McGill University
School of Computer Science
Sable Research Group

Dynamic Profiling and Trace Cache Generation for a Java Virtual Machine

Sable Technical Report No. 2002-8

Marc Berndl and Laurie Hendren

October 17, 2002

Contents

List of Figures

List of Tables

Abstract

Dynamic program optimization is becoming increasingly important for achieving good runtime performance. One of the key issues in such systems is how it selects which code to optimize. One approach is to dynamically detect *traces*, long sequences of instructions which are likely to execute to completion. Such traces can be stored in a trace cache and dispatched one trace at a time (rather than one instruction or one basic block at a time). Furthermore, traces have been shown to be a good unit for optimization.

This paper reports on a new approach for dynamically detecting, creating and storing traces in a Java virtual machine. We first elucidate four important design criteria for a successful trace strategy: good instruction stream coverage, low dispatch rate, cache stability and optimizability of traces. We then present our approach which is based on *branch correlation graphs*. Branch correlation graphs store information about the correlation between pairs of branches, as well as additional state information.

We present the complete design for an efficient implementation of the system, including a detailed discussion of the trace cache and profiling mechanisms. We have implemented an experimental framework to measure the traces generated by our approach in a direct-threaded-inlining Java VM (SableVM) and we present experimental results to show that the traces we generate meet the design criteria.

1 Introduction

Although program optimization has traditionally been performed by static compiler optimizers, recent work has shown the power and effectiveness of dynamic optimizers which find and optimize important or "hot" pieces of code during execution of the program. Critical to the effectiveness of a dynamic optimizer is how it selects which code to optimize. It must quickly find a relatively small set of instructions that cover a maximal amount of the stream of executed instructions. This paper suggests a refined trace-based strategy applicable to any bytecode virtual machine. Like the successful Dynamo optimizer [3] we run entirely in software; however, our technique produces trace sequences similar to those found in the rePLay [9] hardware architecture. Specifically, we describe a light-weight, dynamic trace cache and profiling implementation for an interpreting Java Virtual Machine which finds, constructs, and updates a set of traces extracted from profiling data. We present experimental results supporting that the trace cache contains the set of appropriate traces for optimization and that the profiling and trace generating mechanisms have a reasonable overhead cost.

Our approach combines some successful ideas from previous dynamic optimizers. Like rePLay, we search for traces which tend to execute to completion, and like Dynamo we focus on minimizing program profiling overhead. The main differences between our technique and theirs are: 1) instead of using simple counters or extensive 32-bit branch histories we maintain recent branch/branch correlation statistics and use them to calculate where to delimit traces, 2) we use a dynamic profiler placement strategy to minimize the overhead of our profiling, and 3) we target a Java Virtual Machine while they target a particular native instruction set.

Our approach is based on a JVM interpreter, and is different from a standard JIT compilation approach. In a standard JIT approach, frequently executed methods (or parts of methods) are compiled and optimized on-the-fly. This method-by-method approach fits very nicely into traditional static optimization techniques, where most optimizations and transformations are applied at the basic block or procedure level. Thus, JITs can leverage traditional static optimizer technology. Our approach is different in that we use a modified JVM interpreter to find and construct highly optimizable traces. Traces do not correspond to basic blocks or method bodies, but rather correspond to long, frequently executed, sequences of instructions. These traces seamlessly cross basic block and method boundaries and have an ideal unit shape for dynamic optimization. Unlike static optimization which must consider all possible control flow paths, the traces provide instructions along a specific dynamic path, thus potentially provide better opportunities for optimization.

The contributions of the paper include:

- We discuss four design considerations which must be taken into account when designing a trace construction algorithm which targets a dynamic trace cache.
- We provide the specification for, and simulation of, a new hybrid trace generation algorithm/statistical branch correlation profiling mechanism. This approach has three novel aspects:
  - we model program behaviour with a branch correlation graph;
  - we control trace completion rate by calculating its expected rate during trace construction; and
  - we minimize profiling overhead with a dynamic profiler placement scheme.
- We experimentally measure the behaviour of our approach by modifying a JVM.

The rest of this paper is organized as follows. In Section 2 we give a discussion of related work. Section 3 provides a discussion of constraints imposed on a trace construction algorithm targeting a dynamic software trace cache and we introduce the basic ideas behind our approach. In Section 4 we provide details about our new profiler and trace construction algorithm. In Section 5 we discuss experimental results: in Section 5.3 we show that the algorithm effectively finds reasonable traces and in Section 5.4 we demonstrate that the algorithm is practical, providing evidence that the overhead is low. Finally, Section 6 gives conclusions and future work.

2 Related Work

This section describes several recent projects which have focused on the problems of dynamic hot code selection. The three works which are most closely related to the approach presented in this paper were: Dynamo [3], rePLay [9], and Whaley's work on online instrumentation and feedback-directed optimization of Java [13]. These three papers provide independent confirmation of the effectiveness of several of the techniques discussed in the paper.

While our technique finds sequences which can contain code from many methods, Whaley's technique [13] finds the *not-rare* basic blocks within hot methods. His dynamic optimizer identifies hot code in two phases: first discovering hot methods and then flagging the executed blocks in those methods. First, counters are placed at the method entrance points and back edges. When the counter passes a threshold the entire method is compiled using a baseline compiler. Second, the baseline compiler resets the counter and instruments all the method's basic blocks so that they are flagged when executed. When the counter passes a second threshold all code which was at some point executed are considered *not-rare* and is compiled by the optimizing compiler.

The Dynamo optimization system is a software HP PA-8000 instruction set interpreter running on a HP PA-8000 [3]. Like Whaley's approach it finds initially finds hot code by placing counters on back edges and other potential hot points; however, instead of methods and basic blocks, Dynamo focuses on the construction of traces, long frequently executed sequences of instructions extracted from the instruction stream which terminate on program back edges. Their intuition is that after a counter indicates that a point has become hot the instructions executed immediately afterwards often define a frequently executed sequence. These unoptimized traces are placed into a trace cache of such traces which are optimized and relinked to jump to other traces as appropriate. The interpreter, on reaching the hot point, dispatches the recorded trace. The resulting trace cache tends to cover most of the instruction stream; however, often the tail of a trace remains unexecuted. Due to their efficient use of counters Dynamo runs an average overhead of only 1.5%, dominated entirely by the trace generation mechanism.

While Dynamo's treatment of branch behaviour is lightweight, our use of a branch correlation graph allows us to verify that a sequence is in fact frequently executed. We end trace construction when adding another bytecode will likely cause the resulting trace to execute incompletely.

The proposed rePLay system also prefers the construction of completely executing traces and it simulates a powerful hardware framework to construct completely executing traces which they call frames. Frames are constructed by finding heavily biased branches, and replacing them with assertions. Trace construction is very similar to the Dynamo approach except that trace construction stops on reaching a branch which has not been promoted to an assertion. Branch bias is determined by considering its branch history. A branch which takes the same edge 32 times in a row when correlated with its 6 preceding branches is treated as heavily biased and is replaced with an assertion. Sequences of these assertion are then recorded by the framework from which it constructs frames. Frames are excellent targets for very aggressive optimizations because no calculations of values for branches which leave the frame are needed. The framework handles assertion failures with a rollback buffer which, on an assertion failure, restores the system state to what it was before the frame began executing and continues execution with unoptimized safe code. The resulting frame cache covers the instruction stream very effectively; furthermore frames can be aggressively optimized because they tend to execute to completion.

RePLay has few efficiency concerns because in the proposed framework profiling, frame construction, and frame rollback all happen in hardware, running parallel to the process. Like rePLay we model the program by examining branch history; however, we work entirely in software and thus are limited in the branch history depth available to us. We model program behaviour with a branch correlation graph which provides less branch history, but more detail per branch.

Other related work, which is less directly related, includes the following. Mojo [5], Microsoft's dynamic optimization engine, claims to use different stop conditions than Dynamo. DyC suggests a technique for dynamic code specialization which uses programmer annotations to provide specialization points and techniques [8]. The problem of limiting the instrumentation overhead is often handled with sampling. For example Traub et. al. suggest an ephemeral technique for sampling using self modifying code which periodically inserts and removes profiler hooks [12]. Arnold et. al. suggest instead that code versioning be used to control the execution of profiled code [1, 2]. Significant work has gone into examining the specific problem of path profiling including the classic work by Ball and Larus [4]. Path Based Compilation suggests techniques for profiling fixed length paths [14]. Duesterwald and Burening studied different optimization units and concluded that traces that inline small methods provide an optimal dynamic unit of optimization [6].

3 Generating Good Traces for Cached Dispatch

The purpose of this section is to motivate our choice of trace construction algorithm. We describe the effect of a trace cache on a Virtual Machine and present four design considerations that should be taken into account when designing an effective trace cache. We discuss the impact of the constraints on our design and introduce our approach which is based on *branch correlation graphs*.

3.1 Trace Caches and Java Virtual Machines

A software trace cache acts as a specialized instruction cache which reorders blocks to minimize the number of taken branches. An ordinary virtual machine interpreter dispatches one instruction at a time. In Figure 1 we illustrate this, each node in the graph corresponds to an instruction, and a dispatch happens for each edge in the graph.

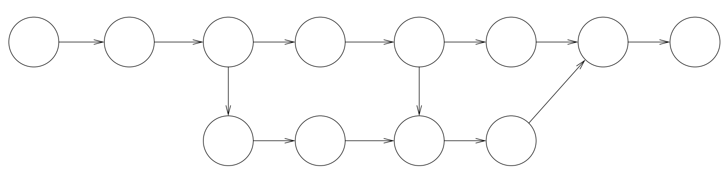

Figure 1: An Interpreter Normally Dispatches one Instruction at a Time

A direct-threaded-inlining interpreter finds all the basic blocks in the program and appends dispatch code as illustrated in Figure 2 [10]. In this case there is only one dispatch for each edge between basic blocks. Each dispatch executes an entire basic block and each basic block contains the code necessary to dispatch its successor.

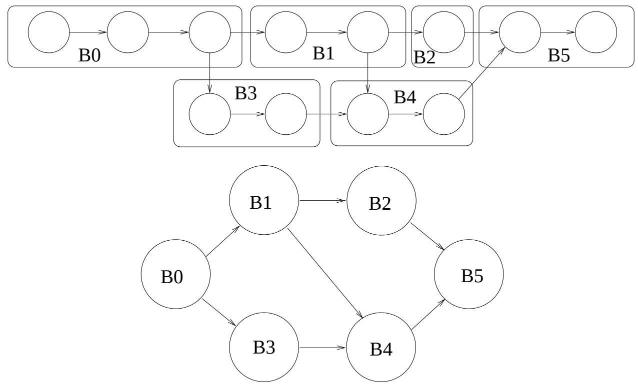

Figure 2: A Direct Threaded Inlining Interpreter Dispatches Basic Blocks

A virtual machine which implements direct threaded inlining can be readily modified to dispatch cached traces instead of basic blocks. A trace cache is a natural extension to such an interpreter because a trace can be thought of as an extended basic block; by placing the code in sequence we eliminate the need for dispatch code between inlined basic blocks.

Dispatching traces instead of blocks can improve Virtual Machine performance in several ways: it drastically decreases the number of dispatches by inlining sequences of frequently executed blocks together. This improves the performance of the underlying processor by improving code locality and increasing fetch bandwidth. In addition to these inherent improvements, traces can be excellent targets for dynamic optimization. However, a trace cache is only as effective as the traces in it.

3.2 Design Constraints

There are four trace-cache design constraints which guide our trace construction algorithm. Our most important concern is we want a small trace cache which contains traces covering as much of the instruction stream as possible, i.e. good instruction stream coverage. Second, we want a low dispatch rate; we want to execute many blocks without having to take a branch. Third, we want the trace cache to be stable; we need to minimize the number of times we replace traces because they no longer satisfy the previous two criteria. Fourth, we want to select traces which we can optimize aggressively; to do so we need to be able to predict the trace's future behavior. In addition to all these constraints we need for our technique to have a low overhead; our mechanism is implemented in software so we compete directly with the process we are trying to optimize.

3.3 Instruction Stream Coverage

Good instruction stream coverage is an extremely important factor for effective trace caches; the impact of a trace cache is strictly bounded by the amount it is used (i.e. the proportion of executed code that comes from the trace cache). Furthermore, practical concerns require that the cache contain as little rarely executed code as possible; the handling this code wastes both time and cache space.

In our approach, we profile our code to separate all the regions of code which are frequently executed from those which will have little impact on the programs performance. Both rePLay and Whaley's profiler both handle this problem by profiling at the basic block level. Whaley flags blocks as *not rare* while rePLay requirement of 32 consecutive branches in the same direction correlated with a depth 6 branch history implicitly proves that the branch is hot. We flag blocks which have not yet been executed more than the *start state delay* number as *newly created*.

3.4 Dispatch Rate

A low dispatch rate is the key to an efficient optimization model; fewer dispatches lead to less execution overhead. Dynamo and rePLay handle the problem of a obtaining a low dispatch rate differently: Dynamo assumes that instructions executing after a hot node will probably re-occur, while rePLay checks that the trace consists of extremely biased branches. The fact that we are working software and the polymorphic nature of Java code [11] where we find a virtual method call approximately every 9 bytecode instructions that suggests that neither of these methods are directly appropriate. Dynamo's technique is extremely lightweight; however, polymorphic branches are traditionally harder to predict than direct branches.

RePLay's use of powerful hardware solves the branch prediction problem; however, hardware parallelism is not available to our software JVM. A compromise is needed; something which strengthens Dynamo's assumption at a lower cost than rePLay's solution.

3.5 Branch Correlation Graphs

Our solution is maintain is a branch correlation graph (BCG), effectively a depth one per address history table. Each new branch we encounter is added a new node in the graph; thus for each pair of blocks (X, Y) executed in sequence we include node N_{XY} in the graph. Each node has execution counter and a tag that summarizes its state. Successive branches are recorded as directed edges; thus for each sequence of of blocks (X, Y, Z) we have a directed edge E_{XYZ} from N_{XY} to N_{YZ} in the graph. Each edge E_{XYZ} has a counter used to store the degree of correlation between branch XY and branch YZ. We carefully represent blocks, nodes, and edges to minimize memory overhead and maximize lookup speed.

Initially we set the counter to the start state threshold and tag N_{XY}'s state as *newly created*. We then wait for counter to reach 0 before declaring it "not rare". This has a similar effect to basic block flagging in Whaley's optimizer; it filters out branches which are rarely executed so we can focus our attention on branches likely to be executed again.

After a branch node becomes hot we examine the edges to determine if there is a strong branch correlation between it and its successors. We store the result of our examination in the branch's state tag. Our *strong correlation* tags serve a similar purpose to assertions in rePLay; they indicate that if we have executed N_{XY} it is reasonable to follow branch N_{YZ} in a trace. The branch correlation graph is a compromise, it uses less resources than rePLay but provides more assurance of the regularity of the trace than Dynamo.

3.6 Cache stability

The branch correlation graph allows us to perform informed trace cache maintenance so we can ensure its performance without changing the traces in it too often. The BCG allows us to make refined choices about where to begin and end our traces, if we discover that the behavior of a branch has changed we can quickly find all the code it might affect and limit any changes to those traces. This is handled very differently in Dynamo and rePLay. In rePLay it is handled in hardware. Dynamo has a mechanism which detects the rapid creation of new traces and simply flushes the trace cache.

3.7 Optimizable Traces

The appropriate optimization for a trace depends on its runtime behavior; we wish to construct traces whose runtime behavior maximizes the power of the optimizations available for it. All traces are very optimizable because they only have one entry point. Both Dynamo and rePLay generate sequences whose branches are, on average unlikely to be taken. However, by ensuring that all the branches are taken with a very high probability rePLay allows for stronger optimizations. Traces which have a well bounded completion frequency can be optimized under the assumption that improving the performance of the trace along the main branch can be traded off against costs along branches which exit the trace. However, to ensure that these these speculative trade-offs are worth while, rePLay requires 32 consecutive branches with respect to the 6 branch history. While this precise mechanism is not practical in software, the 16 bit counters in the branch correlation graph can be used to very accurately estimate the probability that a trace will complete. This is the basis for our trace construction algorithm; we limit the length of traces by ensuring that their probability of completion is high. If a trace contains sequence of branches $N_{X_0X_1}, N_{X_1X_2}, ..., N_{X_{k-1}X_k}$, we can calculate the probability that a sequence which enters $N_{X_0X_1}$ will execute completely by taking the product of the correlations between nodes $E_{X_0X_1X_2}, E_{X_1X_2X_3}, ..., E_{X_{k-2}X_{k-1}X_k}$ ie. multiplying all the edge weights together and dividing by the product of the node weights.

3.8 Summary of our Approach

Our approach to trace construction has three parts: we profile the branches and filter for those which have executed a threshold number of times before declaring them "not rare", then when a branch behavior changes we traverse the graph to find all the highly correlated branches related to the affected branch, and finally we regenerate the traces by partitioning the branches into sequences of blocks whose probability of completion above the trace completion threshold. We discuss the implementation and details of our approach in the next section.

4 Implementing Trace Generation

Our trace cache technique relies on three mechanisms. The first is the use of direct-threaded-inlining in the interpreter (we have based our implementation on SableVM [7]), the second is a profiler and the third is the actual trace cache mechanism.

A direct-threaded-inlining interpreter dispatches code one basic block at a time by appending the dispatch code to the basic block. The profiler works by augmenting the dispatch code to update profiling information and including lazy hooks into the profiling code. The trace cache responds to a series of signals from the profiling mechanism. In this paper we provide details of the profiler and our trace cache's trace constructor, which are new ideas. For details of direct threaded inlining see the work by Piumarta and Riccardi [10] and for details of how this is implemented in SableVM see the thesis by Gagnon [7].

4.1 Profiling Mechanism

The profiling mechanism maintains a branch correlation graph, as introduced in Section 3.5, and summarizes the important branch correlation state changes to the trace cache. One can view the branch correlation graph as a description of the behaviour of the program's branches which are kept current through the use of decay.

4.1.1 Decay

Each branch context node N_{XY} has a series of branch correlations edges E_{XYZ}. Each branch correlation E_{XYZ} stores the probability of taking branch (Y, Z) given that last branch taken was (X, Y). The stored correlations and their states are not cumulative, but rather they are weighted in favour of recent behaviour in the program. This is implemented via periodic exponential decay. Every 256 executions of a branch, all the correlations associated with a branch are shifted one bit to the right. This maintains the relative ratios of the distributions while doubling the importance of current branches. During the decay process the correlation state of the branch is rechecked; we verify that the cached branch correlation is the best correlated one and that its state has not changed. Only the maximally correlated branches are interesting to the trace cache; the branch correlations are summarized to the trace generator with one of four states. In descending degree of correlation they are: *unique*, *strongly correlated*, *weakly correlated*, and *newly created*. If either the maximally correlated branch or its state changes the profiler signals the trace cache to update itself.

4.1.2 Profiler hooks

Part of the profiler is executed with each trace dispatch. Thus, there is an interesting connection between improving dispatch overhead and minimizing profiling overhead. As the system finds better traces, the number of dispatches is reduced. This also eliminates profiling points and thus profiling overhead.

We use lazy construction to keep the memory overhead low because during execution many potential branches and branch correlations are never realized. The interpreter's hook into the profiler is the branch context pointer which reflects the last branch taken by the program. Cached in the branch context is the address of the instruction currently believed to be most likely to be executed and a reference to the corresponding branch correlation.

If the PC and the expected instruction do not match, the PC is checked against the list of previously unpredicted, but previously encountered, branches for the branch context. If it is not found in this list, then the branch has never been encountered before in this context, and a new branch correlation is constructed and inserted into the branch context. The branch context is then checked for decay and the appropriate branch correlation is updated. Then the new context is loaded from the branch correlation.

These steps keep the expected per dispatch overhead low in three ways. First, the number of function calls is limited to the number of distinct correlations discovered during that execution plus those needed to implement periodic decay. Second, most of the branches are immediately predicted by the branch context's inline cache greatly limiting searching required. Third, each branch correlation contains the address of its target branch context so finding the correct correlation immediately allows us to find its respective branch context.

4.2 Trace Cache Mechanism

The trace cache ensures that its traces maintain a minimum threshold of probable completion by reconstructing the appropriate traces when it receives a state change signal from the profiler. When the profiler indicates that the state of a certain branch has changed eg. the branch is no longer strongly correlated but now weakly correlated, the trace cache takes the following three steps:

1. a list of all possible trace entry points which may be affected is generated;

2. a list of new traces to be created is generated; and

3. the new traces are compared to those in the cache and all newly discovered trace cache entries are reconstructed.

In the first step we find the trace entry points by back tracking in the branch correlation graph along all strongly correlated edges. In the terminal element list generally there is only element; however, this technique ensures that everything that may be affected is examined. This list now consists of all those branch contexts which are likely to eventually execute the modified branch.

For efficiency, the second and third step and interleaved. The path of maximum likelihood is followed from each start point until it either discovers a branch already in the trace or a weakly correlated branch. This path now contains the set of instructions which may affect the trace cache starting from that entry point. If the path terminates in loop, that loop is processed first; it is unrolled once and passed to the trace cache. The remaining instructions are then cut into blocks whose cumulative completion probability is above the trace completion threshold.

The block parsing mechanism works by linearly adding bytecodes to the block until, by adding another block, the traces probability of completion would be lower than the completion threshold. After a block is constructed it is passed into the trace cache, which checks if the trace cache hash table already contains that sequence. If so, the trace is retrieved and linked. Otherwise a new trace is constructed and linked into the code. Finally, to prevent cascades of state changes, all the instructions found to be related to the process have their state updated as their trace is currently up to date. Although technically this algorithm could theoretically require the reconstruction of the entire trace cache at any one step, we found that the number of traces generated per signal was rarely greater than five and often less then one in ten.

By waiting for signals from the profiler the trace cache needs only construct that which needs reconstruction as it knows that the remainder of the trace is relatively up to date. By traversing the branch correlation graph it ensures that everything that may need updating has, in fact, been examined.

5 Experimental Results

We tested our algorithm on six benchmarks and found that it generated long traces which covered a large percentage of the execution stream with a low execution overhead. We performed our tests using an implementation based on SableVM, a direct threaded inlining Java interpreter designed for speed. The implementation added our trace cache dispatch approach to SableVM and allowed us to examine the behaviour of the trace cache. The benchmarks were all run on a HP Pavilion N5495 with 512MB of RAM running at 1.06GHz.

5.1 Benchmarks

We chose four SPECjvm benchmarks (`javac`, `compress`, `mpegaudio`, and `raytrace`); one large application, `soot`; and one scientific application, `scimark`. `Compress` and `raytrace` are simple programs which exhibit predictable behaviour. `Javac` is traditionally one of the more challenging SPECjvm benchmarks, selected to show the strength of the model. We selected `soot`, a bytecode-to-bytecode analysis and optimization framework, as a an example of a large real world application. Finally, we selected `scimark` to examine the effectiveness of the technique on scientific calculations.

5.2 Parameters and Dependent Values

The algorithm has two important parameters: *start state delay*, and *threshold*. The *start state delay* determines how many times a branch must be taken before it can be included in a trace, larger values filter rarely executed code while smaller values increase the responsiveness of the algorithm to phase changes. The *threshold* controls the minimum expected trace completion rate. We say that a trace completes if it executes to completion, once started. The *threshold* also controls how the profiler detects state changes. A low completion threshold generates longer traces and many signals from the profiler, whereas a high completion threshold produces fewer signals and more predictable traces.

For our experiments we ran the benchmarks with delays of 1, 64, and 4096 branches. A delay of 1 indicates that branches immediately are declared "not rare". We found that a delay of 64 gave the best results, and so we then examined the results for a variety of threshold values. For the threshold we tried values of 100%, 99%, 98%, 97%, 95%. With a signal threshold of 100% the algorithm does not distinguish between the unique and strong states.

We measured five dependent values:

average trace length: The average executed trace length is measured by finding the sum of the lengths of the traces which execute to completion and dividing by the number of traces. A high average executed trace length indicates that the algorithm is finding traces appropriate for optimization.

instruction stream coverage: The instruction stream coverage is found by dividing the total number of instructions executed by completed traces by the total number of instructions executed in the program. Stream coverage indicates the portion of the code being executed from the trace cache and bounds the performance gains available.

dynamic trace completion rate: The dynamic trace completion rate is calculated by dividing the number of completed traces by the number of entered traces. A high completion rate suggests that the optimizer can implement optimizations with a performance trade off e.g. if a trace's completion rate is over 99% then a optimization which doubles performance along the completion path can degrade performance off the completion path by a factor of ten and still improve performance by 40%.

state signal rate: The state signal rate is the frequency at which the profiler detects changes, the lower the signal rate, the fewer times that the trace cache needs to check its consistency.

trace event interval: The trace event interval is calculated by dividing the number of instructions executed over a programs life time by the number of traces constructed by the trace cache plus the number of signals generated by the profiler. The greater this interval, the lower net overhead of the trace cache.

5.3 Effective Trace Generation

The purpose of the experiments in this section is to demonstrate that for suitably selected values for the threshold, the traces in the trace cache are suitable units for optimization. This analysis is done in two steps: first we verify that we can choose values which generate long traces that effectively cover the execution stream, then we validate our preference for completely executing traces by showing that the trace completion rate is very high.

Tables I and II examine the effect of the threshold on the average trace lengths and instruction stream coverage respectively. The trace lengths are given in terms of the number of Java basic blocks, which consist of a number of Java bytecode instructions. Note that some bytecode instructions, such as `invokevirtual`, can correspond to a large number of machine instructions. The data in these tables supports our assertion that we can find values which make our algorithm effective.[1]

threshold	`compress`	`javac`	`raytrace`	`mpegaudio`	`soot`	`scimark`	average
100%	5.0	2.9	2.9	3.1	3.2	10.8	4.7
99%	12.0	4.0	8.0	3.4	3.9	10.8	7.0
98%	12.0	4.1	8.1	3.4	4.3	10.8	7.1
97%	12.1	4.3	8.4	4.8	4.5	10.8	7.5
95%	11.7	5.9	8.5	5.3	4.8	10.8	7.8

Table I: Trace Length vs. Threshold

Table I shows that the threshold has a negligible effect on the average trace length except when it is 100%. At 100% we see that the average trace length drops to less than 4.7 basic blocks per trace, and if we exclude `scimark` the average falls to 3.4. This is because only traces which branch over 2048[2] times in the same direction are considered inlinable. The fact that 4.6 blocks can be inlined suggests that there are a large number of branches which are never taken, eg exceptions. Thus any of the values between 99% and 95% seem to be reasonable choices in terms of trace length; however, instruction stream coverage suggests that 97% is best.

The most important values to notice in Table II is the benchmark average at a threshold of 97%. At this point we find the completed traces cover over 87.1% of the instruction stream. This value is quite pleasing as, for comparison, rePLay's trace cache captures between 82% and 90% of the instruction stream. Furthermore, because the trace cache also includes partially executed traces, it covers even more of the instruction stream than completely executing traces alone; the trace cache captures 90.7%..

Table III shows how frame completion rate varies with the threshold. From this data

[1] The current implementation does not handle virtual method call linkages as well as it should, and thus the values for `javac`, `mpegaudio` and `soot` are lower than they should be. We are currently modifying our implementation to fix this problem and the updated numbers will be released in our technical report at `http://www.sable.mcgill.ca/publications/#reports`, and will be in the final version of the paper.

[2] it takes up to $2048 = 256 log_2 256$ iterations to completely clear a history. $log_2 256$ shifts are needed and a shift occurs every 256 dispatches

Threshold	`compress`	`javac`	`raytrace`	`mpegaudio`	`soot`	`scimark`	average
100%	78%	72%	79%	90%	76%	98%	82.1%
99%	90%	73%	82%	90%	80%	98%	85.5%
98%	90%	76%	79%	92%	81%	98%	86.0%
97%	91%	79%	80%	92%	83%	98%	87.1%
95%	90%	77%	80%	90%	83%	98%	86.3%

Table II: Instruction Stream Coverage vs. Threshold

we can infer that for threshold values above 97%, the completion rate is sufficiently high to justify the more complex algorithm needed to search out completely executing traces.[3]

threshold	`compress`	`javac`	`raytrace`	`mpegaudio`	`soot`	`scimark`	average
100%	80%	89%	90%	99%+	95%	99%	92%
99%	99%	91%	91%	99%+	95%	99%+	96%
98%	99%+	93%	91%	99%+	95%	99%+	96%
97%	99%+	92%	92%	99%+	94%	99%+	96%
95%	99%+	91%	91%	99%+	92%	99%+	95%

Table III: Frame completion rate vs. Threshold

Our technique generates traces suitable for optimization. We found that we get best performance with a threshold of 97%; however, the algorithm is consistently effective is between 97% and 99%. It is interesting to note that 97% is also the threshold value implicitly used by rePLay, ie their choice of 32 consecutive branches as the assertion threshold.

5.4 Algorithmic Efficiency

In the previous section we showed that the trace cache behaves well. In this section we argue that the cost of the algorithm is acceptably low. The overhead induced by the algorithm is partitioned into three parts: the first is the cost of the trace construction mechanism, the second is the cost of the periodic checks induced by the profiler, the third is the cost of the augmented dispatch statement which provides the hooks into the trace generation mechanism. Of these three, we show that the dispatch statement dominates by a large margin. As the overhead induced by dispatch statements dominates, we then find the per dispatch overhead induced by the profiler dispatch code.

The cost of code dispatch dominates both periodic checks and trace construction, this is not surprising as over 95% of the overhead in Dynamo is profiling. The expected behaviour of the code dispatch is two comparisons, two pointer evaluations, and one assignment. This is fortunate because this code is executed after every block dispatched by the interpreter, which in the case of a direct threaded inlining interpreter means after every basic block executed

[3] 99%+ indicates value over 99.9%

by the program. Similarly a trace dispatch executes a single profiling statement, all of the inlined ones are removed. The periodic check function is significantly more expensive than profiler hook; however it is only called approximately every 256 dispatches. During the check function the branches are decayed and the state of the maximally predicted branch is evaluated. Since most branches in a program are direct there are, on average, only 2 branches to check and so the average execution time is well bounded. Even is we assume that average periodic check time is 25 times longer than a the dispatch code we still spend less than a tenth of our overhead here compared to the dispatch.

The trace construction mechanism can be significantly more costly than the periodic checks however it is only invoked when a branch changes state which occurs very rarely compared to the periodic checks. As Table IV shows at a 97% threshold we guarantee less than one signal per 11100 dispatches and on average less than one signal every 114700 dispatches.

Threshold	`compress`	`javac`	`raytrace`	`mpegaudio`	`soot`	`scimark`	average
100%	37.3	10.4	39.4	30.0	11.5	11.9	23.4
99%	39.8	11.0	41.7	31.6	10.5	369.3	83.9
98%	40.5	11.1	43.3	33.4	10.5	415.5	92.3
97%	38.0	11.1	43.3	31.6	10.5	554.0	114.6
95%	40.5	10.9	43.3	34.3	10.7	415.5	92.5

Table IV: Thousands of Dispatches per State Change Signal

Signals and trace generation are independent events. Most important is the total number of these events as any number of traces can be generated from a signal. As multiple (or no) traces can be generated from a signal the overhead can be dominated by either, so we must attempt to limit the rate of trace events of any kind. We can increase the time between events by adjusting the *start state delay*, the lag between the discovery of a new branch and its potential inclusion in a trace. As Table V shows as we increase the delay from 1 to 4096 the *trace event interval* falls off dramatically.

important than Although signals are lightweight, the costly procedure is trace construction.

Start Delay	`compress`	`javac`	`raytrace`	`mpegaudio`	`soot`	`scimark`
1	36.0	4.6	17.8	18.6	1.6	317.5
64	37.6	6.6	20.0	13.5	5.2	512.0
4096	32.1	12.4	25.5	30.6	6.0	468.1

Table V: Thousands of Dispatches per Trace Event at 97% *threshold*

From table V we find that a delay of 4096 seems to have the greatest event interval. At a delay of 4096 the interval between periodic checks, 256 dispatches, is 25 to over 1500 times smaller than the trace construction interval. The profiling overhead dominates the algorithm's execution time.

In order to calculate the profiling overhead, we modified SableVM to include the profiler code at the end of each *basic block*, and then we timed the unmodified interpreter vs. the profiling version. These results are given in Table VI. The result is that profiling appears to cost between 0.018 and 0.075 seconds per million dispatches. We then counted how many threaded dispatches would be made if the trace dispatching was used (traces include many basic blocks), and preliminary results indicate an overhead between 2% and 7%.

Across all benchmarks, as shown in Table VI, the per dispatch overhead is about 0.044s per million dispatches, while the per instruction overhead is around 0.154s per million dispatches. Thus roughly the cost of executing the profiling code 28.6% of the cost of executing a basic block.

Benchmark	No Profiler (sec)	#dispatches (millions)	Profiler (sec)	Overhead per 10^6 dispatches
`compress`	248	1906	303	0.029s
`javac`	123	621	158	0.058s
`raytrace`	204	866	269	0.075s
`mpegaudio`	240	2404	312	0.030s
`soot`	96	513	124	0.055s
`scimark`	261	3324	321	0.018s

Table VI: Profiler overhead per basic block dispatch in modified SableVM

In Table VII we substitute the per dispatch overhead values into the trace dispatch model to find the expected overhead induced by the trace dispatching model. From table VI we have an estimate of the per million dispatch overhead of the profiler. We calculate the overhead of a pure trace dispatching model taking the number of trace dispatches and multiplying it by the per dispatch cost. We find that the across all benchmarks the predicted overhead is quite acceptable, everywhere less than 7% and averaging 4.5%.

Benchmark	Trace Dispatches in Millions	Overhead per Million Dispatches	Expected Overhead	% Overhead
compress	142	0.029	4.12s	1.7%
javac	144	0.058s	8.35s	6.8%
raytrace	103	0.075s	7.73s	3.8%
mpegaudio	500	0.03s	15.00s	6.2%
soot	114	0.055s	6.27s	6.5%
scimark	308	0.018s	5.54s	2.1%

Table VII: Profiler dispatch overhead

Because trace events and periodic decay checks happen several orders of magnitude less frequently than profiling statements, profiler dispatches dominate overhead. Across a million dispatches we found the overhead of profiling the program to be 28.6%. By moving to trace dispatch we eliminate many of the dispatches, and significantly decreased the profiling overhead to between 1.7% and 6.5%.

5.5 Summary

Our algorithm produces generates long traces 7.5 basic blocks long on average, and it has good instruction stream coverage: 87.1% of the instruction stream is covered by traces which execute to completion and 91% of the instruction stream is executed within the trace cache. Furthermore, the overhead for our algorithm is acceptably low for use in a dynamic optimizer; the expected profiler trace dispatch overhead is low, 4.5% on average.

6 Conclusions and Future Work

In this paper we have presented a new approach for dynamically detecting, creating and maintaining an instruction trace cache. Central to our approach is the use of branch correlation graphs to locate highly correlated blocks that should be placed in the same trace. Our profiling mechanism uses a decay function to allow the system to adapt to more recent program behaviour, and the profiling code itself is attached to the dispatch code for each thread. Thus, as the trace cache builds better traces, the overhead for dispatch and profiling decreases.

We have implemented the mechanism in a direct-threaded-inlining JVM, and measured the behaviour of the trace cache. We found that with a delay of 64 and a threshold of 97%, we could find reasonably long traces with good instruction stream coverage and good trace completion rates. Furthermore, the trace cache was very stable, with very infrequent changes to the traces. Finally, we showed that the biggest overhead is from dispatch and profiling, but that by finding good traces this overhead could be kept to around 5%.

We are very pleased with our initial results, which shows that a software solution can lead to good traces and have reasonable overhead. Our next step will be to fully integrate the mechanism into SableVM, by enabling the VM to execute the traces we can find. We will first investigate performance improvements over the direct-threaded-inlined interpreter, and then we will measure what further improvement can be achieved by applying optimizations to the traces.

Acknowledgements

References

[1] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online instrumentation and feedback-directed optimization of java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '02)*, November 2002.

[2] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 168–179. ACM Press, 2001.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[4] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57. ACM Press, 1996.

[5] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gillies. Mojo: A dynamic optimization system. In *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[6] Evelyn Duesterwald and Derek Bruening. Exploring optimal compilations unit shapes for an embedded just-in-time compiler. In *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[7] Etienne Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, October 2002.

[8] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, October 2000.

[9] Sanjay J. Patel and Steven S. Lumetta. rePLay: A hardware framework for dynamic program optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001.

[10] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300. ACM Press, June 1998.

[11] Ramesh Radhakrishnan N. Vijaykrishnan Lizy K. John Anand Sivasubramaniam Juan Rubio Jyotsna Sabaqrinathan. Java runtime systems: Characterization and architectural implications. In *IEEE Transactions on Computers*, pages 131–146. 502, February 2001.

[12] Omri Traub, Stuart Schecter, and Michael D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard, June 2000. Unpublished technical report, available at: `http://www.eecs.harvard.edu/hube/publications/muck.pdf`.

[13] John Whaley. Partial method compilation using dynamic profile information. In *Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '01)*, pages 166–179, 2001.

[14] Reginald Clifford Young. *Path Based Compilation*. PhD thesis, Harvard University, January 1998.