Reducing Trace Selection Footprint for Large-scale Java Applications without Performance Loss

Peng Wu Hiroshige Hayashizaki Hiroshi Inoue Toshio Nakatani
IBM Research
pengwu@us.ibm.com,{hayashiz,inouehrs,nakatani}@jp.ibm.com

Abstract

When optimizing large-scale applications, striking the balance between steady-state performance, start-up time, and code size has always been a grand challenge. While recent advances in trace compilation have significantly improved the steady-state performance of trace JITs for large-scale Java applications, the size control aspect of a trace compilation system remains largely overlooked. For instance, using the DaCapo 9.12 benchmarks, we observe that 40% of traces selected by a state-of-the-art trace selection algorithm are short-lived and, on average, each selected basic block is replicated 13 times in the trace cache.

This paper studies the size control problem for a class of commonly used trace selection algorithms and proposes six techniques to reduce the footprint of trace selection *without* incurring any performance loss. The crux of our approach is to target redundancies in trace selection in the form of either short-lived traces or unnecessary trace duplication.

Using one of the best performing selection algorithms as the baseline, we demonstrate that, on the DaCapo 9.12 benchmarks and DayTrader 2.0 on WebSphere Application Server 7.0, our techniques reduce the code size and compilation time by 69% and the start-up time by 43% while retaining the steady-state performance. On DayTrader 2.0, an example of large-scale application, our techniques also improve the steady-state performance by 10%.

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

General Terms Algorithms, Experimentation, Performance

Keywords Trace selection and compilation, profiling, Java

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

1. Introduction

1.1 Trace compilation for large applications

How to effectively optimize large-scale applications has always posed a great challenge to compilers. Although method inlining expands the compilation scope of a method JIT, its effectiveness can be limited when the compilation target lacks hot-spots and has numerous calling contexts and deep call chains, all of which are common in large-scale Java applications.

While trace-based compilation was traditionally explored where mature JITs are absent, such as binary translators [1, 6, 7], easy-to-develop JITs [10, 21], and scripting languages [2, 4, 5, 8, 17], we explore trace compilation for Java, focusing on large-scale applications. To our problem domain, the promise of trace-based compilation lies in its potential to construct better compilation scopes than method inlining does. In a trace compiler, a *trace* is a single-entry multiple-exit region formed out of instructions following a real execution path. The most appealing trait of traces is its ability to span many layers of method boundaries, naturally achieving the effect of partial inlining [20], especially in deep calling contexts.

The challenges of trace compilation for Java are also aplenty. As a first step, recent work in [13, 16] has significantly bridged the performance gap between trace compilation and the state-of-the-art method compilation for Java, where a trace JIT is able to reach 96% of the steady-state performance of a mature product JIT on a suite of large-scale applications. This is achieved primarily by aggressively designing the trace selection algorithm to create larger traces.

1.2 Space Efficiency of Trace Selection

A trace selection design needs to optimize all aspects of system performance including steady-state performance, start-up and compilation time and binary code size. While optimizing for steady-state performance often leads to selection algorithms that maximize trace scope, such a design often increases start-up and compilation time, and binary code size. The latter three all relate to one trace selection metrics as defined below.

Definition 1 *Selection footprint* is defined as the cumulative size of all traces formed by a selection algorithm.

We use *space efficiency* to refer to a trace selection algorithm's ability to maximize steady-state performance with minimal selection footprint. Space efficiency is especially important for large-scale applications where memory subsystem and compilation resources can be stressed. For example, code size bloat can degrade the steady-state performance due to bad instruction cache performance.

While space efficiency of trace selection was not extensively studied before as most trace JITs target small or medium size workloads, space considerations have been incorporated into existing selection algorithms. The common approaches fall into the following categories:

- *Selecting from hot regions.* Several trace JITs [2, 8, 10] select traces only out of hot code regions, such as loops. This approach achieves superb space efficiency when dealing with codes with obvious hot spots, but not for large-scale applications, which often exhibit large, flat execution profile.
- *Limiting trace size.* This approach limits individual trace size using heuristics expressed as trace termination conditions, such as terminating trace recording at loop headers, at existing trace heads (known as *stop-at-existing-head*), or when exceeding buffer length. These heuristics, however, sometimes can significantly degrade the performance. For instance, we observe up to 2.8 times slower performance after applying the *stop-at-existing-head* heuristic. The space and performance impact of existing trace termination heuristics are summarized in Section 6.
- *Trace grouping.* This approach groups linear traces so that common paths across linear traces can be merged [2, 8, 10, 14]. Existing trace grouping algorithms focus solely on loop regions. However, they have yet to demonstrate the ability to reach the required level of selection coverage for large-scale non loop-intensive workloads.

When dealing with large-scale applications, existing approaches are either ineffective or insufficient in reducing selection footprint, or otherwise degrade steady-state performance. In this paper, we focus on improving the space efficiency of trace selection by reducing selection footprint *without* degrading the steady-state performance for large-scale applications.

1.3 Key Observations

We focus on a class of commonly used trace selection algorithms, pioneered by Dynamo [1] and subsequently used in [5, 12, 14, 17, 21] as well as the Java trace JIT mentioned earlier. In the paper, the specific selection algorithm used is derived from [16] and is referred to as the *baseline* algorithm throughout the paper.

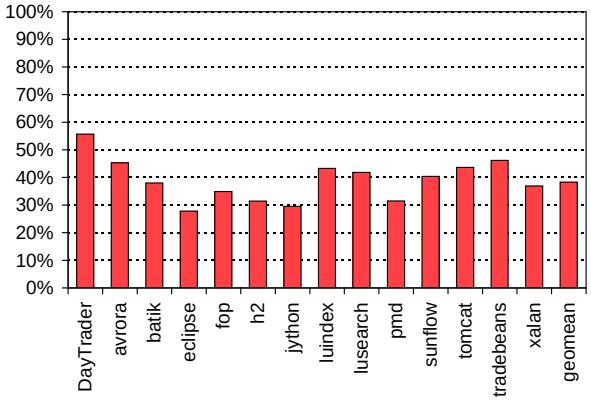

Figure 1. Percentage of traces selected by the baseline algorithm with less than 500 execution count during steady-state execution.

While the baseline algorithm is one of the best performing of its kind, it exhibits serious space efficiency issues. Figure 2 shows the traces selected by the baseline algorithm for a simple example of 5 basic blocks. In total, the baseline algorithm creates four traces (A-D) with a selection footprint of 18 basic blocks and a duplication factor of 3.6.

We identify two sources of space inefficiency in the baseline algorithm that we will briefly describe below.

Formation of short-lived traces refers to a phenomenon where some traces are formed but seldom executed. To quantify this effect, we measured the execution count of traces formed by the baseline algorithm. Figure 1 shows that 38% traces formed for the DaCapo 9.12 benchmarks and the DayTrader benchmark have less than 500 execution counts during steady-state runs.[1]

Intuitively, a trace becomes dead when its entry point is completely *covered* by traces formed later but whose entry points are topologically earlier. At that point, the original trace is no longer executed. This is analogous to rendering a method "dead" after inlining the method to all its call-sites.

Non-profitable trace duplication refers to the duplication of codes within or across traces that do not improve performance. While previous work focuses primarily on traces that are created unnecessarily long, we identified another cause of the problem, that is, duplication due to convergence of a selection algorithm. In this context, convergence refers to the state where a working set is covered completely by existing traces so that no more new traces are created.

[1] For cyclic traces, execution counts include the number of times the trace cycles back to itself.

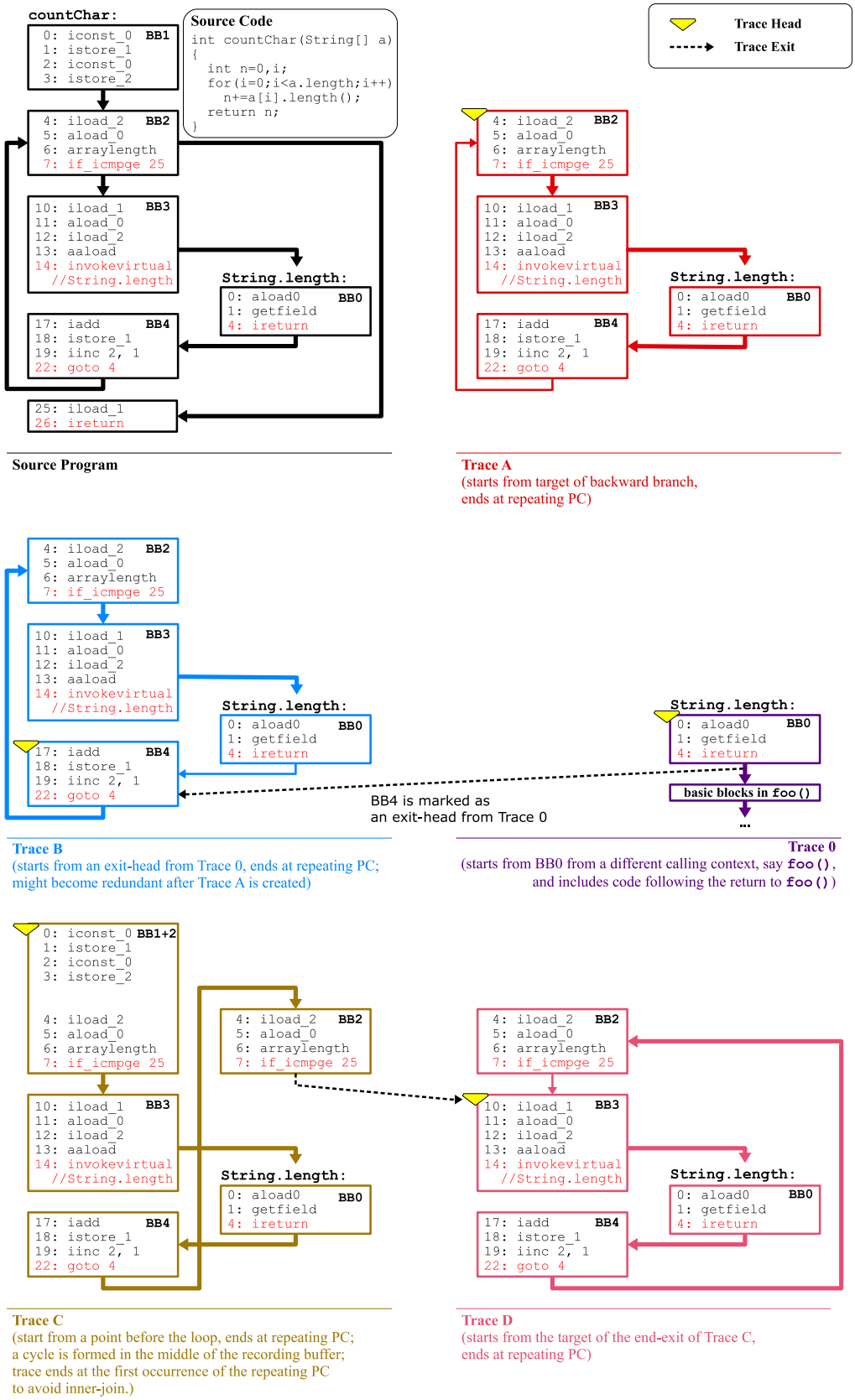

Figure 2. A working example: trace formation by the baseline algorithm.

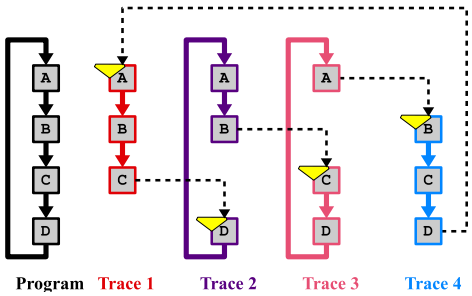

Figure 3. An example of low convergence due to forming max-length traces: when the trace buffer size (3BB) is smaller than loop body size (4BB), the algorithm takes 4 traces to converge.

The problem stems from the fact that a trace can start and end at arbitrary program points, which in the presence of tracing through cyclic paths could lead to pathological convergence. Figure 3 gives such an example: when the critical path of the loop body is too large to fit into a single trace, many largely overlapping traces are formed, all of which start and end at slightly different program points.

1.4 Evaluation and Contribution

In this paper, we proposed six techniques to collectively address the problems of short-lived trace formation (in Section 3) and trace duplication (in Section 4).

We have implemented the proposed techniques in a Java trace JIT based on IBM J9 JVM and JIT compiler. The techniques are applied to the baseline algorithm from [16], which has been heavily optimized for steady-state performance. After applying our techniques, we are able to reduce the code size and compilation time by 69%, and the start-up time by 43%, and with no degradation to the steady-state performance of the DaCapo 9.12 benchmarks and a real-world large-scale Java application, DayTrader on top of IBM's Websphere application server. On DayTrader, our techniques improved the steady-state performance by 10% due to better cache performance.

The paper makes the following contributions:

- Deepen the understanding of the space efficiency problem for a class of commonly used selection algorithms and expanded the solution space.
- Identify the problem of short-lived trace formation and proposed techniques to reduce short-lived traces.
- Identify new sources of trace duplication problem and proposed trace truncation to reduce non-profitable duplication.
- Demonstrate the effectiveness of our techniques on a set of large-scale Java applications.

2. The Baseline Trace Selection Algorithm

The baseline algorithm is a one-pass selection algorithm derived from the one used in [16]. One-pass selection algorithms form traces that are either straight-line or cyclic and contain no inner join or split points. One-pass trace selection algorithms are the most common type of selection algorithms, which include NET [1], LEI [14], YETI [21], PyPy [5], and LuaJIT [17]. It has been demonstrated that one-pass trace selection can scale to large applications and achieve the kind of coverage and performance comparable to those of a mature method JIT [13, 16].

A typical one-pass trace selection algorithm consists of two steps. First, it identifies a set of program points called *potential trace head*, and monitors their execution counts. Once the execution count of a potential trace head exceeds a threshold, trace recording is triggered, where it simply records every instruction following the execution of the trace head into a buffer called the recording buffer. Trace recording continues until a trace termination condition is satisfied, at which point, a new trace for the selected trace head is formed out of the recording buffer. The rest of the section describes these two components in detail.

The rest of the paper uses a working example to illustrate various aspects of trace selection. Figure 2 shows a code fragment that sums the number of characters in an array of strings (`countChar`) and the traces formed by the baseline selection algorithm.

2.1 Trace Head Selection

The main driver of the baseline algorithm is shown in Algorithm 1. TraceSelection(e) is invoked every time the interpreter dispatches a control-flow event, which can be 1) control-flow bytecodes such as invoke, branch and return, or 2) when an exit is taken from a trace.

The baseline algorithm identifies two types of potential trace heads (lines 17-19 in Algorithm 1). The first type is the target of a backward branch, such as $bb2$ on trace A in Figure 2. This heuristic approximates loop headers without constructing a control-flow graph. The second type is the target of a trace-exit (*exit-head*), such as $bb4$ on trace B in Figure 2.[2] This allows traces to be formed out of side branches of a trace.

The algorithm profiles the execution counts of potential trace heads such that those whose execution counts exceed a threshold, T_h, trigger a new trace recording (lines 21-27 in Algorithm 1). The algorithm assumes that counters and traces are kept in a PC-indexed global map that is manipulated by *getCounter* and *getTrace* that return the counter and trace associated with a PC, respectively, and by *getAd-*

[2] Prior to enter the loop in Figure 2, trace 0 is already formed from the entry point of `String.length` and includes codes following the return of `String.length` from a different calling context. Therefore, When the execution enters trace 0 from the loop, a trace exit is taken at the end of $bb0$.

Algorithm 1 TraceSelection(e), baseline

Input: Let e be a control-flow event, *PC* be target PC of e, and buf be the recording buffer

```
1:  /* trace recording and termination */
2:  if mode = record then
3:     if ShouldEndTrace(e) then
4:        mode ← default
5:        create trace from buf then submit to compilation
6:     else
7:        append(buf,e)
8:     end if
9:     return
10: end if
11: /* trace dispatch */
12: if getTrace(PC) ≠ null then
13:    dispatch to trace execution
14:    return
15: end if
16: /* identify potential trace head */
17: if (isBackwardBranch(e) or isTraceExit(e)) then
18:    ctr ←getAddCounter(PC)
19: end if
20: /* trace head selection and start recording */
21: if (ctr ← getCounter(PC)) ≠ null then
22:    ctr++
23:    if ctr > T_h then
24:       mode ← record, buf ← ∅
25:       append(buf,e)
26:    end if
27: end if
28: return
```

dCounter that, in addition, allocates a new counter if none exists for a PC.

The rest of the algorithm covers trace dispatch (lines 12-15 in Algorithm 1) and trace recording (lines 2-10 in Algorithm 1), where its principle component, trace termination conditions, are described in Section 2.2 and Algorithm 2.

2.2 Trace Termination conditions

Trace termination conditions are the primary mechanism in a selection algorithm to control the formation of a trace for a given trace head.

Algorithm 2, ShouldEndTrace(e), describes the termination conditions used in the baseline algorithm. This particular choice of termination conditions is intended to maximize the number of cyclic traces and the length of linear traces, both of which imply large scope for compilation and less trace transition overhead [13, 22]. Like any other selection algorithm, the baseline algorithm must address the following key aspects of when to end a trace recording.

Repetition detection ends the recording when a cyclic repetition path is detected in the recorded trace. Repetition detection is necessary for the convergence of the selection algorithm as well as the formation of cyclic traces.

The baseline algorithm (lines 1-5 in Algorithm 2) detects repetition when current program counter (PC) is already recorded in the buffer (*stop-at-repeating-pc*) [14] and when the cycle is not a false loop [13].

When a repeating PC appears at the beginning of the recording buffer, such as traces A, B, and D in Figure 2, a cyclic trace is formed. Sometimes the repeating PC appears in the middle of the recording buffer (*rejoined*), then the recording buffer is backtracked to the rejoined point to avoid introducing inner join to the trace, such as trace C in Figure 2.

Algorithm 2 ShouldEndTrace(e), baseline

Input: Let e be a control-flow event, *PC* be target PC of e

Output: **true** if the trace should be terminated at e, or **false** if trace recording should continue.

```
1:  if PC is already recorded on the trace and not a false-loop then
2:     if PC is not the trace head then
3:        set trace length be the index of the repeating PC
4:     end if
5:     return true
6:  else if recording buffer overflows then
7:     return true
8:  else if e is an JNI call that cannot be included on trace then
9:     return true
10: else if e is an irregular event (e.g., exception) then
11:    return true
12: else
13:    return false
14: end if
```

Buffer overflow ends the recording when the recording buffer reaches its size limit (lines 6-7 in Algorithm 2).

Tracing out-of-scope ends the recording when an event outside the tracing scope has occurred, such as invoking a native call.

Tracing scope is a property of the trace system and may not be violated as it may result in incorrect traces. In our system, tracing beyond an exception throw or a JNI call is not allowed (lines 8-11 in Algorithm 2).

2.3 Characteristics of the Baseline Algorithm

The baseline algorithm is designed to maximize the steady-state performance and reuses many existing heuristics in other systems.

Table 1 summarizes the basic characteristics of traces selected by the baseline algorithm for the DaCapo 9.12 and the DayTrader 2.0 benchmarks (setup details in Section 5).

Benchmark	Description	coverage	# traces	bbs/trace	dup factor	call/trace
avrora	simulates programs running on AVR microcontrollers	100.0%	1853	43	11	27
batik	produces SVG images based on unit tests in Apache Batik	99.7%	4817	43	11	26
eclipse	executes the jdt performance tests for Eclipse	99.9%	27862	40	15	21
fop	parse and format an XSL-FO file and generate a PDF file	99.7%	6096	62	19	41
h2	a JDBCbench-like in-memory benchmark	100.0%	4124	57	17	34
jython	interprets the pybench Python benchmark	99.7%	10109	77	20	50
luindex	uses lucene to indexes a set of documents	99.9%	1376	31	7	17
lusearch	uses lucene to do a text search of keywords	100.0%	1447	39	9	23
pmd	analyzes Java classes for a set of source code problems	98.6%	7029	56	25	33
sunflow	renders a set of images using ray tracing	100.0%	1624	44	11	29
tomcat	query against Tomcat to retrieve/verify webpages	99.4%	17155	49	13	28
tradebeans	DayTrader via Java Beans on top of GERONIMO and h2	100.0%	8621	47	9	28
xalan	transforms XML documents into HTML	99.5%	3119	59	15	35
DayTrader	DayTrader 2.0 on IBM WebSphere Application Server 7.0	100%	19809	69	15	40
geomean		99.7%	3869	49	13	30

Table 1. Characteristics of traces formed by the baseline algorithm for DaCapo 9.12 and Daytrader benchmarks. *Coverage* is the percentage of total bytecodes executed from traces; *# traces* is the number of compiled traces; *bbs/trace* is average number of basic blocks per trace; *dup factor* is the ratio between the number of bytecodes on traces and that of bytecodes in distinct basic blocks on traces; and *call/trace* is the number of invoke or return bytecodes per trace.

One important characteristic is the coverage of a trace selection. High coverage is a particular requirement for a bytecode trace JIT like ours where more than ten-fold performance gap exists between being "covered" (compiled) and "not covered" (interpreted) by the trace JIT. As shown in Table 1, the algorithm achieves close to 100% coverage at the steady-state, similar to that of the method JIT.

Table 1 also shows other static characteristics of the trace selection. The number of traces selected ranges from 1400 to 27K, indicating the algorithm's ability to support large working sets. It is also observed that traces formed by the baseline algorithm are quite long with an average 49 basic blocks per trace and can span many layers of method boundaries, where, on average, 30 invoke or return bytecodes are included per trace.

3. Reducing Short-lived Traces

In this section, we propose a set of techniques that reduce selection footprint without degrading the performance by targeting traces that are short lived.

3.1 Short-lived Trace Formation

Short-lived trace formation refers to the phenomenon that the selection algorithm creates traces that become *dead* short after their creation. For instance, the baseline algorithm first creates trace B in Figure 2, and shortly after trace A is created.[3] Since the head of trace B ($bb3$) is dominated by that of trace A ($bb1$) and is included in trace A, the creation of trace A renders trace B dead.

[3] Trace B is created before A because the baseline algorithm identifies $bb3$ as a potential trace head first, before the first backward branch targeting $bb1$ is taken.

Intuitively, a trace becomes dead when the head of the trace is completely covered by later formed traces such that the trace is no longer dispatched. A formal definition of dead traces is given below.

Definition 2 An instruction x is an *infeasible dispatch point* at time t, if after t, there is no invocation of *TraceSelection*(e) where the target of e is x.

Definition 3 A trace A starting from x becomes *dead* at time t if, after t, x becomes an infeasible dispatch point.

Short-lived trace formation is an inherent property of a trace selection algorithm that satisfies the following two conditions.

Condition 1 Two traces may be dispatched in the reverse order of how their corresponding trace heads are selected.

Condition 2 The head of an earlier trace can be part of a later trace.

The baseline algorithm satisfies both conditions. Condition 1 is a property of trace head selection. Most selection algorithms satisfies this condition because there is no guaranteed ordering on how a trace head is selected. Potential trace heads may accumulate counter values at different speed. For instance, basic blocks at control-flow join are executed more often than their predecessors.

Condition 2, on the other hand, is a property of trace termination conditions. Certain termination conditions, such as *stop-at-existing-head*, can prevent the condition to be satisfied.

Algorithm 3 TraceSelectionWithSizeControl(e), optimized

Input: Let e be a control-flow or an exact-bb event, *PC* be target PC of e, P_h and T_h be the profiling and trace-head threshold, and buf be the recording buffer.

```
1: /* trace recording */
2: if mode = record then
3:    if ShouldEndTrace(e) then
4:       StructureTruncation(buf, e)
5:       clear counters (if any) of bbs in buf
6:       create trace from buf, mode ← default
7:    else
8:       append(buf,e)
9:    end if
10:   return
11: end if
12: /* trace path profiling */
13: if curr_tr ≠ null then
14:   if PC = getStartPC(curr_tr, curr_pos) then
15:      curr_pos++
16:      return
17:   else
18:      incExitFreq(curr_tr,curr_pos)
19:      curr_tr ← null
20:   end if
21: end if
22: /* trace head selection and dispatch */
23: if isBackwardBranch(e) or isTraceExit(e) then
24:   if (tr ← getTrace(PC)) ≠null then
25:      if isCompiled(tr) then
26:         dispatch to binary address of tr
27:      else
28:         /* enter trace path profiling mode */
29:         tr.entryFreq++
30:         if tr.entryFreq > P_h then
31:            profileTruncation(tr)
32:            submit tr to compilation queue
33:         end if
34:         curr_pos++, curr_tr ← tr
35:      end if
36:      return
37:   end if
38:   ctr ← getAddCounter(PC)
39:   if (++ctr) > T_h then
40:      mode ← record, buf ← ∅, append(buf, e)
41:   end if
42: end if
43: return
```

3.2 Short-lived Trace Elimination

In this section, we identify the causes of short-lived trace formation in the baseline algorithm and propose a more space efficient algorithm, TraceSelectionWithSizeControl(e), as shown in Algorithm 3.

The new algorithm creates only 2 traces for the working example, trace A and a trace that contains bytecode 0-4, with a selection footprint of 5 basic blocks and a duplication factor of 1. This is in contrast to the selection footprint of 18 basic blocks by the baseline algorithm as shown in Figure 2. The rest of the section describes the new algorithm in detail.

3.2.1 Constructing Precise Basic Blocks

In the baseline algorithm, TraceSelection(e) is invoked every time the interpreter executes a control-flow bytecode. The bytecode sequence between consecutive control-flow bytecodes is called a dynamic instruction block. Dynamic instruction blocks can be partially overlapping, such as $bb1+2$ and $bb2$ of trace C in Figure 2. We refer to such dynamic instruction blocks as *imprecise* basic blocks.

Dynamic instruction blocks can trigger the formation of short-lived traces. Consider the formation of trace C and D in Figure 2. A trace recording starts from bytecode 0 and continues through two iterations of the loop. The recording is terminated when a repeating PC, bytecode 10, is detected in the middle of the recording buffer. Trace C is formed by backtracking the recorded trace to bytecode 10, and then trace D is created from the target of the end-exit from trace C. Once trace C and D are formed, trace A and B become dead because their respective entry points, $bb2$ and $bb4$ become infeasible dispatch points.

Such short-lived traces are caused by the termination condition that detects repetition by checking repeating PCs (as line 1 of Algorithm 2) at control-flow bytecodes. This termination condition works fine only when two distinct basic blocks are disjoint. Because of imprecise basic blocks, the baseline algorithm detects bytecode 10 as the repeating PC in the recording buffer, whereas bytecode 4 is the actual first repeating PC.

To address this problem, we identify boundaries of precise basic blocks and call TraceSelection(e) at the end of each precise basic block. The new algorithm correctly detects that bytecode 4 is the first repeating PC in the recording buffer.

3.2.2 Trace-head Selection Optimization

The baseline algorithm performs two lookups for each invocation of TraceSelection(e): 1) look up and dispatch the trace for event e (lines 12-15 of Algorithm 1), and 2) look up and update the counter associated with e (lines 21-27 of Algorithm 1). While this design dispatches traces and accumulate frequency counts as fast as possible, it can cause pathological formation of short-lived traces.

Consider the formation of trace A and B. Despite the loop body having only a single execution path, the baseline algorithm identified two potential trace heads, $bb1$ (the target of a backward branch) and $bb3$ (the target of a side-exit from trace 0). Selecting multiple potential trace heads along one critical path can result in short-lived traces because only

the one that dominates the others along the same path can survive as the head of a long-lasting trace.

To address this problem, we propose to dispatch traces and update counters only when a trace exit or a backward branch is taken, shown as lines 24-41 of Algorithm 3. The algorithm has two nice properties:

- A counter is always available at the time of counter and trace lookup because if one is not available, a new counter is allocated (as line 38 of Algorithm 3). This property bounds the number of failed counter and trace lookups per PC to T_h, thus reduces lookup related runtime overhead. In a trace runtime, failed lookups can be expensive as well as pervasive when trace selection coverage is low such as during start-up time.
- It imposes a partial order on how potential trace heads are selected. For instance, the restriction of dispatching traces only at trace-exit and backward branch events prevents trace 0 from being dispatched in the loop body. As a result, $bb3$ is not marked as a potential trace head.

3.2.3 Clearing Counters along Recorded Trace

The third technique we propose is a simple heuristic: when a new trace is formed, we clear the counter value of any basic block on the trace (if any) when the basic block is topologically after the head of the recorded trace (line 5 of Algorithm 3).

One cause of short-lived traces is that trace head selection is based on counter values combining execution counts of a given PC along *all* paths. By clearing counter values of any potential trace head on a newly formed trace, execution counts contributed by the newly formed trace are excluded.

The rationale of using topological ordering to decide whether to clear a counter is to prevent a short-lived trace from clearing the counter of a long-lived trace. We use a simple heuristic to approximate topological ordering: a basic block x is considered topologically earlier than a basic block y, if x and y belong to the same method and if the bytecode index of x is less than that of y.

3.2.4 Trace Path Profiling

While the techniques proposed before all reduce the formation of short-lived traces, the next technique, *trace path profiling*, prevents short-lived traces from being compiled after they become dead.

The algorithm for trace path profiling is shown as lines 13-21 and 29-34 in Algorithm 3. It works as follows.

1. A newly formed trace is not immediately submitted to compilation, instead it is kept in a "nursery" and interpreted for a short time.
2. When the trace is being interpreted, the interpreter records the entry and exit counts for each basic block on the trace, but does not update counters associated with any basic block on the trace and does not dispatch to other traces.
3. A nursery trace is compiled only if its entry count exceeds a predefined threshold, at which point, the profiling mode ends. Traces that never leave the "nursery" are dead traces.

Intuitively, trace path profiling can identify dead traces because it mimics the execution of a compiled trace, therefore short-lived traces that manifest in a binary trace execution also manifests in trace path profiling. The more fundamental explanation of this effect has to do with more accurate accounting by excluding execution frequencies from infeasible dispatch points. In trace path profiling, the implementation detail of not updating counters associated with any basic blocks on the trace is crucial to dead trace elimination. It has the effect of preventing execution counts from other paths to be counted towards that of a potential trace head or the entry count of a nursery trace.

A second key aspect of trace path profiling is that while traces may be formed out of sync with respect to the topological ordering of trace heads, program execution always follows topological orders. As such, during trace path profiling, traces that start from a topologically earlier program point are dispatched first, thus render those starting from topologically later program points dead.

4. Reducing Trace Duplication

In this section, we study the problem of code duplication across traces and propose techniques to reduce unnecessary duplication.

4.1 The problem of Trace Duplication

Trace duplication refers to the phenomenon that a program point is included in many traces in a given trace formation. The degree of duplication by the baseline algorithm is quite significant. As shown in Table 1, in our benchmarks, on average, each distinct PC is duplicated 13 times across all traces.

Trace duplication is an inherent property of trace selection and often a desirable feature as it reflects a selection algorithm's innate ability to specialize. Fundamentally, duplication happens when tracing through the merge points of a flow graph, such as loop headers and the entry and return points of methods with multiple call-sites. The inlining effect of trace selection, for instance, is the result of tracing through the entry point of a method.

However, not every form of trace duplication is beneficial. In addition to short-lived traces, we identify three other forms of trace duplication that are likely not beneficial:

Duplication due to slow convergence refers to duplication caused by convergence issues of a selection algorithm.

One particular form of convergence problem is triggered by max-length traces as shown in the example in Figure 3. Max-length traces have the property that those that start from different program points often end at differ-

Algorithm 4 StructureTruncation(buf,bb)

Input: Let buf be the trace recording buffer with n bbs, ML be the maximal trace length, and bb be the bb executed after $buf[n-1]$

Output: returns the length of the truncated trace.

```
1: if buf is cyclic or n = 1 then
2:   return n
3: end if
4: for i ← 1 to n − 1 do
5:   if isLoopHeader(buf[i]) then
6:     let L be the loop whose header is buf[i]
7:     if isloopExited(L, i, buf) = false then
8:       if trunc-at-entry-edge and isEntryEdge(buf[i − 1],buf[i]) then
9:         return i
10:      end if
11:      if trunc-at-backedge and isBackEdge(buf[i − 1],buf[i]) then
12:        return i
13:      end if
14:      if trunc-at-loop-header then
15:        return i
16:      end if
17:    end if
18:  end if
19: end for
20: if n = MLand isTraceHead(bb) = false then
21:   for i ← n − 1 to 1 do
22:     if isTraceHead(buf[i]) then
23:       Let tr be the trace whose head is buf[i]
24:       if match(buf[i : n),tr[0:n − i]) and isMethod-Returned(i, buf)=false then
25:         return i
26:       end if
27:     end if
28:   end for
29: end if
30: return n
```

ent program points. When combined with tracing along cyclic paths, this property can cause slow convergence of a selection algorithm.

Loop-related duplication refers to duplication as the result of tracing through a common type of control-flow join, loop headers.

One form of unnecessary duplication happens when tracing through the entry-edge of a loop. This is analogous to always peeling the first iteration of a loop into a trace.

Another form of duplication happens when tracing through the backedge or exit-edge of a loop (also known as tail duplication).

Trace segment with low utilization refers to the case where the execution often takes a side-exit before reaching the tail segment of a trace.

The most common scenario of this problem manifests when a mega-morphic control-flow bytecode, such as the return bytecode from a method with many calling contexts, appears in the middle of the trace. For example, trace A in Figure 2 contains a `return` bytecode from `String.length` that has many different calling contexts. As a result, the return bytecode on trace A is a hot side-exit and a good candidate for truncation.

4.2 Trace Truncation

We propose *trace truncation* that uses structure or profiling information to determine the most profitable end point of a trace. We propose two types of trace truncation. One is *structure-based* that applies truncation based on static properties of a recorded trace (shown as StructureTruncation in Algorithm 4). The other is *profile-based* that truncates based on trace path profiling information (lines 31 in Algorithm 3).

Traditionally, a selection algorithm controls duplication by imposing additional termination conditions. Compared to this approach, trace truncation has the advantage of being able to *look ahead* and use the knowledge on the path beyond to decide the most profitable trace end-point.

Since trace truncation may shorten lengths of active traces, care must be taken to minimize degradation to performance. For this consideration, we define the following guidelines of where *not* to apply truncation:

- *Do not truncate cyclic traces.* Cyclic traces can capture large scopes of computation that are disproportional to its size, therefore the performance cost of a bad truncation may outweigh the benefit of size reduction.
- *Do not truncate between a matching pair of method entry and return.* The rule preserves the amount of partial inlining in a trace, which is a key indicator of trace quality in our system.
- *Do not truncate at non trace-heads.* This rule prevents truncation from introducing new potential trace heads (thus new traces) and worsening the convergence of trace selection.

4.2.1 Structure-based Truncation

Structure-based truncation is applied immediately after a trace recording and ends before the trace is created. We propose the following heuristics for structure-based truncation. The first three exploit loop structures for truncation. The last one is specifically designed for max-length traces with no loop-based edges.

- *trunc-at-loop-entry-edge* that truncates at the entry-edge to the first loop on the trace with more than one iteration.

  This is based on the consideration that peeling the first iteration of a loop is likely not profitable.

- *trunc-at-loop-backedge* that truncates at the backedge to the first or last loop on the trace with more than one iteration.

  This is based on the consideration that the backedge is a critical edge that forms cycles. Therefore, truncation at backedge may improve the convergence of the algorithm. This heuristic allows cyclic traces to be formed on loop headers, but not on other program points in the loop.

- *trunc-at-loop-header* that truncates at the header of the first/last loop on the trace with more than one iteration.

  This is a combination of the previous two heuristics.

- *trunc-at-last-trace-head* that truncates at the last location on the trace, where 1) it is the head of an existing trace, 2) the existing trace matches the portion of the trace to be truncated, 3) it is not in between a matching pair of method enter and return.

The structure-based truncation algorithm is given in Algorithm 4, where *isMethodReturned* checks whether a potential truncation point is between the entry and return of a method on the trace; *isLoopExited(L,i,buf)* assumes that the ith basic block in buf is the header of loop L and checks if the remaining portion of the trace exits from the body of L[4], and *isEntryEdge* (*isBackEdge*) checks whether an edge is the loop entry-edge (backedge).

4.2.2 Profile-based Truncation

Profile-based truncation uses the profiling information collected by trace path profiling to truncate traces at hot side-exits (as line 31 in Algorithm 3).

For a given trace, trace path profiling collects the entry count to a trace as well as trace exit count of each basic block on the trace, i.e., the number of times execution leaves a trace via this basic block. From trace exit counts, one can compute the execution count of each basic block on the trace. Profile-based trace truncation uses a simple heuristic: for a given basic block x on a trace, if the entry count of x on the trace is smaller than a predefined fraction of the entry count of the trace, we truncate the trace at x. In our implementation, we use a truncation threshold of 5%.

5. Evaluation

5.1 Our Trace JIT System Overview

Figure 4 shows a component view of our trace JIT, which is built on top of IBM J9 JVM and JIT compiler [11]. Traces are formed out of Java bytecodes and compiled by the J9 JIT, which is extended to compile traces. Our trace JIT supports both trace execution and interpretation, as well as all major functionality of the method JIT. Compilation is done by a dedicated thread, similar to the method JIT.

[4] In our implementation, we check if the remaining portion of the trace includes codes from the same method but outside the loop body or whether the owning method of the loop header has returned. Both indicate that loop L has been exited.

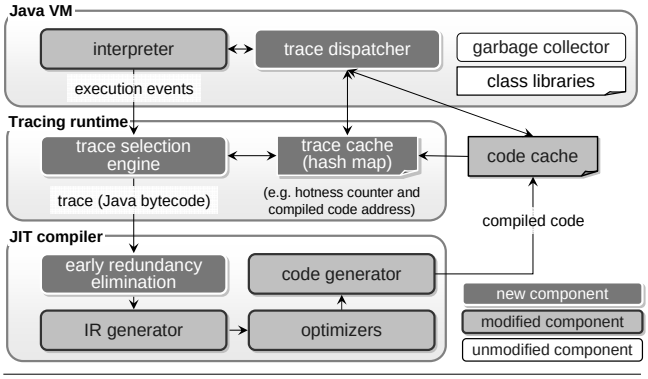

Figure 4. Overview of our trace JIT architecture

Trace head threshold	500 (BB exec freq)
Trace buffer length	128 (BBs)
Structure trunc. (rejoined)	1st loop-header
Structure trunc. (max-length)	1st loop-header or last trace-head
Trace profiling window	128 (trace entry freq)
Profile trunc. threshold	5%

Table 2. Trace selection algorithm parameters.

The trace compiler enables a subset of "warm" level optimizations of the baseline method JIT such as various (partial) redundancy elimination optimizations, (global) register allocation, and value propagation. Our system is aggressively optimized to reduce runtime overhead due to trace execution and trace monitoring (including trace linking optimizations). The current implementation also has limitations compared to the method JIT. For example, the trace JIT does not support recompilation. It does not support escape analysis and enables only a subset of loop optimizers in the J9 JIT. Detailed design of the trace JIT is described in [16].

Table 2 summarizes some of the key parameters of the selection algorithm for the evaluation.

5.2 Experiment Setup

Experiments are conducted on a 4-core, 4GHz POWER6 processors with 2 SMT threads per core. The system has 16 GB of system memory and runs AIX 6.1. For the JVM, we use 1 GB for Java heap size with 16MB large pages and the generational garbage collector. We used two benchmarks:

DaCapo 9.12 benchmark [3] running with the default data size. We did not include the `tradesoap` benchmark because the baseline system with the method-JIT sometimes failed for this benchmark.

DayTrader 2.0 [19] running on IBM WebSphere Application Server version 7.0.0.13 [15]. This is an example of large-scale Java applications. For DayTrader, the DB2 database server and the client emulator ran on separate machines.

In this paper, we use the following metrics to evaluate our techniques. For each result, we report the average of 16 runs along with the 95% confidence interval.

Selection footprint: the total number of bytecodes in compiled traces.

Compiled binary code size: the total binary code size.

Steady-state execution time: For DaCapo 9.12, we executed 10 iterations for `eclipse` and 25 iterations for the other benchmarks, and reported the average execution time of the last 5 iterations. For DayTrader, we ran the application for 420 seconds that includes 180-second client ramp-up but excludes setup and initialization, and used the average execution time per request during the last 60 seconds.

Start-up time: the execution time of the first iteration for DaCapo 9.12, and the time spent before the WebSphere Application Server becomes ready to serve for DayTrader.

Compilation time: the total compilation time.

5.3 Reduction in Selection Footprint

We evaluated the six techniques proposed in this paper as summarized in Table 3. First, we measured the impact of each individual technique on selection footprint. Figure 5 shows the normalized selection footprint when we apply each technique to the baseline.

We observe that each technique is effective in reducing selection footprint, with the average reduction ranging from 12% (*exact-bb*) to 40% (*head-opt*). The only exception is when applying *head-opt* to `jython`, where selection footprint increases by 2%.

Second, we measured the combined effects of the techniques in reducing selection footprint, as shown in Figure 6. In this and following figures, the techniques are combined according to the natural order (left to right) by which they are applied during the selection. For example, the bar *+struct-trunc* stands for the case where we apply *exact-bb*, *head-opt*, and *struct-trunc* to the baseline.

With all techniques applied, the average selection footprint is reduced to 30% of the baseline's. We also observe that each technique is able to further reduce selection footprint over the ones applied before it.

Figure 8 shows a detailed breakdown on where the reduction in selection footprint comes from.

- The bottom bar *our algo w/ all-opt* is the selection footprint of our algorithm relative to the baseline's.
- *Short-lived traces eliminated* represent the total bytecode size of short-lived traces eliminated by our optimizations.
- *Structure truncated BCs* and *profile truncated BCs* account for bytecodes eliminated by structure-based and profile-based truncation, respectively.

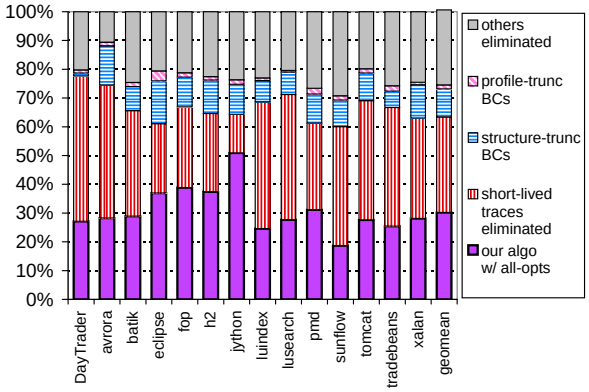

Figure 8. Breakdown of selection footprint reduction normalized to that of the baseline.

- *Others* represent the rest of the reduction, which is likely due to improved convergence of the baseline algorithm that generates fewer new traces.

5.4 Impact on System Performance

Figure 7 shows the combined impact of our techniques on compiled binary code size. With all our techniques combined, the compiled code size is reduced to 30% of the baseline's, which is consistent with the degree of reduction on selection footprint.

Figure 9 shows the combined impact of our techniques on steady-state execution time. The steady-state performance was unchanged on average after all techniques are applied, with a maximal degradation of 4% for `luindex`.

It is also notable that the steady-state performance of `DayTrader` is improved by 10%. This is because L2 cache misses were reduced and thus clock per instruction was improved, due to reduced code size. This shows that code size control is not only important for memory reduction itself but also important for the steady-state performance in large-scale applications.

Figure 10 and Figure 11 show the normalized start-up time and compilation time when the techniques are applied in combination, respectively. Using our techniques, compilation time and start-up time was reduced to 32% and 57% of the baseline's, respectively.

5.5 Discussions

Our results show that the reduction in selection footprint is linear to that of compilation time and binary code size. Start-up time is closely related to but not linear to selection footprint because it is influenced by other factors such as the ratio of interpretation overhead to native execution and how fast bytecodes are captured as traces and compiled to binaries. Only very large-scale applications, such as `DayTrader` and `eclipse`, experience an improvement in

Name	Description	Described in	Main effect
exact-bb	exact basic block construction	Section 3.2.1	Reduced short-lived traces & duplication
head-opt	counter/trace lookup at backward branch and exit-heads	Section 3.2.2	Reduced short-lived traces
struct-trunc	structure-based truncation	Section 4.2.1	Reduced short-lived traces & duplication
clear-counter	clearing counters of potential trace heads on a recorded trace	Section 3.2.3	Reduced short-lived traces
profile	trace profiling	Section 3.2.4	Reduced short-lived traces
prof-trunc	trace profiling with profile-based truncation	Section 4.2.2	Reduced duplication

Table 3. Summary of evaluated techniques

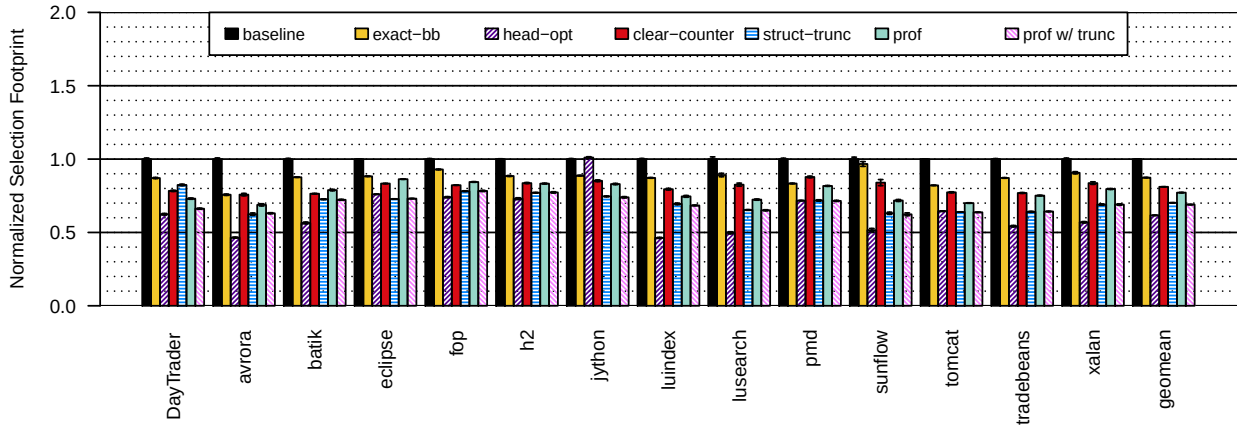

Figure 5. Selection footprint (normalized to the baseline) when applying each technique (shorter is better).

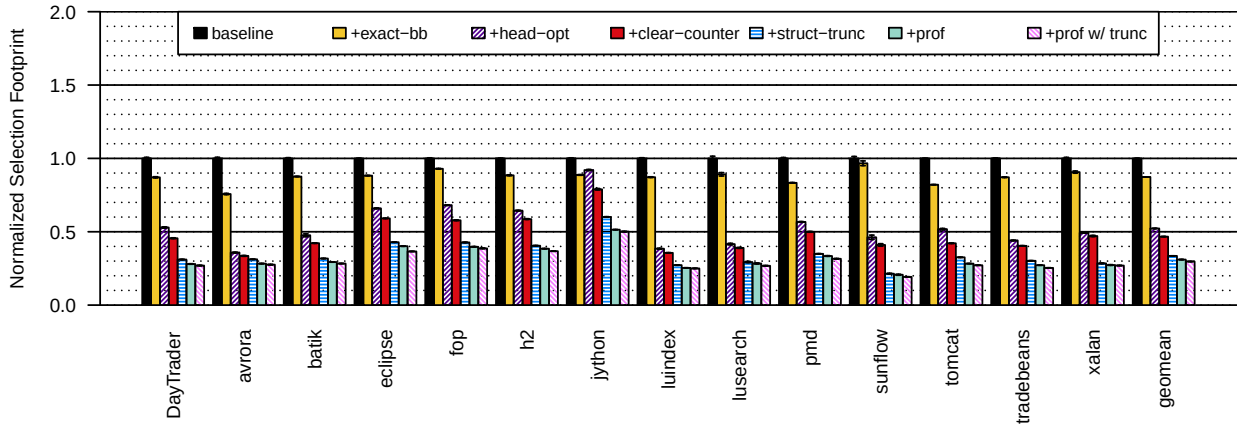

Figure 6. Selection footprint (normalized to the baseline) after combining techniques over the baseline (shorter is better).

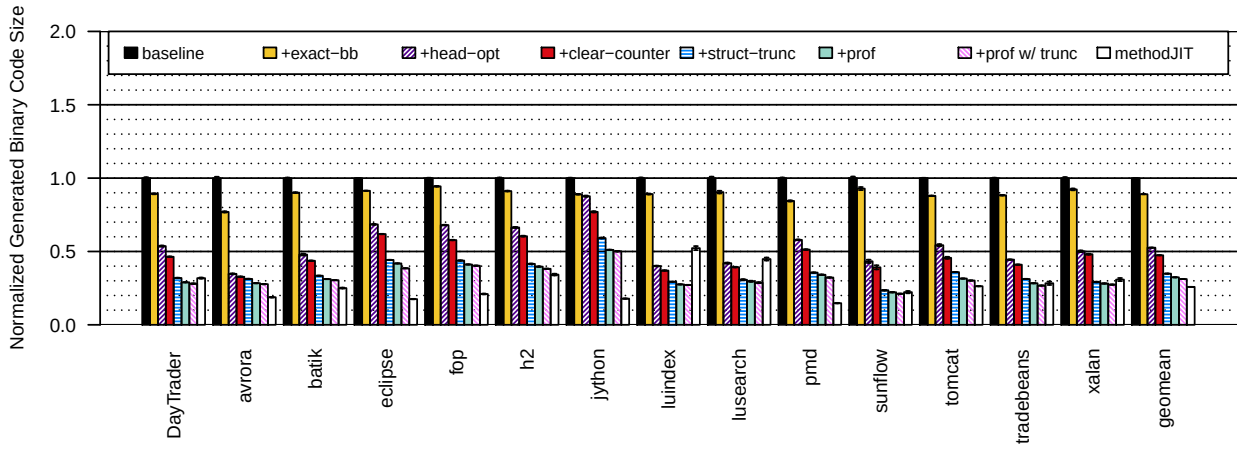

Figure 7. Binary code size (normalized to the baseline) after combining techniques over the baseline (shorter is better).

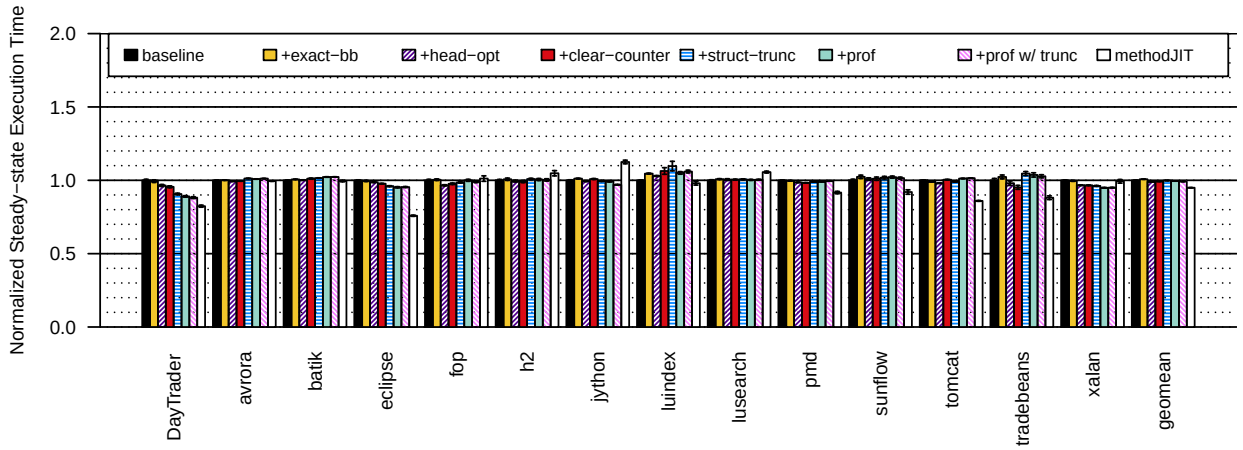

Figure 9. Steady-state execution time (normalized to the baseline) after combining techniques over the baseline (shorter is better).

steady-state performance as the result of selection footprint reduction.

Eliminating short-lived traces have the biggest impact in footprint reduction. Of all the techniques that eliminate short-lived traces, ensuring proper ordering by which to select trace heads (*head-opt*) addresses the root cause of short lived traces.

We would also like to point out that some of the proposed techniques may potentially degrade start-up performance because they either reduce the scope of individual traces (e.g., truncation) or prolongs the time before a bytecode is executed from a binary trace (e.g., profiling). But our results show empirically that footprint reduction in general improves start-up performance because, for large-scale workloads, compilation speed is likely a more critical bottleneck to the start-up time than other factors. However, the cost-benefit effects may change depending on the compilation resource of the trace compiler, the coverage requirement of the selection algorithm, and the characteristics of the workload.

6. Comparing with Other Selection Algorithms

While our techniques are described in the context of the baseline algorithm, many design choices are also common in other trace selection algorithms. Table 4 summarizes important aspects of trace selection discussed in the paper for all existing trace selection algorithms.

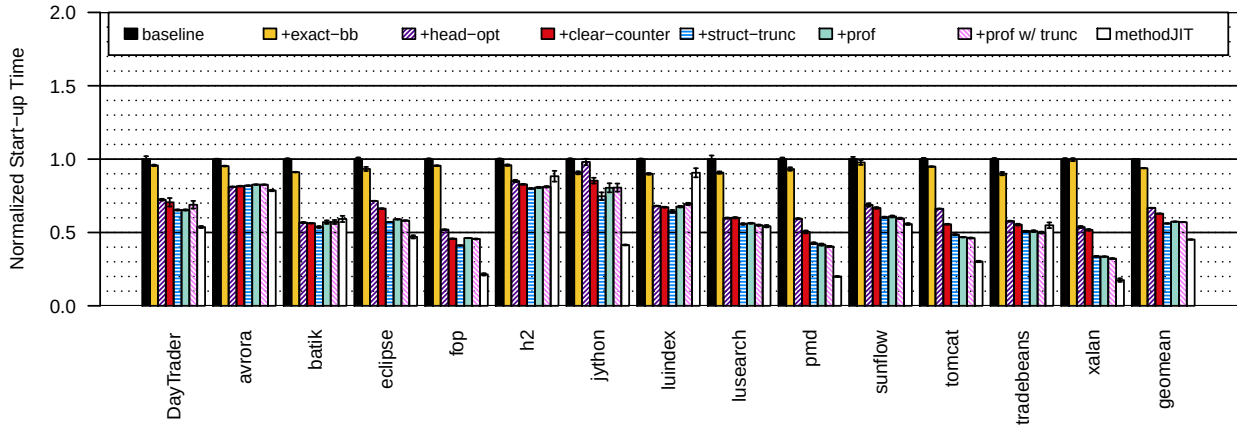

Figure 10. Start-up time (normalized to the baseline) after combining techniques over the baseline (shorter is better).

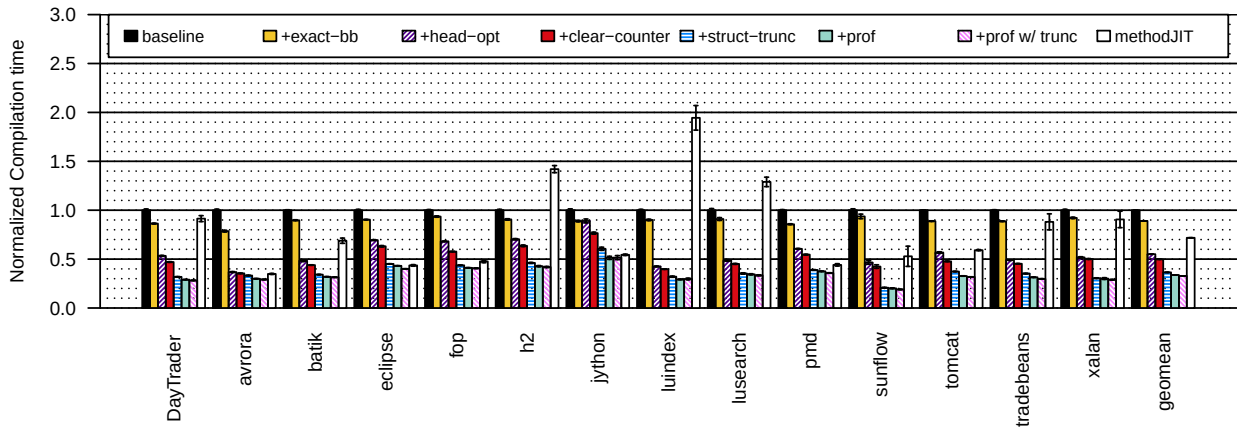

Figure 11. Compilation time (normalized to the baseline) after combining techniques over the baseline (shorter is better).

6.1 One-pass selection algorithms

We first compare our techniques with size control heuristics used in existing one-pass selection algorithms, all of which are based on termination conditions.

Stop-at-existing-head terminates a trace recording when the next instruction to be recorded is the head of an existing trace. This heuristic was first introduced by NET [1]. It is most effective size-control heuristic because it minimizes duplication across traces. It does not generate short-lived traces either because Condition 2 of short-lived trace formation is no longer satisfied. However, *stop-at-existing-head* can have significant impact of performance due to reduced trace scope.

Figure 12 shows the relative execution time and code size of *stop-at-existing-head* normalized to that of our algorithm. It shows that *stop-at-existing-head* excels at space efficiency, but degrades the performance by up to 2.8 times.

A main source of the degradation comes from the reduction in the inlining effect of the trace selection. In particular, once a trace is formed at the entry point of a method, *stop-at-existing-head* prevents the method to be *"inlined"* into any later traces that invoke the method.

Stop-at-loop-boundary terminates a trace recording at loop boundaries, such as loop headers or loop exit-edges. Variations of this heuristic are used in PyPy, TraceMonkey, HotpathVM, SPUR, and YETI.

We compared *stop-at-loop-head* heuristic, which is one type of stop-at-loop-boundary and terminates a trace at loop headers (figure not included in the paper). On the

System		our baseline [13, 16]	SPUR [2]	HotpathVM [10] and [9]	LuaJIT [17]	TraceMonkey [8]	PyPy [5]	YETI [21]	NET [1]	Merrill+ [18]	LEI [14]
Target Program		Java	CIL	Java	Lua	JavaScript	Python	Java	Binary	Java	Binary
Multi-pass or one-pass selection		one	multi	multi	N/A	multi	one	one	one	one	one
Trace Head	loop head	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	exit head	Y	Y	Y	Y	Y	N/A	Y	Y	Y	Y
Condition	method entry		Y							Y	
Trace Termination Condition	stop-at-any-backward-branch							Y	Y		
	stop-at-repeating-pc	Y					Y			Y	Y
	Stop-at-loop-back-to-head		Y	Y	Y	Y					
	stop-at-existing-trace-heads		Y		N/A	call	N/A		Y	Y	Y
	stop-when-leaving-static-scope		loop	method	loop	loop	N/A	N/A		method	
	stop-when-exceeding-max-size	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	stop-at-native-method-call			Y	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Imprecise BB problem		Y			N/A			Y	Y		Y

Table 4. Comparison of Trace Selection Algorithms

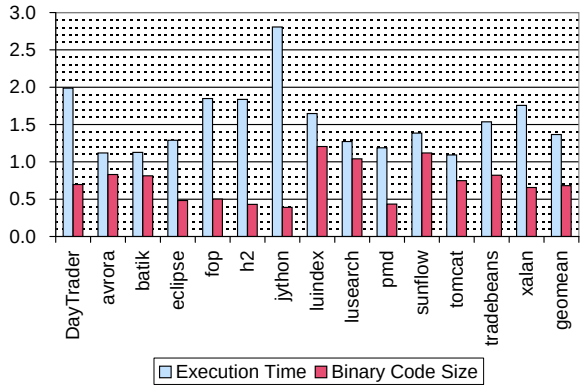

Figure 12. Execution time and code size of *stop-at-exitsing-head* relative to our algorithm.

DaCapo 9.12 benchmark and DayTrader, *stop-at-loop-head* is 3% slower than ours in the steady-state performance, and its binary code size is 2.45 times of ours.

Stop-at-return-from-method-of-trace-head terminates a recording when the execution leaves the stack frame of the trace-head. This heuristic is used in PyPy, HotpathVM and Merrill et al.

This heuristic (figure not included in the paper) is on average 6% slower in steady-state performance compared to ours. The binary code size is 1.6 times of ours, ranging from 1.22 times (`h2`) to 2.4 times (`sunflow`).

6.2 Multi-pass selection algorithms

Multi-pass trace selection algorithms form traces after multiple recordings and combine individual traces into groups. Traces formed by such algorithms can allow split paths or inner-join within a trace (group). While direct comparison with multi-pass selection algorithms is beyond the scope of this work, multi-pass selection algorithms conceptually have more compact footprint than one-pass selection algorithms because paths can be joint in a trace group.

On the other hand, existing multi-pass selection algorithms are designed primarily with loops in mind. SPUR, HotpathVM, and TraceMonkey [2, 8, 10] are three such systems, all of which build trees of traces anchored at loop headers. For non-loop intensive workloads, some computation may happen outside any loops, some may occur in loops whose bodies are too large to be captured into one trace tree. It is an open question whether existing multi-pass selection algorithms can achieve high coverage on large-scale non loop-intensive applications.

7. Conclusion

Designing a balanced trace selection algorithm largely boils down to meeting the competing needs of creating larger traces to maximize performance and reducing the selection footprint to minimize resource consumption. This paper focuses on the latter problem of controlling the footprint of trace selection.

In this work, we discovered some of the most intriguing aspects of trace selection algorithms. Our first insight comes from the observation of trace "churning", where a significant amount of traces, shortly after being created, are no longer executed. A careful study of the baseline algorithm reveals pitfalls of several common-sense trace selection design choices that could lead to pathological formation of short-lived traces.

Our second insight comes from studying the cause of excessive duplication in traces that are not necessarily short-

lived. While trace duplication has been studied before in the context of tail duplication, we identified new sources of unnecessary duplication due to poor convergence property of a trace selection algorithm.

By addressing these sources of footprint inefficiency in common trace selection algorithms, our techniques are able to reduce the selection footprint and the binary code size to one third of the baseline and the startup time to slightly over half of the baseline with no performance loss. In one large-scale enterprise workload based on a production web server, our techniques improve the steady-state performance by 10%, due to improved instruction cache performance.

References

[1] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A Transparent Runtime Optimization System. In Proceedings of Conference on Programming Language Design and Implementation (PLDI) (June 2000).

[2] BEBENITA, M., BRANDNER, F., FAHNDRICH, M., LOGOZZO, F., SCHULTE, W., TILLMANN, N., AND VENTER, H. SPUR: a trace-based JIT compiler for CIL. In Proceedings of International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) (2010), pp. 708–725.

[3] BLACKBURN, GARNER, HOFFMANN, KHANG, MCKINLEY, BENTZUR, DIWAN, FEINBERG, FRAMPTON, GUYER, AND HOSKING. The DaCapo benchmarks: java benchmarking development and analysis. In Proceedings of Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) (Oct. 2006).

[4] BOLZ, C., CUNI, A., FIJALKOWSKI, M., LEUSCHEL, M., PEDRONI, S., AND RIGO, A. Allocation removal by partial evaluation in a tracing JIT. In Proceedings of Workshop on Partial Evaluation and Program Manipulation (2011).

[5] BOLZ, C., CUNI, A., FIJALKOWSKI, M., AND RIGO, A. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (2009).

[6] BRUENING, D., AND AMARASINGHE, S. Maintaining Consistency and Bounding Capacity of Software Code Caches. In Proceedings of International Symposium on Code Generation and Optimization (CGO) (Mar. 2005).

[7] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An Infrastructure for Adaptive Dynamic Compilation. In Proceedings of International Symposium on Code Generation and Optimization (CGO) (Mar. 2003).

[8] GAL, A., EICH, B., SHAVER, M., ANDERSON, D., MANDELIN, D., HAGHIGHAT, M. R., KAPLAN, B., HOARE, G., ZBARSKY, B., ORENDORFF, J., RUDERMAN, J., SMITH, E. W., REITMAIER, R., BEBENITA, M., CHANG, M., AND FRANZ, M. Trace-based just-in-time type specialization for dynamic languages. In Proceedings of Conference on Programming Language Design and Implementation (PLDI) (2009), pp. 465–478.

[9] GAL, A., AND FRANZ, M. Incremental dynamic code generation with trace trees. Tech. rep., University of California Irvine, November 2006.

[10] GAL, A., PROBST, C., AND FRANZ, M. HotPathVM: An Effective JIT Compiler for Resource-constrained Devices. In Proceedings of International Conference on Virtual Execution Environments (VEE) (June 2006).

[11] GRCEVSKI, N., KIELSTRA, A., STOODLEY, K., STOODLEY, M., AND SUNDARESAN, V. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In Proceedings of International Conference on Virtual Execution Environments (VEE) (June 2004).

[12] GUO, S., AND PALSBERG, J. The essence of compiling with traces. In Proceedings of International Symposium on Principles of Programming Languages (POPL) (2011), pp. 563–574.

[13] HAYASHIZAKI, H., WU, P., INOUE, H., SERRANO, M. J., AND NAKATANI, T. Improving the performance of trace-based systems by false loop filtering. In Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (March 2011).

[14] HINIKER, D., HAZELWOOD, K., AND SMITH, M. D. Improving region selection in dynamic optimization systems. In Proceedings of 38th International Symposium on Microarchitecture (MICRO) (Dec. 2005).

[15] IBM CORPORATION. WebSphere Application Server. `http://www-01.ibm.com/software/webservers/appserv/was/`.

[16] INOUE, H., HAYASHIZAKI, H., WU, P., AND NAKATANI, T. A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler. In Proceedings of International Symposium on Code Generation and Optimization (CGO) (April 2011).

[17] LuaJIT design notes in lua-l mailing list. `http://lua-users.org/lists/lua-l/2008-02/msg00051.html`.

[18] MERRILL, D., AND HAZELWOOD, K. Trace fragment selection within method-based jvms. In Proceedings of International Conference on Virtual Execution Environments (VEE) (June 2008).

[19] THE APACHE SOFTWARE FOUNDATION. DayTrader. `http://cwiki.apache.org/GMOxDOC20/daytrader.html`.

[20] WHALEY, J. Partial Method Compilation using Dynamic Profile Information. In Proceeding of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) (Oct. 2001), pp. 166–179.

[21] ZALESKI, M., DEMKE-BROWN, A., AND STOODLEY, K. YETI: a graduallY Extensible Trace Interpreter. In Proceedings of International Conference on Virtual Execution Environments (VEE) (2007), pp. 83–93.

[22] ZHAO, C., WU, Y., STEFFAN, J., AND AMZA, C. Lengthening Traces to Improve Opportunities for Dynamic optimization. In 12th Workshop on Interaction between Compilers and Computer Architectures (Feb 2008).