TAO: Two-level Atomicity for Dynamic Binary Optimizations

Edson Borin Youfeng Wu Cheng Wang Wei Liu Mauricio Breternitz Jr. Shiliang Hu
Esfir Natanzon Shai Rotem Roni Rosner

Intel Corporation

{edson.borin, youfeng.wu, cheng.c.wang, wei.w.liu, mauricio.breternitz.jr, shiliang.hu, esfir.natanzon, shai.rotem, roni.rosner}@intel.com

Abstract

Dynamic binary translation is a key component of Hardware/Software (HW/SW) co-design, which is an enabling technology for processor microarchitecture innovation. There are two well-known dynamic binary optimization techniques based on atomic execution support. Frame-based optimizations leverage processor pipeline support to enable atomic execution of hot traces. Region level optimizations employ transactional-memory-like atomicity support to aggressively optimize large regions of code. In this paper we propose a two-level atomic optimization scheme which not only overcomes the limitations of the two approaches, but also boosts the benefits of the two approaches effectively. Our experiment shows that the combined approach can achieve a total of 21.5% performance improvement over an aggressive out-of-order baseline machine and improve the performance over the frame-based approach by an additional 5.3%.

Categories and Subject Descriptors B.1.4 [*Microprogram Design Aids*]: Languages and compilers

General Terms Design, Experimentation, Measurement, Performance

Keywords Hardware/software co-design, dynamic binary optimization, atomic execution, large region optimization

1. Introduction

Hardware/software co-design is a promising technology for processor microarchitecture innovations [7, 9, 15, 16, 25, 27]. In this technology, the traditional source ISA, e.g. x86, is dynamically translated into an internal implementation ISA, and may be optimized by SW before the implementation ISA is executed on the microarchitecture. Intel's P6 microarchitecture family converts x86 instructions into internal micro-operations (called μops), and may apply a number of μop level optimizations in the HW pipeline when they are executed [6]. This can be viewed as an early dynamic optimization system, although the optimizations are simple and done by HW.

Frame-level optimizations (FAO), explored by rePLay [25, 29] and PARROT [1, 27], rely on atomic execution support to enable aggressive μop optimizations. For example, rePLay collects frequently executed μop traces, or frames, and converts branches in them into "asserts" to verify the branch conditions [26]. When a frame executes, its results, e.g. store values, are buffered in pipeline buffers (re-order buffer, store/load buffers, etc). If an assert fails to verify the branch condition, the architectural state is rolled back to the beginning of the frame, the buffered results are discarded, and the execution is steered to the corresponding non-optimized μops. If no failure is detected, the frame can commit its results after the last μop is executed. The frame level atomicity support enables fast and aggressive optimizations within the frame without concern about control flow among the μops as well as the memory order among the memory operations being optimized. Another key feature explored by rePLay and PARROT is that they use well known branch prediction techniques to predict future paths and their associated frames, which enables path-based optimizations. Slechta [29] et al. and Almog [1] et al. showed that μop optimizations with frame level atomicity support can improve performance by $\sim$17% on machine models that implement the x86 ISA.

Buffering the results of frame execution into the processor's pipeline buffers enables fast checkpointing, commit and rollback. However, the pipeline buffers are limited in size, which restrict the size of the frames, reducing the scope for optimizations. Furthermore, a frame of μops is a straight-line of code that can only commit at the end of the frame. This requires that only highly predictable branches be included inside the frame and must be converted into assertions. Fahs et al. [10] shows that partial matching, i.e. allowing a frame to optionally exit early, may significantly improve the potential performance.

Region level atomic execution, such as implemented by Transmeta Efficeon [16], supports a fully programmable internal ISA. The internal ISA allows branches within a region, which enables partial match. A region is also normally larger than a frame. Consequently, it increases the optimization scope for advanced optimizations. To support atomic execution, region level atomic execution usually utilizes a speculative data cache to buffer the execution results until the whole region is committed. A checkpoint is created at the beginning of the region execution. When unexpected execution events happen, such as exceptions or uncacheable memory accesses, the data in the speculative cache is discarded, the execution rolls back to the beginning of the region and continues with less aggressive optimizations, e.g. interpretation. Due to its larger scope, region level atomic execution is likely to have higher checkpointing, rollback, and commit overheads.

Clearly, frame-level and region-level atomic execution have their respective advantages and disadvantages. Frame-level atomic execution benefits greatly from path based optimizations, asserts, and fast checkpointing/commit. Region level atomic execution enables large region optimizations, supports internal branches and enables memory optimizations across large code regions.

CGO'10, April 24–28, 2010, Toronto, Ontario, Canada.

In this paper, we propose an integrated Two-level Atomic Optimization approach, or TAO for short. We support frame-level atomicity with processor pipeline buffers, and region level atomicity with speculative cache HW support to optimize large regions constructed from frames. TAO not only achieves the advantages of both approaches, but also enhances each so the combined benefits are higher. For example, the combined approach enables even larger region level optimizations because the frame level atomicity support allows partial commit of the region execution, which minimizes the overhead caused by the region roll back.

The contributions of the paper can be summarized as follows.

- We developed a framework to extend frame level optimizations to large region optimizations, which takes advantage of both levels of support for atomic execution.
- We implemented a list of global optimizations to optimize large regions of code dynamically, and also extended several frame level optimizations to the region level.
- We demonstrated the potential benefit of TAO over frames by measuring the performance improvements of two-level atomic optimizations using an industrial strength cycle accurate simulator. Our experiments show that TAO can achieve a total of 21.5% performance gain over an aggressive out-of-order baseline machine and improve upon the FAO approach by an additional 5.3%.

The paper is organized as follows. Section 2 provides an overview of the frame based optimizations framework and Section 3 shows how we extended the frame based framework to optimize and execute large regions. Section 4 discusses the two-level atomicity support and its benefits for global region optimizations. Section 5 depicts the TAO framework, including the optimizations used to optimize large regions and the hardware support to execute the regions. Section 6 discusses the experimental results. Section 7 describes the related work and Section 8 concludes the paper.

2. Frame-level Atomic Optimizations

Frame-level Atomic Optimizations (FAO) were previously introduced in [1, 25, 27, 29]. It extends the trace cache [28] idea to store decoded and optimized μop traces in a processor local frame cache to reduce instruction decoding overhead as well as benefit from μop optimizations. We implemented a FAO framework on top of an x86 out-of-order simulator with the following HW support.

Frame formation logic: This logic collects profile information for retired μops, and uses branch predictability information to form hot frames. This logic also tracks the branch direction history to train the frame prediction logic.

Frame optimizer: Once the frames are collected, they are optimized. The optimizer converts predictable branches into asserts and optimizes the frame code as a single basic block, which can be done easily and efficiently. Since the μops are beneath the x86 ISA, many microarchitecture level resources, such as internal extra registers, and special operations, such as fused and SIMDized μops, can be used. The optimized μops are stored in a frame cache.

Frame cache: Since frames have variable sizes, the frame cache organizes frames into chunks of fixed number of μops each. When the size of a frame is not a multiple of the chunk size, the last chunk may be not fully utilized, causing frame cache and fetching fragmentation. This could be more severe for smaller frames.

Since the atomic scope for frames is constrained inside the HW pipeline, the checkpointing and commit can be done as efficiently as those for branch misprediction handling. Specifically, the checkpointing only needs to remember a few points in the pipeline, e.g. renaming buffer, re-order buffer location, etc. The commit allows buffered stores to be retired to cache and memory.

Frame predictor: Given an x86 instruction pointer (IP) and the current global branch history, the frame predictor tries to predict a frame for execution. If the prediction fails, regular branch prediction is used to predict x86 code for execution. Note that it is possible to have multiple frames starting at the same x86 IP, distinguished only by the internal branch directions. This allows specialized frames with the same starting IP to be formed for path-specific optimizations. For a correctly predicted frame, the direction information for the branches (converted to assert) within the frame is used to update the global history register for future branch and frame prediction. The predicted frame is fetched from the frame cache and fed into the processor pipeline for execution.

2.1 Limitations of frame-level optimizations

Although FAO has distinctive advantages, such as fast checkpointing/commit, straight-line code optimizations and path-specialized optimizations, it also has several limitations.

First, it does not allow internal branches inside frames. This forces frames to be terminated at un-predictable branches, leading to smaller frames. Small frames can limit optimization benefits and reduce the effective frame fetch bandwidth. Not being able to partially execute a frame can constrain the frame execution coverage as the partial execution has to abort and execute un-optimized code.

Second, there are other issues caused by relatively small size of frames. For example, when the iTLB (instruction translation lookaside buffer) flushes (e.g. due to process context switches), the page mapping for the x86 instructions may be changed and the translation may be invalidated. This may require flushing the frame cache. One optimization is to add "page mapping check" (PMC) code in the beginning of a frame to examine whether or not the page mapping has changed for the frame code and only if the mapping has changed the optimized code would be discarded. The checking can quickly verify if the iTLB changed since the last time the frame was executed. If so, a more complete check, including the mapping of all the pages that contain the frame code, must be performed. As our experiments in Section 6.4 indicate, this checking cost is usually negligible for large regions but it can be significant for small frames. Notice that PMC does not handle self-modifying code (SMC). To handle SMC we may use techniques such as described by Dehnert et al. [7].

3. Extending FAO to Region Level

The two-level atomic optimization (TAO) framework extends the FAO framework to enable global optimizations on regions composed of multiple atomic frames. A TAO region (or region for short) is defined as a set of frames connected by control-flow edges.

In this section, we describe the extensions to the FAO framework to enable the optimizations and execution of TAO regions. We believe this approach can overcome most of the FAO limitations.

Public and private frames: We use the terms public frame and private frame to distinguish the region entry frames from the internal frames. A public frame is defined as a frame that can be reached from outside of the region (through un-optimized code or any other frame), while a private frame is a frame that can only be reached though one of its predecessor frames in the region control flow graph. A region may contain multiple public (entry) and private (internal) frames. Figure 1 (a) shows a region composed of one public (F_1) and two private frames (F_2 and F_3). In this region, the private frame F_2 can only be reached through its predecessor frame F_1. We represent public frames as boxes with a black circle and

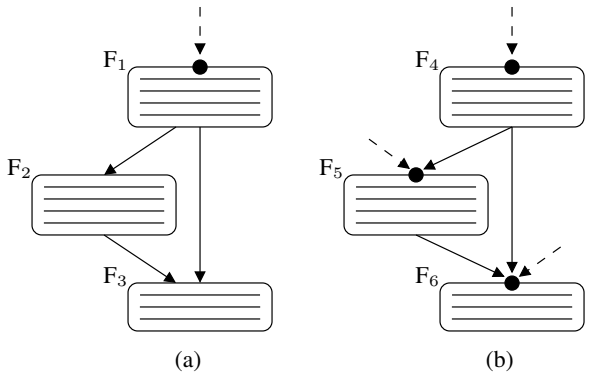

Figure 1. TAO regions. Boxes with black circles (F_1, F_4, F_5, and F_6) represent public frames and boxes without the circles (F_2 and F_3) represent private frames.

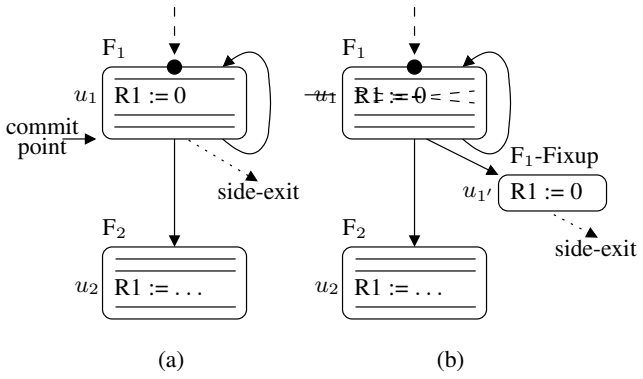

Figure 2. (a) Region with partially dead code. (b) Region after partial dead code elimination.

a dashed income edge. The dashed edge indicates that the control flow can reach the frame from outside of the region.

Private frames are basically used to trim side entries in the control flow graph, creating more optimization opportunities. The region at Figure 1 (a) illustrates the benefits of private frames. Suppose that frame F_1 computes a value that is recomputed at F_3. Since F_1 dominates F_3, we can apply common sub-expression elimination (CSE) to remove the redundant computation in F_3. The key property that enables the CSE and other forward optimizations (like partial redundancy elimination - PRE) is that F_1 dominates F_3. It is easy to see that, if all frames are public, as shown in Figure 1(b), a frame cannot dominate other frame in the region. Notice that all the frames can be directly reached from outside of the region (dashed edges), therefore they cannot be dominated by the other frames in the region.

Fixup code: We introduce the concept of fixup code to allow the execution of compensation code on side-exits of regions. As we show below, the fixup code mechanism relaxes the frame commit semantics, enabling more aggressive optimizations to be performed.

In the FAO framework, the architectural state (registers, memory, etc.) is required to be precise at frame boundaries, as the code executed after a frame may rely on precise architectural state for correct execution. As an example, assume the region in Figure 2 (a). If frame F_2 raises an exception, the framework rolls the architectural state back to the last checkpoint (beginning of F_2) and transfers the execution to un-optimized code, which properly handles the exception. Since the framework supports precise exceptions, the architectural state has to be precise when the execution is transferred to un-optimized code, therefore, the commit points must hold precise architectural states.

Requiring precise architectural state on commit points prevents many optimizations from being applied to the region. As an example, the value produced by μop u_1, in the region at Figure 2 (a), is killed by itself (through the back-edge) and by μop u_2 (in frame F_2). Unfortunately, u_1 cannot be eliminated, as the value of R1 would not be precise at the commit point.

To model the potential control transfer to un-optimized code, we annotate commit points with implicit side-exits to un-optimized code (as seen in Figure 2 (a)). Also, to ensure precise architectural state on the side-exits, we modify the global optimizations to assume that all the architectural values (memory and architectural registers) are alive on the side-exits.

The fixup code is a piece of compensation code associated with a commit point and it is executed whenever the execution rolls back and the precise architectural state is required. Figure 2 (b) shows how the fixup code can be used to optimize the dead code in Figure 2 (a). Notice that μop u_1 was removed from F_1 and the μop $u_{1'}$ was inserted into the fixup code (F_1-Fixup). After the removal of u_1, the commit point at the end of F_1 does not hold the precise architectural state anymore (R1 value is not precise). However, in case F_2 fails and the execution rolls back to F_2 checkpoint, F_1-Fixup is identified and executed, computing the precise value for R1 and recovering the precise architectural state. For simplicity, since F_1 commit happens immediately before F_2 checkpoint, we may say that the execution was rolled back to F_1 commit point, instead of F_2 checkpoint.

We can see from the previous example that the fixup code allows the F_1 commit point to hold an imprecise architectural state. This is possible because before any external inspection to the value of register R1 (exception handler, context switch, etc.) the framework executes the fixup code and corrects the value of R1.

Although fixup code enables more optimizations across frames, it is limited by the fact that it cannot contain μops that may generate exceptions (Divide by zero, loads, stores, etc.). Notice that if an exception happens in the fixup code, the valid architectural state cannot be recovered, which could cause an incorrect execution.

Besides the limitation on the fixup code, the global optimizations are also limited by the fact that optimizing memory operations across frames may violate the memory ordering. This happens because whenever a frame commits, the memory state becomes visible to other processors. In order to remove these limitations, we propose the usage of a second level of atomicity support at region level.

Region atomicity support: To provide atomic execution at region level, a checkpoint is created whenever the execution enters the region, the data is buffered during execution and committed when the execution leaves the region. The region atomicity support provides two main benefits:

- The memory operations inside the region are only made visible to other processors when the execution leaves the region. This semantic enables the aggressive optimizations to freely optimize (reorder, remove or insert) memory operations across frames inside the region without compromising the memory ordering. In case there is a memory ordering conflict, the execution rolls back to the checkpoint at the beginning of the region. This semantics also allows instructions with lock prefix to be included into the atomic regions.
- The region checkpoint provides a secondary valid architectural state that can be used to recover the execution in case the fixup

code cannot perform the architectural state recovery. This property enables the optimizations to insert code that may generate exceptions inside the fixup code. Notice that if the fixup code for a frame fails, the TAO framework can still recover the architectural state saved by the checkpoint at the region entry.

Successor list: In order to represent the control-flow edges of the region we annotate each frame in the region with a list of successor frames. In this sense, each frame in a region has a list of identifiers representing its valid successor frames in the region. The framework uses the successor list to check if the next predicted frame is a valid successor frame. If not, the framework takes a side-exit and executes the proper fixup code to fix the architectural state.

Region formation: The TAO framework builds and optimizes atomic frames for hot code. In order to form regions, TAO profiles the retired frames and triggers the region optimizer.

TAO collects hot frames and stores them into a hot frame buffer. Whenever the buffer is full, or one of the frames becomes very hot, the region formation module is triggered. The region formation module uses the edge frequency information between frames to analyze the hot frames in the buffer and forms one or more connected regions. Non-cacheable memory operations, I/O operations and system calls should not be included in the region, otherwise, the region will abort when they are executed.

Region optimizer: Once a region is formed, the region optimizer is invoked to optimize the region. Section 5.3 describes the optimizations implemented into our TAO Framework.

Gear shifting: The TAO framework implements a system that executes the code using three gears. Unlike Transmeta Efficeon [16], our first gear is the native execution of x86 code (a.k.a. un-optimized code), rather than interpretation. The second gear is the execution of optimized frames. The third gear is the execution of hot regions. The frame profiler shifts the gear from 1 to 2 by building and optimizing hot frames. The region profiler shifts the gear from 2 to 3 by building and optimizing hot regions. Besides the profilers shifting the gears up, the TAO framework also employs a gear down profiler to detect bad frames/regions that frequently roll back. The gear down profiler triggers the removal of frames/regions from the frame cache and prevents the bad frames/regions from being formed again. Although the shift down rarely happens, it is important for certain benchmarks.

4. Benefit of Two-level Atomicity

The two-level atomicity consists of a mechanism to provide atomicity at region and frame levels. In this section we describe the importance of providing the two-level atomicity support for efficient region and frame execution and how the two atomicity mechanisms can make each other more effective.

Region level atomicity: As discussed at Section 3, the main benefit of region level atomicity is that it enables the optimizations to freely optimize memory operations across frames.

Frame level atomicity: TAO framework provides atomicity support at frame level similar to FAO. The frame μops are executed speculatively and the frame commits only if all the μops were executed successfully.

The TAO framework relies on the frame atomicity and the fixup code support to record local checkpoints that enable the recovery of precise architectural states without rolling the execution back to the beginning of the region. This is important to minimize the amount of work discarded during a roll back. As an example, Figure 3 (a) shows a region with a loop that was optimized by sinking the store μop from loop F_2 to frame F_3. There is a region checkpoint in the beginning of the region execution and a region commit in the end of the region. After entering the region, the loop starts to run and all the memory operations are stored in the speculative cache. Suppose that, after 1000 loop iterations, the speculative cache runs out of resources and the execution cannot proceed. A safe way to proceed is to roll the execution back to the last checkpoint (region entry) and continue the execution with un-optimized code. However, in this case, the amount of work wasted due to the roll back to the region entry would be very large (1000 loop iterations).

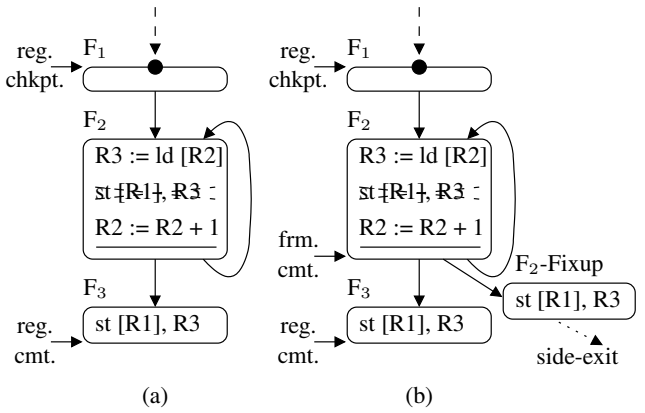

Figure 3. (a) optimized loop region. (b) optimized loop region after introducing the speculative checkpoint.

In order to reduce the amount of work discarded on a roll back, we use the frame atomicity support to record a speculative checkpoint that enables us to recover a more recent precise architectural state, instead of rolling back to the beginning of the region. The speculative checkpoint is recorded by the frame checkpoint mechanism and is called speculative because it contains speculative states. Figure 3 (b) shows how the speculative checkpoint is used to recover the precise architectural state. In this figure, F_2-Fixup is the fixup code associated with the commit point at frame F_2. Whenever a roll back is required, the framework rolls the execution back to the last frame checkpoint and execute the fixup code associated with the previously executed frame to recover the full precise architectural state. Different from the example in Figure 3 (a), if the speculative cache runs out of resource after 1000 loop iterations, the execution rolls back to the last frame checkpoint (entry of F_2 at iteration 999), executes the fixup code associated with the previously executed frame (F_2-Fixup) and commits the region. Although the execution leaves the region, it managed to commit the work produced by 999 loop iterations. This ability to save useful work even when overflowing the speculative cache is important, as it gives the TAO optimizer the freedom to construct large regions without concern about speculative cache capacity.

If the recovery is not possible (e.g. the fixup code fails), the framework rolls the execution back to the checkpoint in the beginning of the region. In this sense, the fixup code is allowed to contain μops that may generate exceptions, like memory μops.

Similar to FAO, the results produced by the frame execution are buffered by the pipeline buffers until the frame commits. However, when the frame commits, the retired memory data is transferred to the speculative cache. Figure 4 shows the storage and migration of computed data. The data generated by frames execution is buffered on the pipeline until the frame commits. After the frame commits, the data produced by store operations is migrated to the speculative cache. After the region commits, the speculative data is promoted to non-speculative and becomes visible to other processors.

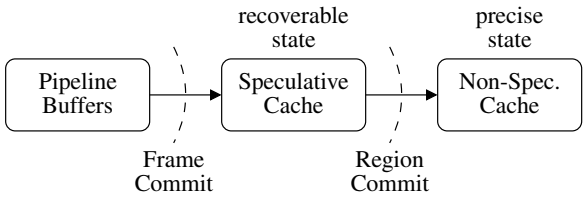

Figure 4. Storage and migration of computed data.

With region level atomicity support, frame level optimizations can also be more effective. For example, the fixup code at frame level could not contain any potential excepting operations, as the fixup code is meant to recover precise architectural state. Consequently, frame level optimizations cannot optimize memory operations across frame boundary even with fixup code support. If the frames are inside a region, however, the fixup code is allowed to have operations that may potentially generate exceptions. So if no exception happens in the fixup code for a frame, the execution can fix the precise machine state and exit the region. If an exception happens, the execution can roll back to the region entry and then exit the region.

The two-level atomicity support could be implemented using a multi-version speculative cache [11], dedicating one version for the frame level atomicity and another to the region level atomicity. Although this approach may remove the size limitation of the frames, it would be certainly more expensive in terms of hardware, and the checkpoint/commit overhead for the frame could be significantly increased, diminishing the benefits of the frame execution.

5. TAO Implementation

As we described in Section 4, the TAO framework relies on the frame atomicity support to record speculative checkpoints that enable the recovery of precise architectural states without rolling the execution back to the beginning of the region. However, these speculative checkpoints add side-exits to the control flow graph, which may reduce the optimization opportunities in the region. Section 5.1 discusses the limitations imposed by these side-exits and shows that sinking and some speculative hoisting optimizations can still be applied after the introduction of speculative checkpoints with fixup code.

5.1 Speculative checkpoint limitation on optimizations

In order to ensure that the fixup code can recover the full architectural state, the optimizations must ensure that the transformations applied to the region do not prevent the architectural state from being recovered. As an example, Figure 5 shows a hoisting transformation that prevents the architectural state from being recovered by the fixup code. In this example, the store operation u_1 was hoisted out of the loop. In this case, u_1 changes the contents of the cache when F_1 commits, however, the fixup code cannot undo this speculative cache write, which prevents the precise architectural state from being recovered by the F_1 fixup code.

In Section 4 we showed that if the architectural state cannot be recovered for the side-exit, the execution can still roll back to the beginning of the region. This is also valid for the example of Figure 5. In case the transformation prevents the fixup code from recovering the architectural state, we could simply force the fixup code to fail, which would roll the execution back to the beginning of the region in case the fixup code is executed. However, this behavior may cause performance degradation (as discussed in Section 4), thus, we avoid transformations that prevent the fixup code from recovering the precise state.

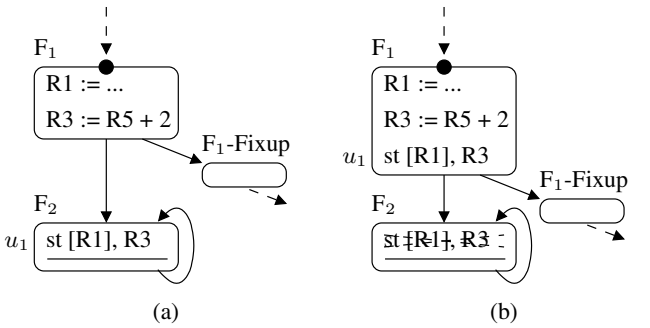

Figure 5. Hoisting transformation that prevents the fixup code from recovering the precise architectural state.

Sinking optimizations: Optimizations that sink code in the control flow graph, like the partial dead code elimination in Figure 2, are known as sinking optimizations. These optimizations do not cause unrecoverable modifications to the architectural state before the frame commits. In fact, they eliminate or postpone these modifications to after the commit point. Therefore, they do not prevent the fixup code from recovering the architectural state.

Hoisting optimizations: Optimizations that perform code hoisting, like the loop invariant code motion in Figure 5, are known as hoisting optimizations. As seen before, these optimizations could speculatively modify the architectural state, preventing it from being recovered by the fixup code. In order to avoid this scenario, we adopt the following rules in our hoisting optimizations:

- *Rename the destination register when hoisting μops that write to architectural registers:* whenever hoisted across a commit point, the destination register is renamed to a non architectural temp register and a new μop is introduced to copy the value from the temp register to the architectural register.
- *Do not hoist μops that have unrecoverable side-effects:* these μops include store μops and μops in which the destination architectural register cannot be renamed. The optimizations do not hoist these μops.

Figure 6 (b) shows an example where a load that modifies an architectural register (R1) is hoisted from a loop. The load μop (u_1) is hoisted out of the loop and a copy μop (u_2) is introduced to copy the value from the temp register (T4) to the architectural register (R4). Although the load is speculatively executed, the architectural state is not modified by the hoisted operation.

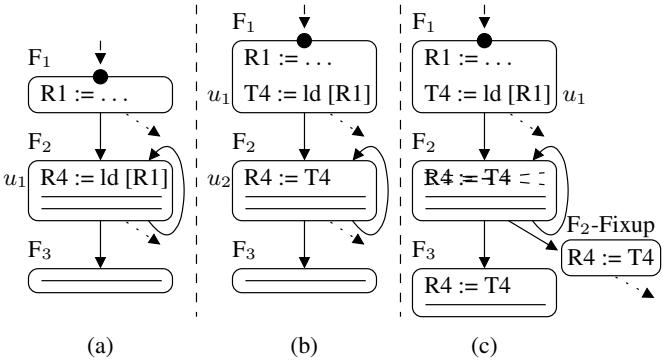

Figure 6. (a) Original code. (b) Hoisted load. (c) Partially dead code eliminated.

The copy operation introduced by the hoisting transformation is dead across the loop back edge and we remove it by applying partial dead code elimination to the region. Figure 6 (c) shows the region code after applying partial dead code elimination.

Although we do not hoist store operations, often, these μops can be sunk in order to achieve the same benefits of hoisting. Notice that the store μop at Figure 5 could be sunk out of the loop, instead of hoisted. In fact, our framework implements the partial dead store elimination (PDSE), an optimization that performs this sinking transformation.

5.2 Hardware support for TAO regions

In this section we describe the extra hardware proposed to support building and executing regions on top of a frame-level optimizations framework.

Prediction of private frames: During the region formation, if a frame is not chosen to be one of the entry frames, it is privatized into the region, and the original frame is left in the frame cache as a stand-alone frame. For a given stand-alone (public) frame, there may be multiple private copies in multiple regions, each with a unique identifier (ID). Since the frame predictor only predicts public frames in the FAO framework, we need an additional mechanism to determine whether the predicted frame should be one of the privatized versions of the frame. This is accomplished by keeping a list of valid successor frames. If the predicted frame matches one of the valid successors of the last predicted frame, then the private frame corresponding to the valid successor should be the predicted. The IDs of the valid successor frames are stored in each frame, in the "successor list", as described in Section 3. After a private frame is predicted, it is fetched and executed in the same way as public frames. This approach enables us to keep the same frame prediction accuracy achieved by FAO.

The frame predictor could also be modified to directly predict private frames. In this case, the frame predictor would have to be trained with private frame IDs. The direct private frame prediction eliminates the match operation on the successor frames list, but may require a bigger prediction table in order to store the extra frame IDs and could affect the frame prediction accuracy. Other IP remapping techniques could also be used to predict private frames [14, 21], however, we leave the evaluation of these approaches for future work.

Execution of fixup code: Fixup code will only be executed when one of the following unexpected region exit events happens: 1) a frame misprediction, 2) a frame cache fetch miss, 3) a roll back caused by a frame control assert failure, an exception or interrupt, or by lack of speculative cache resources. When this happens, the fixup code associated with the previous frame will be injected into the fetch flow in the processor. If an exception occurs inside the fixup code, the execution rolls back to the region entry and exits the region.

Region formation profilers: The FAO framework uses branch predictability information to form hot traces before converting the highly predictable branches into asserts. For region level optimizations, branch frequency/probability information is necessary for constructing tight regions where the execution stays inside for extended time before exiting. We currently maintain the edge frequency information for all frames retired since the last time a region was constructed. When enough hot frames are collected, new regions are formed for optimizations and the edge frequency information is cleared. Although the edge frequency information is convenient for region formation, it may be inefficient in HW implementation. We leave the evaluation of more efficient region profiling techniques, like in [20], for future work.

Memory disambiguation: Many memory optimizations, such as load hoisting, redundant and dead loads/stores eliminations, etc, require precise memory alias information. We use "alias assertions" to dynamically detect if any memory dependency is violated in an optimization range. For example, if we move a load from frame B to frame A, assertions are inserted to monitor all of the potentially aliased stores to ensure that no stores from A to B may store to the same location as the load. If any assert fails during execution, the execution needs to roll back. If a memory optimization range is within a frame, only the frame needs to be rolled back. Otherwise the whole region needs to be aborted and rolled back.

Atomic region execution: In our framework, TAO relies on a speculative cache and a register checkpoint mechanism to provide atomic region execution support. Notice that regions may execute loops with thousands of μops and it would be impractical to buffer the execution using the processor's pipeline buffers. Speculative caches usually require more hardware than traditional, non-speculative caches, however, we expect that more and more processors will readily provide speculative caches for hardware transactional memory support (HTM) [11] on multi-cores. In this sense, the TAO framework can take advantage of HTM support.

5.3 TAO optimizations

TAO includes most of the frame level optimizations as described in [1]. In addition, it supports a number of region level optimizations. Here we outline the region level transformations and optimizations implemented for our TAO. All the optimizations are implemented by a processor internal software, like the Transmeta Code Morphing Software [7].

Before a region is optimized, the region is transformed to enable optimizations. Specifically, a fake entry frame is created and connected to every public frame in the region, and empty fixup frames are created and connected to each frame in the region. The fixup frames and the fake entry are removed if they stay empty after applying the optimizations to the region. These created entry frame and fixup frames allow code to be hoisted to the region entry or sunk to the fixup frame after the exit frames. The following is a brief description of the region level optimizations.

Once the control flow graph for a region is constructed, function inlining is applied. This optimization expands non-recursive functions into their call sites, to 1) reduce function call/return overhead, 2) provide context sensitive data analysis information surrounding the called functions.

For every post-test loop inside a region, a pre-header frame is created. This frame initially is empty, and later optimizations, such as partial redundancy elimination, may move code from inside the loop to the pre-header frame. The pre-header is removed if it remains empty after all the optimizations are applied. Pre-test loops can be converted into post-test loops by applying the loop inversion transformation [23], however, we did not find a significant number of pre-test loops in our experiments and we did not implement this transformation.

Partial Redundancy Elimination (PRE) aggressively removes redundant operations by hoisting code upward to remove redundant computations along some of the paths reaching the split node. This operation subsumes traditional invariant and load hoisting, redundant load/store elimination, and forward propagations.

Partial Dead Code Elimination (PDE) pushes operations, including loads and stores, downwards to remove dead computations in some of the successor paths. When pushing an operation across a commit point of a frame, the operation is also pushed into the fixup frame, generating fixup code.

In addition to the above optimizations, global transformations could extend some of the frame level optimizations to region level. For example, frame level μop fusion [1] combines two or more

μops within the same frame to a single new μop that can be scheduled and/or executed as a single μop to reduce scheduling slots and execution delay. Global optimization framework can be leveraged to hoist or sink fuseable μops into the same frame and apply frame level fusion to accomplish the final fusions. Similarly, code motion can extend frame level SIMDization to region level.

6. Experimental Results

In this section we describe the infrastructure used and the results achieved with TAO. We first introduce our experimental framework and show the performance improvements, measured in terms of instructions per cycle (IPC), for FAO and TAO. Then, we compare the average dynamic sizes and the execution coverage for regions and frames. Finally, we discuss the impact of page mapping check on frames and regions.

We implemented the TAO framework on top of an x86 simulator. The simulator is a very detailed execution driven cycle accurate timing simulator for an out-of-order microarchitecture. The simulator was first modified to include the FAO framework, similar to PARROT [1, 27], and later extended to support the TAO framework. The atomicity for FAO and the first level atomicity for TAO is provided by the re-order buffer. The second level atomicity for TAO is modeled using a speculative cache with unbounded resources. We used the product quality simulation methodology, which includes around 350 single-threaded simulation traces distributed among 140 applications. The 140 applications belong to 8 different benchmark classes: SPEC 2006 floating point (FSPEC06) and integer (ISPEC06) applications, games, multimedia, office, productivity, server and workstation.

In our experiments, we use the following configurations:

- *base:* models an aggressive state-of-the-art 4-wide out-of-order microarchitecture.
- *FAO:* extends the base configuration by adding support for atomic frame execution, as described in Section 2. The frame optimizations are simple and mostly performed by hardware and the optimization overheads are modeled in the simulation.
- *TAO:* extends the FAO configuration by profiling and building large regions composed of frames, as described in Section 5. The TAO configuration uses the same size of frame cache, frame predictor and associated structures as the FAO configuration. We also charge an extra checkpoint operation when entering a region. The overhead associated with software optimizations for TAO is not measured in our experiments.

6.1 IPC improvement

Figure 7 shows the IPC improvement of FAO and TAO over the base microarchitecture. On average (geometric mean), FAO improves the base IPC by 16.3% and TAO increases another 5.3%, a total of 21.5% IPC improvement.

6.2 Dynamic sizes

Figure 8 shows the average dynamic size for TAO regions using the arithmetic mean. On average, each region executes about 8,900 x86 instructions before exiting the region.

Figure 9 shows that the average dynamic size for FAO frames is 27 instructions. The average size for regions is much bigger than for frames, which not only provides a larger optimization scope for TAO, but also helps to hide the extra overhead associated with "page mapping check" (see the effects of this property at Section 6.4).

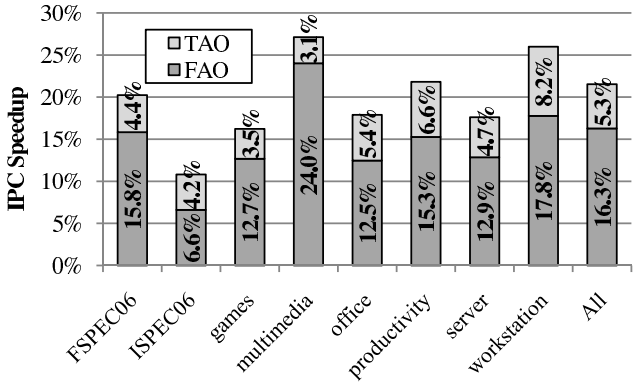

Figure 7. IPC speedup achieved by FAO and TAO.

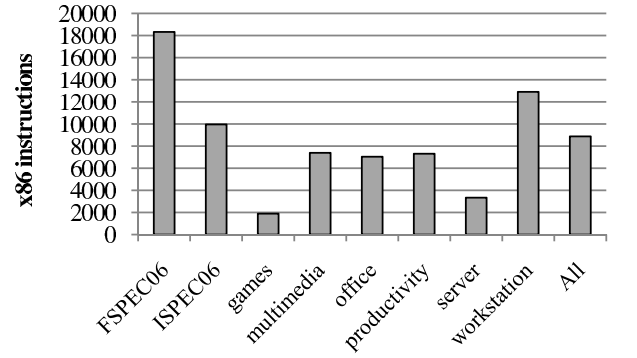

Figure 8. Dynamic region sizes for TAO.

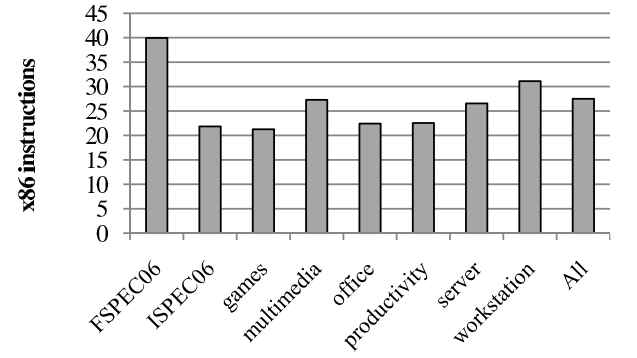

Figure 9. Dynamic frame sizes for FAO.

6.3 Execution coverage

Figure 10 shows the execution coverage for FAO and TAO. TAO has both region and frame coverage because the frames that could not be connected to other frames during the region formation step are left as stand-alone frames in the frame cache. On the average (arithmetic mean), TAO can cover $\sim$84% of dynamic instructions. Note that the coverage for FAO and TAO are very similar. This happens because our current implementation only allows hot frames in regions. We are investigating techniques to also include small basic blocks that are not qualified as frames, despite hot, into regions to further improve region coverage and performance.

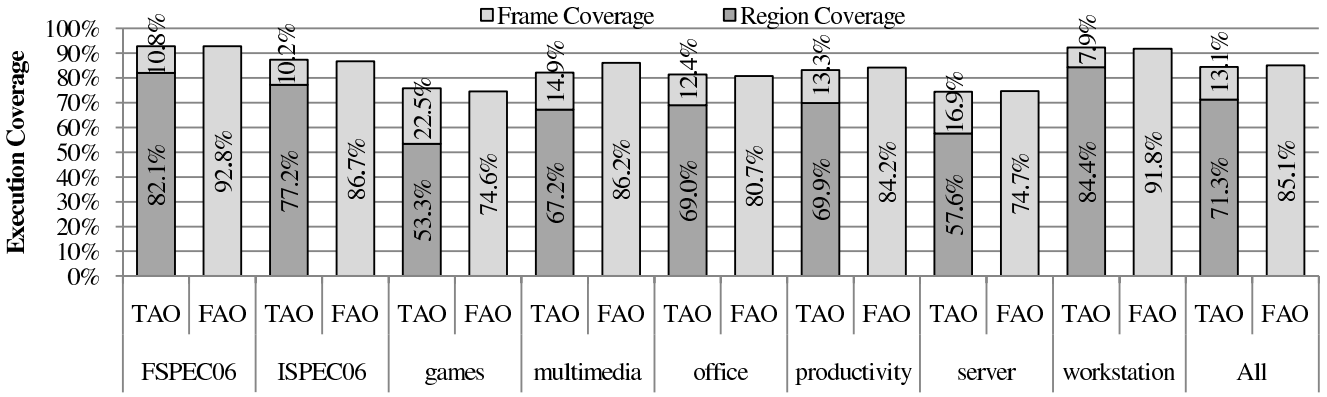

Figure 10. Frame and Region execution coverage for FAO and TAO.

6.4 Page mapping check (PMC)

The frame predictor mechanism predicts frames based on a virtual instruction pointer (IP). In this sense, whenever a frame is executed, the framework needs to ensure that the virtual IP still maps to the same physical address that it mapped during the region/frame formation. We call this task "Page Mapping Check". We experimented with two approaches for PMC:

- *Explicit PMC:* Every time a region or a stand-alone frame is executed, we check if the set of IPs represented by the region/frame still maps to the same physical addresses. In this case, there is an extra overhead for checking the physical IP for every region or stand-alone frame executed. The results reported before assume this approach.
- *Frame Cache Flush:* Every time the iTLB changes, we flush the frame cache. The disadvantage of this approach is the loss of coverage due to the reduced utilization of optimized frames.

Figure 11 shows the results for FAO when performing explicit PMC (FAO-PMC) and when flushing the frame cache on iTLB changes (FAO-Flush). Notice that the overall performance is very similar, but the IPC speedup varies for different classes of workloads. Specifically, for games and server benchmarks, the flush approach is worse due to reduced frame coverage caused by excessive frame cache flushes. The large code footprint in these benchmarks causes an increased amount of iTLB updates, which leads to frame cache flushes on the FAO-Flush. The FAO-Flush approach is better for Spec 2006, office and workstation benchmarks, which have very few iTLB flushes, due to the extra overhead associated with the explicit PMC in the FAO-PMC approach.

Figure 12 shows the results for TAO when performing the explicit PMC (TAO-PMC) and when flushing the frame cache on iTLB updates (TAO-Flush). Notice that the TAO-PMC is often better than TAO-Flush, and the performance difference is bigger than in the FAO experiment. The explicit PMC overhead for regions is amortized due to the large dynamic region size. On average, the checking is only performed at every 8,900 x86 dynamic instructions. As opposed to the FAO experiment, the checking overhead in TAO does not cause significant impact to the IPC speedup on the Spec 2006, office and workstation benchmarks.

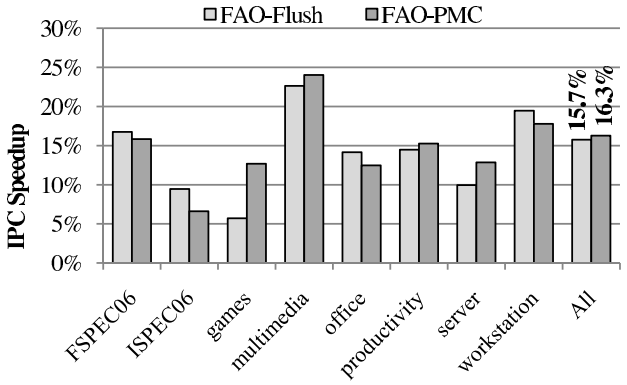

Figure 11. IPC Speedup for FAO-PMC and FAO-Flush.

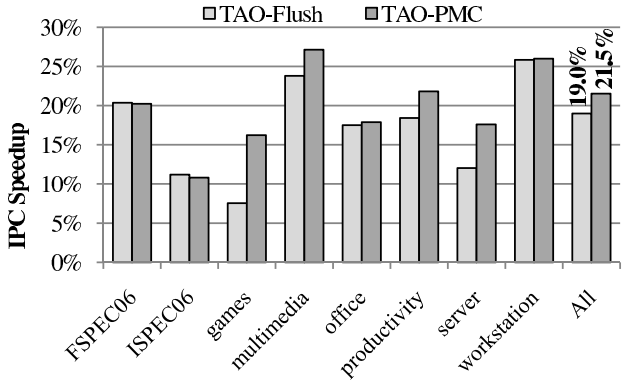

Figure 12. IPC Speedup for TAO-PMC and TAO-Flush.

7. Related Work

This work is related to several research areas, such as dynamic binary translation, dynamic optimization, atomic execution, and nested atomic execution.

The frame-level atomic optimizations were previously introduced by Patel and Lumetta [25] and latter explored by others [1, 27, 29]. Similar to PARROT [27], our experimental results show that frame-level atomic optimizations can achieve an average of 16% IPC improvement. The TAO approach enables large region optimizations and boosts IPC improvement to 21.5%.

HW/SW co-designed systems [7, 15, 16] leverage DBT SW and special HW support to enable efficient emulation of source ISA code at system level and can potentially improve performance,

reduce energy and processor silicon. For example, Transmeta Efficeon [16] used a DBT SW layer to implement x86 processors. The DBT SW translates x86 code into an internal VLIW code for execution with promising power/performance benefits.

IBM Research's DAISY and BOA projects [9] utilize a VLIW architecture to emulate PowerPC processors. A similar project [12] proposed binary translation assistance for an IBM System/390 mainframe design.

In addition to HW based dynamic optimization, such as rePLay [25, 29] and PARROT [1, 27], SW based dynamic optimizations can be done at user level as well as at system level. There are many research papers and systems on dynamic binary translation (DBT), e.g. IA32-EL [2], StarDBT [31], HDTrans [30], DynamoRIO [3], and Pin [18]. User level DBT might not easily benefit from internal HW support and thus usually suffers from translation and emulation overhead. For example, IA32-EL [2] runs IA32 applications on Intel IPF platforms, achieving 40% to 60% of native performance.

Speculative optimizations enabled by atomic execution demonstrate benefits at both source level and binary level. Chen et al. [4] proposed atomicity support in a static compiler to optimize and parallelize C/C++ program aggressively. Neelakantam et al. used HW atomicity to optimize Java programs in a JVM [24] and reported 10-15% average speedup, however, the optimizations were done at bytecode level and cannot be applied to legacy C/C++ and binary code. TAO is implemented at microarchitecture level and can optimize legacy binary code transparently. Although rePLay [25, 29], and PARROT [1, 27] also used atomic HW support at binary (μop) level, TAO enables the optimization of large regions of code (including loops).

Atomicity and recovery are also the foundation of thread-level speculations (TLS) [5, 8, 17, 19]. TLS speculatively runs potentially conflicting regions of code in parallel, and when dependency violation is detected at runtime, some of the threads may roll back to the beginning of the region using the HW atomicity support.

Transactional memory (TM) [13] supports atomicity, isolation and consistency. These features allow TM to achieve opportunistic parallelism in places where locks used to be required. When conflicting data accesses happen within a TM region, the atomicity support allows the transaction to abort and recover from the conflict. Because of its atomicity, a transactional memory system can be leveraged to support atomic optimizations [24].

Transactional memory may also allow nested atomicity [22]. Specifically, closed nested transactions may allow partial abort of inner transactions and roll back to the beginning of the inner transactions (nested atomicity). However, nested atomicity may result in deadlock when an inner transaction aborts and re-executes the same code. Some implementations may flatten nested transactions into the top-level transaction, resulting in a complete abort on conflict (single level atomicity). Although TAO employs two-level atomicity, it avoids any dead lock by fixing the architectural state and going to un-optimized code when an inner atomic frame aborts.

8. Conclusions and Future Work

Dynamic binary optimizations benefit greatly from atomicity support to achieve performance gain. Both frame-level and region level atomic optimizations have their advantages and limitations respectively. Frame-based optimizations can leverage processor's pipeline buffers to support atomic execution of hot traces of straight-line code. It is efficient, but the optimization scope is small and performs poorly when branches are unpredictable. Region level optimizations can aggressively optimize large regions of code, but may suffer from higher rollback overheads. We have not seen any previous attempt to combine the two approaches. In this paper, we present TAO, a two-level atomic optimization framework, which not only overcomes the limitations of the two approaches, but also boosts the benefits of the two approaches effectively. Our experiment shows that the combined approach can significantly improve the frame-based approach by 5.3%, and has the potential to enhance the region level atomic optimization.

This work can be expanded in several directions. First, our current regions consist of frames only. Since frames have to be larger than a minimal size, some hot small blocks are not transformed into frames and excluded from our regions. In the future, we may also include these hot basic blocks into our regions, which would potentially increase region coverage, make regions larger and improve TAO's performance gain. Second, more elaborated optimizations, like code vectorization and parallelization, could also be employed to improve the execution of the atomic regions. These optimizations can take advantage of the region atomicity support, which avoids memory ordering issues. Third, in our current implementation of TAO, a speculative checkpoint is recorded at the entry of every frame. We could assigning speculative checkpoints to only a few selected frames to reduce the constraints imposed on the region optimizations. These checkpoints should be carefully assigned so that the amount of speculative data produced between checkpoints is small enough to fit the pipeline buffers. Finally, HW and SW trade-offs for frames storage (HW frame cache versus memory code cache) and region optimizations (HW versus SW optimizers) should be investigated more closely.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and discussions. We also appreciate the support provided by Jesse Fang at the Programming System Laboratory and colleagues at the Mobility Group at Intel.

References

[1] Almog, Y., Rosner, R., Schwartz, N., and Schmorak, A. Specialized Dynamic Optimizations for High-Performance Energy-Efficient Microarchitecture. In Proceedings of the international symposium on code generation and optimization (CGO'04), Palo Alto, CA, 2004.

[2] Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skalesky, A., Wang, and Y., Zemach, Y. IA-32 Execution Layer: A Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems. In Proceedings of the 36th international symposium on microarchitecture (MICRO'03). San Diego, CA, 2003.

[3] Bruening, D. L. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D thesis, Massachusetts Institute of Technology, 2004.

[4] Chen, L-L. and Wu, Y. Aggressive Compiler Optimization and Parallelization with Thread-Level Speculation. In Proceedings of international conference on parallel processing (ICPP'03). Kaohsiung, Taiwan, 2003.

[5] Chen, M. K. and Olukotun, K. The Jrpm System for Dynamically Parallelizing Java Programs. In Proceedings of the 30th annual international symposium on computer architecture (ISCA'03). San Diego, CA, 2003.

[6] Colwell, B., and Steck, R. A 0.6 um BiCMOS processor with dynamic execution. In Digest of Technical Papers of 1995 IEEE international solid-state circuits conference (ISSCC'95). San Francisco, CA, 1995.

[7] Dehnert, J. C, Grant, B., Banning, J. P., Johnson, R., Kistler, T, Klaiber, A., and Mattson, J. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In Proceedings of the international symposium on code generation and optimization (CGO'03). San Francisco, CA, 2003.

[8] Du, Z.-H., Lim, C.-C., Li, X.-F., Yang, C., Zhao, Q., and Ngai, T.-F. A cost-driven compilation framework for speculative parallelization of sequential programs. In Proceedings of the ACM SIGPLAN 2004

conference on programming language design and implementation (PLDI'04). Washington, DC, 2004.

[9] Ebcioglu, K., Altman, E., Gschwind, M., and Sathaye, S. Dynamic Binary Translation and Optimization. IEEE Transactions on Computers.50, 6 (Jun. 2001), 529-548.

[10] Fahs, B., Mahesri, A., Spadini, F., Patel, S., and Lumetta, S. The Performance Potential of Trace-based Dynamic Optimization. Tech. report, University of Illinois at Urbana-Champaign, 2005.

[11] Gopal, S., Vijaykumar, T. N., Smith, J.E., and Sohi, G.S. Speculative Versioning Cache. In Proceedings of the 4th international symposium on high performance computer architecture (HPCA'98). Las Vegas, NV, 1998.

[12] Gschwind, M., Ebcioglu, K., Altman, E., and Sathaye, S. Binary Translation and Architecture COnvergence issues for IBM System 390. In Proceedings of International Converence on Supercomputing, Santa Fe, NM, 2000.

[13] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In Proceedings of the 20th annual international symposium on computer architecture (ISCA '93). New York, NY, 1993.

[14] Kim, H-S. and Smith, J. Hardware Support for Control Transfers in Code Caches. In proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03). Washington, DC, 2003.

[15] Klaiber, A. The Technology Behind the Crusoe Processors. White Paper, http://www.charmed.com/PDF/CrusoeTechnologyWhitePaper_1-19-00.pdf, Jan. 2000.

[16] Krewell, K. Transmeta Gets More Efficeon. Microprocessor report. v.17, October, 2003

[17] Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., and Torrellas, J. POSH: a TLS compiler that exploits program structure. In Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP'06). New York, NY, 2006.

[18] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney G., Wallace, S., Reddi, V., and Hazelwood K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation (PLDI'05). New York, NY, 2005.

[19] Luo, Y., Packirisamy, V., Hsu, W.-C., Zhai, A., Mungre, N., and Tarkas, A. Dynamic performance tuning for speculative threads. In Proceedings of the 36th annual international symposium on computer architecture (ISCA'09). Austin, TX, 2009.

[20] Merten, M. C., Trick, A. R., George, C. N., Gyllenhaal, J. C., and Hwu, W-m. W. A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. In Proceedings of the 26th annual international symposium on computer architecture (ISCA'99). Atlanta, GA, 1999.

[21] Merten, M. C., Trick, A. R., Nystrom, E. M., Barnes, R. D., and Hwu, W-m. W. A hardware mechanism for dynamic extraction and relayout of program hot spots. In Proceedings of the 27th annual international symposium on computer architecture (ISCA'00). Vancouver, Canada, 2000.

[22] Moravan, M., Bobba, J., Moore, K., Yen, L., Hill, M., Liblit, B., Swift, M., and Wood, D. Supporting nested transactional memory in logTM. In Proceedings of the 12th international conference on architectural support for programming languages and operating systems (ASPLOS'06). San Jose, CA, 2006.

[23] Muchnick, S. S. Advanced compiler design and implementation, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1998

[24] Neelakantam, N., Rajwar, R., Srinivas, S., Srinivasan, U., and Zilles, C. B. Hardware atomicity for reliable software speculation. In Proceedings of the 34th annual international symposium on computer architecture (ISCA'07). San Diego, CA, 2007.

[25] Patel, S. J. and Lumetta, S. S. rePLay: A Hardware Framework for Dynamic Optimization. IEEE Transactions on Computers.50, 6 (Jun. 2001), 590-608.

[26] Patel, S., Tung, T., Bose, S., and Crum, M. Increasing the size of atomic instruction blocks using control flow assertions. In Proceedings of the 33rd annual ACM/IEEE international symposium on microarchitecture (MICRO'00), Monterey, CA, 2000.

[27] Rosner, R., Almog, Y., Moffie, M., Schwartz, N., and Mendelson, A. Power Awareness through Selective Dynamically Optimized Frames. In Proceedings of the 31st annual international symposium on computer architecture (ISCA'04). Mnchen, Germany, 2004.

[28] Rotenberg, E., Bennett, S., and Smith, J. Trace cache: A low latency approach to high bandwidth instruction fetching. In Proceedings of the 29th international symposium on microarchitecture (MICRO'29). Paris, France, 1996.

[29] Slechta, B., Crowe, D., Fahs, B., Fertig, M., Muthler, G., Quek, J., Spadini, F., Patel, S. J., and Lumetta, S. S. Dynamic Optimization of Micro-Operations. In Proceedings of the 9th international symposium on high-performance computer Architecture (HPCA'03), Washington, DC, 2003.

[30] Sridhar, S., Shapiro, J. S., Northup, E., and Bungale, P. HDTrans: An Open Source, Low-Level Dynamic Instrumentation System. In Proceedings of the 2nd international conference on virtual execution environments (VEE'06), Ottawa, Canada, 2006.

[31] Wang, C., Hu, S., Kim, H-S., Nair, S. R., Breternitz Jr., M., Ying, Z., and Wu, Y. StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In Proceedings of Asia-pacific computer systems architecture conference, 2007.