A Loop Correlation Technique to Improve Performance Auditing

Jeremy Lau
University of California, San Diego
IBM T.J. Watson Research Center
jl@cs.ucsd.edu

Matthew Arnold, Michael Hind
IBM T.J. Watson Research Center
{marnold,hindm}@us.ibm.com

Brad Calder
Microsoft
University of California, San Diego
calder@cs.ucsd.edu

Abstract

Performance auditing *is an online optimization strategy that empirically measures the effectiveness of an optimization on a particular code region. It has the potential to greatly improve performance and prevent degradations due to compiler optimizations. Performance auditing relies on the ability to obtain sufficiently many timings of the region of code to make statistically valid conclusions. This work extends the state-of-the-art of performance auditing systems by allowing a finer level of granularity for obtaining timings and thus, increases the overall effectiveness of a performance auditing system. The problem solved by our technique is an instance of the general problem of correlating a program's high-level behavior with its binary instructions, and thus, can have uses beyond a performance auditing system. We present our implementation and evaluation of our technique in a production Java VM.*

1. Introduction

Understanding and predicting the performance impact of compiler optimizations in modern applications is becoming increasingly difficult due to increased hardware complexity and software virtualization. Performance auditing [11] is a general online optimization strategy that attempts to address this challenge by empirically measuring the effectiveness of an optimization on a particular code region, such as a method. This technique greatly improves the ability of runtime systems to both increase performance and prevent degradations.

A performance auditing system creates two (or more) versions of a method: one with the optimization, and one without the optimization. As the program executes, invocations of these versions are timed and a statistical analysis is performed to determine the faster version. However, methods that are infrequently invoked will have a small number of timings, resulting in a larger number of inconclusive comparisons. This happens when a method spends time in loops rather than being frequently invoked, because the baseline performance auditing system only collects method invocation timings. Thus, for such methods, a solution is needed that times loop iterations rather than method invocations.

When collecting loop timings, the challenge is to collect comparable loop timings from two different compilations of the same loop. This becomes very difficult as loop optimizations are introduced: for example, if the optimizer unrolls the loop, the loop's timing data must be adjusted accordingly.

This problem is an example of the general problem of mapping between program source and its optimized binary instructions. Prior work in this more general area falls into two categories: use a heuristic matching algorithm to correlate the source and binary [10, 18, 13, 20], or modify the optimizer to maintain a mapping between the program's source and the optimizer's output [1, 5]. Both of these approaches have significant shortcomings that make them inappropriate for the problem we are addressing.

This paper presents a new solution to this problem. We inject lightweight markers into the code before optimization that have the semantics of an arithmetic operation on a specially allocated global variable. We allow the optimizer to freely manipulate these markers, which are removed after optimization. The optimizer's semantics-preserving property automatically maintains the mapping between source and binary. Because the markers are operations on global variables, the optimizer may not remove the markers, although it may move, copy, or combine markers.

We evaluate our technique in the context of performance auditing. The contributions of this work are

- a technique to accurately map between a program's source and its optimized binary without using heuristic matching, and without modifying the optimizer;

- a technique to predict if loop timings will be needed by a performance auditing system and an algorithm to choose loop timing points to minimize overhead; and
- an evaluation of the effectiveness of our techniques.

The paper is organized as follows: Section 2 describes performance auditing. Section 3 describes our experimental methodology used throughout the paper. Section 4 presents our technique to map between a program's source and binary. Section 5 presents our technique to predict if loop timings will be needed by a performance auditing system, and our algorithm to choose loop timing points to minimize overhead. Section 6 evaluates the effectiveness of our techniques. Section 7 discusses related work on the problem of mapping between source and binary. Section 8 discusses other possible uses for our ideas, and Section 9 concludes this paper.

Program	Methods Executed	Bytecodes Exe (KB)	Hot Methods Total	Hot Methods Loopy
antlr [17]	1702	228	2	0
compress [16]	770	66	3	2
daikon [4]	2108	171	1	0
db [16]	782	67	3	2
hsqldb [17]	1416	147	2	1
ipsixql [3]	828	61	7	1
jack [16]	746	56	3	0
javac [16]	1467	133	2	0
jbb2000 [15]	1197	115	3	2
jess [16]	1140	86	4	1
jython [17]	1777	186	2	1
mpegaudio [16]	866	78	2	1
mtrt [16]	853	76	3	0
phase [12]	450	31	2	0
pmd [17]	2030	128	1	0
ps [17]	946	75	1	0
soot [14]	2061	235	4	2
xalan [17]	2108	171	1	0
xerces [19]	521	36	1	0

Table 1. Benchmark suite

2. Performance Auditing

Performance auditing [11] is typically applied to the critical subset of a program's executing methods. This is most easily accomplished in a modern virtual machine, which automatically detects hot methods as a program executes and recompiles them at various levels of optimization [2]. The small subset of methods that reach the highest level of optimization are candidates for performance auditing.

A performance auditing system compiles two (or more) versions of the method: one with the optimization (A), and one without the optimization (B). As the program runs, one version of the method is randomly selected, invoked, and timed. Periodically, a statistical analysis with a confidence threshold is performed to determine the faster version. This analysis produces one of three possible outcomes: 1) A is faster, 2) B is faster, or 3) there is insufficient statistical evidence to determine which is faster. In the first two cases, the system has converged on a performance decision, and that version of the method can be used.

If it is not clear which version is faster, more timing data is needed to make a statistically significant conclusion. Because timing data is produced whenever the method is invoked, the system will wait until the program produces more timing data, then re-run the statistical analysis. Thus, a small number of timings will typically result in a low convergence rate, which results in a larger number of inconclusive comparisons.

3. Methodology

The experiments in this paper were performed using a development version of IBM's J9 VM and its high-performance optimizing JIT compiler [7]. The VM was run on a Pentium 4 3.0 GHz machine with 2 processors and 1 GB RAM running Linux. We use a suite of 19 benchmarks composed of the complete SPECjvm98 benchmark suite [16], the SPECjbb2000 benchmark [15], and several other Java applications, including six from the DaCapo benchmark suite [17][1]. The first column of Table 1 reports the number of methods executed, which counts all Java methods, including library methods, executed by the JVM to load and execute the benchmark. The second column lists the total size (in KB) of all bytecodes executed for each benchmark. These numbers report dynamic metrics, i.e., they are based on what is executed, not what could be executed.

The third column lists the number of hot methods for each benchmark. We define the hot methods as methods that consume enough cycles to be selected by J9 for aggressive feedback-directed optimizations. For these methods, J9 first performs an additional compilation to instrument the method for profiling, then optimizes the method at one of the two highest optimization levels (O4 or O5).

The last column lists the number of hot methods that are infrequently invoked. These are hot methods, so the processor spends a significant amount of time executing instructions in these methods, yet they are not invoked frequently. This implies that these methods spend most of their time in loops. These methods will be discussed in detail in Section 5.

We use the x86 Read Time-Stamp Counter (RDTSC) instruction to collect timings. The RDTSC instruction reads the processor's 64-bit cycle counter and stores its value in a register. The cycle counter is highly accurate, but RDTSC instruc-

[1]The development version of the VM that we used did not run the remaining benchmarks from this version of the DaCapo suite

tions may execute out-of-order, and only a single RDTSC instruction can execute at a time, so a large number of dynamic instructions should execute between invocations of RDTSC [9].

To collect timings, we define startTimer() which reads the cycle counter and stores the count in the method's stack frame, and stopTimer() which subtracts the current cycle count from the cycle count on the stack, and stores the difference in a circular buffer.

To collect method timings, we instrument the method's entry points with calls to startTimer(), and we instrument the method's exit points with calls to stopTimer(). To collect loop timings, we instrument the method's entry and exit points as with method timings, but also insert back-to-back calls to stopTimer(); startTimer() at each loop timing point. So, when a method that has been instrumented to collect loop timings is run, the sequence of calls to startTimer() and stopTimer() will be

```
startTimer() // method entry
stopTimer()  // loop timing point 1
startTimer() // loop timing point 1
stopTimer()  // loop timing point 2
startTimer() // loop timing point 2
stopTimer()  // method exit
```

We inline all calls to startTimer() and stopTimer().

This methodology measures the total time that the method was active on the stack to ensure fair comparisons in the presence of changing inlining decisions. Measuring only time spent in the instrumented method (excluding callees) would produce incorrect results in the presence of method inlining.

4. Lightweight Markers

This section describes lightweight code markers, which are used to build a mapping between a program's source and its optimized binary. Each lightweight code marker is an arithmetic instruction that increments a specially allocated global variable. Markers are added to the IR at the beginning of optimization, and to the optimizer, these markers are just ordinary arithmetic instructions. Thus, the optimizer is free to transform the markers as it desires, as long as the semantics of the markers are preserved. Because each marker modifies a global variable, the optimizer may not remove markers, but the optimizer may move, copy, or merge markers. After (or during) optimization, the markers can be used to map their code locations back to unoptimized code. The markers are removed before register allocation and final code generation, so lightweight markers do not generate code that executes at runtime.

The markers attempt to satisfy three conflicting goals:

1. The marker should be lightweight; it should have little or no impact on optimization. Inserting instructions that block optimization would allow mapping from source to binary, but it would do so by disabling optimizations. Markers that disable optimization are not a viable solution because our goal is to evaluate performance; any changes to the optimizations performed would change what we are attempting to measure.

2. The markers must maintain the desired mapping information from source to optimized code, thus they must be respected by the optimizer sufficiently to maintain the desired information. If the optimizer deletes markers, or moves markers far from the code they are tracking, mapping information is lost.

3. Marker insertion and removal should be easy to implement. No changes to the optimizer should be required.

To achieve these goals, we use markers that are simple arithmetic instructions that increment a global counter by one. This gives the optimizer significant freedom to optimize the markers. For example, if a marker is placed in a loop and the optimizer unrolls the loop four times, the optimizer should produce four copies of the original marker. The optimizer may then combine the four copies of the marker into a single merged marker, which increments by four. Because each marker increments a different global variable, optimized markers can be mapped back to source locations based on the global variable that they increment.

Although the optimizer is free to optimize the markers, the semantics-preserving nature of the compiler guarantees that certain properties are maintained. Markers are never deleted because they modify a global variable. The markers may be duplicated, moved, or combined, but all optimized markers referencing a global variable X are duplicates of the source-level marker that references X, and merged markers can be identified by markers that increment by more than 1.

Because the optimizer must respect the semantics of each code marker, a marker's total dynamic count can not change across optimizations. For example, if markers are not removed after optimization and code is generated for the markers, and the program is run deterministically, the value of each marker's global variable will be the same at the end of every run, regardless of the optimization settings used.

Every marker's count is the ratio of its current block's execution frequency to its original block's execution frequency. For example, if we place a marker that increments by 1 in block A, and the optimizer moves that marker to block B and modifies the marker so it increments by 5, the marker now indicates that every execution of block B corresponds to 5 executions of block A.

So while code markers may move from their original location, they still provide information on how their new location relates to their original location. This information will be very useful for some applications, such as collecting a source-level block profile from an optimized binary as in [1].

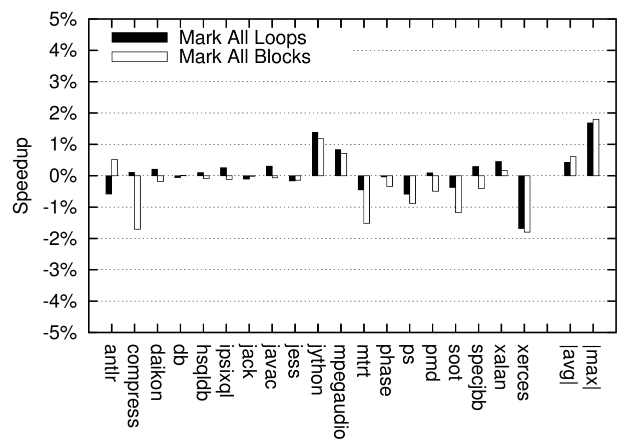


Figure 1. Speedup/slowdown observed when a marker is inserted and removed in every loop or every block in every method

In this paper, markers are used to identify semantically equivalent loops across compilations to collect comparable timing data. It becomes more difficult to achieve this goal if the optimizer hoists markers out of loops. This can happen if the optimizer identifies the number of iterations that the loop will execute for, moves the marker out of the loop, and modifies the marker to increment by the number of loop iterations.

This has not been a problem in our experience, because our application of code markers tracks loops that account for significant portions of program execution, and such loops tend to have characteristics that make it difficult for an optimizer to hoist code out of the loop, such as `break` or `continue` statements, which make it difficult to statically determine the number of loop iterations.

If a marker is hoisted out of a loop, it may still be possible to compensate for it depending on how the markers are used, because the optimizer must respect the semantics of markers. Details on how this is done for performance auditing will be described in Section 5.4.

Figure 1 shows the overall impact on performance of inserting and removing a marker into every loop in every method, or every block in every method. Markers are inserted before the optimizer is run and are detected and removed when the optimizer completes, just before register allocation and code generation. Therefore, markers do not exist at runtime, so any performance differences are the result of different optimization decisions caused by the presence of markers during optimization. When a marker is inserted and removed in every loop in every method, the average performance difference is 0.4%, and the worst-case performance difference is 1.7%. This demonstrates that the markers have a minimal impact on optimization, and thus are lightweight. Inserting and removing a marker in every basic block is significantly more invasive than marking every loop, but this would be necessary if a full block mapping from source to binary was desired. Even when markers are inserted into every block in every method, the behavior of the optimizer still remains largely the same — the average performance difference is 0.6%, and the worst-case performance difference is 1.8%.

4.1. Marker Identification and Removal

Our marker approach requires the ability to identify markers in the optimized code, just before register allocation and final code generation. Identifying markers, even after they have been transformed by the optimizer, is easy because each marker refers to a variable that was specially allocated for the marker. So to identify a marker we just need to find instructions that reference the special global variable associated with that marker.

To remove a marker from the code, the system removes any instructions that reference the global variable, removes all instructions with dependencies only on the instructions that referenced the global variable, and deallocates the global variable.

5. Applying Code Markers to Performance Auditing

The previous section described a technique for correlating code between a program's source and its binary. This section describes how the correlation technique is applied in the context of a performance auditing system.

Our goal is to improve convergence time for a performance auditing system by collecting loop timings. To do this, we identify methods with poor convergence, and collect timing data for a subset of the method's loops. This section presents our technique to predict if loop timings are needed, and our algorithm to select loops for timing instrumentation.

The goals of the loop selection algorithm are:

1. Collect loop timings only when necessary. If method timings provide enough timing data, do not collect loop timings.

2. Collect only as much loop timing data as necessary. Do not insert unnecessary instrumentation.

3. Ensure that a sufficient amount of execution occurs between timings. The timer has a minimum resolution, and tight loops may not contain enough dynamic instructions per iteration to be measurable.

```
void selectLoops(method) {
  if method.invocationsPerSec > invocationThresh
    return
  targetFreq = initialTargetFreq
  while targetFreq > 0
    blockFreq = findLoop(method, targetFreq)
    if blockFreq == -1
      return
    else
      targetFreq -= blockFreq
}
Freq findLoop(method, targetFreq) {
  foreach loop in method in depth first order
    if (loop.freq > targetFreq &&
         block = selectBlock(loop, targetFreq))
      return block.freq
  return -1
}
Block selectBlock(loop, targetFreq) {
  sort loop.blocks by |block - targetFreq|
  foreach block in loop.blocks
    minDist =
      min(distToNearestTimingPoint, distToSelf)
    if minDist > minDistThreshold
      return block
  return nil
}
```

Figure 2. Loop Selection Algorithm

Section 5.1 describes our technique for predicting if loop timings, and thus, the correlation technique, will be required. Section 5.2 describes how we choose loops for timing instrumentation, Section 5.3 describes how we choose which blocks in each loop to instrument, and Section 5.4 describes how the loop selection algorithm and lightweight code markers are used to collect loop timing data.

Our loop selection algorithm makes decisions based on profile data. We are working in a Java virtual machine that interprets bytecodes before compiling them, and while a method is being interpreted, the virtual machine collects profile data for the method.

The raw block profile produced by the interpreter indicates approximately how many times each block was executed. Because methods can be interpreted for variable amounts of time, we normalize the frequencies from the interpreter profile data for each block B as follows:

$$normalizedFrequency_B = \frac{frequency_B}{\sum_{block} frequency_{block}}$$

In other words, we calculate the normalized frequency for a block by summing the frequencies of all blocks in the method, and dividing the block's frequency by the sum. This calculates the fraction of the method's execution that is spent in each block.

Figure 2 gives a high-level overview of the loop selection algorithm.

5.1. Predicting if Loop Timings are Needed

The first step is to determine if loop timings are necessary. When compilation occurs for hot methods, we examine the method's profile data to estimate how frequently the method was invoked. Specifically, the execution frequency of the method entry block is used to predict if loop timings are necessary.

If the method was frequently invoked, the profile data will indicate that the method entry block was executed frequently compared to other blocks in the method. If the method was frequently invoked, loop timings should not be necessary.

On the other hand, if the method was rarely invoked, the profile data will indicate that the method entry block was rarely executed compared to other blocks in the method. If the method was rarely invoked, but the method is still considered hot, method timings will not generate enough timing data, and we need to collect loop timings for the method.

5.2. Choosing Loops to Instrument

If loop timings are needed for a method, we need to collect timing data for some subset of the method's loops. Also, we must decide where to place the timing instrumentation in each loop — the loop head block is not always the best choice.

Our algorithm works as follows. We first define a target frequency threshold to ensure that we do not perform too much instrumentation or too little instrumentation. Ideally, the total frequency of instrumented blocks should just exceed the target frequency threshold.

We examine the loops in the method in depth-first order, starting from outermost loops. If the normalized loop head block frequency exceeds the instrumentation target threshold, we select the loop for instrumentation, and we call a subroutine to determine which block in the loop should be instrumented. The implementation of this subroutine will be described in the next subsection. The subroutine may or may not find a satisfactory block for the instrumentation code. If the loop is very tight, for example, the subroutine will not be able to find a good block in the loop for the instrumentation code. If no satisfactory block can be found, the depth-first search continues. If a satisfactory block was found, we decrease the target frequency threshold by the satisfactory block's weight. If the target frequency threshold is less than or equal to zero, or if no satisfactory blocks were found, the algorithm terminates. Otherwise, the algorithm runs again.

5.3. Choosing a Block to Instrument

We define a subroutine which, given a loop, determines where to best place the timing instrumentation. To collect useful timing data, there must be enough dynamic instructions between any two timing points. There may not be a best place to put the timing instrumentation — in this case, the subroutine returns an error code and no timing instrumentation is attempted for the loop.

To find the best block to place timing instrumentation in a loop, this subroutine first collects a list of the blocks in the loop, and sorts them by $|frequency - targetFrequency|$. In other words, the subroutine sorts the blocks by the distance between each block's frequency and the target frequency. This ensures that we consider blocks that closely match the target frequency first.

Next, we iterate over the sorted list of blocks, and we run Dijkstra's algorithm to compute single source shortest paths from each block in the list to all other blocks in the control flow graph. Then we examine the results of Dijkstra's algorithm to determine the minimum distance between each block and itself, and the minimum distance between each block and all timing points.

If the minimum distance between a block and its nearest timing point exceeds the minimum distance threshold, the block can be instrumented safely, because we know that there will always be enough dynamic instructions between instrumentation points.

We measure the distance between blocks in terms of instruction *cost* rather than instruction count. For example, a `load` instruction puts more distance between timing points than an `add` instruction. Instruction costs are static estimates based on instruction opcode, and they are commonly used in instruction scheduling. Also, we do not consider edge weights — we just calculate the minimum distance between two points, without considering the frequency or even the feasibility of the path. We do this for safety, because interpreter profile data may not completely capture the method's behavior.

Figure 3 shows control flow graphs for three infrequently invoked methods that can not be safely instrumented to collect loop timing data. In these control flow graphs, circles represent basic blocks, rectangles indicate loops, and the number in each circle indicates the cost of each block, which is just the sum of the cost of the block's instructions.

These three methods are infrequently invoked, but we can't collect additional loop timings from these methods because all their loops are tight. Inserting instrumentation code into these loops would not provide useful timing data — if we tried to insert timing instrumentation into these tight loops, the timing code would not actually time anything, because these loops do not execute enough dynamic instructions per iteration to compensate for timing and instrumentation overheads.

Our techniques automatically identified the three methods shown in Figure 3 as infrequently invoked methods where loop instrumentation could not be safely applied.

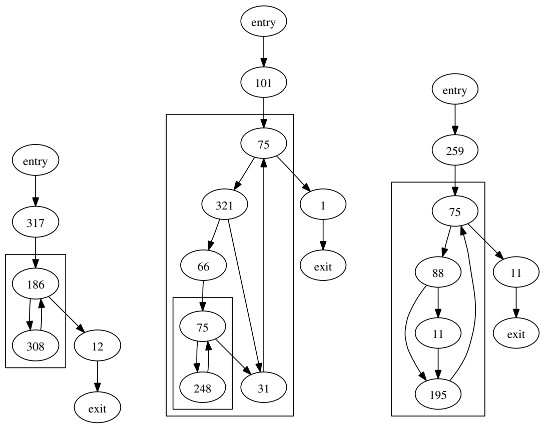


Figure 3. Three CFGs selected for loop instrumentation, where our algorithm was unable to find a good timing point

Our current approach does not attempt to collect timing data for tight loops, because of the likelihood of perturbation; however it may be possible to collect timing data in these situations with an extension of our approach. Our current approach calculates the minimum distances between potential timing points before the optimizer is run, so the effects of optimizations are not accounted for. It would also be possible to calculate minimum distances after the optimizer is run, in case the optimizer has increased the size of the loop through optimization such as loop unrolling or inlining.

The primary disadvantage of such an approach is that the minimum distance between timing points will have to be recalculated for every binary generated: a timing point is acceptable if the minimum distance to the nearest timing point is sufficiently large for *every* instance of that timing point across *all* binaries. A potential timing point can be evaluated only after compilation and optimization of all binaries. Timing instrumentation code is generated only for selected timing points, so code generation for all binary versions must be delayed until all binary versions have been compiled and optimized. Alternatively, timing instrumentation can be patched into the binaries in a post-processing step. To avoid these engineering complexities, the baseline approach described in this section calculates minimum timing point distances once, regardless of the number of binaries generated, by performing the calculation before the optimizer is run.

5.4. Collecting Timing Data

The loop selection algorithm is run before optimization, and it identifies timing points and places lightweight code markers at those timing points. Next, the optimizer is run, and afterwards the lightweight code markers are replaced with full timing instrumentation code.

The optimizer is free to transform each lightweight code marker as it desires, but it must respect the semantics of each lightweight code marker. Before optimization, each lightweight code marker increments a specially allocated global variable by one, but after optimization the increment amount could be another constant, or it could be variable. As a simple example, if a `X++` marker is inserted into a loop, and the loop is unrolled 4 times, the optimizer should merge the 4 copies of the `X++` marker into a single `X += 4` marker. As another example, if a `X++` marker is inserted into a loop that always iterates N times, the optimizer should hoist the code marker out of the loop, resulting in a `X +=` N marker.

In these cases where a marker's increment amount is greater than 1, additional code is generated that records the code marker's increment amount. For example, if a code marker is found that increments by N, additional code will be generated that records the dynamic value of N, in addition to the code that collects timing data. Timings collected in this manner are *merged timings*, because they represent multiple loop iterations.

Before the performance auditor's statistical analysis consumes the timing data, a simple preprocessor normalizes merged timings so they can be compared across compilations. The preprocessor splits a single merged timing T that represents N loop iterations into N timings of T/N. So, for example, if a merged 10ms timing represents 10 loop iterations, that merged timing will be expanded into 10 timings of 1ms each.

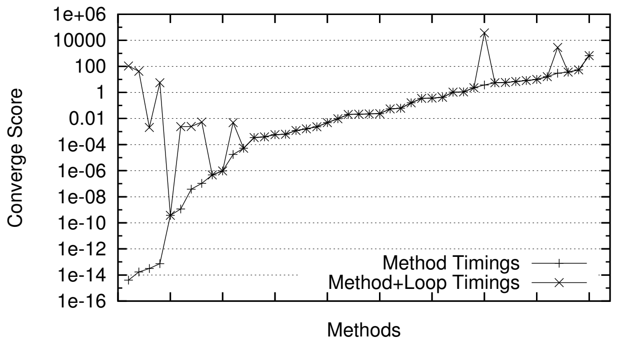


Figure 4. Converge scores for hot methods, higher is better

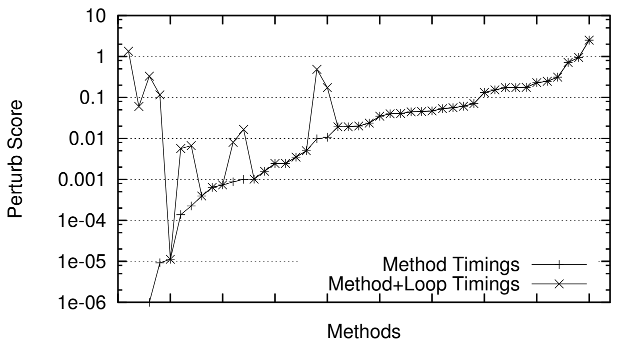


Figure 5. Perturb scores for hot methods, lower is better

6. Evaluation

This section evaluates the effectiveness of our technique for collecting comparable loop timings. We present three sets of results. Section 6.1 presents a metric analysis, where we evaluate the quality of our loop timings based on their statistical properties. Section 6.2 presents an accuracy analysis, where we determine if our loop timings can detect speedups and slowdowns as effectively as method timings. Finally, Section 6.3 presents a convergence study, where we determine if our loop timings will improve convergence time in a performance auditing system.

6.1. Metric Analysis

This section evaluates the loop selection algorithm using metrics that measure the primary objectives discussed in Section 5: collecting enough, but not too many, data points. We want to collect a sufficient number of timings so that the statistical analysis has enough data to converge quickly, yet we do not want to start and stop the timer too frequently, because the timer has a minimum resolution, and the accuracy of the timer decreases as we approach the minimum resolution.

Thus, we define two metrics to evaluate the effectiveness of the loop selection algorithm: *converge score* and *perturb score*. Converge score models how quickly the set of timing data is likely to converge when run through our statistical analysis. Higher is better. It is computed by dividing the number of timings collected by the variance in timings:

$$convergeScore = \frac{numTimings}{variance(timings)}$$

The amount of time required for our statistical analysis to determine if one set of timings is faster than another is determined by the number of timing points available, and the variance in the timing data. So, when more timing points are available, or the variance in timing data decreases, the statistical analysis converges faster. Thus, the converge score metric is defined so that timing data with higher converge scores will tend to converge more quickly.

Figure 4 shows converge scores for our hot methods. Results are shown for method timings, and method+loop timings (Section 3 described how our loop timings always include method timings). This figure shows that we only collect loop timings when necessary — methods with poor convergence using method timings (the left 1/3 of the graph) are improved by our loop selection algorithm, while most of the remaining methods are left alone. There are two mispredictions on the right side of the graph, where we collect loop timings when they are not actually needed.

There are three methods on the left that could benefit from loop timings, yet we do not collect loop timings for these methods. These three methods were discussed in Section 5.3: our prediction scheme correctly predicts that loop timings are needed for these three methods, but our timing point selection algorithm is unable to find code points where timing instrumentation can be safely inserted into these methods, because these three methods spend all their time in tight loops.

Figure 4 shows that our loop timings are effective. We find that the addition of loop timings drastically increases the number of timings without significantly affecting the variance in timings. This results in a large boost to converge score for methods selected for loop instrumentation.

Our second metric is *perturb score*. It models the perturbation introduced by the increased frequency of timing collection. The minimum observable timing with our cycle counting infrastructure (MIN_TIMING) is 88 cycles on our test machine, which is the number of elapsed cycles we observe between two consecutive reads of the cycle counter. If our timing data contains many timings that are close to the minimum observable timing, then they are timing very little program execution and are most likely inaccurate.

We define perturb score as follows:

$$perturbScore = \frac{\sum \frac{100}{timing-(MIN\_TIMING-0.1)}}{numTimings}$$

The perturb score metric is defined so that timing data that contains many timings close to the minimum possible timing will have a high perturb score. The maximum possible perturb score is 1000, which occurs when every timing we collect is the minimum of 88 cycles. The constant 100 normalizes perturb scores so that a perturb score of 1 or lower is likely to be acceptable (If the perturb score is 1, then the average timing collected is 187.9 cycles).

Figure 5 shows perturb scores for our hot methods. Increases in perturb score are inevitable when loop timings are used, since the loop timings require starting and stopping the timer much more often. This figure shows that our loop selection algorithm increases the number of timings for the key methods without having a significant negative impact on perturbance. The perturb score for the leftmost loop timing data point is 1.34, which corresponds to an average cycle count of 168.5, which is still nearly double the minimum observable timing.

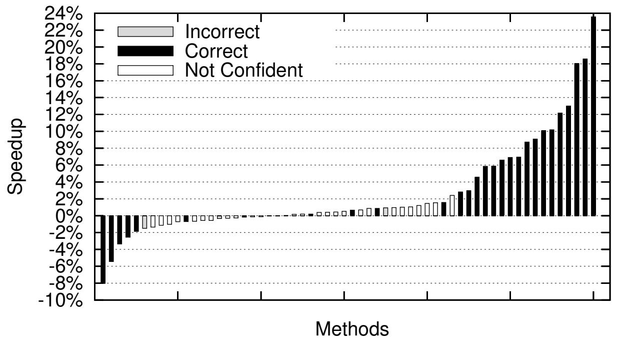


Figure 6. Accuracy of our approach, where markers are inserted before optimization, and markers are replaced with full timing instrumentation after optimization

6.2. Accuracy Analysis

To evaluate the accuracy of our approach, we collected timing data for various optimization settings (for example O1 vs O2) and compared their performance to see which is faster. For each hot method, we first evaluate its performance across different optimization settings using timings collected with method-only timing instrumentation. This result is considered the correct answer because the method-only instrumentation perturbs the execution the least, and is our most accurate timing mechanism.

We also evaluate the performance of the hot methods across different optimization settings with timings collected with method+loop instrumentation, and we check if the two techniques agree on the performance difference between the differently compiled versions of each hot method.

If both techniques indicate a speedup, or both techniques indicate a slowdown, the method+loop timings are correct. But if one technique indicates a speedup and the other indicates a slowdown, or vice versa, the method+loop timings are incorrect. If method+loop timings indicate that the magnitude of the speedup or slowdown is less than 1%, the method+loop timings are not confident.

Figure 6 presents the accuracy of method+loop instrumentation. This figure shows the overall accuracy of our approach compared to the method-only instrumentation used in [11]. This figure shows the overall accuracy of our approach, including the effects of loop timing point selection, code marker insertion, and replacing code markers with loop timing instrumentation. Hot methods are on the x-axis, and the y-axis shows the speedup detected by the method timings (i.e., the true speedup). The bars are solid black if the method+loop timings are correct, gray if they are incorrect, and white if they are not confident. This figure shows that,

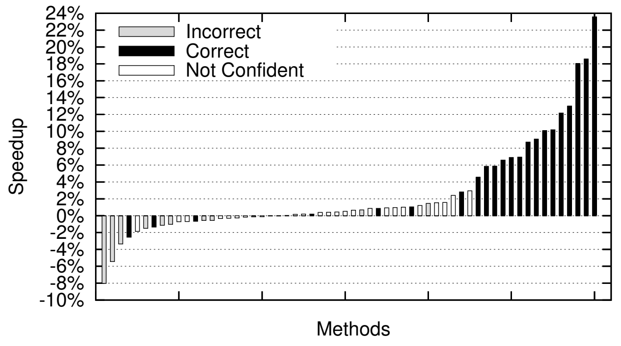


Figure 7. Accuracy of the naïve approach, where full timing instrumentation is inserted before optimization

overall, our approach is highly accurate. The worst-case error is a failure to detect a 1% speedup.

To demonstrate the importance of using lightweight markers, Figure 7 shows the accuracy of the naïve approach — inserting full timing instrumentation before optimization, then running the optimizer.

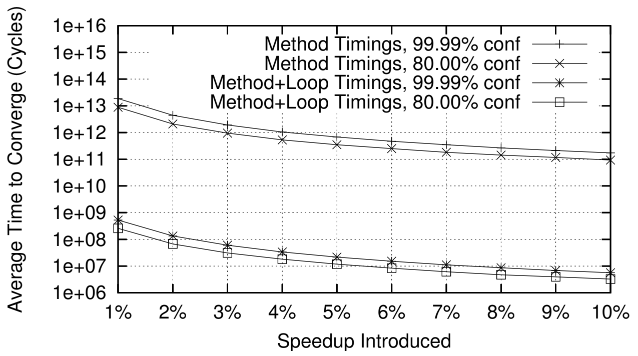
The naïve approach results in significant changes to optimization decisions, resulting in incorrect performance evaluations. With the naïve approach, the worst case error is failure to detect an 8% slowdown. Our use of lightweight markers reduces the optimization perturbation, and enables significantly more accurate speedup detection.

6.3. Convergence Study

To demonstrate the importance of increasing the number of timings, we calculate the amount of time needed for our statistical analysis to detect a range of speedups. This is similar to the offline convergence study presented in [11].

For each hot method, we collect method and method+loop timing data. We take each set of timing data, and randomly partition the data into two sets, A and B. Since A and B came from the same data set, the average timings in each set should be very similar. Next we apply an artificial speedup of $X\%$ to the timings in set B. At this point, the timings in set B represent an optimized version of the method that runs $X\%$ faster than the original.

We set a confidence threshold $Z\%$, 80% and 99.99% in this study, and we run our experiment by adding samples to A' from A, and to B' from B until we are $Z\%$ confident in our performance prediction. We start with 100 samples each in A' and B', and perform the confidence evaluation. If the confidence is below our threshold, we add 100 more samples to both A' and B', and repeat. We continue adding more samples to A' and B' until the confidence is above our threshold.



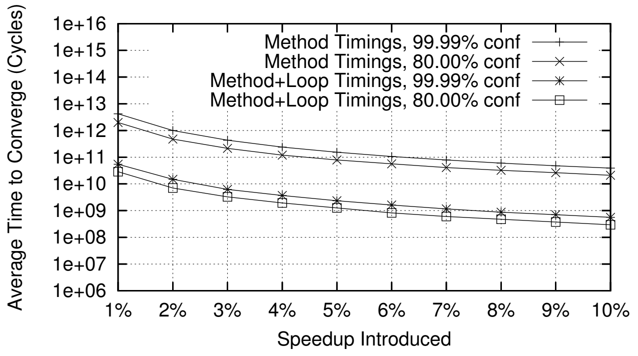


Figure 8. Convergence time, comparing method+loop timings to method timings. Top figure: convergence time for hot methods selected for loop instrumentation. Bottom figure: convergence time for all hot methods

Unlike the offline convergence study in [11], for this experiment we collect enough timing data so we always have enough samples to reach the desired confidence threshold.

When the statistical analysis confidently detects a performance difference, we report the time to converge, which is just the sum of the cycle counts in A' and B'. We repeat each convergence experiment 100 times with different random seeds and average the results.

The top graph in Figure 8 shows convergence time only for methods that were selected for loop instrumentation. This graph shows that loop timings improves convergence rate for infrequently invoked methods by about four orders of magnitude on average, which is consistent with the results of Figure 4.

The bottom graph in Figure 8 shows convergence time for all hot methods, including methods that were not selected for loop instrumentation. A relatively small number of hot methods require loop instrumentation, so when methods that were not selected for loop instrumentation are included in the study, the results are more modest. Still, loop timings improve the overall average convergence rate of all hot methods by about two orders of magnitude on average.

7. Related Work

This section discusses related work in mapping between a program's source and binary. We discuss two general approaches to the problem: *matching*, which uses heuristics to match the source and binary, and *bookkeeping*, which requires the optimizer to maintain this mapping.

The primary advantage of the matching approach is that it works if there are source-level changes. Both the bookkeeping approach and our new technique are intolerant of source-level changes. The primary disadvantage of the matching approach is that it is very difficult to achieve high accuracy, especially if different optimizations are used, and that it is fragile — matching is driven by heuristics, and the heuristics will likely need to be updated as compilers change.

The primary advantage of the bookkeeping approach is accuracy. The optimizer has full knowledge of all the transformations that it performs to convert the source to binary, so it can maintain a fully accurate mapping between the two. The primary disadvantage is that this approach modifies the optimizer to track all of its transformations, which is difficult and time-consuming, because it requires fairly detailed understanding of every optimization.

7.1. Matching with Heuristics

The first approach uses heuristics to correlate the source and binary. Wang et al. [18] propose BMAT, a technique to match binaries. They use BMAT to recycle profile data: if quality profile data has been collected for an old binary of a program, but programmers have made relatively small source-level changes to the code and produced a new binary, their system can effectively use the old binary's profile data with the new binary.

The matching algorithm is complex, yet it works well for their application. We believe this is because they are considering source-level changes, not optimization-level changes — the differences in their binaries shouldn't be as drastic as the changes we see when we change optimization settings.

We experimented with a matching algorithm driven by bytecode index data, which is similar to line number information, and loop structure for our application. We compiled two versions of the same source with different optimizations, and we attempted to match the loops in the two compiled versions. We found it very difficult to achieve over 70% match accuracy, and we require much higher accuracy to collect comparable loop timings for performance auditing.

7.2. Bookkeeping: Modifying the Optimizer

Albert [1] collects profile data from optimized binaries. For the compiler to use the profile data, a mapping from basic blocks in the optimized binary to source-level basic blocks is required. Basic block split, merge, and delete operations are tracked to build and maintain a mapping from blocks in the optimized binary to source-level blocks.

In Albert's work, only basic block operations are modified, which greatly reduces the amount of effort required to implement this solution. However, this approach also decreases its accuracy. Heuristics are needed to prevent information loss, because it is impossible to accurately track block mappings across block operations without higher-level knowledge of what the optimizer is doing.

Engblom et al. [5] present a variant on the modify-the-optimizer approach: they define their optimizer in a domain-specific language (Optimization Description Language, ODL), and generate code for their optimizer from ODL. Because they are generating code for their optimizer, they can easily generate additional code to accurately maintain mappings between source and binary. The downside of this approach is that the optimizer must be written in ODL, and ODL is not general enough to describe all possible optimizations.

We considered modifying the optimizer to maintain a mapping between source and optimized binary for our work, but we decided it would be very difficult to achieve high accuracy with such an approach in the J9 optimizer.

7.3. Heavyweight Code Markers

A variant of lightweight code markers have been used to implement on-stack replacement (OSR) [8, 6]. In a system that supports OSR, certain points in a method are designated as OSR points, i.e., points where a mapping back to unoptimized code is available. Like lightweight markers, OSR points are inserted into the IR before optimization and the semantics of inserted instructions dictate what optimizations are possible. Unlike lightweight markers, OSR points will likely preclude many optimizations because they are modeled similar to a call instruction that uses all live variables needed to recover unoptimized state. Because they will impact the applicability of optimizations, they are not a viable solution to the problem addressed by lightweight markers.

8. More Possibilities

Lightweight code markers are a new solution to the problem of correlating a program's source and its optimized binary, and they have unique tradeoffs compared to existing approaches. They should prove useful for much more than just improved performance auditing.

Albert [1] modifies the optimizer to track basic block operations to collect a source-level block profile from an optimized binary. This should be easy to do with code markers. We simply insert a code marker into every basic block and allow the optimizer to freely manipulate the markers. Regardless of where the markers end up, when we find a marker that increments its global variable by N, that corresponds to the marker's original block being executed N times (as described in Section 4), so we can replace each marker with appropriate instrumentation code to collect a source-level block profile.

BMAT [18] uses binary matching to reuse stale profile data with new versions of a program. The matching approach is key because it works when there are source-level changes, unlike the bookkeeping approach and our approach, although these two approaches are superior to the matching approach when optimizations can change. But with code markers' ability to map between source and binary, we can do *source* matching instead of binary matching. Source matching is much easier than binary matching, because the matching algorithm only needs to worry about source-level changes at source-level — the effects of compilers and optimizers do not have to be considered at all.

Perelman et al. [13] propose a system to meaningfully compare the results of accelerated architectural simulations when differently-compiled binaries are used for the same program. The different binaries are used to compare the advantages of different ISAs as well as compiler optimizations on future processor designs. Architectural simulators are very slow, so accelerated architectural simulation simulates many small representative samples of program behavior to approximate whole-program simulation. In [13], the challenge is to find the same sample of program behavior in differently-compiled binaries produced from the same source code. The authors use a matching system driven primarily by profile data. High accuracy is achieved, but special profiling runs of each binary are required. Applying a code marker-based approach to this problem should allow for high accuracy without requiring profiling runs.

9. Conclusions

This work presented lightweight code markers, a new approach to correlate a program's source and its optimized binary, and algorithms to determine when and where to insert code markers to improve performance auditing. We evaluated these techniques in a production Java virtual machine and showed that they improve the effectiveness of a performance auditing system. We also discussed how lightweight code markers could be applied to related problems.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful feedback. This research was funded in part by DARPA contract No. NBCH30390004, and by NSF grants CNS 0311683 and CCF 0541434.

References

[1] E. Albert. A transparent method for correlating profiles with source programs. In *2nd ACM Workshop on Feedback-Directed Optimization (FDO-2)*, Nov. 1999.
[2] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, Feb. 2005. Special issue on Program Generation, Optimization, and Adaptation.
[3] http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench.
[4] http://pag.csail.mit.edu/daikon.
[5] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating worst-case execution time analysis for optimized code. In *EuroMicro Workshop on Real-Time Systems*, June 1998.
[6] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *The International Symposium on Code Generation and Optimization with Special Emphasis on Feedback-Directed and Runtime Optimization*, pages 241–252, 2003.
[7] N. Grcevski, A. Kilstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.
[8] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. *ACM SIGPLAN Notices*, 27(7):32–43, July 1992. In *Conference on Programming Language Design and Implementation (PLDI)*.
[9] http://softwarecommunity.intel.com/isn/Community/en-US/forums/thread/30235396.aspx.
[10] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *International Workshop on Mining Software Repositories (MSR)*, May 2006.
[11] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: using hot optimizations without getting burned. *SIGPLAN Not.*, 41(6):239–251, 2006.
[12] P. Nagpurkar, M. Hind, C. Krintz, P. F. Sweeney, and V. Rajan. Online phase detection algorithms. In *The International Symposium on Code Generation and Optimization*, Mar. 2006.
[13] E. Perelman, J. Lau, H. Patil, A. Jaleel, G. Hamerly, and B. Calder. Cross binary simulation points. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2007.
[14] http://www.sable.mcgill.ca/software/#soot.
[15] Standard Performance Evaluation Corporation. SPECjbb2000 Java Business Benchmark. http://www.spec.org/jbb2000.
[16] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks. http://www.spec.org/jvm98.
[17] The DaCapo Project. DaCapo Benchmark Suite, version beta051009. http://www-ali.cs.umass.edu/DaCapo/gcbm.html.
[18] Z. Wang, K. Pierce, and S. McFarling. BMAT - a binary matching tool for stale profile propagation. *Journal of Instruction-Level Parallelism*, Apr. 2000.
[19] http://xml.apache.org/xerces2-j/index.html.
[20] X. Zhang and R. Gupta. Matching execution histories of program versions. In *International Symposium on Foundations of Software Engineering (FSE)*, Sept. 2005.