generate a larger compilation scope, we first invalidate the current compiled trace. Then the next time the execution reaches the trace head of the invalidated trace, we record the execution path starting from that location using a larger maximum trace length than the initial compilation to find a long trace. Alternatively, we could generate a lerger compilation scope by concatenating existing short traces, but our tests indicated this approach was inferior. An advantage of the trace-JIT over the method-JIT is that each trace is specialized for a specific execution path and hence the profile results are more important in the trace-JIT. Concatenating multiple traces may inefficiently mix traces from different execution paths, and this may cause the profiling results to be inaccurate or even illegal. In our system, we again profile for a new trace at the time of upgrade recompilation.

After the new trace is recorded for recompilation, it is compiled by the compilation thread using the hot optimization level. At the time of trace invalidation, we also invalidate the branch instructions that jump into the invalidated trace. These branch instructions were originally generated by trace linking optimization [7], which directly dispatches the next compiled trace at the trace exit, and they revert to their original states. After generating the new trace, those branch instructions are rewritten again to jump into the new trace at the next execution.

3. Trace Recompilation via Trace-Transition Graph

As shown in Section 4, the basic scheme for the adaptive multi-level compilation can improve the peak performance significantly compared to the case of using only the cold-level compilation. However we found that there were still more opportunities to improve the trace selection engine to generate better compilation scopes for recompilation.

To address the limitation of the basic trace recompilation, we introduce a new recompilation scheme based on the *trace-transition graph (TTgraph)*, which is a control flow graph using traces as nodes. We use the TTgraph to select the longest possible traces that capture the hot code paths without causing significant code duplication.

3.1 Motivation

Although our basic recompilation technique described in Section 2 successfully improved the peak performance, we discovered that it sometimes generated a sub-optimal compilation scope. Figure 1 shows an example of the

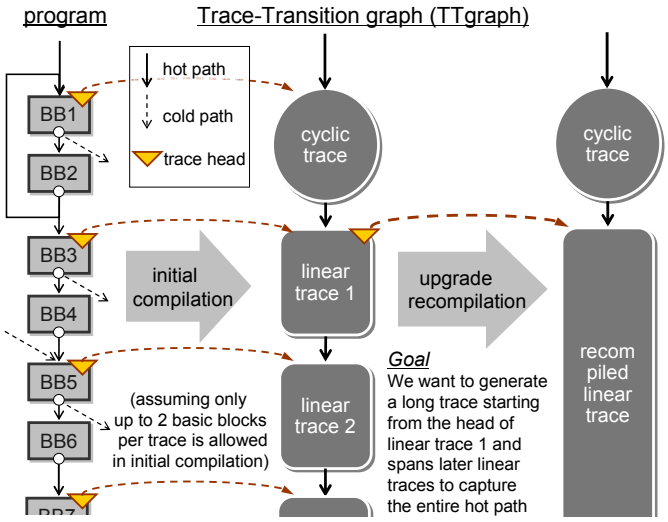

Figure 1. Example of desired trace selection for upgrade recompilation. The basic recompilation may start new trace from other location and generate a sub-optimal trace.

desired trace selection for recompilation. In this example, the program executes a loop, which consists of BB1 and BB2, and the following long linear hot path. When we start to select and compile traces with our multi-level compilation, we generate lots of short and fragmented traces to achieve faster start up by avoiding excess JIT compilation overhead. In this example, we limit the trace length up to two basic blocks and hence each trace includes only two basic blocks.

We present the status of the selected traces using a weighted directed graph called a *Trace-Transition graph* (*TTgraph*). The TTgraph resembles a control flow graph but uses a trace as a node and the transfer of the control between the two traces as an edge. The weight of the edge shows the relative frequency of that transfer of control. We generate the TTgraph at runtime based on trace linking information and timer-based sampling. Then we use the TTgraph for better trace selection.

The middle of Figure 1 shows the TTgraph of the program after the hot path was initially compiled as short traces. Because the code is a long linear hot path, we want to generate a long linear trace capturing the entire hot path as shown in the figure on the right. The basic recompilation technique, however, may pick a different location as the trace head for the new trace based on the timer-based sampling. For example, if a sufficient number of timer ticks occur in linear trace 2, the new trace starts from the head of the linear trace 2, excluding linear trace 1. Such trace selection may miss optimization opportunities. Also, if a sufficient number of timer ticks occur later in linear trace 1 after generating the new trace starting from linear trace 2,

then another trace may be generated from the head of linear trace 1. This could cause duplicated traces.

By using the TTgraph, we aim to generate better traces for upgrade recompilation as shown in the example. In a nutshell, we determine where to start a new trace by traversing the edges in the TTgraph backward to capture the entire hot path. We will explain the detailed algorithm of the backward traversal in Section 3.4.

3.2 Profiling Trace Transition

To construct the TTgraph, we need to profile trace transitioning events. We extended the timer-based sampling profiler used to identify hot traces to profile the transitions between the compiled traces.

Here is an explanation of the timer-based sampling. The runtime system configures a hardware timer for periodic interrupts. When the timer interrupt occurs during the execution of compiled code, a thread-local flag called the *yield flag* is set to show that this thread needs to stop at the next *yield point*, where the runtime system can safely stop the thread and identify the exact program location. The locations of the yield points determine the accuracy and overhead of the profiler [8]. To accurately identify the trace executed when a timer interrupt occurs, yield points are required for the trace entry point and for all of the exit points, as shown in Figure 2(a). We use this approach in our basic recompilation described in Section 2. We do not insert yield points at the trace exit points for handling exceptions to avoid code size bloat. Most of these exits will never be triggered, but they would require a large amount of yield-point code.

To generate the TTgraph, we use the timer interrupt to capture the frequent transitions between traces. We do not insert yield points at the exit points of the traces, but we use the *branch-and-link* instruction, which records the instruction pointer in the link register, to implement a jump between traces when we do trace linking optimization [7]. The branch-and-link instruction is typically used for method invocation in the method-JIT. When the execution stops at the yield point at the entry of the next trace, we can identify the previously executed trace by looking at the value in the link register. Unlike a method-JIT, the execution never returns to a previously executed trace in the trace-JIT, and hence we use the value in the link register only for generating the TTgraph. Since the execution never returns to a previously executed trace, the trace-JIT does not need to maintain any information such as an execution stack to record the execution history. This means we cannot identify previously executed traces by walking the stack. One potential drawback of this approach is that the frequent use of the branch-and-link instruction without matching method returns could confuse the return-address-stack-based branch prediction of the processor. However

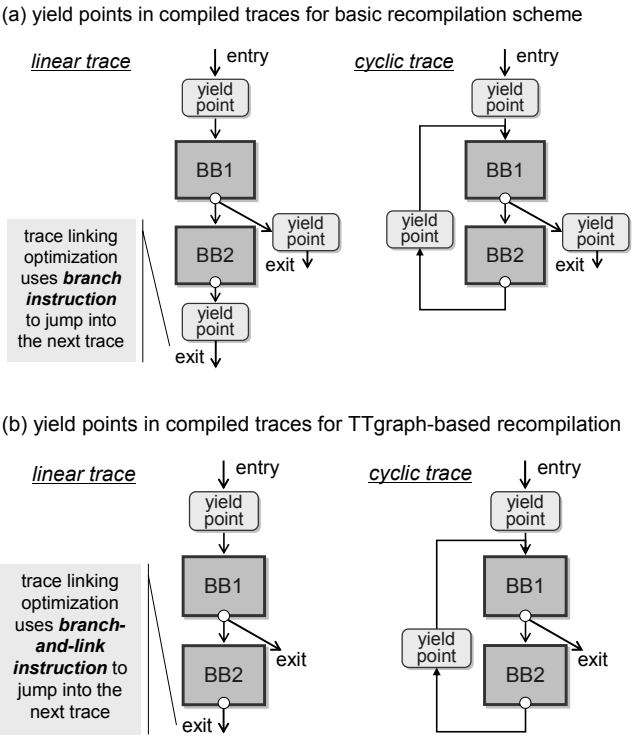

Figure 2. Locations of yield points in compiled traces.

we did not observe any significant increase in the number of branch mispredictions. Figure 2(b) depicts the locations of the yield points to generate the TTgraph.

3.3 Generating Trace-Transition Graph

To generate the TTgraph, we first generate the basic structure of the TTgraph as we link the traces. Then we adjust the weights of the edges using timer-based sampling.

Figure 3 shows an example of the steps to generate a TTgraph. Our trace-JIT, like many other trace-based systems, starts generating traces to compile by selecting a hot loop as a cyclic trace. Then the trace-JIT selects a frequently taken exit point of the first trace as a trace head for the next trace and starts a trace from that trace head. When a trace is generated just after an exit point of another trace, we can link the two traces. At the same time, we add a directed edge between each pair of linked traces with the weight of 1 in the TTgraph as shown in Figure 3(a). By repeating this process, the trace-JIT can generate the TTgraph with sufficiently high coverage of the compiled traces for the frequently executed code.

A node in a TTgraph (a trace) may have multiple incoming edges, like the linear trace 2 in the example. Because a trace is a single-entry-multiple-exit block in our system, all incoming edges must jump to the head of the trace. A node may also have multiple outgoing edges such as the linear

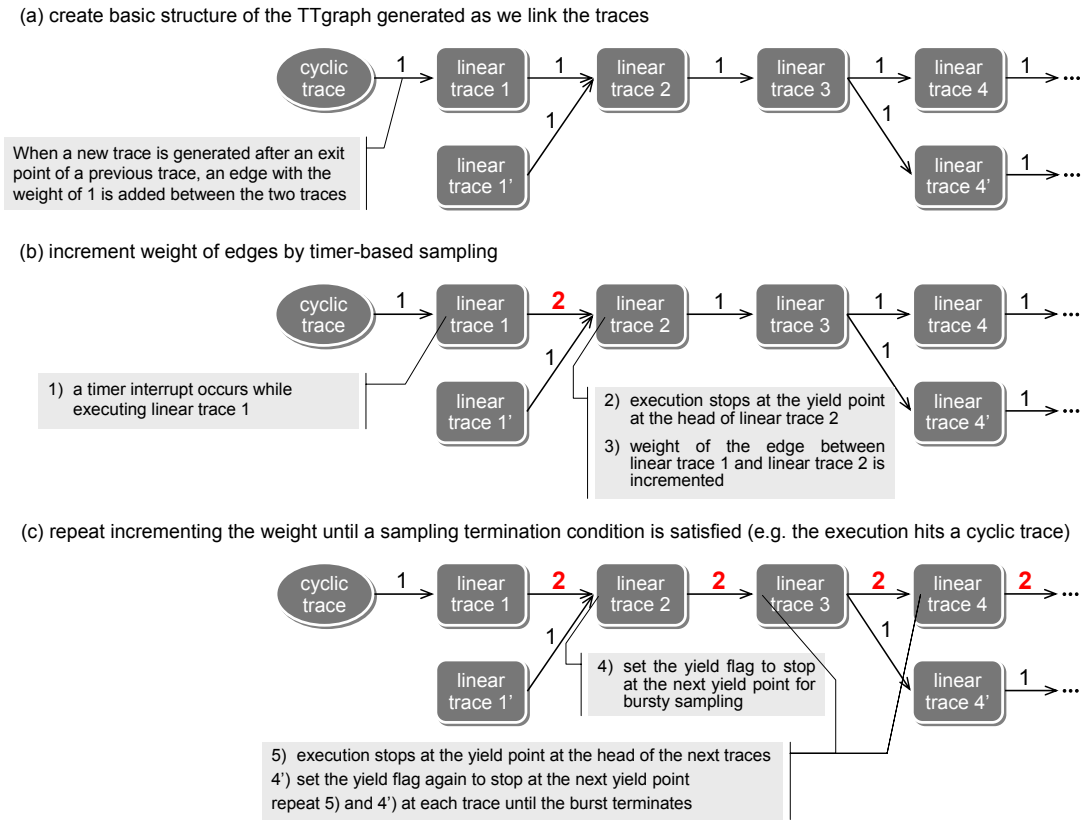

Figure 3. Steps to generate TTgraph.

trace 3 in the example. An outgoing edge may cause an exit from the middle of the trace or from the end of the trace.

After building the basic structure of the TTgraph, we adjust the weights of the edges in the TTgraph to assess the hot paths in the program. As described in Section 3.2, we use the timer-based sampling profiler to find the transitions between traces. When the profiler samples a transition between two traces, we increment the weight of the edge representing this transition. For example, in Figure 3(b), a timer interrupt occurs while the thread is executing linear trace 1 and then the execution stops at the yield point at the head of the next trace, linear trace 2. The runtime identifies the current trace from the current instruction pointer value and the previous trace by the value in the link register. Then the weight of the edge between linear trace 1 and linear trace 2 is incremented by 1.

To more quickly build a sufficiently accurate TTgraph, we sample multiple edges to increment the weights at each timer interrupt. This is similar to bursty tracing [9], which allows low-overhead temporal profiling by sampling a sequence of consecutive events (a *burst*) instead of just one event when it starts sampling. We sample a sequence of transitions between traces by repeatedly stopping the execution at the yield point at the head of each trace. To implement this behavior, we set the yield flag again when the execution stops at any yield point in the trace head, which then halts at the next trace head. In the example of Figures 3(b) and 3(c), first the yield flag is set when the timer interrupt occurs, and then the execution stops at the head of linear trace 2. After incrementing the weight of the edge between linear trace 1 and linear trace 2, the yield flag is set again to stop at the next yield point. The execution stops at the head of the linear trace 3 and the weight of the edge between linear trace 2 and linear trace 3 is incremented. By repeating this steps until one of the following burst termination conditions is satisfied, we can sample the consecutive transitions without increasing the frequency of the timer interrupts. The burst termination conditions we used are: (1) the number of transitions in this burst reaches a predefined threshold (32 in the current implementation), (2) the execution reaches a trace already encountered in this burst, (3) the execution reaches a cyclic trace or already recompiled trace, or (4) the execution uses a trace exit at which no next trace is linked. Once most of the hot paths are recompiled, most bursts quickly terminate at an already recompiled trace and so the overhead due to the sampling is not significantly larger than the basic sampling method without repeated stops in the steady state.

TTgraph consumes 96 bytes of memory per node (a trace) in the graph. To reduce the memory consumption,

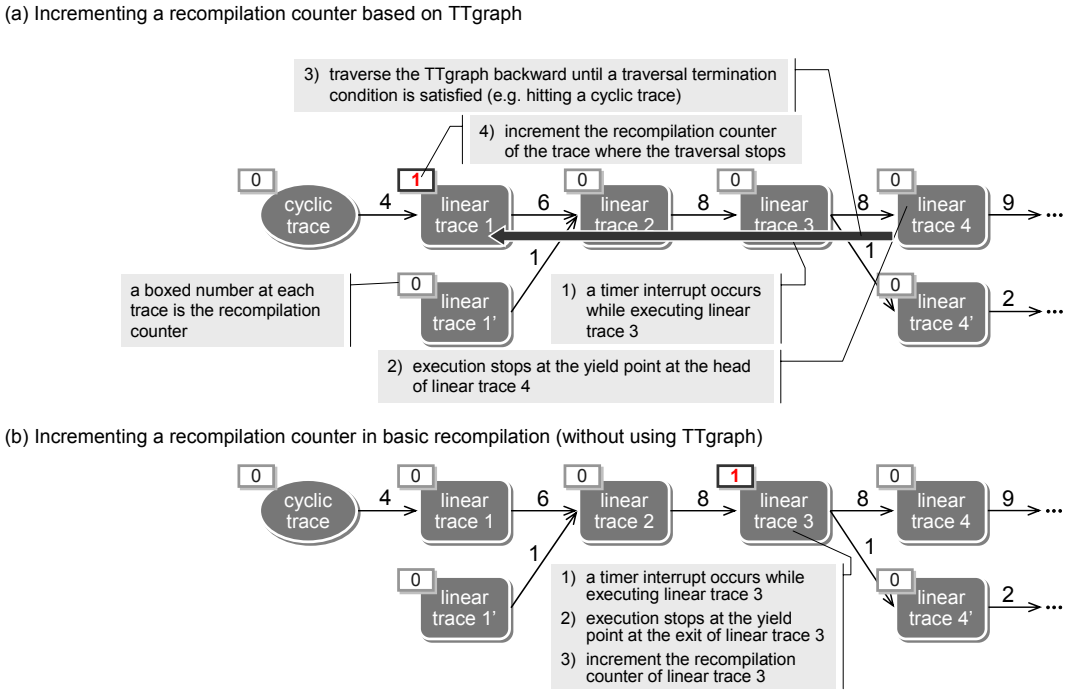

Figure 4. Recompilation using the TTgraph.

we track only a few of the hottest incoming and outgoing edges if the number of the edges becomes large. Also, we allocate these data associated with a trace only on demand during the TTgraph construction, instead of allocating it at the JIT compilation time.

3.4 Selecting New Trace Heads for Recompilation Based on Trace-Transition Graph

We use the generated TTgraph to decide on better locations to start new compilation scopes for the upgrade recompilation. In this section, we assume that the TTgraph already exists as described in Section 3.3 for ease of explanation. In the real implementation we generate and use the TTgraph at the same time.

Figure 4(a) is an example depicting how we select a trace to charge the execution time using the TTgraph in the timer-based sampling profiler. When execution stops at a yield point due to a timer interrupt, we traverse the TTgraph backward starting from the current location of the yield point, and decide on the trace whose execution time should be increased (by incrementing the recompilation counter). We stop the backward traversal if there is no incoming edge in a node or if the edge to track starts from a cyclic trace, an invalidated trace, or an already recompiled trace. For the example in Figure 4(a), we start the backward traversal from linear trace 4 and stop the traversal at linear trace 1. The recompilation counter of linear trace 1 is incremented. If the counter value reaches the recompilation threshold, then we will use the trace head address for linear trace 1 as the starting point to start recording a long trace for upgrade recompilation. We do not continue the backward traversal into the cyclic trace in this example, because our trace-JIT allows only cyclic or linear traces as compilation units, and hence a cyclic execution path and a linear execution path cannot be captured in the same trace even if we allow very long traces for upgrade recompilation.

To contrast to our technique, Figure 4(b) shows how the basic recompilation works. Note that we do not generate the TTgraph for the basic recompilation, so the TTgraph was only provided to clarify this explanation. In this example, we charge the execution time for linear trace 3 where the timer interrupt occurred. This behavior is similar to that of the timer-based sampling profiler for a method-JIT. If the compilation scope for recompilation starts from the head of linear trace 3, then the new trace does not cover linear traces 1 and 2, which is a sub-optimal selection.

In the TTgraph, a node may have multiple incoming edges or outgoing edges. Figure 5 shows how we handle a node with multiple edges during the backward traversal. If a node has multiple incoming edges, we track all of the edges with sufficiently large weights. We used 20% of the total weight of the incoming edges as the threshold. In the example of Figure 5(a), linear trace 2 has two incoming edges, from linear traces 1 and 1'. We track both edges because they have weights larger than the threshold. When tracking multiple edges from a node, the recompilation

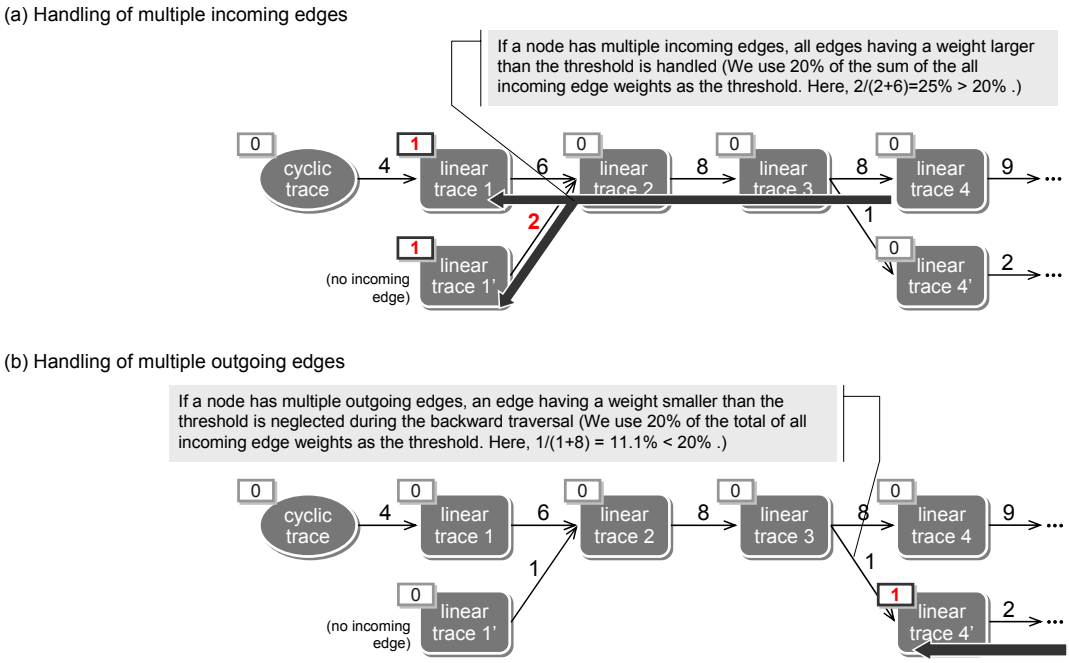

Figure 5. Handling of a node having multiple incoming or outgoing edges.

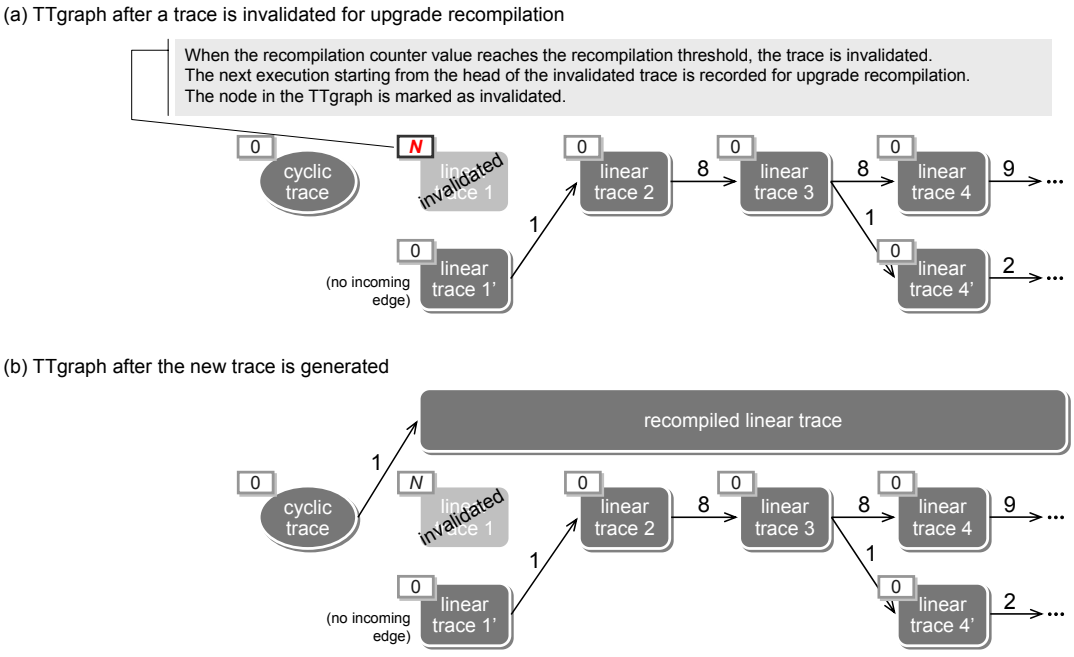

Figure 6. Trace invalidation and recompilation.

counters of multiple traces may be incremented. Here the counters for both linear traces 1 and 1' are incremented.

If a node has multiple outgoing edges and some of their weights are less significant than the others, we can ignore those edges while traversing the TTgraph. We again used 20% as the threshold value. For example in Figure 5(b), linear trace 3 has two outgoing edges, while the edge to linear trace 4' has a weight smaller than the threshold. Therefore, we do not track this edge during the backward traversal of the TTgraph. The rationale for this is that if an edge has a much smaller weight, then the edge is likely to represent a rarely-executed path. Therefore, a trace starting from linear trace 3 (or its predecessors) will not cover linear trace 4' and its successors. Hence we do not track

```
timer_interrupt_handler() {
  // this method is called when the timer interrupt occurs
  • set the yield flag of the current thread;
  • burstCounter = 0;
}

yield_point_handler() {
  // this method is called at a yield point if the yield flag is set
  • reset the yield flag of the current thread;
  if (the current yield point is at the head of a trace) {

    // Recompilation Based on TTgraph (see Section 3.4 for detail)
    if (burstCounter == 0) { // if this is the first yield point in this burst
      // charge the execution time to trace(s) based on the TTgraph
      • identify the trace(s) to charge the execution time by the backward traversal from the current trace;
      • increment the recompilation counter of the identified trace(s);
      • invalidate the trace(s) for recompilation if the recompilation counter exceeds the threshold;
    }

    // Generating TTgraph (see Section 3.3 for detail)
    if (the previous trace can be identified by the link register value) {
      • increment the weight of the TTgraph edge between the previous trace and the current trace;
      if (none of the burst termination conditions is satisfied) {
        // continue bursty tracing
        • set the yield flag of the current thread;
      }
    }

    • burstCounter++;
  }
  else if (the current yield point is in a compiled trace) { // at a loop backedge
    if (burstCounter == 0) { // if this is the first yield point in this burst
      • increment the recompilation counter of the current trace;
      • invalidate the current trace for recompilation if the recompilation counter exceeds the threshold;
    }
  }
}
```

Figure 7. Pseudocode of the TTgraph generation and the TTgraph-based recompilation.

this edge to more efficiently capture the hot path in the new trace.

When the recompilation counter for a trace reaches the recompilation threshold, we mark the node as invalidated and remove all of the related edges in the TTgraph. The next time the execution reaches the invalidated trace, the runtime will start to record that execution starting from the trace head address of the invalidated trace as a new trace for upgrade recompilation, as already described in Section 2. After the new trace is generated, we add the new trace in TTgraph as a node marked as already recompiled. Figure 6 explains how the TTgraph changes when a trace is invalidated and then the new trace is generated by upgrade recompilation.

To summarize how we generate the TTgraph based on the timer-based sampling and use it for trace recompilation, Figure 7 shows simplified pseudocode for our algorithm as executed at a yield point when the execution stops due to the timer interrupt.

4. Performance Results

4.1 Implementation

We implemented our adaptive multi-level compilation scheme with the TTgraph-based trace selection in our trace-JIT [1, 6, 10], which is based on the IBM J9/TR Java VM and JIT compiler [3]. Figure 8 shows a component view of our trace-JIT. In our system, traces are formed out of Java bytecode and compiled by our trace-JIT. As already described, our trace compiler has two optimization levels, a *cold level* and a *hot level*. The cold level is used in the initial compilation and focuses on balancing the compilation time and the code quality to achieve faster JVM startup. The cold-level compilation executes some basic optimizations including value propagation, dead store elimination, and common subexpression elimination. The hot level is used in the upgrade recompilation and puts the emphasis on the code quality over the compilation cost. Hence the hot-

Table 1. Summary of two optimization levels.

optimization level	maximum trace length	compiler optimizations	used in	Focus
cold	16 basic blocks	value propagation, dead store elimination, and common subexpression elimination etc	initial compilation	fast startup (balanced compilation cost and code quality)
hot	256 basic blocks	optimization used in *cold* + loop optimizations, escape analysis, and global register allocation	upgrade recompilation	best peak performance (code quality)

level compilation performs more costly optimizations, such as global register allocation, escape analysis and loop optimizations, to achieve higher peak performance. We also use different values for the maximum trace length in the cold-level and hot-level compilations. We use 16 basic blocks for the cold level and 256 basic blocks for the hot level. The maximum trace length is one of the most important parameters for the overall performance. We selected the maximum trace length for cold traces (16 BBs here) to minimize the startup time with no upgrade recompilation. After fixing the maximum length for cold traces, we select the maximum length for hot trace (256 BBs) to maximize the peak performance. From our experience, using trace lengths larger than 256 BBs did not produce obvious performance gains. The effects of the other parameters, including the threshold for bursty tracing and timer interrupt interval, were not important compared to the maximum trace length. Table 1 summarizes the configurations for the two optimization levels. For the recompilation threshold, we used 5 based on our experiments, so the upgrade recompilation starts when five timer ticks are hit within one trace, to balance the peak performance and the compilation time.

We ran the evaluation on an IBM BladeCenter JS22 using four 4.0-GHz POWER6 cores [11] 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 1 GB using 16-MB pages and we used the generational garbage collector. In our evaluation, we used DaCapo 9.12 [12] 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 encountered an error with this benchmark. For each result, we report the average of 16 runs along with the 95% confidence interval.

4.2 Performance With and Without Recompilation

In this section, we describe the performance of the trace-JIT with our adaptive multi-level compilation, which recompiles selected code using the TTgraph. For comparison, we used configurations with only one optimization level. One configuration compiles all traces at the cold level, which is the baseline in all of the graphs, and the another configuration compiles all traces at the hot level,

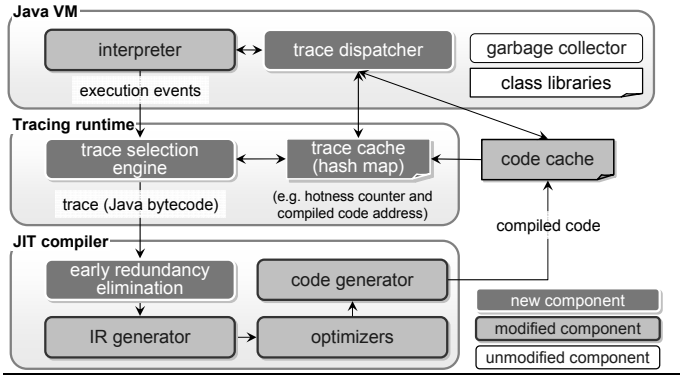

Figure 8. Overview of our trace-JIT system architecture.

which yields higher peak performance in exchange for slower startup. In addition, we show the performance of the method-JIT. For the method-JIT, we added the JVM option `-Xjit:count=1000,bcount=250,milcount=1` only for Jython to avoid pathological behavior of the method-JIT, in which the method-JIT does not compile many methods over a long time period.

Figure 9 compares the steady-state performance and the startup time (the execution time of the first iteration). With the adaptive multi-level compilation, the steady-state performance was improved by up to 58.5% and 22.2% on average (geometric mean) for all of the benchmarks compared to the case compiling all of the traces at the cold level. Compiling all of the traces at the hot level further improved the steady-state performance. However, as shown in Figure 9(b), using only the hot-level compilation greatly slowed the startup, by up to 2.57x and 1.46x on average. In contrast, the adaptive multi-level compilation offers startup times comparable to the baseline times. These results showed that the adaptive multi-level compilation benefited from both optimization levels, with faster startup times at the cold level and higher peak performance at the hot level. Comparing the performance of our trace-JIT using the adaptive multi-level compilation against the method-JIT, the trace-JIT was 5.3% slower in the steady state and 2.0% slower in the startup than the method-JIT on average. One reason for the performance differences was that our current system uses only linear traces and cyclic traces to avoid join points in the control flow in middle of a

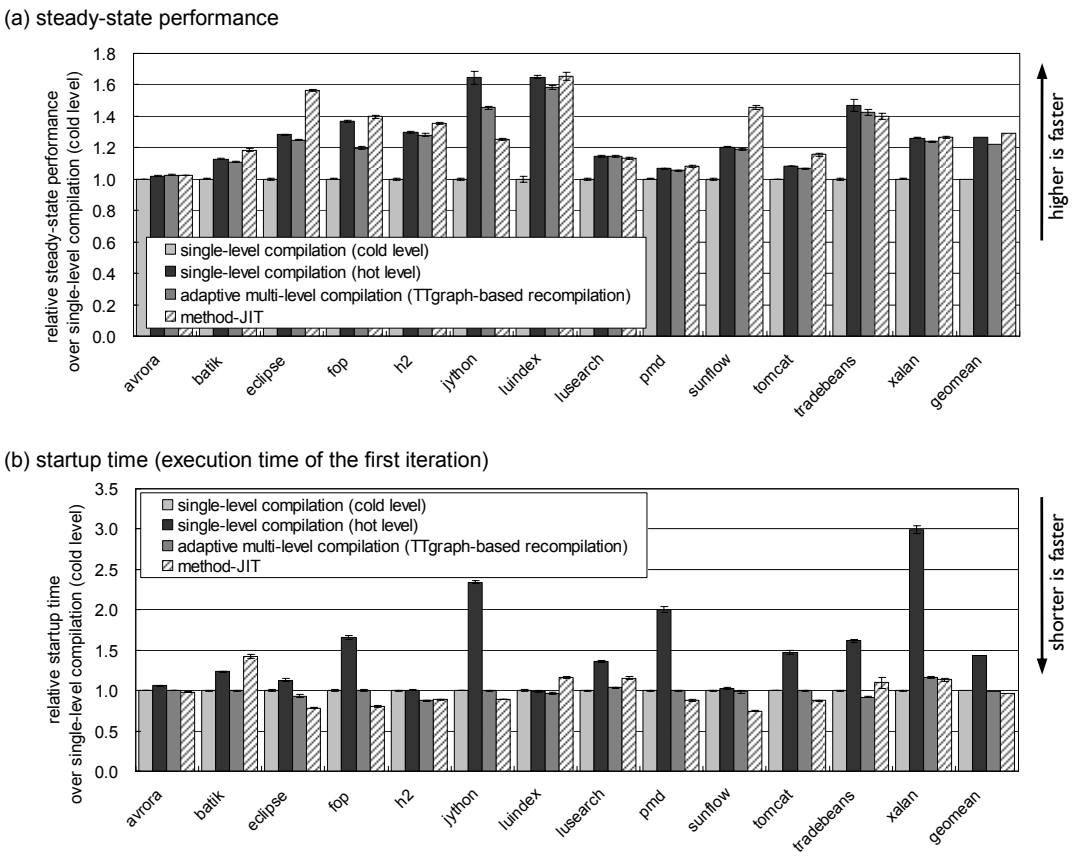

Figure 9. Steady-state performance and startup time with and without adaptive multi-level compilation.

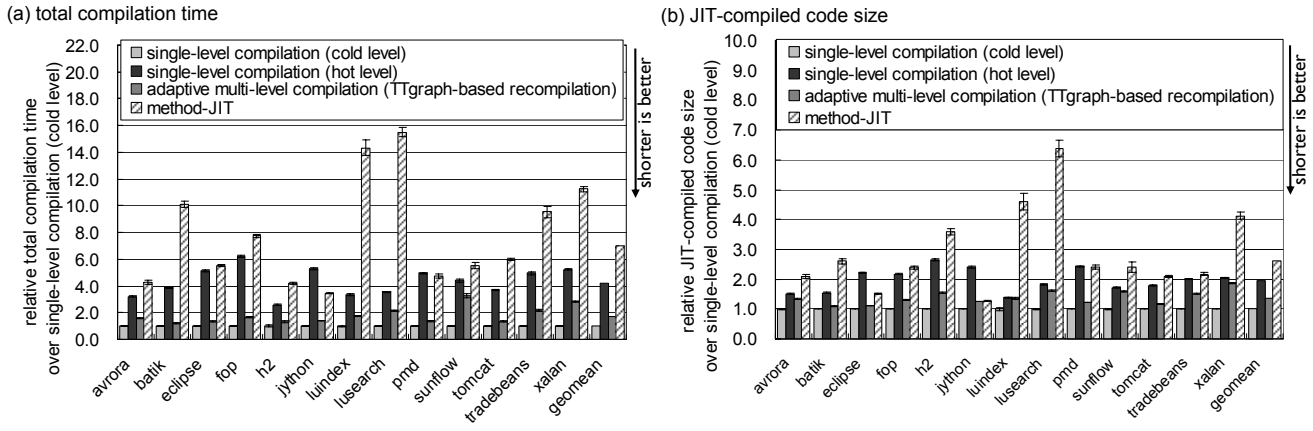

Figure 10. Compilation time and compiled code size.

trace. Hence, even with very large maximum trace length, the longest trace will be limited. Allowing more complicated control flow may create more optimization opportunities in exchange for additional compilation work. Allowing complicated control flow in a trace will not only enlarge the compilation scope but also help loop optimizations. This is because a cyclic trace does not include code sequence just before the loop, such as initialization of the loop counter, and hence lots of optimization opportunities are missed. Another reason for the slower startup was lower performance of the interpreter in the trace-JIT due to the trace selection overhead.

Figure 10 shows the total compilation time and the total size of the JIT-compiled code. These graphs are quite similar to those of the startup times except for the method-JIT because the startup time is included most of the compi-

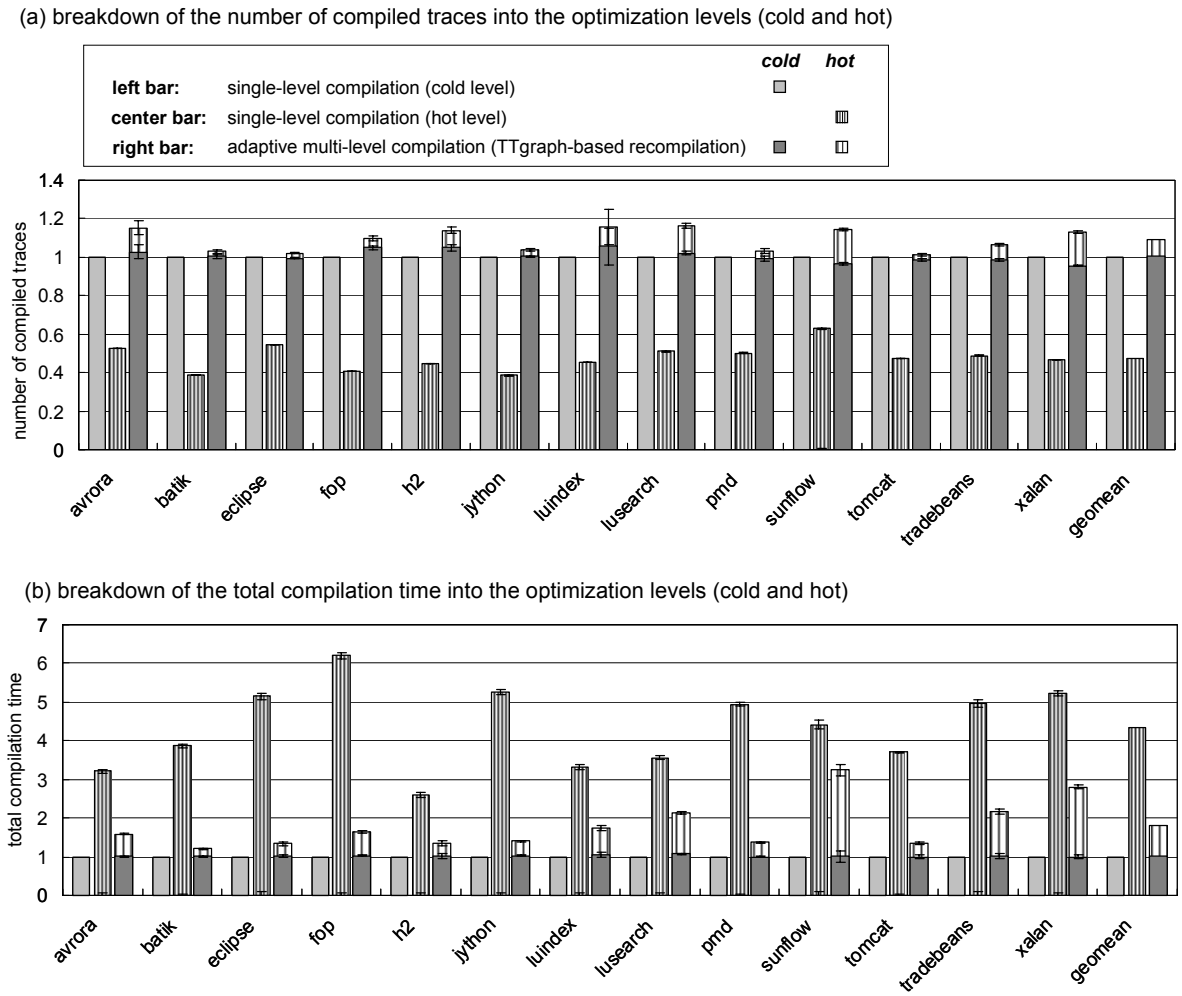

Figure 11. Numbers of compiled traces and total compilation times for each optimization level.

lation times and longer compilation times often lead to slower starts. The adaptive multi-level compilation consumed about 71.1% more compilation time on average than the single-level compilation at the cold level due to the additional compilation time consumed for the upgrade recompilations. However, the compilation time of the adaptive multi-level compilation was much shorter than the compilation time when compiling all of the traces at the hot level. Similarly, the JIT-compiled code size was increased by using the upgrade recompilation but was still much smaller than when using only the hot level. The compilation time and code size of the method-JIT were significantly larger because the method-JIT has more optimization levels for recompilation and hence one hot method was compiled repeatedly. Another reason for the shorter compilation times of the trace-JIT was that optimizations in the trace-JIT were less costly than the same optimizations in the method-JIT due to the simpler control flow in the trace compilation scope.

For further insight into the characteristics of the adaptive multi-level compilation, Figure 11 is a breakdown of the number of compiled traces and total compilation time for the two optimization levels. There are three bars for each benchmark. The left bar represents the single-level compilation at the cold level, the center bar represents single-level compilation at the hot level, and the right bar represents our adaptive multi-level compilation.

As shown in Figure 11(a), the number of traces recompiled at the hot level was not significant compared to the total number of traces compiled. The number of upgrade compilations was 6.0% of the all compilations on average (though up to 15.6%). Such relatively small numbers of upgrade compilations gave significant performance improvements in the steady state, as shown in Figure 9.

Although the number of traces compiled at the hot level was only 6.0% on average, the compilation time used for upgrade compilations was 34.8% on average and up to 68.8% of the total compilation time. This mens that one

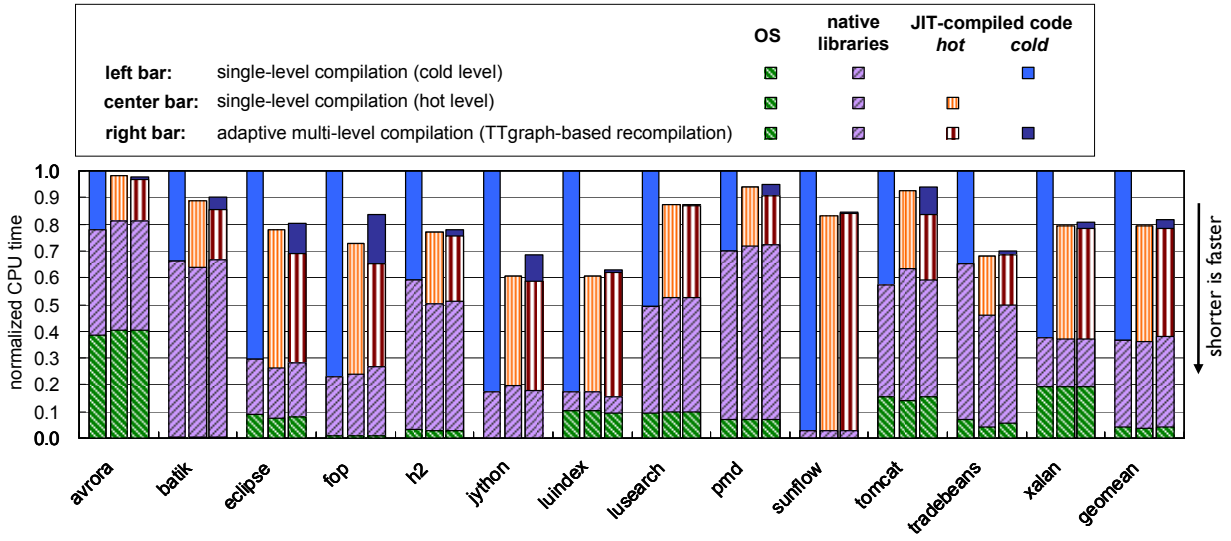

Figure 12. CPU time breakdown into JIT compiled code for the hot and cold levels, native library, and OS.

upgrade compilation at the hot level used about five times more compilation time than one cold-level compilation due to the more aggressive optimizations and larger compilation scope. Despite the visible increase in the total compilation time due to the upgrade compilations, the startup time with multi-level compilation was not increased much, as shown in Figure 9(b). The upgrade compilations were invoked by the timer-based profiler and hence the compilation time of the upgrade compilation was typically excluded from the startup time. Unlike our multi-level compilation, the startup time of the single-level compilation using the hot level was badly affected by the increase in the compilation time because costly hot-level compilations frequently occurred during the startup phase of the JVM.

Figure 12 shows profiles of the CPU time broken down into JIT compiled code at the cold level, JIT compiled code at the hot level, native library in the JVM, and the OS, based on how much active CPU time was spent in each component as measured by using the hardware performance monitor of the processor. Only in this figure, we averaged the results of four measurements. On average, 85.9% of the CPU time for JIT compiled code is spent in hot-level compiled code, while only 14.1% of the time is spent in cold-level compiled code. These results show that our multi-level compilation technique successfully captured really hot code paths and recompiled them by using the higher optimization level.

4.3 Comparison of the basic recompilation and the TTgraph-based recompilation

This section focuses on how our recompilation technique using the TTgraph described in Section 3 improved the overall performance compared to the basic recompilation. In this section, we use the multi-level compilation with the basic recompilation as the baseline for all of the figures. In addition, the TTgraph-based recompilation tends to use more compilation time than the basic recompilation. For the fair comparison, we tested the basic recompilation with a lower recompilation threshold to use roughly the same amount of the total compilation time on average (labeled "basic recompilation with aggressive threshold" in figures).

Figure 13 shows the speedup in the steady-state performance over the basic recompilation. A positive value means better performance than the basic recompilation. The TTgraph-based recompilation improved the performance by up to 6.5% and 1.8% on average. Even with the same basic recompilation, using the aggressive threshold gave the performance improvement of 0.5% on average. However the improvement was smaller than the TTgraph-based recompilation in all benchmarks.

Figure 14 depicts the improvements in the startup performance. The performance of the TTgraph-based recompilation spanned from 8.0% slower to 3.5% faster than the basic recompilation. The average for all of the benchmarks for startup performance was not much different from that of the basic recompilation. It was only 0.4% faster. The startup times for the basic recompilation with the aggressive threshold were also quite similar to the baseline.

Our results showed that with the given total compilation time, the TTgraph-based recompilation technique achieved better steady-state performance than the basic recompilation with almost similar startup performance.

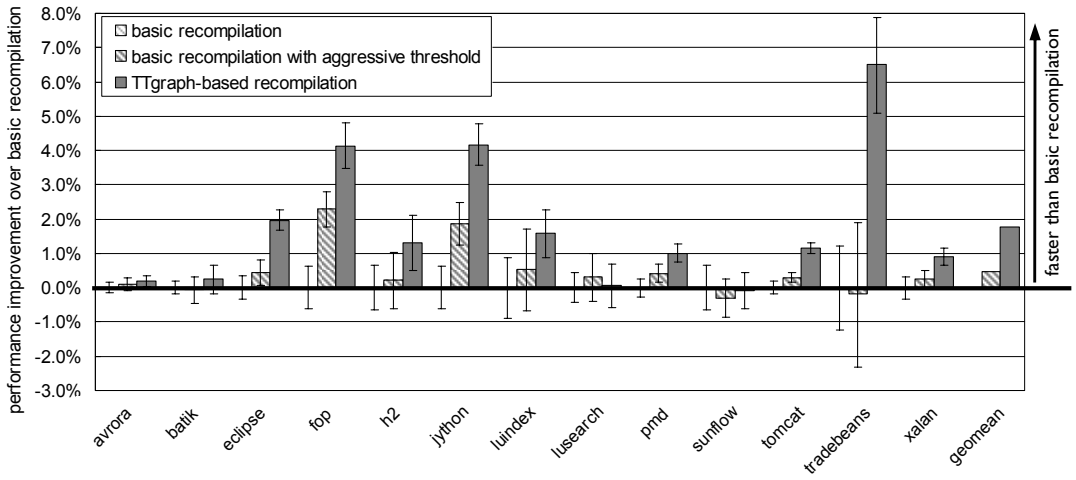

Figure 13. Changes in steady-state performance with multi-level compilation using various recompilation techniques.

5. Related Work

Upgrade compilation is a standard technique that is supported in many modern dynamic compilers for Java and other languages, such as Sun Hotspot JVM [2], IBM JVM [3, 4], Jikes RVM [5], and SELF [13]. In all of these systems, upgrade compilation is triggered by hotness detection, after which recompilation selects an optimization strategy. In our work, the hotness and optimization strategy selection used by the trace-JIT are similar to those used in the method-JIT. But upgrade compilation in the trace-JIT has another degree of freedom in deciding how to re-record a trace for recompilation, such as the selection of trace head during trace re-recording.

The trace-based compilation was first introduced by binary translators and optimizers [7], where method structures are not available. Recently, trace-based compilation has gained popularity in dynamic compilers because it can potentially provide more opportunities for specialization [14, 15] with smaller resource requirements [16, 17] compared to the method-based compilation.

Some existing trace-JIT support recompilation. In [18-20], a trace that is aborted from the compiled code into the interpreted execution too frequently may be recompiled to include a trace starting from the side exit point in the compilation scope. Such recompilation is used to limit the penalty of a badly formed trace by extending the coverage by the compiled code, not for upgrading compilation for really hot traces. Because the trace recompilation is invoked by taken side exit in these existing systems, the recompiled traces are not necessarily hotter than other traces and hence using a higher optimization level for them cannot be justified. In contrast, the hotness-based upgrade recompilation described in this paper identifies really hot

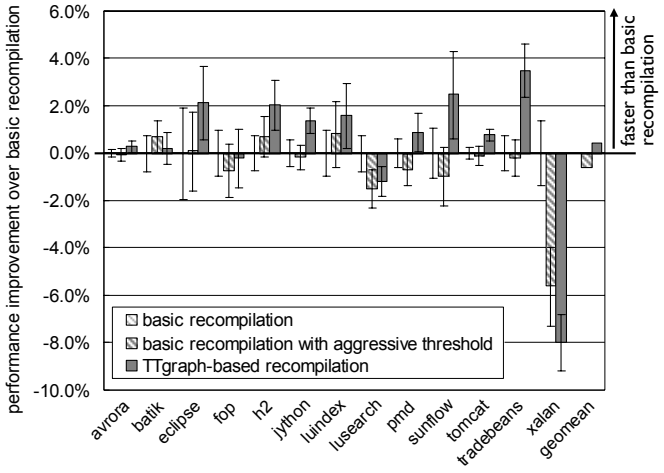

Figure 14. Changes in startup performance with multi-level compilation using various recompilation techniques.

paths by using timer-tick-based sampling. Hence, applying costly optimizations for recompilation can be justified.

Most of the trace-based systems, including our trace-JIT, have an interpreter to monitor the program execution. In contrast, SPUR [21] and Maxpath [22] do not have interpreters and use instrumented non-optimized compiled code generated by a method-JIT to monitor the execution. After the hot paths are detected by the instrumented compiled code, the detected hot paths are recompiled as a trace by an optimizing trace-JIT. Therefore, the goals of recompilation are different from ours. Also, we do not need to insert instrumented code in the compiled code to monitor execution. Instead, we use lightweight timer-based sampling to detect the hot paths for recompilation.

6. Summary

In this paper, we described our adaptive multi-level compilation technique for a trace-based JIT compiler. As in the existing multi-level compilation for a method-JIT, we start JIT compilation with a small compilation scope and a low optimization level to achieve fast startup. Then we identify hot paths with a timer-based sampling profiler, generate long traces that capture the hot paths, and recompile them with a higher optimization level to yield better peak performance. Our multi-level compilation technique successfully identified and recompiled a few traces at a higher optimization level. Our multi-level compilation accelerated the code by up to 58.5% in the steady state and 22.2% on average without degrading the startup performance compared to compiling all of the traces at the cold level. We also showed that our technique to build and exploit the TTgraph for selecting the compilation scope improved the performance over the basic recompilation technique.

Our results showed that the adaptive multi-level compilation is a practical technique in trace-based systems to balance the steady-state performance and startup time by taking advantage of different optimization levels.

References

[1] P. Wu, H. Hayashizaki, H. Inoue, and T. Nakatani, "Reducing Trace Selection Footprint for Large-scale Java Applications with no Performance Loss", in *Proceedings of the ACM Object-Oriented Programming, Systems, Languages & Applications,* pp. 789–804, 2011.

[2] M. Paleczny, C. Vick, and C. Click, "The Java Hotspot(tm) Server Compiler", in *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium*, pp. 1–12, 2001.

[3] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. "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.

[4] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, "A dynamic optimization framework for a Java just-in-time compiler", in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 180–195, 2001.

[5] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. "Adaptive optimization in the Jalapeño JVM", in *Proceedings of the ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. pp. 47–65. 2000.

[6] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani, "A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler", in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 246–256, 2011.

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

[8] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. "Evaluating the accuracy of Java profilers", in *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*. pp. 1879–197, 2010.

[9] M. Hirzel, and T. M. Chilimbi, "Bursty tracing: a framework for low-overhead temporal profiling", *in Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization*, pp. 117–126, 2001.

[10] H. Hayashizaki, P. Wu, H. Inoue, M. Serrano, and T. Nakatani, "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*, pp. 405–418, 2011.

[11] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. "IBM POWER6 microarchitecture". *IBM Journal of Research and Development*, Vol. 51 (6), pp. 639–662, 2007.

[12] S. M. Blackburn *et al*., "The DaCapo Benchmarks: Java Benchmarking Development and Analysis", in *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 169–190, 2006.

[13] U. Holzle and D. Ungar, "A third generation self implementation: Reconciling responsiveness with performance", in Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 229–243, 1994.

[14] C. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "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.

[15] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz., "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.

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

[17] B. Cheng and B. Buzbee, "A JIT Compiler for Android's Dalvik VM", *Google I/O developer conference*, 2010. http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html

[18] C. Häubl and H. Mössenböck, "Trace-based Compilation for the Java HotSpot Virtual Machine", in *Proceedings of the*

International Conference on the Principles and Practice of Programming in Java, pp. 129–138, 2011.

[19] C. Häubl, C. Wimmer, and H. Mössenböck, "Evaluation of trace inlining heuristics for Java", in *Proceedings of the Annual ACM Symposium on Applied Computing*, pp. 1971–1876, 2011.

[20] A. Gal, "Efficient bytecode verification and compilation in a virtual machine", PhD thesis, University of California, Irvine, 2006.

[21] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter, "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.

[22] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz, "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.