MATS: Multicore Adaptive Trace Selection

Jason Mars
University of Virginia
jom5x@cs.virginia.edu

Mary Lou Soffa
University of Virginia
soffa@cs.virginia.edu

ABSTRACT

Dynamically optimizing programs is worthwhile only if the overhead created by the dynamic optimizer is less than the benefit gained from the optimization. Program trace selection is one of the most important, yet time consuming, components of many dynamic optimizers. The dynamic application of monitoring and profiling can often result in an execution slowdown rather than speedup. Achieving significant performance gain from dynamic optimization has proven to be quite challenging. However, current technological advances, namely multicore architectures, enable us to design new approaches to meet this challenge.

Selecting traces in current dynamic optimizers is typically achieved through the use of instrumentation to collect control flow information from a running application. Using instrumentation for runtime analysis requires the trace selection algorithms to be light weight, and this limits how sophisticated these algorithms can be. This is problematic because the quality of the traces can determine the potential benefits that can be gained from optimizing the traces. In many cases, even when using a lightweight approach, the overhead incurred is more than the benefit of the optimizations. In this paper we exploit the multicore architecture to design an aggressive trace selection approach that produces better traces and does not perturb the running application.

1. INTRODUCTION

The goal of dynamic optimization is to take advantage of information available only at runtime to perform optimization. At the core, this information usually involves identifying hot regions of executing code. When using bytecode with source level information, these regions can take the form of a frequently executed method, as in Java dynamic optimizers [18]. On the other hand, when dynamically optimizing a native binary, a basic block trace representing a path through the control flow graph of the binary is used, as in Dynamo RIO [3]. We call the regions in the latter example a *trace*.

A hot trace is formed by taking a frequently executed sequence of basic blocks and coalescing them into a single unit for optimization with one entrance and multiple exits [2, 8]. The benefits of forming traces, among others, include improved instruction locality, branch elimination, compaction, and new optimization opportunities.

The potential benefits of dynamically optimizing traces depend heavily on the quality of the traces produced. To evaluate the quality of traces, a number of metrics have been proposed. These metrics include the amount of code duplication, the number of transitions off traces, the amount of time spent on traces, and the length of the traces [2, 6, 8], In addition to these we propose another metric, the amount of *ideal trace executions*. An ideal trace execution occurs when a trace is executed to completion. In many cases it is critical to get ideal trace executions because the key of many trace optimizations is the assumption that the entire trace will be executed. If there is an early exit from the trace, *compensation code* must be executed [13, 2]. Executing compensation code can diminish the possible improvements achieved by optimizing a trace.

The two primary challenges of identifying hot traces include the obtrusion of the executing application and the development of good analysis techniques to produce high quality traces. In current systems, detection of hot traces is achieved through monitoring the application execution using instrumentation [2, 8, 6, 3]. This instrumentation usually requires saving some process state, executing injected code, and restoring process state.

When stealing cycles from the application, a second problem arises. To minimize the overhead of monitoring the application dynamically, simple trace analysis algorithms are used. This further constrains the quality of traces that are produced. If too much overhead is incurred, there may be no overall performance improvement, and in some cases performance degradation can occur.

With the advent of multicore architectures comes new opportunities for dynamic optimization. In this paper, we propose a new approach for trace selection. Our approach takes advantage of the multicore architecture to address both of the challenges mentioned above. Using the multicore architecture, we design a cross core trace selection technique that produces better traces and incurs no direct overhead to the running application. Using a variation of currently available hardware structures, our approach can monitor application execution, analyze its branch behavior patterns, and construct high quality traces without instrumenting the application. This is achieved by moving the application monitoring, execution path analysis, and trace selection, into a separate core. By doing this our approach can take advantage of idle cores while our application executes undisturbed.

We experimentally compared our approach with the widely used next executed tail (NET) algorithm [3, 2]. We show that our approach produces traces that cover 33% more dynamic instructions (e.g. 33% more instructions are executed within traces), and more than doubles the amount of ideal trace executions, with 2.2x more ideal trace execution coverage.

The specific contributions of this work are as follows:

- An analysis and study of *ideal trace executions* and a demonstration of its importance as a trace evaluation metric.

- A multicore framework for the implementation of trace selection techniques.

- New trace selection algorithms that are more aggressive and produce better traces than previous work.

- An experimental analysis of the improvement of traces when using more sophisticated algorithms versus the more traditional techniques.

Next, Section 2 describes the multicore configuration of our approach and discusses cross core monitoring. Section 3 motivates our trace selection technique and describes the desired characteristics of the traces we produce. Section 4 describes our pattern based aggressive trace selection approach. In Section 5 we describe our experimental setup, evaluate our approach and present preliminary results. In Section 6 we present related work, and finally, we conclude in Section 7.

2. CROSS CORE MONITORING

The key requirement of our approach is the ability to monitor and react to the behavior of an application that resides on a neighboring core. This would allow our trace selector to perform more complex and expensive computation without directly effecting the application. This new capability is the key of our trace selection technique. We first develop a cross core monitoring framework, and then we develop algorithms that produce better traces than current techniques.

The question arises as to how we achieve cross core monitoring and whether there are algorithms that produce better traces than current techniques.

The hardware/software interface of the proposed log-based architectures [5] could be used for the cross core monitoring. Unfortunately there is significant hardware overhead when implementing log-based architectures, and thus they have not been implemented in real systems. We propose a different approach. The branch trace buffer (BTB) of the Itanium and Core 2 Duo processors have proved to be a useful tool to detect the dynamic basic block trace of an application [4, 10]. However current multicore hardware configurations do not allow cross core reading of a core's branch trace buffer. This capability can be achieved by placing the individual buffers together and assigning a core id to each buffer. In this way a trace selector could simply specify which BTB it would like to read. The branch trace of the host application is read by our trace selector thread that may reside on a neighboring core. We call this the *chip-wide branch trace buffer* (CWBTB).

In figure 1, the CWBTB is shown. It is a centralized unit that is globally readable. Through the use of the CWBTB, our trace selection technique can seamlessly collect the targets and PC's of the branches of the application. We use this dynamic program behavior to observer, collect and form traces unlike any that are currently formed by modern dynamic optimizers.

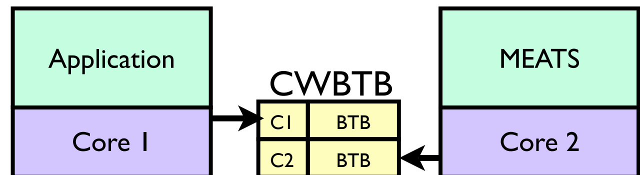

Figure 1: The Core Wide Branch Trace Buffer serves as a centralized performance monitoring tool that is available to all cores.

3. HOT TRACE SELECTION

To understand the need for better dynamic trace selection techniques, we first consider what is currently used and the characteristics of the traces they produce.

```
foo()
{
  for(int i=0; i<8; i++)
  {
    if(i<4)
    {
      block A
    }
    else
    {
      block B
      unpredictable();
    }
  }
}
```

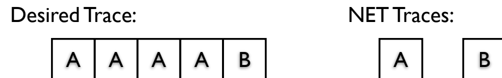

Figure 2: This is a partially unrolled trace.

3.1 Next Executing Tail

The next executed tail (NET) algorithm described by Bala et al. [2] NET has 2 phases, a profiling phase and a trace collection phase. In the profiling phase, each branch that is a backedge is instrumented. A counter is kept for each of these backedges and incremented every time that backedge is taken. When this counter hits a predefined threshold, the next trace is collected. To collect the trace, the code is instrumented and monitored block by block until another backedge is executed. Note that it is possible to collect a cold path during this collection even if another, more hot, path was primarily responsible for reaching the backedge.

A number of shortcomings of the NET approach have been outlined by previous work [2, 6]. Most notably, NET traces cannot span the code before a interprocedural call and its corresponding return. Also, NET traces cannot span outer and inner loops. In addition to these limitations, there are others that have not been fully addressed by previous work. When collecting a trace, that trace is ended on the first backedge; for this reason traces only average 2-3 basic blocks in length. When using NET, it is impossible to collect com-

```
foo()
{
  for(int i=0; i<20; i++)
  {
    block A
    if(!i%3)
    {
      block B
    }
    else
    {
      block C
    }
    block D
    unpredictable();
  }
}
```

Desired Traces:

NET Traces:

Figure 3: Here we have the competition for selection.

```
foo()
{
  for(int i=0; i<4; i++)
  {
    block A
    if(!i%2)
    {
      block B
    }
    else
    {
      block C
    }
    block D
  }
  unpredictable();
}
```

Desired Trace:

NET Traces:

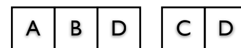

Figure 4: This is a trace of a complex path through a loop.

plex control flow paths that revisit blocks. As we will show, our approach overcomes these limitations by using the global context of the trace.

3.2 Trace Characteristics

There are some key trace characteristics we aim to achieve that are not present in NET traces or other dynamic trace selection algorithms. The first is the ability to form traces that contain partially unrolled loops. For example, in Figure 2, we have a function `foo` with a loop that would be partially unrolled into a trace by our technique. This loop has the same behavior for the first few iterations and then the behavior is erratic. If `foo` is called often, a trace `AAAAB` could be formed, partially unrolling the loop. This kind of trace cannot be formed by NET; Net is only able to produce two small traces that span 1 basic block each, presenting no code optimization opportunities. Our approach is able to detect the desired trace and apply optimization.

Another trace characteristic we aim to capture with our technique is to take advantage of the global temporal context of each trace. For example if we have two hot paths with the same head block, we want to have information about which is 'more hot.' For example in Figure 3, we have 2 candidate hot streams: `ABD` or `ACD`. As can be seen, `ACD` is executed twice as often as `ABD`; therefore it is hotter. NET is not able to identify the hotter trace. Our technique, however, is able to observe this. As we will discuss shortly, our approach has an entire window sample to analyze and automatically collects information about the number of occurrences of each potential hot stream. This window view can be very beneficial when electing which trace to use dynamically.

Beyond identifying what traces are hotter to break ties, we would also like to collect traces that span multiple paths through a loop. In Figure 4 we show a code example where we would like our approach to generate the trace `ABDACD` as one trace. This kind of trace selection has not been done by software dynamic optimization. However this type of pattern sometimes presents itself in executing loops. This is one of the main features of our technique and it is the key to the power of our technique. The information necessary to detect this trace is present when considering the global context of the branch sequences.

4. PATTERN BASED ADAPTIVE TRACE SELECTION

Our trace selection approach produces traces with the desired characteristics mentioned in section 3. This is achieved by using pattern detection to extract hot sequences of basic block executions. When inspecting taken branch targets over a period of time, patterns emerge. These patterns in taken branch targets represent a sequence of basic blocks executed in succession. Using our cross core technique, recurring sequences of basic blocks can be detected and traces can be formed without instrumenting the application. This presents a solution that is unobtrusive to the application's execution.

4.1 Using Patterns

The first step is to collect a small history of taken branch targets called a branch target window. We use our pattern detection algorithm to learn candidate trace patterns from this window.

To detect patterns we use the sequitur algorithm, developed by Nevill-Manning et al. [14] The sequitur pattern detection approach is very well suited to our problem because it is fast (linear execution time), concise, and intuitive. Sequitur builds a tree that represents the hierarchical structure from any sequence of elements. Grammatical symbols are associated with subsequences in the elements that occur more than once, and a hierarchical structure emerges. Whether these are sequences of numbers, letters, or any other data objects that can be compared, the sequitur algorithm can build a tree where each node in the tree can

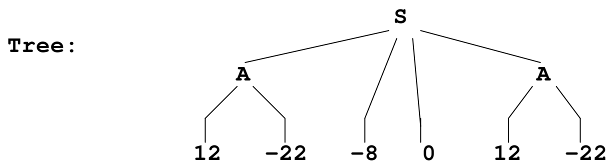

Figure 5: An Example of Sequitur

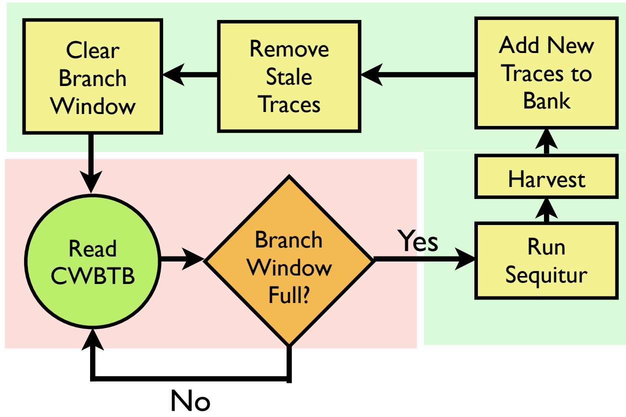

Figure 6: Our Adaptive Trace Selection Approach

be expanded to a subsequence that occurs in the window of elements more than once. For example in Figure 5, a simple example of a sequitur tree is built from a small number of elements. In this example, one pattern emerges: `12 -22`. Sequitur associates the grammatical symbol `A` to this pattern. Grammatical symbols represent our candidate traces.

The sequitur pattern detection algorithm is a part of our trace selection approach. Beyond detecting recurrent subsequences, we must also harvest these candidates traces that meet predefined criteria; only a subset of these are chosen for final selection.

4.2 Adaptive Trace Selection

Hot traces are placed in a *trace bank*. The trace bank holds the currently active traces. When a trace becomes cold, i.e. it has not recently been executed, it is invalidated and removed from the trace bank. Remember that since we are not obtruding our application, we can perform this trace bank management with no penalty to the application.

In Figure 6 we give an overview of our adaptive approach. In the monitoring phase, we simply probe the CWBTB, collect new branch targets, and place them in the taken branch window. The branch window size can be fixed or dynamically resized to adjust to the desired pattern granularity. In our case this size ranges between 20 and 500 branch targets.When the taken branch history window is full, we go into analysis phase. Here we run our pattern detection algorithms, harvest for good trace forming patterns, add them to the trace bank, remove stale traces, clear the current window, and return to monitoring phase.

Our approach also tries to automatically detect the best window sizes for producing a number of patterns that is space efficient, but also captures the relevant hot regions of code. In our experiments we have set the ideal number of hot traces to detect per window to be between 5 to 30. This means if we detect any less than 5 hot traces we increase the window size; if we find any more than 30 we decrease the window size.

5. EVALUATION

To evaluate our approach, we use the Pin instrumentation framework [11]. Pin provides an API interface to the dynamic profiling of native application binaries. Using this API we implement our baseline trace selection algorithm, as well as our multicore enabled approach. This evaluation methodology has been validated by previous work [6].

With Pin we are able to simulate the core wide branch trace buffer (CWBTB) by simply instrumenting the branches of the application to send the instruction address and branch target of every executed branch to our trace selection engine. Our trace selection engine then analyzes this branch information to execute our trace selection algorithms and generate the necessary statistics.

For our baseline approach we use the next executing tail algorithm. Similar to previous work, the benchmarks we use are the SPEC2006 test suite [6]. We compile the SPEC benchmarks using GCC 4 on Pentium Xeons.

5.1 Metrics

The first metric we use to evaluate and compare our trace selection techniques is the percent of time, represented by dynamic instructions, spent on traces. If traces are formed but infrequently executed, little benefit can be gained. The second metric involves the *ideal trace execution* metric. As mentioned before, an ideal trace execution occurs when a trace is executed in its entirety.

The final metric we use to evaluate our trace quality is the length of our traces. We identify exactly how many basic blocks our typical trace spans. Longer traces present better optimization opportunities which can better be exploited when we have ideal trace executions.

5.2 Results

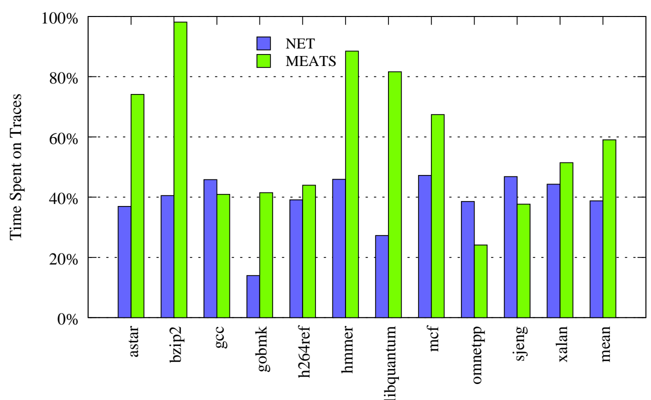

Figure 7: Here we show the percent of dynamically executed code spent on traces.

In Figure 7, we compare the dynamic code coverage of our approach to NET. In this graph we consider traces that span

at least 2 basic blocks as most code optimization opportunities on a single block can be exploited by the static compiler. As can be seen in Figure 7, our approach outperforms NET by a considerable margin in all but 3 benchmarks. On average, we spend 33% more time on traces than NET.

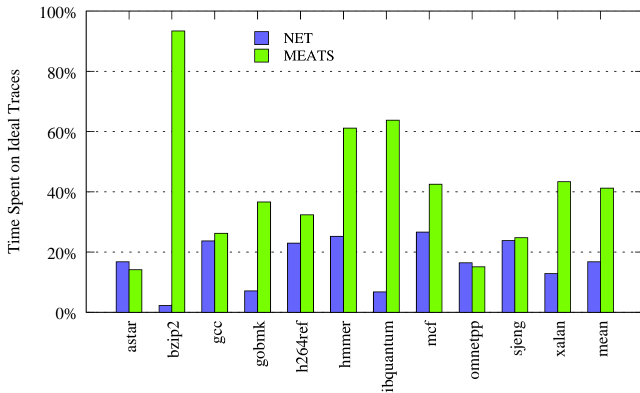

Figure 8: Here we show the percent of dynamically executed code spent on ideal trace executions.

Our ideal trace execution cover is also quite good compared to NET's. Figure 8 shows that the amount of execution time spent executing traces to completion. Achieving more ideal trace executions is one of the major goals of our approach. Figure 8 shows that on average, we more than double the ideal trace executions of NET. This naturally raises the question of what the typical size for these 'ideal' traces are.

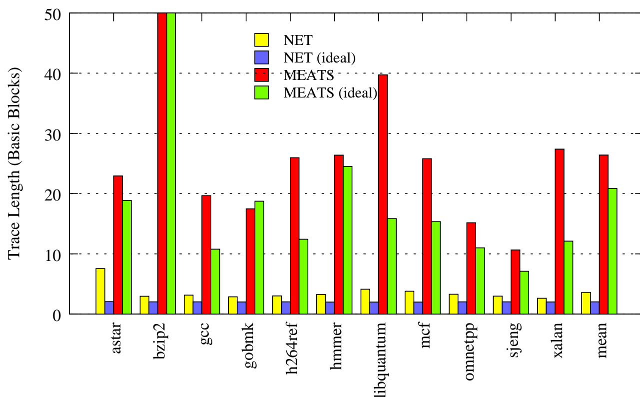

Figure 9: The trace lengths of our approach are significantly longer than NET's.

Our traces are also longer than NET's. In Figure 9 we show the average length of our hot traces. Our technique produces traces that are 3x to 10x as long as NET. This is due to the partial unrolling that can occur, and the complex paths that may revisit basic blocks. Most interestingly, the average length of even our ideal trace executions is quite a bit longer than those of NET's. Having such long traces creates much more optimization potential than those presented NET's traces.

6. RELATED WORK

In this section we first discuss the relevant research in dynamic optimization and then we discuss work dealing specifically with the runtime trace selection problem.

There are two classes of hot code based optimizers: those that are method based and those that are trace based. Dynamic optimization frameworks that target bytecode [15, 18, 1] detect frequently executed methods and identify them as hot. These hot methods are then compiled and recompiled at higher levels of optimization, depending on how often they are executed.

There is also a class of dynamic optimizers and optimization frameworks that deal with native binaries directly [2, 3, 17]. This class is known as binary translators. These dynamic optimizers typically form program traces and use these traces to apply dynamic optimization techniques.

Some research attention has been paid to dynamic optimization approaches using multicore architecture. Work in the Java VM community dealing with parallel recompilation shows the importance of intelligently scheduling the optimizer thread [9]. However, this work deals with the method level dynamic optimizers.

There has also been some research proposing a number of hardware extensions to support software dynamic optimization on multicore architectures [19, 20]. This work proposes that trace selection occurs entirely in hardware and uses a number of hardware extensions that, considering commercial interest, are not available now and may not be available in the future.

Trace selection and formation can occur in hardware specifically as a hardware optimization. This type of trace formation been well studied [12]. Techniques such as the trace cache [16] and trace preconstruction [7] use hardware functional units and memory buffers to form traces. However, these traces are not made available to software dynamic optimizers, thus limiting trace selection to hardware only.

Dynamic trace selection, namely the NET algorithm, was developed and used by Bala et al. for the Dynamo project [2], the first dynamic binary optimizer. Other software-based trace selection techniques have also been developed. Chen et al. [4] developed a technique to take advantage of the branch trace buffer of the Itanium to assist in trace selection. The Adore system [10] also proposes using the branch trace buffer to identify hot traces. These approaches deal more with the efficiency of detecting traces and less about the quality of the traces themselves. Hiniker et al. [6] developed the LEI trace selection algorithm which deals with the question of trace quality. However, this work had the same constraints of the NET algorithm of Dynamo, and thus a lightweight algorithm for trace selection was developed. While LEI produced better loop spanning traces than NET, it also shares many of NET's limitations.

7. CONCLUSION

The hot spot monitoring requirement of dynamic optimizers is one of its most expensive and inhibiting tasks. In this paper we show that taking advantage of the multicore architecture can enable us to develop more effective trace selection techniques while remaining unobtrusive to the executing application. In addition we have demonstrated that more sophisticated trace selection techniques can give us higher quality traces. We presented a novel online trace

selection approach that takes advantage of global branch target patterns to extrapolate longer traces that excel along all current metrics. Producing higher quality traces can enable beneficial dynamic optimization

Multicore architectures are evolving. One major new feature of this evolution is the capability to use hardware introspection. Through the use of performance monitoring hardware we are able to monitor the effects and behaviors of applications while incurring minimal to no overhead. We believe that this evolution is moving towards the inclusion of core wide hardware monitors, given the advantages they offer to software.

8. REFERENCES

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeño jvm. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[3] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[4] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 79–90, Washington, DC, USA, 2003. IEEE Computer Society.

[5] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 63–65, New York, NY, USA, 2006. ACM.

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

[7] Q. Jacobson and J. E. Smith. Trace preconstruction. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 37–46, New York, NY, USA, 2000. ACM.

[8] T. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Trans. Comput.*, 50(6):549–566, 2001.

[9] P. Kulkarni, M. Arnold, and M. Hind. Dynamic compilation: the benefits of early investing. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 94–104, New York, NY, USA, 2007. ACM.

[10] J. Lu, H. Chen, P. Yew, and W. Hsu. Design and implementation of a lightweight dynamic optimization system. *The Journal of Instruction-Level Parallelism*, 6, 2004.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[12] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. mei W. Hmu. A hardware mechanism for dynamic extraction and relayout of program hot spots. *SIGARCH Comput. Archit. News*, 28(2):59–70, 2000.

[13] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 174–185, New York, NY, USA, 2007. ACM.

[14] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[15] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 150–162, Washington, DC, USA, 2007. IEEE Computer Society.

[16] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.

[17] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[18] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, 2005.

[19] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society.

[20] W. Zhang, B. Calder, and D. M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 50–64, Washington, DC, USA, 2006. IEEE Computer Society.