Improving the Performance of Trace-based Systems by False Loop Filtering

Hiroshige Hayashizaki
IBM Research - Tokyo
1623-14 Shimotsuruma, Yamato,
Kanagawa, Japan 242-8502
hayashiz@jp.ibm.com

Peng Wu
IBM Research - Watson Research Center
P.O. Box 218, Yorktown Heights,
NY, USA 10598
pengwu@us.ibm.com

Hiroshi Inoue
IBM Research - Tokyo
1623-14 Shimotsuruma, Yamato,
Kanagawa, Japan 242-8502
inouehrs@jp.ibm.com

Mauricio J. Serrano
IBM Research - Watson Research Center
P.O. Box 218, Yorktown Heights,
NY, USA 10598
mserrano@us.ibm.com

Toshio Nakatani
IBM Research - Tokyo
1623-14 Shimotsuruma, Yamato,
Kanagawa, Japan 242-8502
nakatani@jp.ibm.com

Abstract

Trace-based compilation is a promising technique for language compilers and binary translators. It offers the potential to expand the compilation scopes that have traditionally been limited by method boundaries.

Detecting *repeating cyclic execution paths* and capturing the detected repetitions into traces is a key requirement for trace selection algorithms to achieve good optimization and performance with small amounts of code. One important class of repetition detection is *cyclic-path-based* repetition detection, where a cyclic execution path (a path that starts and ends at the same instruction address) is detected as a repeating cyclic execution path.

However, we found many cyclic paths that are not *repeating* cyclic execution paths, which we call *false loops*. A common class of false loops occurs when a method is invoked from multiple call-sites. A cycle is formed between two invocations of the method from different call-sites, but which does not represent loops or recursion. False loops can result in shorter traces and smaller compilation scopes, and degrade the performance.

We propose *false loop filtering*, an approach to reject false loops in the repetition detection step of trace selection, and a technique called *false loop filtering by call-stack-comparison*, which rejects a cyclic path as a false loop if the call stacks at the beginning and the end of the cycle are different.

We applied false loop filtering to our trace-based Java^{TM}JIT compiler that is based on IBM's J9 JVM. We found that false loop filtering achieved an average improvement of 16% and 10% for the DaCapo benchmark when applied to two baseline trace selection algorithms, respectively, with up to 37% improvement for individual benchmarks. In the end, with false loop filtering, our trace-based JIT achieves a performance comparable to that of the method-based J9 JVM/JIT using the corresponding optimization level.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0266-1/11/03. . . $10.00

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

General Terms Algorithms, Performance

Keywords Trace-based Compilation, Trace Selection, Repetition Detection

1. Introduction

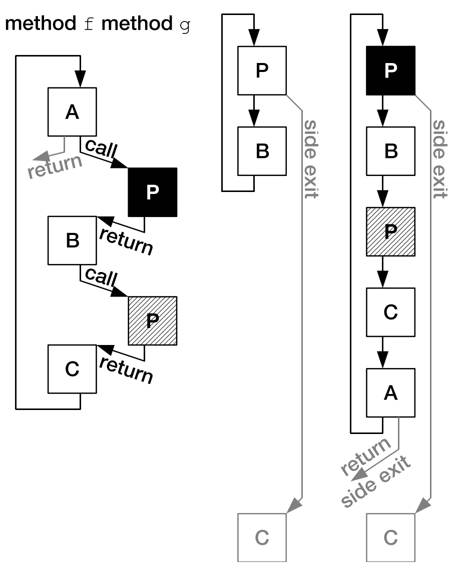
Trace-based compilation has been explored in binary translation systems [1, 7, 8], lightweight JIT for embedded JVMs [12, 20], and more recently in compilers for dynamically typed languages such as JavaScript [4, 13], Lua [17], and Python [6]. In a trace-based compiler, a *trace*, typically defined as a single-entry multiple-exit region formed from executed instructions, is the basic unit for compilation. This is the most distinctive aspect of a trace-based compiler. In this paper, we start with *trace selection*, the process that forms traces out of hot execution paths, and focus on improving the steady-state performance of trace-based compilation systems by improving the trace selection algorithms.

All trace selection algorithms aim to detect *repeating cyclic execution paths* (or *repetitions*) from runtime execution information (*repetition detection*), and capture the detected repetitions into traces, typically by terminating traces when a repetition is detected. By doing so, a trace can capture a large amount of computation in a compact form, which leads to good space efficiency and good optimization.

Repetition detection algorithms used in existing trace selection algorithms fall into the following three categories:

- *Stop-at-backward-branch* repetition detectors [1] consider every backward branch as an indicator of a repeating cyclic execution path, and terminate a trace at every backward branch.
- *Cyclic-path-based* repetition detectors [6, 15, 18] consider an execution sequence as a repeating cyclic execution path when the sequence is *cyclic*, that is, the sequence starts and ends at the same program counter (PC).

```
 1: void f(){
 2:  for(A){
 3:   g();
 4:   B;
 5:   g();
 6:   C;
 7:  }
 8: }
 9: void g(){
10:  P;
11: }
```



(a) Program 1 (b) CFG of Program 1 (c) Trace 1A (without false loop filtering) (d) Trace 1B (with false loop filtering)

Figure 1. False loop and false loop filtering

- Trace selection algorithms with *static-scope-based* repetition detectors [4, 11–13, 17] leverage the information about the program structures such as loops and methods, and construct a trace starting from a loop header and spanning the loop body to capture repeating cyclic execution paths.

While static-scope-based repetition detectors have worked well in recent trace compilers, in this paper, we focus on cyclic-path-based repetition detectors, because they can achieve high performance and are applicable to a wide variety of trace-based systems including binary tracing that static-scope-based repetition detectors may not be suitable for.

The False Loop Problem

While cyclic-path-based repetition detectors detect cyclic execution paths and capture such paths into traces, they do not always detect *repeating* cyclic execution paths. Consider the example in Figure 1. Suppose a trace starts from instruction `P` of method `g` that is called by method `f` at Line 3. The execution goes as follows:

1. execute `P`
2. return to method `f`
3. execute `B`
4. invoke method `g` (at Line 5)
5. execute `P`

At Step 5, a cyclic execution path `P-B-P` (from Step 1 to Step 5) is detected, and `P-B-P` is formed into a cyclic trace Trace 1A as shown in Figure 1 (c). The cyclic execution path `P-B-P`, however, does not capture a real repetitive execution pattern in this example. Consider mapping the execution sequence `P-B-P-C-A-P` into trace `P-B-P`. The cycle in the trace is exercised exactly once and the trace is exited when `C` is executed.

A better trace formation starting from `P` (Trace 1B) is shown in Figure 1 (d). Trace 1B spans one iteration of a natural loop starting from `A`, where the instruction sequence `P-B-P-C-A-P` can be repeated many times in consecutive executions. Not only is Trace 1B a much better target for optimization, it also incurs much lower overhead because trace exit occurs only when the execution finishes iterating the loop.

Intuitively, `P-B-P` does not represent a real repetitive execution pattern because the cyclic path does not come from a natural loop or a recursion, but rather is the effect of multiple invocations of the same method on one execution path. The rest of the paper refers to a cyclic execution path that does not represent a real repetitive execution pattern as a *false loop*.

Another problem with false loops is that they might prematurely terminate a trace that could have otherwise captured real repetitive execution patterns. This occurs, for example, when a trace selection algorithm terminates a trace when it finds a cyclic path in the middle of a trace, as in the *Rejoined* case discussed in [18]. Consider again the example in Figure 1, and suppose the trace is recorded from `A` instead. The trace `A-P-B-P` might be created, because the sequence `P-B-P` inside the path `A-P-B-P` is identified as a cyclic path, whereas `A-P-B-P-C-A` is a better structure.

False Loop Filtering

In this paper, we propose *false loop filtering* to reject false loops in repetition detectors and to improve the quality of the traces.

Various heuristics can be used to detect false loops. We propose a false loop filtering algorithm, named *call-stack-comparison*. For a cyclic execution path, it rejects the cyclic path as a false loop if the call stacks at the beginning and the end of the cyclic path are different. This rejects the trace shown in Figure 1 (c), which is a major class of false loops. This is one of the best performing algorithms among those being tested, and the most direct and exact solution to the false loop problem.

Using call stacks to detect false loops is based on the following insights. Although there is only one instruction address for `P` in Figure 1, there are two instances of `P` on the trace: `P` called from Line 3 (shown in a black box) and `P` called from Line 5 (shown in a slashed box). When the call stacks of two dynamic instructions are different, the execution paths following the two dynamic instructions would diverge after some method returns (that lead to the differing callers on the call stacks). Therefore it is a strong indicator that forming a trace out of such a cycle is unlikely to capture a long repetitive execution sequence. Distinguishing the two instances of `P` from different calling contexts results in the superior trace shown in Figure 1 (d).

We also show an approximation algorithm that achieves almost the same performance (at risk of a small number of false loops), and algorithms that can be applied to systems where less information on executed instructions is available such as binary tracing systems.

We evaluated false loop filtering on a trace-based just-in-time compiler for Java (*trace JIT*) that is built on top of IBM's J9 JVM [14]. The trace JIT supports the basic functions of the original J9 JVM including GC, threading, exceptions, monitor events, and JNI, as well as basic optimizations from the original method-based JIT in J9. Details of the trace JIT are described in [16].

On the DaCapo benchmark [5, 9], we demonstrated that false loop filtering achieved an average speedup factor of 1.16x and 1.10x, respectively, when applied to our two linear trace selection algorithms, with a reasonable increase of code size and compilation time. With our best performing trace selection configuration with false loop filtering, the trace JIT achieved a performance comparable to the method JIT at the corresponding optimization levels.

Contribution and Organization

The paper makes the following contributions:

1. We describe the false loop problem in existing trace selection algorithms and propose the concept of false loop filtering. To

the best of our knowledge, this paper is the first one that explicitly addresses the false loop problem in trace selection.

2. We propose a false loop filtering algorithm which we named *call-stack-comparison*, and showed the impact of false loops and false loop filtering in a real trace-based compiler.

The rest of the paper is organized as follows. Section 2 gives an overview of trace-based systems and repetition detection. We propose false loop filtering and describe false loop filtering algorithms in Section 3. Section 4 describes our implementation of the trace-based system for Java. Evaluations of false loop filtering and the trace JIT are in Section 5. Section 6 discusses the related work and we conclude in Section 7.

2. Background

2.1 Trace-based Compilation

In a trace-based system, the basic unit for compilation and execution is a trace as opposed to the method of traditional method-based compilers. At the heart of any trace-based system lies *trace selection*, the mechanisms that form traces out of executed instructions.

The next-executing-tail (NET) selection used in Dynamo [2] pioneered many early concepts of trace selection that are still in use today. Dynamo interprets the instruction stream until a taken branch is encountered. The execution transfers to compiled code if the branch target corresponds to the start of a compiled trace. Otherwise, a counter is maintained for the target of each backward branch and trace exiting branch, and the counter of the branch target is incremented every time the branch is executed. When the counter value of a branch target exceeds a threshold, a trace is recorded from the branch target until one of these conditions is encountered: (a) a backward branch is taken, (b) a branch whose target address is the head of another trace is found, or (c) a maximum trace length is exceeded.

There have been many advances in trace selection algorithms since Dynamo. In general, a trace selection algorithm involves these three steps.

Trace-head selection identifies a set of program counter (PC) addresses that are the likely starting points of hot regions (*potential* trace-heads). A potential trace-head can be the target of a backward branch or a trace exiting branch (*exit-head*). The system monitors the execution frequency of each potential trace-head. A potential trace-head becomes a trace-head if its execution frequency reaches a predefined threshold, and trace recording is started.

Trace recording forms a trace by recording every instruction[1] that follows the execution of a newly selected trace-head until a trace termination condition is met. The recorded instructions are stored in the *trace recording buffer*, and everytime an instruction is executed, the instruction is put into the buffer and trace termination conditions are checked.

Trace termination conditions generally fall into the four categories:

1. when a repeating cyclic execution sequence is detected in the trace recording buffer
2. when the recorded trace exceeds the maximum length
3. when there is an unusual or unsupported instruction sequence such as exception handling
4. when other heuristics apply, such as a rule terminating a trace at the head of another trace (*stop-at-existing-head*)

Trace grouping combines multiple individually recorded traces into one trace (group) for compilation [4, 11, 12, 17]. Trace grouping requires multiple recording and sometimes recompilation. Grouped traces often have a complex topology that allows split paths, such as trees or general graph. Some trace selection algorithms do not involve trace grouping.

Once a trace or a trace group is formed and compiled, the generated code is stored in the *trace cache*. A compiled trace is dispatched either from the interpreter or at the exit point of another trace. Trace dispatch often involves looking up a trace based on the original PC of the next instruction. This overhead is unique to trace-based systems, but can be reduced by several techniques such as trace linking [2] for trace exits with very few targets. When the execution exits from a trace, there is a *trace transition*. For example, the execution may return to the interpreter or jump directly to the next compiled trace.

2.2 Repetition Detection in Trace Selection

Every trace selection algorithm uses heuristics to detect repeating cyclic execution paths (or *repetitions*) during trace recording. We refer to such mechanisms as *repetition detection*. Repetition detection is a critical element of trace selection for these reasons.

Time and space efficiency. Since traces are formed from runtime executions, capturing repeating cyclic execution paths into a compact form is key to the time and space efficiency of a trace selection algorithm.

Trace quality. Repeating cyclic execution paths typically represent a large amount of computation. Capturing such execution patterns into one trace reduces the runtime overhead due to trace exits, and improves code quality by exposing a larger compilation scope to the compiler.

There are three approaches to repetition detection in existing trace selection algorithms: stop-at-backward-branch, cyclic-path-based, and static-scope-based repetition detection.

2.2.1 Stop-at-backward-branch Repetition Detector

The NET selection [2] uses backward branches as an indicator of a repeating cyclic execution pattern, where a trace is terminated at a backward branch (*stop-at-backward-branch*) during trace recording.

Stop-at-backward-branch has good time and space-efficiency, but it tends to terminate traces before repeating cyclic execution paths are detected [15]. In many cases, it does not form traces that are cyclic. This repetition detector has been largely superseded by the cyclic-path-based repetition detector.

2.2.2 Cyclic-path-based Repetition Detector

A *cyclic path* refers to an execution path that starts and ends at the same PC. A *cyclic-path-based repetition detector* uses cyclic paths detected at trace recording as the indicator of a repeating cyclic execution pattern.

A cyclic-path-based repetition detector was first proposed for the last-execution-iteration (LEI) selection method [15] and subsequently used in PyPy [6] and by Merrill et al. [18]. When a PC appears twice in the trace recording buffer, the LEI selection creates a trace corresponding to the path between the repeating PCs.

The main benefit of cyclic-path-based repetition detection is its simplicity and flexibility. It can be applied to a wide variety of trace selection algorithms, tracing systems and target workloads. The main drawback is the accuracy of the repetition detection. This is because not every cyclic path represents a true repeating cyclic

[1] The unit for trace recording and trace selection can be an instruction, a (taken) branch, or a basic block. In this paper, our description uses an instruction as the unit, but can be naturally applied to other units.

execution pattern, as shown in Section 1. When a repetition detector terminates a trace due to a falsely detected repeating pattern, it hurts the selection algorithm's ability to form long traces.

2.2.3 Static-scope-based Repetition Detector

A *static-scope-based* repetition detector leverages the information about the static scopes and program structures (such as loops and methods), and constructs a trace starting from a loop header and spanning the loop body to capture repeating cyclic execution paths. The traces resemble loop structures in the source program. This is based on the observation that loops are the primary sources of repeating cyclic execution paths.

HotpathVM [12] was the first to use a static-scope-based repetition detector. The system forms traces in a tree topology, where the root is a loop header and the branches are different recorded execution paths that start from the loop and end when the execution (1) returns to the loop header, (2) leaves the owning method of the loop header (*stop-when-leaving-method*), or (3) the number of backedges in the tree exceeds the limit. Traces formed in this way resemble the control-flow representation of a loop region that includes only hot execution paths and inlined methods called by the loop. SPUR [4] and TraceMonkey [13] also use trace tree formation but with some variations, such as (4) ending a trace when leaving the loop scope (*stop-when-leaving-loop*) or (5) when the stack depth of the inlined calls exceeds a threshold.

Unlike stop-at-backward-branch or cyclic-path-based repetition detectors, static-scope-based repetition detectors do not suffer from false loop problems. A false loop like the one shown in Figure 1 always contains a method return from the owing method of the starting point of the false loop. Therefore, such false loops are filtered out implicitly by the stop-when-leaving-owning-method or the stop-when-leaving-loop conditions.

A static-scope-based repetition detector typically requires the selection algorithm to form only traces starting from loop headers. For this reason, they are most suitable to forming trace trees, where paths along the side-branches of a loop can be incorporated into a trace tree that is rooted at a loop header. Such trace trees are close to traditional program representations and are typically good for compiler optimizations and implementations.

The main limitation of a static-scope-based repetition detector is its applicability. When a static-scope-based repetition detector is used, the traces always start from a loop header and must fit within the maximum trace size. For workloads that are not particularly loop intensive, trace trees representing loops may only capture a small fraction of the dynamic execution paths. Therefore, they are not applicable to systems forming traces that do not start from non-loop headers or that do not align with static program structures. Static-scope-based repetition detectors also require knowledge of the loops and the methods of the target programs, and thus they are hard to apply to systems where such information is not available, such as binary tracing systems. For these reasons, a static-scope-based repetition detector is not a replacement for a cycle-based repetition detector in many systems.

3. False Loop Filtering

We propose *false loop filtering* to improve the accuracy of cyclic-path-based repetition detectors by rejecting the false loops described in Section 1.

We first formalize a repetition detector and false loop filtering. We define a *repetition detector* as a component that, for a given instruction sequence *insts*$[0, ..., n]$ (typically given by the trace recording buffer) and an integer $0 \leq s < n$, determines whether the instruction subsequence *insts*$[s, ..., n]$ is a repeating cyclic execution sequence. Typically the repetition detector is used in the trace recording step of the trace selection algorithm, and the input sequence for a repetition detector is a trace recording buffer. The trace selection algorithm uses the result of a repetition detector to determine whether or not to terminate a trace. For example, when the repetition detector returns **true** for some s, the recording is terminated and a trace is formed.

The parameter s indicates the starting point of a repeating cyclic execution sequence inside the instruction sequence. We use this formalization to support both a repeating cyclic execution sequence starting from the head of a trace ($s = 0$) and the middle of a trace ($s > 0$). The parameter s is given by the trace selection algorithm that uses a repetition detector, for example by enumerating all possible values of s or by using a fixed value $s = 0$.

A cyclic-path-based repetition detector without false loop filtering checks whether the address of the first and the last instructions of the subsequence are the same, as shown in Algorithm 1. If that is the case, the repetition detector considers the sequence as a repeating cyclic execution path (though it can be a false loop).

Algorithm 1 Cyclic-path-based Repetition Detector without False Loop Filtering

```
Input: insts[0, ..., n]: a sequence of dynamic instructions
Input: s: the candidate for the starting point of a repeating cyclic
    execution sequence ($0 \leq s < n$)
Output: true if the subsequence insts[s, ..., n] is a repeating cyclic
    execution sequence, or false otherwise
 1: if insts[n].address $\neq$ insts[s].address then
 2:     return false // Not a cyclic path
 3: else
 4:     return true // Cyclic path
 5: end if
```

Algorithm 2 describes a repetition detector with false loop filtering. In addition to checking the instruction address at Line 1, it further applies false loop filtering, as shown in Line 5 in Algorithm 2, to reject false loops.

Algorithm 2 *isRepetitionDetected*(*insts*$[s, ...n]$): Cyclic-path-based Repetition Detector with False Loop Filtering

```
Input: insts[0, ..., n]: a sequence of dynamic instructions
Input: s: the candidate for the starting point of a repeating cyclic
    execution sequence ($0 \leq s < n$)
Output: true if the subsequence insts[s, ..., n] is a loop, or false
    otherwise
 1: if insts[n].address $\neq$ insts[s].address then
 2:     return false // Not a cyclic path
 3: else
 4:     apply false loop filtering
 5:     if insts[s, ..., n] is a false loop then
 6:         return false // False loop
 7:     else
 8:         return true // True repeating cyclic execution sequence
 9:     end if
10: end if
```

3.1 Design Considerations for False Loop Filtering

In this section, we cover the basic aspects of false loop filtering. False loop filtering can be used with trace selection algorithms in a wide variety of systems, so different false loop filtering algorithms can be used depending on the system requirements.

First, the false loop filtering algorithm depends on the type of information that is available for each instruction *insts*$[i]$. For example,

- ***insts*$[i]$*.address***: The address of the instruction.

- ***insts***$[i]$***.isBackwardBranch***: Whether or not the instruction is a backward intra-procedural branch. This is available in most systems including binary code, but is unavailable in a few systems.
- ***insts***$[i]$***.isMethodInvoke***: Whether or not the instruction is a method invocation.
- ***insts***$[i]$***.isMethodReturn***: Whether or not the instruction is a method return.
- ***insts***$[i]$***.callStack***: The call stack (that is, the stack of return addresses) at the instruction. This can be obtained via a stack walk, or reconstructed from *insts*$[0, .., n]$*.isMethodInvoke*, *insts*$[0, .., n]$*.isMethodReturn*, and *insts*$[0, .., n]$*.address*, as shown in Section 3.2.2.
- ***insts***$[i]$***.method***: the method to which the instruction belongs.

Second, the false loop filtering algorithm depends on the trace selection algorithm being used. For example, some trace selection algorithms always start traces from loop headers, while others can create traces starting from arbitrary program points.

Third, the precision of false loop filtering algorithms affects the performance and code size.

- **Accepting false loops as true repetitions** has a large negative impact on the performance (though it probably leads to smaller code), because the false loops terminate traces prematurely and are suboptimal for compiler optimizations. In this case, a true repeating cyclic execution path is split into multiple traces, and trace exits occur at every iteration.
- **Missing true repetitions** (that is, considering true repeating cyclic execution paths as a non-loop sequence or a false loop) has a relatively small negative impact in the performance. Even when true repeating cyclic execution paths are missed, trace recording continues and a longer trace may be created in most trace selection algorithms (unless the trace recording is aborted), and thus trace exits are not so frequent compared to accepting false loops. This, however, leads to larger amounts of code.

3.2 False Loop Filtering Algorithms

We propose a false loop filtering algorithm, *call-stack-comparison*, designed for systems where all of the information described in Section 3.1 is available:

#1: Call-stack-comparison considers a cyclic execution sequence as a repeating cyclic execution path if the top k elements of the call stacks at the first and last instructions are the same, in addition to the instruction addresses. The number k is set so that this algorithm can support both natural loops and recursive calls. We describe this in more detail in Section 3.2.1.

We think this is a direct and accurate way to detect false loops such as the one shown in Figure 1. This call-stack-comparison algorithm is also one of the best false loop filtering algorithms among the false loop filtering algorithms we tested. The call-stack-comparison algorithm achieved the best steady-state performance among all of them, and the generated binary code size was the smallest among the algorithms with the best steady-state performance. Therefore, we use this as the default false loop filtering algorithm.

We can design simpler approximation algorithms with similar performance and small number of false loops. Here is one example of such approximation algorithms:

#2: The **one-backward-branch-in-head-method** algorithm considers a cyclic execution path as a repeating cyclic execution path if it contains at least one backward branch in the path and the backward branch belongs to the same method as the first instruction of the path. This is based on the intuition that a repeating cyclic execution path in a loop contains at least one backward branch in the path. Formally, the algorithm considers *insts*$[s, ...n]$ as a repeating cyclic execution path if *insts*$[s]$*.address* = *insts*$[n]$*.address* and, for some t such that $s \leq t < n$, *insts*$[t]$*.isBackwardBranch* is **true** and *insts*$[t]$*.method* = *insts*$[s]$*.method*.

This algorithm requires a separate mechanism to detect recursive method calls, and cannot filter out some false loops. However, this worked well when applied to our trace selection algorithm and the benchmark we used, as shown in the evaluations.

This algorithm does not require call stack information, and thus the implementation can be simpler than the call-stack-comparison algorithm.

These two algorithms require information related to methods and call stacks. In some systems such as binary tracing in some architectures, such information might be not directly available. In such systems, one approach is to obtain the information about methods by approximation or heuristics. For example, when the instruction set architecture has explicit call and/or return instructions such as IA-32, or when we can assume a standard binary application interface (that is, a standard calling convention) and typical code patterns for method invocation and return, then we might be able to infer method calls and returns. This approximation is safe, because the approximation only affects the precision of false loop filtering, and does not affect the correctness of program execution itself.

Another approach is to use false loop filtering algorithms that do not use information about methods. We present two false loop filtering algorithms below that do not require such information. We can use these algorithms, at the cost of performance or code size.

#3: Repeated-twice considers an instruction sequence as a repeating cyclic execution path if the sequence is executed twice consecutively. Formally, the algorithm consider *insts*$[s, ...(n+s)/2]$ as a repeating cyclic execution path if, for all i such that $s \leq i \leq (n+s)/2$, *insts*$[i]$*.address* = *insts*$[i+(n-s)/2]$*.address*.

This algorithm requires only the instruction address information and thus is applicable broadly. It incurs no false loops, because detected repeating paths can be repeated at least twice. However, it tends to create longer traces and results in larger amounts of code, as shown in the evaluations.

#4: End-with-backward-branch considers an instruction sequence as a repeating cyclic execution path if the sequence goes back to the head at a backward branch. In other words, this checks repeating cyclic execution path only at backward branches. Formally, the algorithm considers *insts*$[s, ...n]$ as a repeating cyclic execution path if *insts*$[s]$*.address* = *insts*$[n]$*.address* and *insts*$[n-1]$*.isBackwardBranch* is **true**.

This is intuitive, but we show this also leads to larger code size in the evaluations. It can also accept false loops, and it also misses repeating cyclic execution paths not starting from backward branch targets.

3.2.1 Call-stack-comparison False Loop Filtering

We show the call-stack-comparison false loop filtering algorithm in Algorithm 3. Our algorithm first compares the addresses of the instruction at the end (*insts*$[n]$) and possible start instruction of a repeating cyclic execution path (*insts*$[s]$) at Line 1. If they are the same, the algorithm compares the top k elements of the call stacks of both instructions (Line 10). If the call stacks are the same,

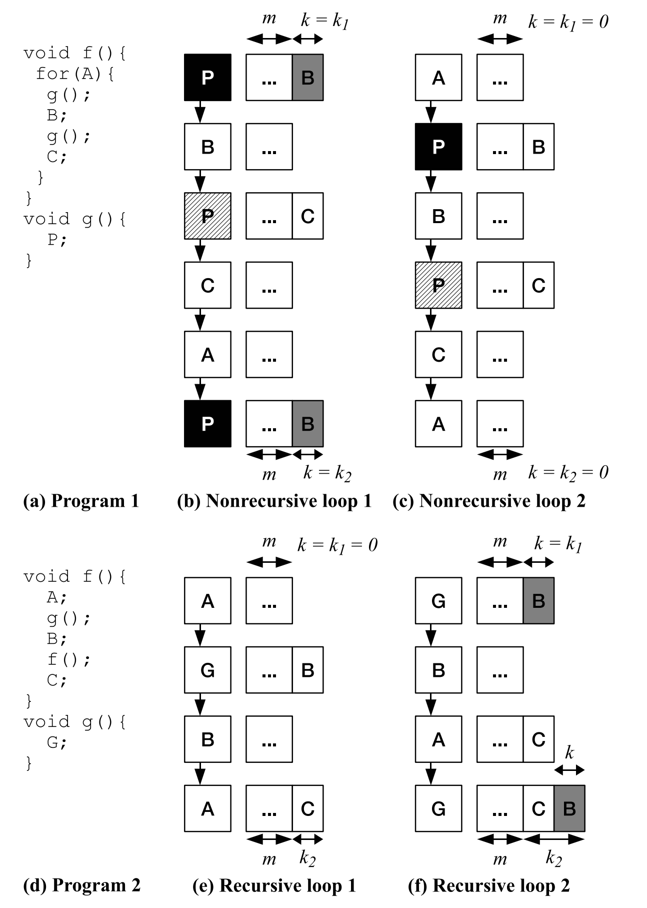


Figure 2. Examples of call stack comparison

our algorithm considers it as a repeating cyclic execution path, or otherwise rejects it as a false loop.

Algorithm 3 Cyclic-path-based Repetition Detector with False Loop Filtering by call-stack-comparison

```
Input: insts[0, ..., n]: a sequence of dynamic instructions
Input: s: the candidate for the starting point of a repeating cyclic
    execution sequence ($0 \leq s < n$)
Output: true if the sequence is a repeating cyclic execution
    sequence, or false otherwise
 1: if $insts[n].address \neq insts[s].address$ then
 2:     return false
 3: else
 4:     // false loop filtering begins
 5:     // $insts[i].callStack$ is the call stack at $insts[i]$.
 6:     $m = \min_{s \leq i \leq n} insts[i].callStack.depth$
 7:     $k_1 = insts[s].callStack.depth - m$
 8:     $k_2 = insts[n].callStack.depth - m$
 9:     $k = \min(k_1, k_2)$
10:     if the top $k$ elements of call stacks $insts[s].callStack$ and
        $insts[n].callStack$ are the same then
11:         return true
12:     else
13:         return false // false loop
14:     end if
15: end if
```

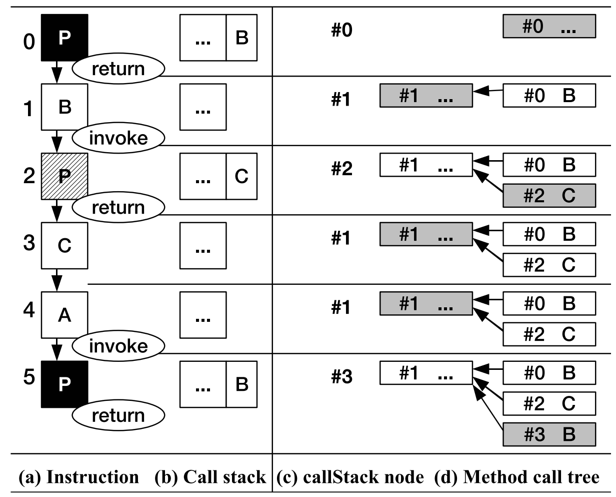


Figure 3. Call stack building

By comparing only the top k elements and calculating k as shown in Lines 6-9, our algorithm can detect repeating cyclic execution paths in non-recursive loops and recursive method calls. For non-recursive loops, our partial call stack comparison is the same as comparing the entire call stacks. For recursive loops, call stack elements added by recursive calls are not compared and thus recursive loops can be detected as true loops.

Figure 2 shows examples of call stack comparisons, with the values of k, k_1, k_2 and m. Examples of non-recursive loops are shown in Figures 2 (a), (b), and (c) (for the same program as in Figure 1), and examples of recursive calls are shown in Figures 2 (d), (e), and (f). Figures 2 (b), (c), (e), and (f) show execution paths on the left and the call stacks for each instruction on the right (the rightmost element is the top element in the stack).

The top k call stack elements being compared are shown in grey boxes. Call stack elements added due to recursive calls (for example, the `C`s in Figures 2 (e) and (f)) are not compared.

In fact the call stacks are longer in actual execution (there are call stack elements corresponding to the callers of method `f`), but only the relevant parts are shown in Figure 2, and the irrelevant parts are shown as boxes with "`...`". The irrelevant parts are always excluded from the top k elements compared.

Each element of a call stack contains a return address. For example, the top element `B` of the call stack "`...-B`" in the first row of Figure 2 (b) means that control will return to `B` after the next method return.

3.2.2 Incremental Call Stack Generation

Call stack information (*insts*$[i]$.*callStack*) can be obtained by doing a stack walk when recording each instruction *insts*$[i]$. It can also be incrementally reconstructed using these two types of information for each instruction *insts*$[i]$.

1. the instruction address (*insts*$[i]$.*address*) and
2. the instruction type, that is, whether the instruction is a method invocation, method return, or other (*insts*$[i]$.*isMethodInvoke* or *insts*$[i]$.*isMethodReturn*).

We do not need to start our call stack generation from a known stack state, but rather can infer it as we encounter new instructions.

Figure 3 (a) shows the instruction sequence `P-B-P-C-A-P` for the sample code shown in Figure 1, starting with the first instruction in Row 0. Figure 3 (b) shows call stacks for each instruction that we are going to reconstruct.

We do not initially know the call stack state at the first instruction `P`. We will later find it is "`...-B`", because the control returns to `B` at the return instruction from `P` to `B`. This method return implies that the caller of P was actually B, so we can update the call stack information. There is still no information on the callers of B.

We can determine the call stack at Row 2 by pushing `C` onto the call stack at Row 1, because we know the instruction at Row 1 (`B`) is a method invocation and `C` is the return address (`C` is the next instruction of `B`). Similarly, we can calculate the call stack at Row 3 by simply popping one element from the call stack at Row 2, because we know the instruction at Row 2 (`P`) is a method return.

Formally, Algorithm 4 shows the algorithm to incrementally build the call stacks from a given sequence of instructions.

Algorithm 4 Call Stack Building

```
Input: insts[0,...,n]: a sequence of instructions, where address,
    isMethodInvoke, and isMethodReturn information is available.
Output: tree: a call stack tree
Output: insts[0,...,n].callStack: insts[i].callStack points to a
    node in tree that represents the call stack at insts[i]
 1: tree ← a call stack tree with only one node
 2: p ← the root node of tree
 3: for $i = 0$ to $n - 1$ do
 4:     insts[i].callStack ← p
 5:     if insts[i].isMethodReturn then
 6:         if p.parent does not exist then
 7:             p.address ← insts[i + 1].address
 8:             create a new parent node newp
 9:             p.parent ← newp
10:         end if
11:         p ← p.parent
12:     else if insts[i].isMethodInvoke then
13:         create a new child node newc
14:         newc.address ← address of the next instruction of insts[i]
15:         // insts[i].address + 4 in a 32-bit fixed length instruction
16:         newc.parent ← p
17:         p ← newc
18:     else
19:         // do nothing
20:     end if
21: end for
22: insts[n].callStack ← p
```

For efficiency, we represent call stacks by nodes in a method call tree. We define the *method call tree* as a directed tree where each node has an ID (left field) and a return address value (right field), as shown in Figure 3 (d). Edges are drawn from child to parent (that is, callee to caller), and the root node is always "`...`" to represent an irrelevant part of the call stack. A node represents a call stack, which means that the path from the root node to that node corresponds to a call stack. For example, Node #2 represents the call stack "`...-C`".

First, we create an initial call stack tree *tree* in Lines 1-2, as shown in Row 0 in Figure 3. The p indicates the node that corresponding to the current call stack. Then, for each instruction i, we update *tree* and p and set *insts*$[i]$.*callStack* (Lines 3-21), according to the instruction addresses and instruction types. In other words, at method invocation we move p to a child node corresponding to the callee method (corresponding to pushing a new element to the call stack), and at method return we move p to a parent node (popping an element from the call stack). If needed, a new node is created. For each i, the call stack tree (at Line 4) is shown in Figure 3 (d) Row i, where p is indicated by the grey node, and *insts*$[i]$.*callStack* is shown in Figure 3 (c) Row i.

The time and space complexity of this call stack building algorithm is $O(n)$, and it is independent of the maximum depth of the call stack. Also, this algorithm collects only the necessary portion of the call stacks. This can be faster and space-efficient than recording all of the call stack elements at every instruction.

3.3 False Loop Filtering for Trace Trees

In the evaluation in Section 5, we will evaluate false loop filtering algorithms by using them with linear trace selection algorithms. However, false loop filtering can also be applied to the trace trees when a cyclic-path-based repetition detector is used.

Figure 4 shows an example of the false loop problem in trace tree formation. Here we collect the execution paths starting from each trace head and terminated when a repeating cyclic execution path is detected (using a cyclic-path-based repetition detector), and merge those paths into one tree-shaped trace. We do not use the stop-when-leaving-owning-method or the stop-when-leaving-loop conditions.

Program 3 shown in Figure 4 (a) contains interprocedural nested loops. The inner loop is executed more frequently than the outer loop, so `P` in method `g`, the loop head of the inner loop, is selected as the trace head. In cyclic-path-based repetition detectors, the sequences `P-Q-P` (a true iteration of the loop in method `g`), `P-B-P`, and `P-C-A-P` (these two are false loops) are considered as cyclic execution sequences, and Trace 3A is formed by merging them, as shown in Figure 4 (c). However, if we can exclude false loops, the sequences `P-Q-P`, `P-B-P-C-A-P`, `P-B-P-Q-P-C-A-P`, `P-B-P-Q-P-Q-P-C-A-P`, ... are detected as repeating cyclic execution sequence, and Trace 3B is formed, as shown in Figure 4 (d). The three sequences `P-B-P-C-A-P`, `P-B-P-Q-P-C-A-P`, and `P-B-P-Q-P-Q-P-C-A-P` are true iterations of the loop in method `f`, with zero, one, and two iterations of the inner loop are unrolled and inlined, respectively. Trace 3B is longer and more suitable for optimization than Trace 3A.

4. Implementation

We prototyped a trace-driven system based on IBM's J9 JVM. The original J9 interpreter was modified to send control-flow events such as branches and method invocations to the trace selection engine that forms traces out of the executed Java bytecode. Once a trace has been recorded, it is added to a compilation queue shared by all of the Java threads, which is compiled by a compilation thread. To compile the traces, we extended the original method-based JIT in J9 to take traces as the basic unit for compilation [16].

Once a trace is compiled, the binary code is placed in a global trace cache. Dispatches to compiled code are initially done by the interpreter that checks if the next bytecode to be executed corresponds to the head of a compiled trace. When the execution exits from a trace, a runtime helper is invoked to link the traces if a compiled trace is available at the exit target.

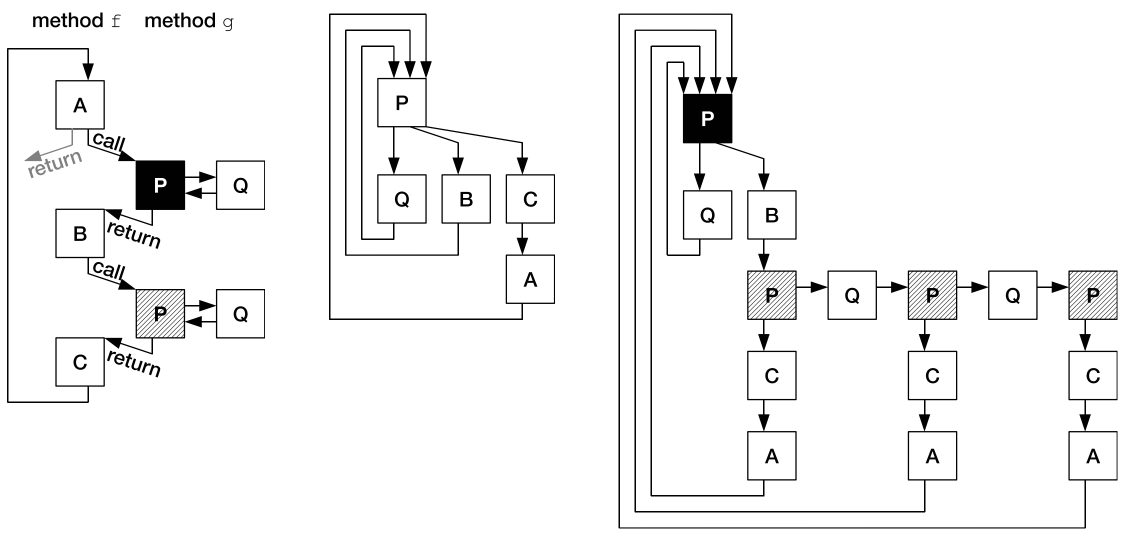
The rest of the section discusses a few implementation details of the system.

4.1 Interpreter event monitoring

The trace selection engine is driven by these control-flow events generated by the interpreter.

- **method invocation**: when executing an invoke bytecode.
- **method enter**: when entering a Java method.
- **method return**: when executing a return bytecode or returning from a JNI method.

```
void f(){
 for(A){
  g();
  B;
  g();
  C;
 }
}
void g(){
 for(P)
  Q;
}
```



(a) Program 3 (b) CFG of Program 3 (c) Trace 3A (without false loop filtering) (d) Trace 3B (with false loop filtering)

Figure 4. False loop filtering for trace tree selection

- **branch**: when executing a branch bytecode (such as `goto`, `ifeq`, or `tableswitch`).
- **exception throw**: when an exception occurs or the interpreter executes an `athrow` bytecode.
- **exception catch**: when an exception is caught by a catch block.
- **trace exit**: when returning from a compiled trace to the interpreter (i.e., via unlinked trace exits).

For each monitored event, the interpreter invokes a call-back function to the trace selection engine with the event information and its context, such as the current and next PCs, the caller/callee methods, and the receiver object class.

4.2 Trace Selection

The trace selection engine is implemented as a runtime library driven by the control-flow events described in Section 4.1. The selection engine maintains a counter to track the execution frequency for each target PC of a backward branch (at branch events) or a trace exit (at trace exit events). Currently our trace selection engine only creates linear traces.

When the counter of the current PC exceeds a predefined threshold, the selection engine switches to trace recording mode and records subsequent interpreter events until a termination condition is met. Since the interpreter monitors every control-flow bytecode, a trace of bytecodes (basic blocks) can be constructed out of the sequence of recorded control-flow events.

The trace selection engine can be configured to use the various subsets of trace termination conditions defined here:

1. When the repetition detector detects a repeating cyclic execution sequence
   - 1a. that cycles back to the head of the buffer (the repetition detector returns **true** for $s = 0$), or
   - 1b. that cycles back to the middle of the buffer (the repetition detector returns **true** for $s > 0$).
2. When the recorded events exceed the maximum length.
3. When there is an unusual or unsupported instruction sequence such as
   - 3a. an exception-throw event,
   - 3b. certain JNI invocation event[2],
   - 3c. returning to JNI event, or
   - 3d. an unexpected event sequence.
4. When the last basic block is the head of another existing trace (i.e., *stop-at-existing-trace-heads*).

Conditions (1a) and (1b) use a repetition detector. Whether or not to use false loop filtering in repetition detectors is another configurable parameter of the trace selection engine. Condition (4), stop-at-existing-trace-head, is a heuristic used in NET, LEI, and [18] for space efficiency.

4.3 Trace JIT

To compile the traces selected by the trace runtime, we implemented our trace JIT compiler by enhancing the existing (method-based) JIT compiler in the IBM's J9 JVM. Our trace JIT compiler assumes the mixed execution of the interpreter and the trace JIT. It does not support mixed execution of the method-based JIT and the trace JIT.

The trace JIT is implemented from the same code base as the method JIT and the interpreter. It supports the basic functions of the method JIT including exception handling, GC, synchronization, and JNI calls. Compilation is done by a separate dedicated thread, similar to the method JIT.

Our trace JIT guarantees that the JVM states, such as the Java stack and the program counter value, are compatible with the interpreter at the trace exit points. We do not maintain the JVM states in the middle of the trace to improve the performance. If a stop-the-world garbage collection begins while a Java thread is executing a compiled trace, we reconstruct the Java stack before the garbage collector walks the stack frames of this Java thread. We prepare the

[2] We allow a small set of JNI methods in the Java standard library to be included in traces. Invocation and return events for these JNI methods do not terminate a trace [16].

Benchmark	Description
antlr	A parser generator and translator generator
bloat	A bytecode-level optimizer and analyzer for Java
chart	A graph plotting toolkit and pdf renderer
eclipse	An integrated development environment
fop	An output-independent print formatter
hsqldb	An SQL relational database engine written in Java
jython	A python interpreter written in Java
luindex	A text indexing tool
lusearch	A text search tool
pmd	A source code analyzer for Java
xalan	An XSLT processor

Table 1. Description of the `DaCapo` benchmark.

special metadata for this reconstruction at the JIT compilation time. If an exception occurs during the execution of compiled trace, we update the JVM states and fall back to the interpreter to handle the exception.

Our trace compiler applies optimizations that are almost equivalent to the "warm" optimization level in the existing IBM's Java JIT compiler. These optimizations include common subexpression elimination, dead store elimination, dead code elimination, value propagation, and global register allocation. Currently we only support one optimization level and we do not support recompilation.

Our trace JIT also supports trace linking optimizations to reduce the overhead in the trace runtime. The execution jumps directly from a trace exit to the entry point of the next trace without going back to the interpreter if, for example, the trace exit has only one candidate for the next program counter value.

5. Evaluation

5.1 Benchmarks and Setup

We used the DaCapo benchmark suite [5, 9] with the default data size (`-s default`) to evaluate false loop filtering. Table 1 gives a brief description of the benchmarks. We executed 10 iterations of the long `eclipse` benchmark, and 25 iterations for the other benchmarks.

The trace selection engine uses these default parameters. The trace-head selection threshold is 500, which was determined based on the thresholds used by the original method JIT for method compilation. The maximum trace length is 128 basic blocks, which was determined based on an experimental evaluation. Using these default parameters , the coverage of traces (the ratio of dynamic basic blocks executed in the compiled binary code of traces) was more than 98.9% at steady-state for the DaCapo benchmarks. The trace JIT used its highest optimization level, which is equivalent to the warm-level optimization in the original method JIT.

To study the impact of the false loop filtering, we defined two baseline trace selection algorithms without false loop filtering, *BASE* and *BASE+*.

- **BASE** uses termination conditions (1)-(4) (defined in Section 4.2) to model a LEI-like selection, and uses the repetition detector without false loop filtering (Algorithm 1).
- **BASE+** uses termination conditions (1)-(3) and and uses the repetition detector without false loop filtering (Algorithm 1). Condition (4), stop-at-existing-trace-head, is removed to allow the creation of longer traces.

We then define two variations of the baseline selection algorithms with the false loop filtering by using call-stack-comparison:

	BASE	BASE w/FLF	BASE+	BASE+ w/FLF
Condition (1)	Yes	Yes	Yes	Yes
False loop filtering	No	Yes	No	Yes
Condition (2)	Yes	Yes	Yes	Yes
Condition (3)	Yes	Yes	Yes	Yes
Condition (4)	Yes	Yes	No	No

Table 2. Summary of trace selection algorithms

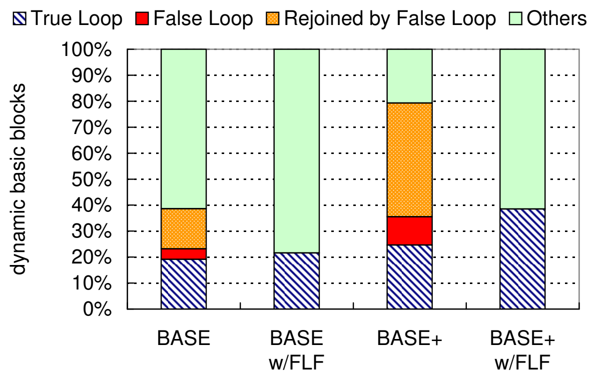


Figure 5. Average distribution of true-loops and false-loop-related traces in `DaCapo`.

- **BASE w/FLF** uses termination conditions (1)-(4) and uses the repetition detector with the false loop filtering by call stack comparison (Algorithm 3).
- **BASE+ w/FLF** uses termination conditions (1)-(3) and uses the repetition detector with the false loop filtering by call stack comparison (Algorithm 3). This is the default trace selection algorithm for our trace JIT.

We summarized the four trace selection algorithms in Table 2.

We also compared the trace JIT with the following configurations for the method JIT.

- **method (warm)** uses the method JIT with warm-level optimizations.
- **method (full)** uses the method JIT with default (full) optimization levels and upgrade compilation.

All performance data was gathered on a 4.0-GHz IBM® POWER6® blade center with 4 cores (8 threads) running on IBM AIX® 6.1. We averaged 16 runs to gather the performance data.

We also collected detailed statistics for the trace JIT (Figures 5 and 13) by executing 16 instrumented runs (only for the trace JIT). In these runs, the trace JIT generates JITed code with statistics collection code.

5.2 Quantify False Loops and FLF

Figure 5 shows the ratios of the numbers of dynamic basic blocks executed in each of the following types of traces, averaged over the `DaCapo` benchmarks:

- **True loop:** A trace is terminated because it forms a cycle and the cycle is a true repeating cyclic execution path (after the false loop filtering by call-stack-comparison). In other words, a trace is terminated by the condition (1a) and the detected repetition is a true repeating cyclic execution path.

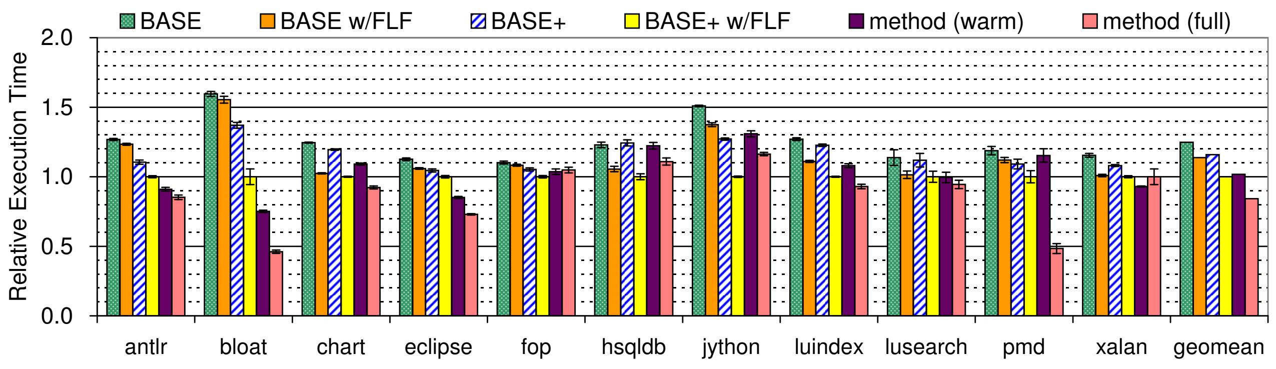


Figure 6. Relative steady-state execution time (normalized to **BASE+ w/FLF**)

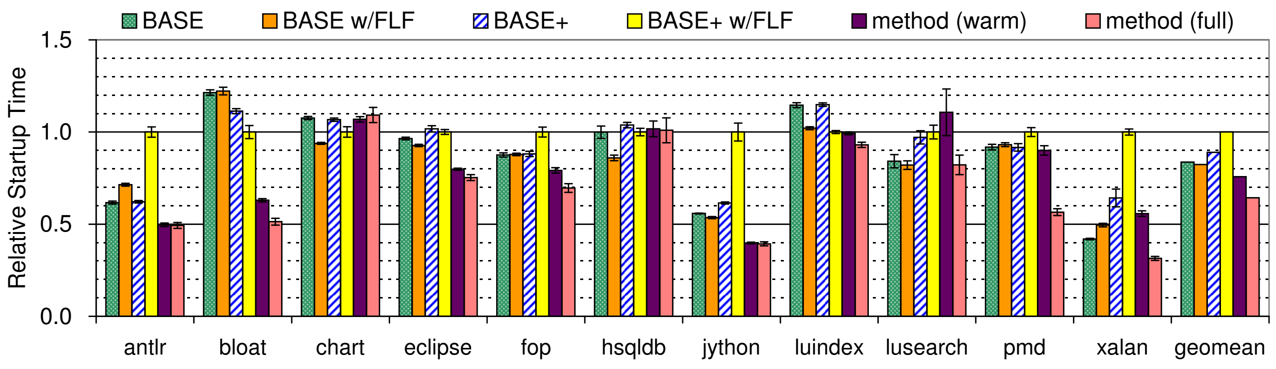


Figure 7. Relative startup time (normalized to **BASE+ w/FLF**)

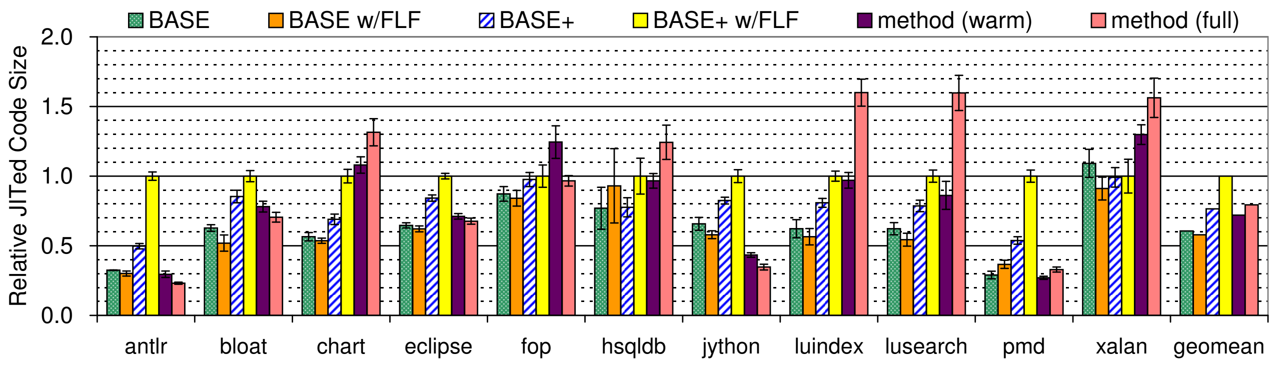


Figure 8. Relative compiled code size (normalized to **BASE+ w/FLF**)

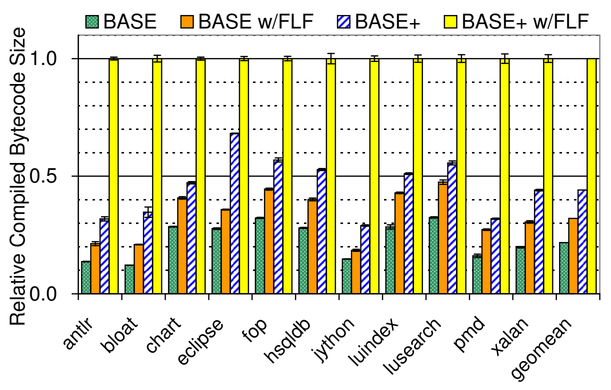


Figure 9. Relative number of bytecodes compiled (normalized to **BASE+ w/FLF**)

- **False loop:** A trace is terminated because it forms a cycle, but is a false loop. In other words, a trace is terminated by the condition (1a) and the detected repetition is a false loop.
- **Rejoined by false loop:** A trace is terminated because a false loop cycle is formed in the middle of the trace recording buffer. In other words, a trace is terminated by the condition (1b) and the detected repetition is a false loop.
- **Others:** A trace is ended by other termination conditions.

Figure 5 quantifies the extent of the false loop problem in the `DaCapo` benchmarks using our baseline selection algorithms: 55% (for *BASE+*) and 19% (for *BASE*) of the dynamic basic blocks are executed on traces terminated by false loops or rejoined by false loops. *BASE* suffers much less from false loops than *BASE+* because *stop-at-existing-trace-head* often terminates a trace before a false loop cycle can be formed.

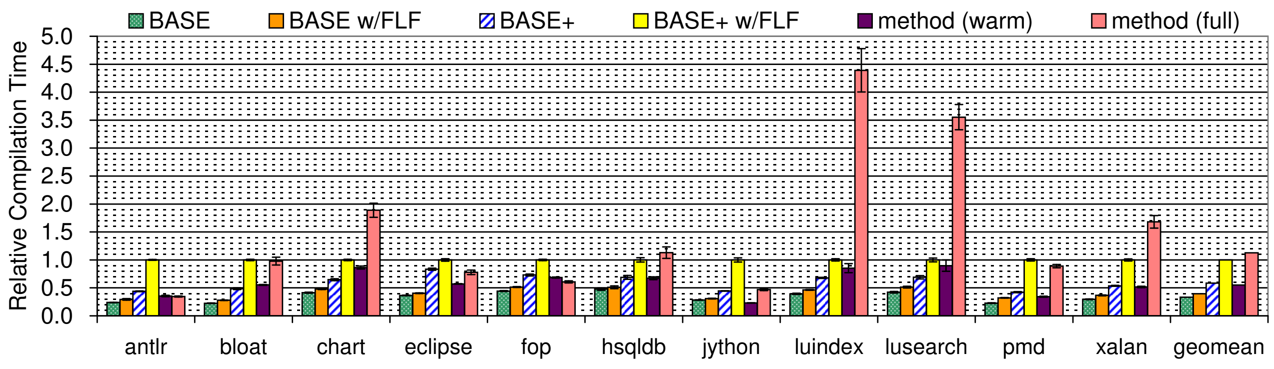


Figure 10. Relative compilation time (normalized to **BASE+ w/FLF**)

Most false loops are caused by multiple invocations of the same method in one execution path. This pattern is quite common in object-oriented programs, which often makes frequent calls to small methods such as accessor methods.

Figure 5 also shows one positive effect of false loops filtering: the ratio of true loop traces has increased from 19% to 22% (for *BASE*) and from 25% to 38% (for *BASE+*).

5.3 Impact of FLF on the Trace JIT

We next evaluate the impact of the false loop filtering on key measurable aspects of the trace-driven system: steady-state performance, start-up performance, compiled code size, and compilation time.

Figure 6 shows the steady-state execution time of the trace JIT and the method JIT normalized to that of BASE+ w/FLF. In Figure 6 and the following figures, the error bars show the sample standard deviations. The steady-state performance is measured as the average execution time of each of the last 5 iterations, to avoid fluctuation due to garbage collection events. On average, false loop filtering improved the steady-state performance by 16% for *BASE+* and 10% for *BASE*. The benefit is much smaller for *BASE* because, as shown in Figure 5, fewer traces are terminated by false-loop related termination conditions due to the use of another termination condition, stop-at-existing-head. At the same time, for *BASE+*, the improvements from false loop filtering are quite significant for some benchmarks, such as 37% for `bloat`, 27% for `jython`, and 24% for `hsqldb`.

For systems with a larger trace runtime overhead, we expect the impact from false loop filtering on the runtime component to be more significant. For example, if we disable a runtime overhead optimization (fast trace look-up using shadow arrays [16]), the steady-state performance improvement from false loop filtering was 64% and 32% for *BASE+* and *BASE*, respectively, which are much larger than the results shown in Figure 6.

In steady-state execution, the execution time of the false loop filtering algorithm itself was negligible (typically less than 0.1%), because trace recording occurs rarely in steady state.

Figure 7 shows the start-up execution time, which is the execution time of the first iteration of each benchmark. False loop filtering increases the start-up time by an average of 12% for *BASE+* and decreases it slightly for *BASE*. This is because in our system the start-up performance is dominated by how soon a trace is compiled. While longer traces are in general preferable for better code quality, they can be detrimental to start-up performance due to the longer compilation time.

Figure 9 shows the total number of bytecodes being compiled. False loop filtering almost doubles the cumulative bytecode size for *BASE+*. This leads to an increase in compilation time (by 71%) and the compiled code size (by 31%) for *BASE+*, as show in Figure 10 and Figure 8, respectively.

Ways to reduce code size expansion and compilation times are also an important aspect of trace compilation and will be future work.

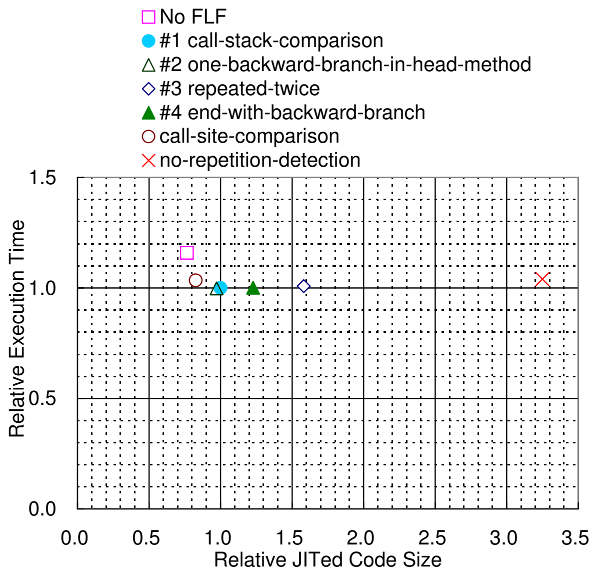


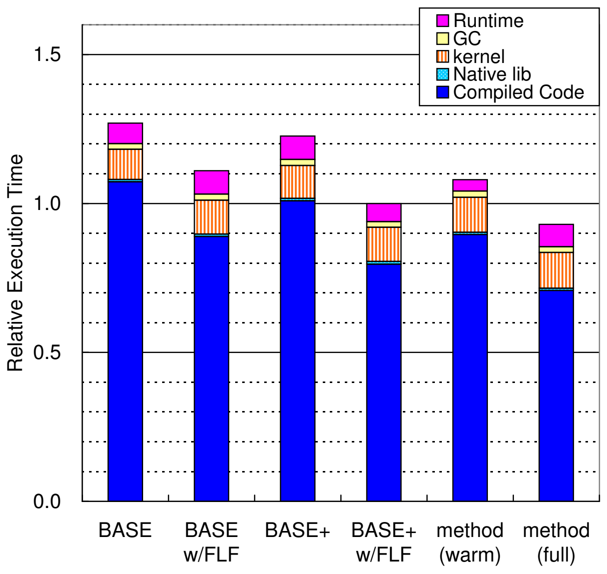
Figure 11. Relative execution time and compiled code size when various false loop filtering algorithms are applied to *BASE+*, normalized to the call-stack-comparison.

5.4 Comparison of False Loop Filtering Algorithms

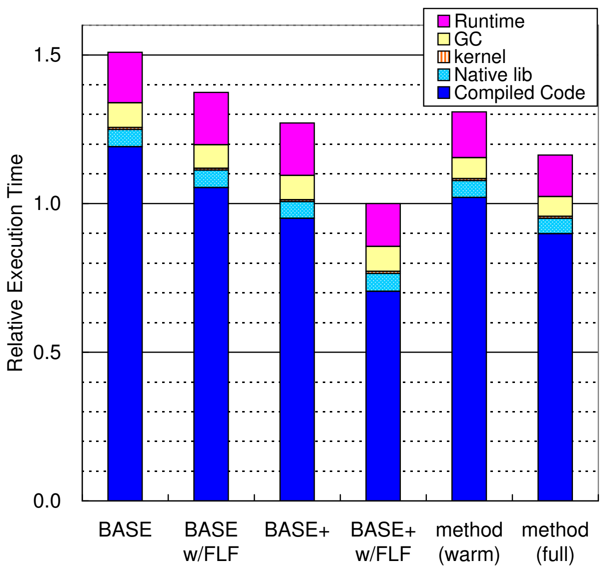
Figure 11 shows the relative steady-state execution times and relative compiled code sizes for the other variations of false loop filtering algorithms described in Section 3.2, relative to *call-stack-comparison*. All schemes are applied to *BASE+*.

Two algorithms, *call-stack-comparison* and *one-backward-branch-in-head-method*, achieve the best performance and the minimum code size among those with equivalent performance. *One-backward-branch-in-head-method* is slightly more conservative in detecting false loops than *call-stack-comparison*, where 4% of its traces are either false loops or rejoined by false loops. But these false loops have little effect on the performance and the the compiled code size.

Two other algorithms, *repeated-twice* and *end-with-backward-branch*, also achieved the best performance, but with a much larger amounts of code. These algorithms are overly aggressive in their



(a) `luindex`



(b) `jython`

Figure 12. Relative steady-state execution time breakdown (normalized to **BASE+ w/FLF**)

false loop detection in the sense that certain true loop cycles (e.g., ones that start from non loop-headers) may be detected as false loops, leading to longer traces. The value of these algorithms is that they require less information for the executed instructions than the first two. For example, *end-with-backward-branch* requires *isBackwardBranch* and *address*, and *repeated-twice* requires only the *address*. Therefore, they may be used for binary trace-based systems where precise information on executed instructions, such as the owning methods or call stack information, is not available.

We added two additional configurations, *call-site-comparison* and *no-repetition-detection*, which are inferior to the other algorithms but are evaluated to demonstrate the inefficiency of simpler heuristics.

Call-site-comparison compares the callers of the first and the last instruction on a cyclic path and detects a false loop if the callers are not the same. This scheme can be viewed as a restricted form of call-stack-comparison (#1) where only the top of the call stacks are compared. Call-site-comparison is 3% slower than call-stack-comparison and fails to detect a significant portion of false loops. For example, with call-site-comparison, 29% of dynamic basic blocks are still executed from traces related to false loops. This indicates that comparing only the call sites is not sufficient and a deeper call stack needs to be inspected.

When traces are not terminated in any repeating cyclic paths (*no-repetition-detection*), the code size is more than 3x larger than *call-stack-comparison*. This demonstrates the importance of repetition detection to achieve good space-efficiency in trace formation.

5.5 Why False Loop Filtering Improves Performance?

To understand the performance benefit of false loop filtering, we profiled the steady-state execution of `luindex` and `jython` in our trace system and divide the time spent into components.

- **Compiled code** represents the execution time of compiled (JITed) traces.
- **Runtime** represents the execution time of the interpreter, helper functions called from compiled code, and trace execution overhead.

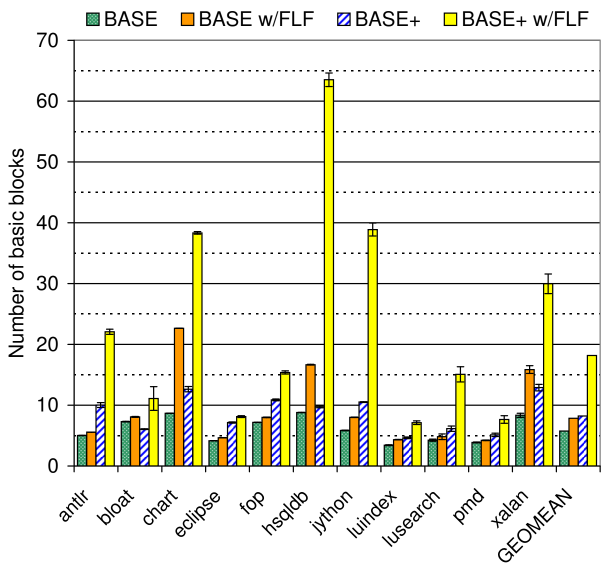


Figure 13. Average dynamic trace length in BBs

- **Native library**, **GC**, and **OS Kernel** represent the execution time of native parts of the Java class libraries, garbage collection, and OS kernel, respectively.

As shown in Figure 12, the steady-state performance improvements from false loop filtering come mostly from the compiled code component.

To further understand these improvements, we measured several metrics of trace execution in the instrumented runs. We found that improvements in the *dynamic trace length*s, defined as the average number of consecutive basic blocks executed between entering and exiting from a trace in a program, is a good indicator of relative performance among different selection algorithms. Figure 13 shows average dynamic trace lengths of the four selection al-

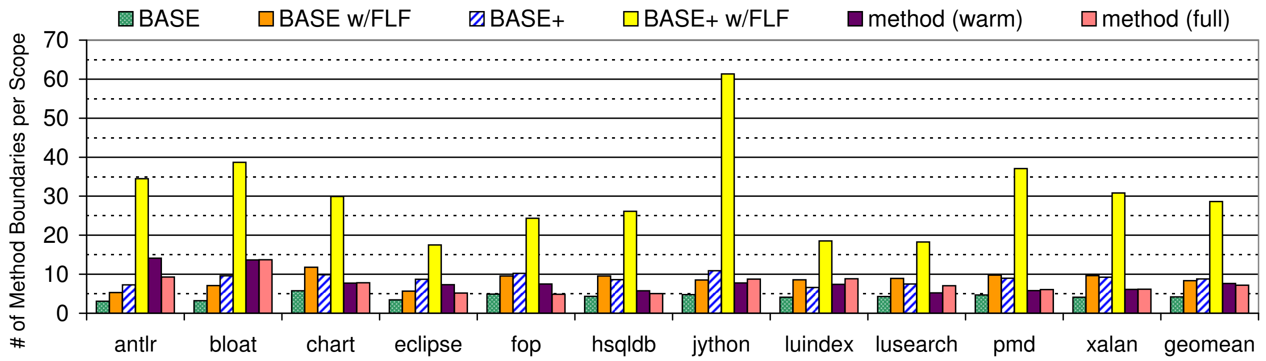


Figure 14. Average number of method boundaries per compilation scope

gorithms, where longer dynamic trace lengths often correspond to shorter running time in Figure 6.

A longer dynamic trace length means fewer trace exits and less runtime overhead. However, Figure 12 shows only a slight reduction in trace runtime overhead (part of Runtime) by false loop filtering despite a significant reduction in the number of dynamic trace exits (the inverse of the dynamic trace length). This is because our system has aggressively optimized the runtime overhead due to trace exits [16] using techniques such as trace linking, fast trace look-up using shadow arrays, and inlining of trace look-up sequence into compiled code. As a result, the trace runtime overhead at steady-state is quite small.

Another indicator is the number of method boundaries (both method invocations and method returns) crossed in one compilation scope. In the trace JIT, a trace can naturally span multiple methods (since we do not terminate traces at method invocations or method returns). In the method JIT, method boundaries are included in a compilation scope by method inlining.

The direct effect of crossing method boundaries in a compilation scope is the elimination of method invocation and return overhead. The indirect but more significant benefit is the expansion of compilation scope that facilitates more effective optimizations.

Figure 14 shows the average number of method boundaries per compilation scope. We counted a method invocation or a method return as one method boundary crossing. For example, when a method is entered and returned in a compilation scope (for example, a method is inlined once in the method-based JIT), the number of method boundaries in the compilation scope is two. Figure 14 shows that our trace JIT (BASE+ w/FLF) crosses many more method boundaries than the method JIT. The improved code quality in the current trace JIT most likely comes from this effect.

6. Related Work

Dynamo is the most influential trace-based compilation system [1, 2, 10]. Traces are formed out of binaries and collected using a native binary interpreter. The system optimizes traces for better code layout and for partial redundancy elimination [19]. Dynamo introduced the next-executing-tail (NET) policy for trace selection. The NET policy achieves much lower profiling overhead than exhaustive path profiling schemes such as the one presented by Ball and Larus [3] with the same prediction quality and faster convergence. The NET policy terminates traces at every backward branch. That is, it uses a stop-at-backward-branch repetition detector.

Hiniker, Hazelwood and Smith [15] introduced the last-executing-iteration (LEI) policy as an improvement over the NET policy. The LEI policy uses a cyclic-path-based repetition detector, instead of a stop-at-backward-branch repetition detector. Merrill and Hazelwood [18] focused on evaluating the potential of using traces for better code layout for a Java virtual machine using the LEI policy.

PyPy tracing JIT [6] applies trace compilation to Python. Pypy's tracing JIT traces a language interpreter executing a user program, and aims to capture the repeating cyclic execution paths in the user program, not those in the language interpreter. This is done by, for example, considering an execution path as a repetition if the program counter in the language interpreter *and* the program counter in the user program are the same at the beginning and the end of the execution path. The program counter in the user program is given as a hint by the implementers of the interpreter. This is also a cyclic-path-based repetition detector, where the program counter used by the repetition detector is that in the user program, rather than that in the direct target program.

Static-scope-based repetition detectors are used in trace compilers that support tree-shaped traces, such as HotpathVM [11, 12] for Java, TraceMonkey [13] for JavaScript, LuaJIT [17] for Lua, and SPUR [4] for CIL (and JavaScript converted to CIL). They utilize loop structures and methods of programs, and start trace recording of a root trace from a loop header. The trace recording is terminated or aborted when the execution leaves the method scope (HotpathVM) or loop scope (TraceMonkey, LuaJIT, and SPUR) which the loop header belongs to.

7. Conclusion

In this paper, we identified the problem of false loops in existing trace selection algorithms, which was not explicitly addressed in previous literature. We quantified the impact of false loops on trace selection algorithms using cyclic-path-based repetition detection. Depending on the trace selection algorithms being used, we showed that on average 55% and 19% of dynamic basic blocks are executed on traces affected by false loops (for BASE+ and BASE selection, respectively). We suggest false loop filtering is an important aspect in trace selection algorithm design.

We proposed the concept of false loop filtering and explored the design space of false loop filtering heuristics. We proposed a technique called false loop filtering by call-stack-comparison as well as several alternatives to satisfy other systems with different requirements. False loop filtering increased the dynamic trace length and method boundary crossings per trace, which produced better code and fewer trace exits.

We showed that our false loop filtering algorithm improved the steady-state performance of our trace-based Java system significantly (by 16% and 10% on average when applied to BASE+ and BASE, respectively) with acceptable code size and compilation time increases (by 31% and 71%, respectively). We also observed that improvements are much larger on systems with more trace runtime overhead (64% and 32% improvement on average when applied to BASE+ and BASE, respectively, when a runtime overhead optimization is disabled).

False loop filtering has a significant positive impact on the performance and its concept is useful for designing trace selection algorithms that maximize steady-state performance.

Trademarks

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical Report HPL-1999-78, HP Laboratories, 1999.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, June 2000. ACM.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[4] 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*, OOPSLA '10, pages 708–725, New York, NY, USA, 2010. ACM.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, Oct. 2006. ACM.

[6] C. F. 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*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[7] D. Bruening and S. Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 74–85, Washington, DC, USA, Mar. 2005. IEEE Computer Society.

[8] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '03, pages 265–275, Washington, DC, USA, Mar. 2003. IEEE Computer Society.

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

[10] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IX, pages 202–211, New York, NY, USA, Oct. 2000. ACM.

[11] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical report, University of California Irvine, November 2006.

[12] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 144–153, New York, NY, USA, June 2006. ACM.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. 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 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[14] 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 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, pages 12–12, Berkeley, CA, USA, June 2004. USENIX Association.

[15] D. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 141–154, Washington, DC, USA, Dec. 2005. IEEE Computer Society.

[16] 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 (to be published)*, CGO '11, Apr. 2011.

[17] LuaJIT. LuaJIT design notes in lua-l mailing list. `http://lua-users.org/lists/lua-l/2008-02/msg00051.html`, `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`, `http://lua-users.org/lists/lua-l/2008-06/msg00228.html`.

[18] D. Merrill and K. Hazelwood. Trace fragment selection within method-based JVMs. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 41–50, New York, NY, USA, June 2008. ACM.

[19] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, New York, NY, USA, June 1990. ACM.

[20] M. Zaleski, A. D. Brown, and K. Stoodley. YETI: a graduallY extensible trace interpreter. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 83–93, New York, NY, USA, 2007. ACM.