Lengthening Traces to Improve Opportunities for Dynamic Optimization

Chuck (Chengyan) Zhao, Youfeng Wu†, J. Gregory Steffan, and Cristiana Amza

Department of Electrical and Computer Engineering
University of Toronto
{czhao,amza,steffan}@eecg.toronto.edu

†Corporate Technology Group (CTG)
Intel Corporation
youfeng.wu@intel.com

ABSTRACT

Program traces—multiple basic blocks that frequently execute in succession as a group—are a useful unit for code manipulation and optimization, especially in *dynamic binary translation* (DBT) systems. Since most of the overhead in such systems is related to managing control flow into and out of traces and performing translation and optimization the first time a trace is accessed, longer traces are desirable to amortize this overhead. More importantly, longer traces provide a larger window of instructions on which to operate when performing dynamic optimizations, potentially improving the impact of such optimizations. However, indiscriminately building longer traces will only increase the frequency of early exits—branches and jumps out of the middle of traces that require the execution of costly compensation code to ensure correctness.

In this paper we propose a method for lengthening program traces in dynamic optimization systems without significantly increasing the frequency of early exits. Our initial study of Spec2000Int benchmarks shows that we can increase the average size of hot traces by more than 50% (from an average of 33 to 50 instructions) while increasing the frequency of early exits by less than 1%. We demonstrate that conventional dynamic optimizations can significantly benefit from this increased trace size, in particular by showing that we can improve the effectiveness of local value numbering by 25% through unrolling only 5% of all existing traces. Furthermore we claim that longer traces can enable novel and aggressive speculative optimizations that capitalize on underlying hardware transactional memory support, potentially allowing legacy applications to exploit such support.

1. INTRODUCTION

Dynamic binary translation (*DBT*) [24, 34, 36, 41] is a runtime technique to translate binary code from a source architecture and execute it on a target architecture, providing a powerful abstraction layer for architecture and ISA independence. DBT can be used to run legacy binary code on modern architectures [43, 44, 45], providing binary-level compatibility without the need for recompilation or software re-engineering. DBT can also be used to provide ISA-level virtualization and enable architecture co-design [43]. DBT can even be used to implement or extend software transactional memory [38, 42, 45] systems that can support legacy code and libraries. One of the main challenges for DBT systems is minimizing translation overhead.

Rather than translate a single instruction or basic block at a time, DBT systems are often implemented to perform translation and optimization on the level of a trace [8, 9, 10, 21, 26]: a sequence of basic blocks that frequently execute in succession. A common implementation allows a trace to have only a single entry point but multiple exit points. Unconditional branches and jumps are *straightened* by rewriting the code to fall-through to the target basic-block, and then removing the original branch or jump. Conditional branches and jumps can also be straightened by rewriting the code to fall-through to the most common target basic-block; however, this requires the generation of costly recovery code to handle the case when the uncommon conditional path is taken, referred to as an *early exit* from the trace.

Operating at the level of traces provides many benefits for DBT systems such as removing uncommon code, increasing locality, and improving opportunities for optimization. However, trace size is a key issue: larger traces can better amortize the overheads of translation and provide more opportunities for optimization—but indiscriminantly creating larger traces will only increase the frequency of early exits, likely negating any benefit of the larger traces.

1.1 Lengthening Hot Traces

We propose to improve opportunities for optimization in DBT systems through a novel approach to lengthening traces. Our key insight is that *hot traces*, traces that execute frequently, can be dynamically measured for their suitability for *unrolling*: that is for traces that normally branch back to their own start instruction, such traces can essentially be unrolled similar to the unrolling of loops. The challenge is to unroll only hot traces with a very small frequency of early exits, since the high cost of such early exists is multiplied with each unrolling. Furthermore, when one hot-trace is frequently followed directly by another specific hot-trace, we can potentially combine them through a form of *straightening*—another technique which can provide lengthened traces. The advantage of DBT systems is that hot traces can be easily instrumented to directly measure trace hotness, trace size, and the frequency of early exits.

We have extended Intel's StarDBT dynamic binary translation framework [43] to measure the potential for lengthening hot traces through unrolling in selected Spec2000Int benchmarks. We show that a significant fraction of hot traces can be lengthened through unrolling with only a minor increase in the frequency of early exits. In addition we investigate the potential for two classes of dynamic optimization to benefit from lengthened traces.

Improving conventional optimizations: Lengthened traces provide a larger window of instructions for conventional dynamic optimizations to operate on. To demonstrate this opportunity we evaluate the potential for improvement for *local value numbering* [14], a common dynamic optimization which efficiently performs copy propagation, common subexpression elimination and dead code elimination.

Exploiting transactional memory: We claim that longer traces can potentially enable a novel use of transactional memory (TM) hardware support [23, 32] when the overheads might be too large otherwise. TM support can potentially be exploited to checkpoint and roll-back a trace, allowing a DBT system to perform aggressive speculative optimizations such as speculative code motion [6, 11, 33], while at the same time allowing legacy binaries to capitalize on hardware transactional memory support.

1.2 Contributions

In this paper we make the following contributions: (i) we show that a significant number of hot traces are amenable to lengthening through unrolling; (ii) we demonstrate that conventional dynamic optimizations can significantly benefit from increased trace size, in particular by showing that we can improve the effectiveness of local value numbering; (iii) we propose that longer traces can enable aggressive speculative optimizations to capitalize on underlying hardware transactional memory support.

2. RELATED WORK

Our techniques build on existing work in two main areas: i) binary translation and ii) trace collection and optimization. In the following we compare our work with the state-of-the-art in these two areas.

2.1 Binary Translation

Binary translation or binary rewriting is a well known method for instrumenting existing application code with the purposes of profiling or transparent optimization of legacy code. User-level instrumentation tools, such as Pin [34], Dynamo RIO [18], ATOM [41], and Valgrind [36] can be used to insert arbitrary instrumentation. Dynamic binary translation discovers basic blocks and indirect branch targets at runtime by following the application's execution. Thus, in contrast to static binary rewriting tools such as ATOM [41], dynamic binary translation does not need to determine the safety of inserting instrumentation by detecting control flow graphs a-priori. It also provides better support for self-modifying code. A similar technique is used by machine virtualization tools i.e., Virtual Machine Monitors, such as Xen [5] or VMWare [37], which dynamically rewrite instructions of the guest operating system.

2.2 Trace Collection and Optimization

Related tools that perform trace-based optimizations include the HP Dynamo system [4], Mojo [12], and DynamoRIO [18]. Most of these tools are based on binary translation, recognize frequently-executed binary traces at runtime, and perform dynamic optimizations on the recognized traces. Dynamo's trace optimizations focus on redundancy eliminations and cache utilization to benefit from traces' simplified control flow and are free of internal join points. Both Dynamo and Mojo include loop unrolling optimization, but the heuristics and mechanism for unrolling are not discussed.

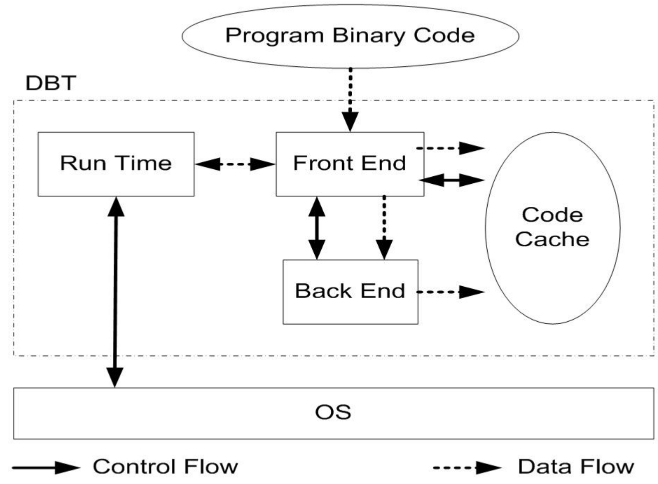

Figure 1: Overview of StarDBT

Specifically, in DynamoRIO when frequently executed traces are detected they are compiled, optimized, and then executed instead of interpreted. DynamoRIO provides four optimizations (copy propagation, dead-code-elimination, call-return matching, and stack cleaning) but does not lengthen traces through unrolling. While we use StarDBT [43] in our experiments, our ideas should be generally applicable regardless of the baseline trace-finding tool.

Our work is also related to recent work on trace collection and optimization based on Java byte-codes [7, 8, 9, 39]. Similar to our work, Bradel *et al.* [7] leverage traces for performing code optimizations—specifically inlining. Additional work includes characterizing traces in terms of trace length, dynamic program coverage, completion rates [8, 39], and available trace-level parallelism [9]. Their goal is to direct optimizations by determining the frequency of various instructions in traces [39] by predicting the control flow of a program [8] or by evaluating traces as a unit for automatic parallelization [9]. Their focus is mostly on off-line feedback-directed systems. In this paper, we provide support for performing aggressive trace-specific optimizations dynamically. These optimizations are both efficient and capable of obtaining cumulative benefits that are not available otherwise. At the same time, our techniques open up unique opportunities to exploit hardware transactional memory support to further leverage these optimizations.

2.3 Dynamic Binary Translation Framework

Our work is based on the StarDBT [43] dynamic binary translation system. The overall structure of StarDBT is given in Figure 1. StarDBT runs on top of the OS as a user-level run-time system. The program binary code is dynamically translated and stored in a code cache. StarDBT controls the execution of the translated code and then applies different dynamic binary translation techniques. StarDBT consists of a *Runtime* module, a *Frontend* module, and a *Backend* module. The Runtime module provides system supports. The Frontend module manages the execution for dynamic binary translation. The Frontend module also collects program profiling information during execution and selects hot traces based on the profiling information. The Backend module then performs run-time optimization on

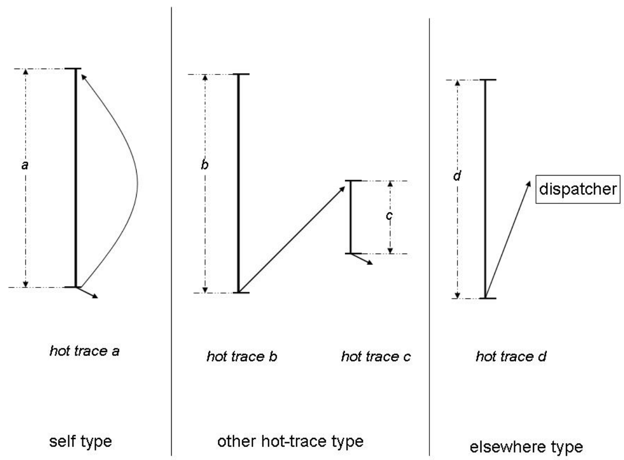

Figure 2: Types of main-exits: (i) to *self*, (ii) to *other hot trace*, and (iii) to *elsewhere*.

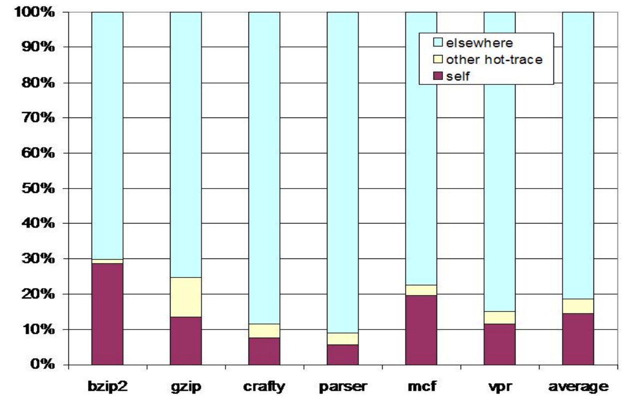

Figure 3: Distribution of hot trace main-exit types.

these hot traces by building an intermediate representation (IR), performing optimizations on the IR, and finally storing optimized code in the code cache.

StarDBT extends the well-known *Most Recent Execution Tail* (MRET) [3] approach for hot trace selection. In MRET, the hot trace heads are first identified based on profiling information. Each loop head (e.g. the backward branch target) is treated as a candidate trace head. Each candidate trace head is instrumented such that a counter is incremented after each execution of the candidate trace head. When the counter exceeds a certain threshold, the candidate trace head becomes a hot trace head. Then the hot trace is simply selected as the execution path from the hot trace head to the most recent execution tail (an instruction that satisfies certain trace tail conditions). Our trace selection is based on $MRET^2$, a two-pass improvement over the MRET. In the first pass, we use the MRET approach to select one trace only as a potential hot trace. We then clear the performance counter, restart the counting and select another potential hot trace using MRET in the second pass. We obtain two potential hot traces with the same hot trace head but possible different trace tails. Our $MRET^2$ approach selects the hot trace as the common path of the two potential hot traces, which is likely to have both hot head and hot tail.

3. HOT TRACE PROPERTIES

In this section we study the hot traces discovered by the StarDBT framework. We first categorize hot traces based on the type of main exit branch or jump. Next we measure the sizes of hot traces. Finally we analyze the frequency of early exits from hot traces, as summarized by the *completion ratio*. In this work we measure the subset of the Spec2000Int benchmark suite that worked reliably with the StarDBT system at the time of writing, using the *MinneSPEC* [30] input set.

3.1 Hot Trace Main-Exit Types

Any trace formed by StarDBT may have multiple *exits*—branches or jumps out of the common-path of the trace. The exit which occurs most often is called the *main* exit, while other exits are called *side-exits*. To identify traces suitable for lengthening we first classify hot traces based on the dynamic destination of the main-exit, as shown in Figure 2.

The type of hot trace that we are primarily interested in is the *self* type where the destination of the hot trace main-exit is the beginning of that same hot trace, similar to a loop: such traces can be effectively unrolled in a somewhat straightforward manner. The second case is where the common destination of the hot trace main-exit is some specific *other hot trace*. These cases can also be lengthened, although through a more difficult transformation explained later. Finally, the remaining cases are comprised of hot traces whose main-exit destination is not a hot trace but *elsewhere*, and hence StarDBT's runtime dispatcher is invoked on the execution following that main-exit.

Figure 3 shows the relative distributions of these three types of hot trace main-exits. On average, there are roughly **15%** *self* main-exits, **4%** main-exits to some *other hot-trace*, and 81% main-exits to elsewhere. Hence a combined 19% of hot traces can potentially be lengthened.

3.2 Hot Trace Size

StarDBT already builds relatively large and aggressive traces by employing the $MRET^2$ algorithm and supporting early-exits from traces (as described earlier in Section 2.3). Figure 4 show StarDBT's average hot trace size in number of instructions, which averages 33 static[1] instructions per hot trace across the benchmarks; note that each trace is comprised of roughly three to four basic blocks.

3.3 Hot Trace Completion Ratio

We are interested in lengthening hot traces by combining hot trace instances. However, doing so will increase the frequency of early-exits from hot traces which will invoke a greater amount of recovery code and potentially nullify any benefits gained from the larger hot traces. Hence we must

[1]Static trace size is defined as the total size of static traces divided by the total number of static traces, while dynamic trace size is a weighted average with respect to the relative execution count of each trace.

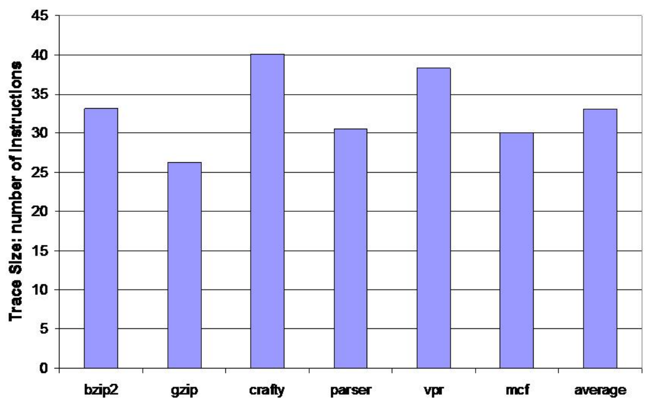

Figure 4: Average number of instructions per hot trace.

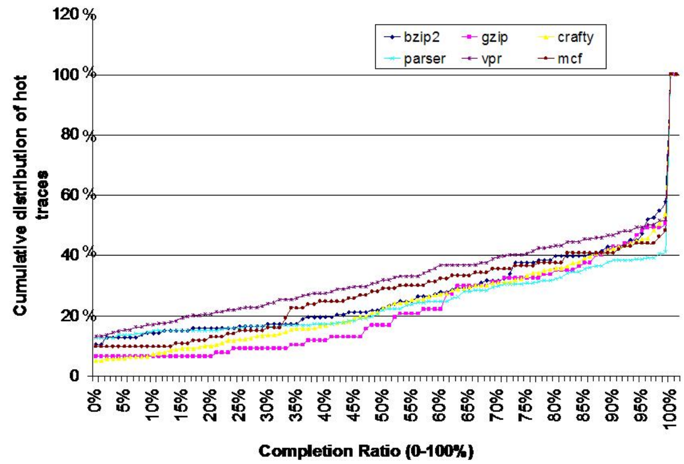

Figure 5: Cumulative distribution of completion ratios for hot traces.

first understand the frequency and distribution of early-exits from hot traces.

Once a hot trace is identified, StarDBT instruments the trace with counters to measure the frequency of early-exits. In particular for each hot trace StarDBT inserts (i) a counter at the top of the hot trace to count instances of the hot trace; and (ii) a counter at each early-exit to count the number of times the early-exit is taken to leave the trace. To reduce measurement overheads, the counters are only present for a small fixed number of instances of the trace (currently limited to 256 instances, but achieving a 94%+ level of confidence [25]). Furthermore, rather than place the early-exit counters inside the hot-trace they are instead placed at the early-exit destination locations. When the limit number of sampling instances is reached, the hot-trace and destination locations are re-translated to remove these measurement counters.

The trace instance counter and early-exit counters can be used to compute a *completion ratio* for the hot trace—an indication of the frequency of early-exits for a hot trace. Let $C_{instances}$ denote the value of the trace-instance counter, and let $C[i]_{early-exit}$ denote the values of early-exit counter i, then the completion ratio CR for the hot trace can be computed by

$$CR = \frac{C_{instances} - \sum_1^n C[i]_{early-exit}}{C_{instances}} \quad (1)$$

where n is the total number of early-exit branches (i.e., excluding the main exit from the hot trace). Hence the completion ratio effectively indicates the fraction of instances of the hot trace that one can expect to execute to completion (i.e., to its main-exit).

Figure 5 shows the cumulative distribution of completion ratios for hot traces. The main insight from the figure is that roughly half of all hot-traces have a fairly high completion ratio of greater than 97%, and that there is a fairly uniform distribution of completion ratios for the other half of hot-traces. These results are encouraging since they indicate that half of all hot traces can be combined without resulting in unacceptable frequency of early-exits.

4. LENGTHENING HOT TRACES

The goal of this work is to obtain longer hot traces while maintaining high completion ratios. However, these two requirements are contradictory: longer hot traces have more side-exit branches, and thus lower completion ratios. Our approach is to lengthen hot traces through unrolling, straightening, or an combination of the two depending on the main-exit type of the hot trace. A *self* type hot trace can be *unrolled* by replicating the trace body multiple times; an *other hot-trace* type hot trace can be *straightened* by appending a replicated copy of the target hot trace. In both cases any early exits must also be repaired. Note that when unrolling or straightening, the main exit for the first original hot trace becomes an early exit for the resulting lengthened hot trace. This iterative process continues until the new hot trace (produced through either unrolling or straightening) has an estimated completion rate below a certain target value.

Figure 6(a) shows a suitable hot trace a being unrolled *twice* (i.e., an *unroll factor* of 3). Figure 6(b) shows a straightening example on an *other hot-trace* type hot trace b. Straightening concatenates b with its target hot trace c, producing a straightened hot trace composed of code from both b and c. The straightening process continues whenever the new target hot trace (e.g., c's target hot trace) is still of *other hot-trace* type. However, since we find that the *other hot-trace* type is fairly uncommon in practice (only 4% of hot trace exits on average according to Figure 2), we do not implement nor evaluate straightening further in this paper.

4.1 Unrolling Algorithm

For any *self* type hot trace, let its original completion ratio be p_{orig}, and let the *target completion ratio* be p_{target}, then there exists a positive integer number U such that

$$p_{orig}^U \geq p_{target} \quad (2)$$

and

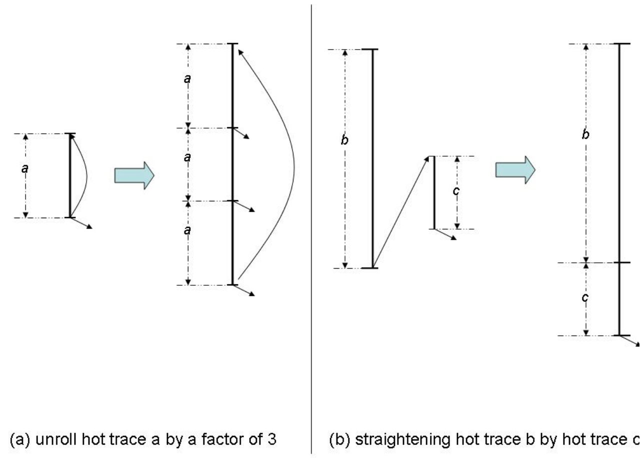

Figure 6: Two types of hot trace can be lengthened: (i) *self* type hot traces suitable to unrolling; and (ii) other hot-trace type hot traces suitable to straightening.

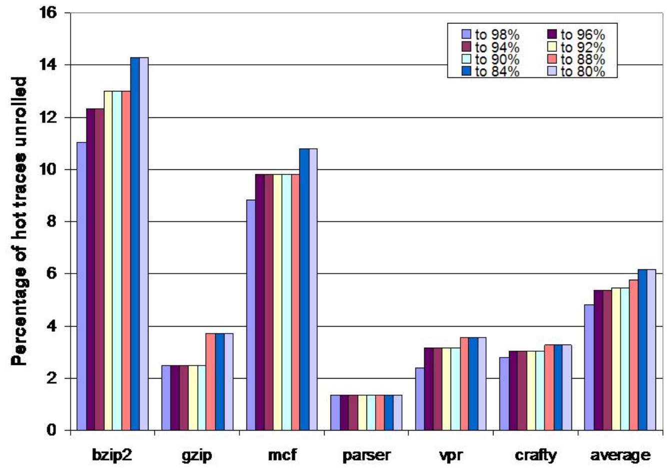

Figure 7: Fraction of hot traces that can be unrolled for varying target completion ratios.

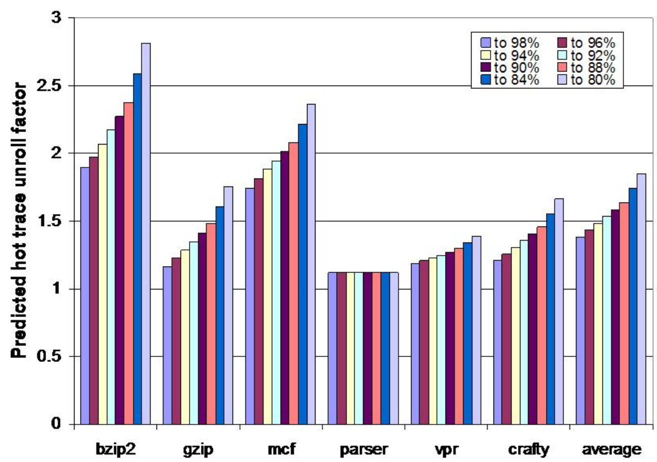

Figure 8: Average predicted unroll factor across all hot traces.

$$p_{orig}^{U+1} < p_{target} \tag{3}$$

U is the *predicted unroll factor*, the maximum unroll factor such that the completion ratio for the unrolled trace is predicted to be above the target value. A predicted unroll factor of 1 means no unrolling, a predicted unroll factor of 2 means to unroll the hot trace once, and so on. For any *self* type hot trace its predicted unroll factor can be computed by the following.

$$U = \begin{cases} 10 & \text{if } p_{orig} = 100\% \\ \lfloor \frac{\log p_{target}}{\log p_{orig}} \rfloor & \text{if } p_{orig} \geq p_{target} \\ 1 & \text{otherwise} \end{cases}$$

For each candidate hot trace with a 100% completion ratio, the formula will heuristically set the unroll factor to 10, hence replicating the trace body 9 times. If $p_{orig} \geq p_{target}$, the predicted unroll factor is estimated through the formula, taking both p_{orig} and p_{target} into consideration. If $p_{orig} < p_{target}$, the formula indicates that unrolling should not be performed (giving a predicted unroll factor of 1).

The formula of

$$U = \lfloor \frac{\log p_{target}}{\log p_{orig}} \rfloor$$

is obtained by taking logarithm on both sides of eq. (2) and eq. (3) with some simplification.

4.2 Lengthening Results

In this section we evaluate the impact of unrolling on hot traces, including the impact on average trace size and predicted completion ratio.

Unrollable Hot Traces Figure 7 shows the fraction of hot traces that can be unrolled for varying target completion ratios. For some benchmarks such as BZIP2 the number of unrollable hot traces increases significantly as the target completion ratio decreases, while for others such as parser decreasing the target completion ratio has little impact. On average across all benchmarks there are roughly 5% of hot traces suitable for unrolling with a 98% target completion ratio, and 6% of hot traces suitable for unrolling with an 80% target completion ratio.

Predicted Unroll Factors Figure 8 shows the impact of decreasing the target completion ratio on the predicted unroll factor. On average across all benchmarks, decreasing the target completion ratio steadily increases the predicted unroll factor from 1.42 for a 98% target completion ratio to 1.8 for an 80% target completion ratio. This impact varies widely across the benchmarks, and is fairly tied to the fraction of unrollable hot traces in each benchmark as shown in Figure 7.

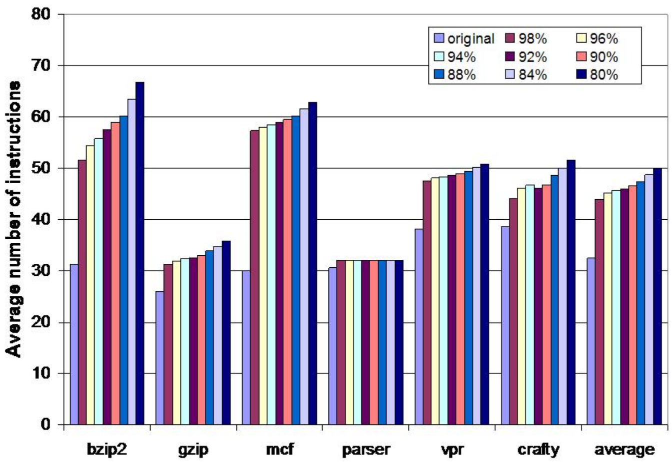

Figure 9: Impact on average hot-trace size of unrolling to varying predicted completion ratios.

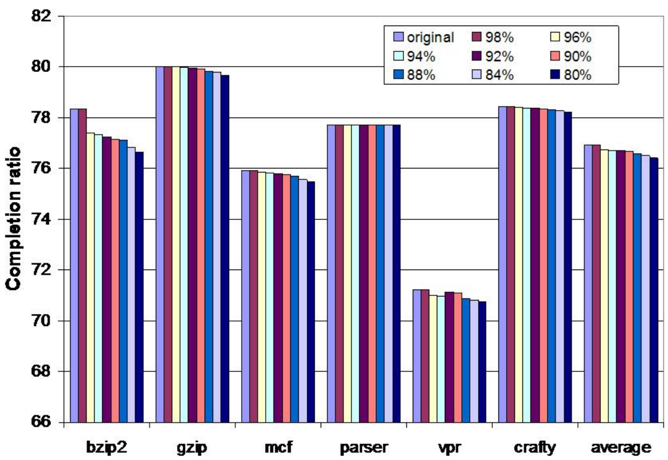

Figure 10: Impact on hot-trace completion ratio of unrolling to the predicted unroll factor.

Impact of Unrolling on Hot Trace Size Figure 9 shows the impact of unrolling on average hot-trace size as the predicted completion ratio is reduced from 98% to 80%. Recall that the original hot traces have an average trace size of roughly 33 instructions. Unrolling to a target completion ratio of 98% increases the average hot trace size to roughly 44 instructions (an increase of 33%); further relaxing the target completion ratio to 80% results in an average hot trace size of 50 instructions (an increase of 52%). These results demonstrate that despite the relatively low fraction of traces that are amenable to unrolling, unrolling can still have a significant impact on average trace size.

Impact of Unrolling on Completion Ratio In Figure 10 we evaluate the impact of unrolling hot traces to the predicted unroll factor on the completion ratio of the unrolled hot trace. On average, reducing the predicted completion ratio from 98% to 80% reduces the actual completion ratio from 77% to 76.5%—a reduction of only 1%, while average hot trace size has increased by 50%.

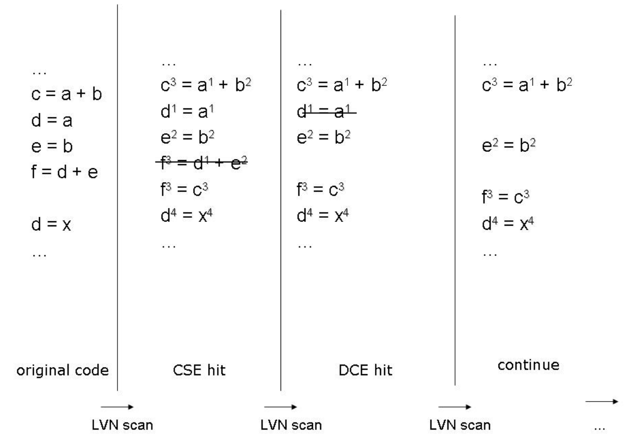

Figure 11: Local value numbering effectively implements (i) common subexpression elimination, (ii) dead-code elimination, and (iii) copy propagation.

5. IMPACT OF LENGTHENED HOT TRACES ON LOCAL VALUE NUMBERING

There are many challenges to attempting optimizations in DBT systems. DBT systems typically operate on binary executables that have already been heavily optimized by a static compiler at its highest optimization levels—hence there are significantly fewer optimization opportunities compared to those available when operating at the level of the original unoptimized source code. Furthermore, the runtime performance of the target executable is highly sensitive to the overheads of any dynamic optimizations attempted by the DBT system; hence any optimization benefits have to outweigh the overheads of implementing them.

In this section we demonstrate that lengthened traces can improve the impact of conventional trace-based dynamic optimizations, in particular by showing the resulting improvement in *local value numbering*—a common optimization implemented in DBT systems and JIT compilers.

5.1 Local Value Numbering

Most existing compiler optimization algorithms need to perform control-flow analysis and build data-flow solvers. But for dynamic optimization systems such as DBT systems, such analyses are too expensive. Local value numbering (LVN) [2, 14, 19] is an analysis that natively and elegantly supports three different optimizations: *constant subexpression elimination (CSE)*, *copy propagation (CP)*, and *dead-code elimination (DCE)*—as illustrated in Figure 11. LVN can be implemented with worst case complexity of $O(N^2)$ (though in practice most LVN analyses complete in $O(kN)$ time where $k << N$).

Briefly, LVN works as follows. Progressing through each statement in order, each new variable is assigned a distinct integer value-number (starting from 1). For any assignment expression, an existing value-number on the right-hand side *(RHS)* is assigned to the left-hand side *(LHS)*. If the *RHS* is a new variable (no existing value-number), a new value-

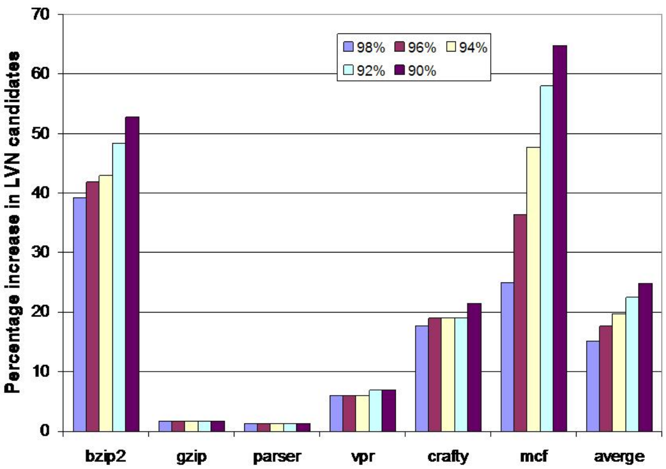

Figure 12: Percentage increase in optimization candidates for local value numbering over all hot traces.

number is created and propagated to both sides of the assignment expression (simulating copy propagation). LVN looks for value-number patterns on all unary expressions and binary expressions. When a match is recognized, LVN searches a history table of value-number versions, and replaces the *RHS* with the corresponding variable that has the matching value-number (effectively implementing CSE). If LVN recognizes that the same variable is assigned different value-numbers in two different assignment expressions and there is no intervening use of that variable, the first assignment expression is marked *dead* and can be removed from the instruction list (effectively implementing DCE).

Figure 11 illustrates the progress of LVN. The left-most column shows the original code in a simplified format. In the 2nd column all variables have been assigned value-numbers, and a common subexpression has been identified and eliminated (a *CSE hit*): binary expression $c^3 = a^1 + b^2$ and binary expression $f^3 = d^1 + e^2$ have the same value-number pattern *(vn(3) = vn(1) + vn(2))*, hence $f^3 = d^1 + e^2$ is replaced with $f^3 = c^3$. The third column shows elimination of dead code (a *DCE* hit): two assignment expressions, $d^1 = a^1$ and $d^4 = x^4$, associate different value numbers to the same variable d, and there is no intervening use of d, hence the 1*st* assignment expression $d^1 = a^1$ is marked *dead* and removed from the list of instructions. The last column shows the resulting optimized code after LVN.

Our prototype LVN implementation in StarDBT covers three forms of expressions: binary expressions of the form *(result = operand1 op operand2)*, unary expression of the form *(result = op operand)*, and assignment expressions of the form *(result = operand)*. An operand can be a memory location, a register, or a constant, and an operator can be any arithmetic or logical operation in the IA32 ISA.

5.2 Impact of Lengthened Hot Traces on LVN

Each time LVN finds a match for any of the three optimizations it covers, we call it a *LVN hit*. In this preliminary evaluation we measure the increase in LVN hits for lengthened hot traces. Figure 12 shows the percentage increase in LVN hits for traces unrolled to decreasing target completion ratios. For MCF, hot-trace lengthening provides 25% additional LVN hits for a 98% target completion ratio and 65% additional LVN hits for a 90% target completion ratio. For other benchmarks such as GZIP and PARSER our approach to lengthening has little impact: as seen in Figure 9, the overall impact on hot-trace size of unrolling is equally limited for these two benchmarks. On average across all benchmarks LVN hits are increased by 16% for a 98% target completion ratio, and by 23% for a 90% target completion ratio. Overall these results demonstrate that lengthened traces indeed result in increased opportunities for dynamic optimization. For now we are unfortunately unable to evaluate the performance impact of LVN optimization because our binary translation infrastructure is incomplete.

6. USING TRANSACTIONAL MEMORY TO SUPPORT SPECULATIVE OPTIMIZATIONS FOR LONG TRACES

Support for instruction-level speculation in current processors is limited, for example processors have on the order of a few tens of entries for load/store queues that support the speculative reordering of loads and stores [16, 28, 29]. Furthermore, support for instruction-level speculation is unlikely to increase significantly given recent industry trends towards multicore processors composed of simpler cores. Transactional Memory (TM) [1, 13, 20, 22, 27, 31, 35, 40] has been proposed to support optimistic synchronization of critical sections in parallel programs, by providing the ability to checkpoint and restore code that can speculatively read and modify kilobytes of data and more. In this paper we propose a novel use for TM to support aggressive speculative optimization of sequential programs, particularly the speculative optimization of legacy sequential programs in DBT systems that are enabled by lengthened traces and with guaranteed safety by TM's support for precise exception.

The basic idea is to exploit TM hardware to provide low-overhead checkpoint and restore, allowing the DBT system to apply speculative optimizations to hot traces. Lengthened hot traces will offer improved opportunities for recently-proposed speculative optimizations such as speculative code motion [17], and speculative execution based on delinquent load value prediction and function return value prediction [15].

We also argue that our techniques for estimating the ideal trace size based on completion ratios could be extended to determine the ideal size of transactions within traces. In particular, we can use the DBT system to profile the *successful completion ratio* of a transaction, which would subsume two factors: i) the completion ratio of the corresponding trace; and ii) the *success ratio* of the speculative optimization applied to that trace (i.e., the fraction of times that the speculative optimization succeeds). We can then throttle the application of speculative optimizations and transactions to be limited only to traces with a minimum successful completion ratio. There are many ways to extend this framework to take other factors into account, such as hardware limits to speculative buffering.

In this preliminary work we demonstrate that the amount of data accessed by lengthened traces is near the limit of what can be supported by typical load and store queues, motivating the use of TM hardware to enable DBT systems to apply more aggressive speculative optimizations than possible in the processor's reorder buffer.

Figure 13 shows the average number of memory accesses (both reads and writes) for the original hot traces, and the hot traces unrolled to varying predicted completion ratios from 98% to 80%. On average, the original hot traces have 22 memory reads and 12 memory writes. After lengthening to a predicted completion ratio of 80%, reads are increased to 34 and writes to 18—roughly a 50% increase for both reads and writes.

Figure 14 shows the average amount of data in bytes of memory accessed (reads and writes), for the original hot traces and those unrolled to predicted completion ratios between 98% and 80%. On average, the original hot traces read 91 bytes and write 46 bytes. Unrolling to 98% increases the reads and writes to 150 bytes and 77 bytes respectively; unrolling to 80% increases memory reads to 183 bytes and writes to 94 bytes.

7. CONCLUSIONS AND FUTURE WORK

In this paper we demonstrate the feasibility of obtaining longer hot traces through unrolling. We achieve significantly lengthened hot traces at a negligible increase in the frequency of early exits from hot traces (i.e., a negligible decrease in the completion ratio). We also demonstrate that lengthened traces can improve opportunities for applying both traditional and speculative optimizations dynamically. We develop a prototype LVN-based trace optimizer and prove that longer hot traces improve opportunities for optimization. Finally, we propose that emerging hardware transactional memory can provide effective support for speculative optimizations on hot traces. Our future research efforts will focus on completing the binary translation infrastructure for trace optimizations and directly evaluating performance on real machines, investigating speculative optimizations on long hot traces with high completion ratio and on automatically determining the optimal transaction granularity for such traces when applying speculative optimizations.

8. REFERENCES

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI 06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. A formal approach to code optimization. In *Proceedings of a symposium on Compiler optimization*, pages 86–100, 1970.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. In *Tech. Report HPL-1999-78*, Cambridge, jun 1999.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI 00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[6] A. Bhowmik and M. Franklin. A fast approximate interprocedural analysis for speculative multithreading compilers. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 32–41, New York, NY, USA, 2003. ACM Press.

[7] B. J. Bradel. The use of traces in optimization. Master's thesis, University of Toronto, 2004.

[8] B. J. Bradel and T. S. Abdelrahman. A characterization of traces in java programs. In *Proc. of the Int'l Conference on Programming Languages and Compilers*, pages 87–93, June 2005.

[9] B. J. Bradel and T. S. Abdelrahman. Automatic trace-based parallelization of java programs. In *Proc. of the Int'l Conference on Parallel Processing*, September 2007.

[10] B. J. Bradel and T. S. Abdelrahman. The potential of trace-level parallelism in java programs. In *Proc. of the Int'l Conference on Principles and Practices of Programming in Java*, September 2007.

[11] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. *SIGPLAN Not.*, 38(10):25–36, 2003.

[12] W. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, december 2000.

[13] J. Chung, C. C. Minh, B. D. Carlstrom, and C. Kozyrakis. Parallelizing specjbb2000 with transactional memory. In *Workshop on Transactional Memory Workloads*. June 2006.

[14] C. Click. Global code motion/global value numbering. In *PLDI 95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 246–257, New York, NY, USA, 1995. ACM.

[15] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25, New York, NY, USA, 2001. ACM.

[16] J. Crummey. Sun nagria and ibm power5. In *Computer Science Lecture Notes 522*, September 2007.

[17] X. Dai, A. Zhai, W.-C. Hsu, and P.-C. Yew. A general compiler framework for speculative optimizations using data speculative code motion. In *Code Generation and Optimization (CGO)*, March 2005.

[18] T. Garrnet. *Dynamic Optimization of IA-32 Applications Under DynamoRIO*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2003.

[19] S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. *Sci. Comput. Program.*, 64(1):97–114, 2007.

[20] L. Hammond, V. Wong, M. Chen, B. D. C. J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, June 2004.

[21] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. mei W. Hwu. Superblock formation using static program analysis. In *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, pages 247–255, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[22] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[23] M. Herlihy and E. J. Moss. Transactional memory: Architectural support for lock-free data structures. In

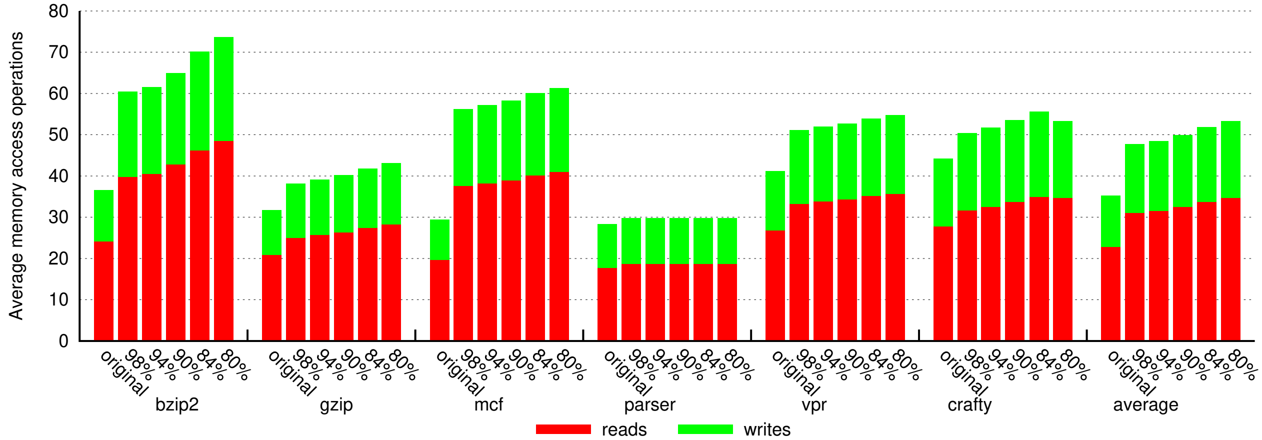

Figure 13: Average number of data locations accessed for varying predicted completion ratios.

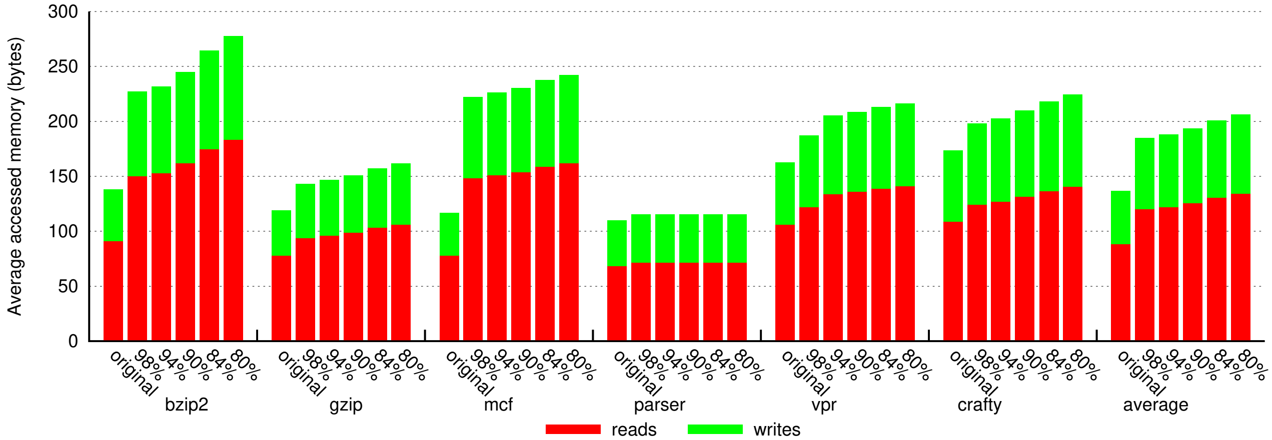

Figure 14: Average number of bytes accessed for varying predicted completion ratios.

Proceedings of the 20th Annual International Symposium on Computer Architecture, pages 289–300, 1993.

[24] 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.

[25] http://www.raosoft.com/samplesize.html. Online sampling confidence calculator.

[26] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, pages 229–248, 1992.

[27] W. N. S. III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC 05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[28] j. Mendler, J. Dodson, J. F. Jr., H. Le, and B. Sinharoy. Power4 system microarchitecture. In *IBM Research*, January 2002.

[29] C. Keltcher. Amd hammer processor core. Aug 2002.

[30] A. KleinOsowski and D. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. In *In Computer Architecture Letters*, 2002.

[31] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP 06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220, New York, NY, USA, 2006. ACM.

[32] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology, Boston, MA, USA, 2004.

[33] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan. A compiler framework for speculative analysis and optimizations. In *PLDI '03: Proceedings of the 2003 conference on Programming language design and implementation*, pages 289–299, New York, NY, USA, 2003. ACM Press.

[34] 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.

[35] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual*

International Symposium on Computer Architecture. June 2007.

[36] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN*, 42(6):89–100, 2007.

[37] J. Nieh and O. Leonard. Examining vmware. In *Dr. DobbŠs Journal*, August 2000.

[38] M. Olszewski, J. Cutler, and J. G. Steffan. Judostm: a dynamic binary rewriting approach to software transactional memory. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2007.

[39] E. Rotenberg and J. Smith. Control independence in trace processors. In *32nd Annual International Symposium on Microarchitecture*, 1999.

[40] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP 06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[41] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *PLDI 94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM.

[42] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Code Generation and Optimization (CGO) 2007*, March 2007.

[43] C. Wang, S. Hu, H.-S. Kim, S. R. Nair, M. B. Jr., Z. Ying, and Y. Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In *Asia-Pacific Computer Systems Architecture Conference*, pages 4–15, 2007.

[44] C. Wang, V. Ying, and Y. Wu. Supporting legacy binary code in a software transaction compiler with dynamic binary translation and optimization. In *Compiler Construction (CC)*, 2008.

[45] V. Ying, C. Wang, Y. Wu, and X. Jiang. Dynamic binary translation and optimization of legacy library code in stm compilation environment. In *Proceedings of the workshop on binary instrumentation and applications*, pages 63–70. IEEE Computer Society, 2006.