TSO-Atomicity: Efficient TSO Enforcement for Aggressive Program Optimization

Cheng Wang, Youfeng Wu, Jaewoong Chung

Programming Systems Lab
Microprocessor and Programming Research
Intel Labs
cheng.c.wang@intel.com, Youfeng.wu@intel.com, Jaewoong.chung@intel.com

Abstract

Strong memory consistency models like Sequential Consistency (SC) and Total Store Ordering (TSO) impose tremendous memory consistency constraints on program optimizations. Existing techniques leverage HW atomicity supports to enable aggressive optimizations while enforcing sequential consistency. By grouping the optimized code into a region and committing the whole region atomically, sequential consistency is preserved. But for optimization in TSO, the sequential consistency enforced by HW atomicity supports imposes stronger consistency than necessary, resulting in in inefficient region commit operations. In this paper, we introduce the HW support for a novel TSO-atomicity, which provides weaker consistency than atomicity for efficient region commit, but still can enforce TSO for aggressive optimizations. We show in our experiments that, dynamic binary optimizations without HW atomicity supports can improve binary program performance by only 6.7% and 8.1% in frame-level (around 20 instructions per frame) and region-level binary optimization (around 70 instructions per region) respectively. With HW atomicity supports, region-level binary optimization can improve the performance further by 7.8%. However, frame-level binary optimization actually losses performance by 2.7% due to the overheads associated with the commit for atomicity. HW supports for TSO-atomicity can reduce the overheads at region commit and achieve a total 12% and 20.4% performance gain for frame-level and region-level binary optimization respectively.

1. Introduction

Memory consistency models describe how threads may interact through shared memory consistently, and thus program optimizations on multi-core systems must consider the memory consistency models for correct and efficient transformations. Strong memory consistency models like Sequential Consistency (SC) [15] and Total Store Ordering (TSO) [26] impose tremendous memory consistency constraints on optimizations. For example, we do not allow any reordering of shared memory operations in optimization for SC, even if the reordering does not violate any control/data dependences. In optimization for TSO, no reordering of memory operations is allowed except between an earlier store operation and a later load operation. Due to that, high level programming languages, such as C++ [6] and Java [17], typically adopt weak memory consistency models for efficient compiler optimizations, and rely on programmer to explicitly specify the synchronizations between concurrent threads to prevent data-racing on shared memory.

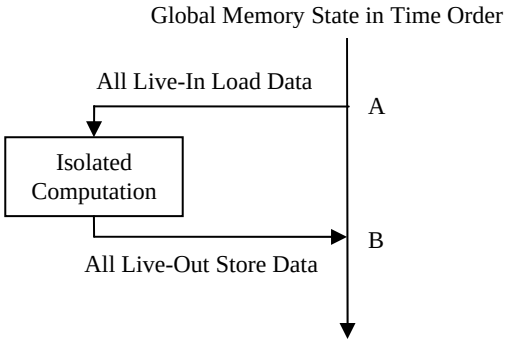

Figure 1: Logical View of TSO-Atomicity

Previous work [2] has shown that the HW atomicity supports can enforce sequential consistency while enabling aggressive optimizations. By grouping the optimized code into a region and committing the whole region atomically, the execution orders of individual instructions within the region cannot be observable by other concurrent threads. Thus sequential consistency is preserved, even with reordered memory operations in the optimizations.

Unfortunately, for optimization in TSO, the sequential consistency enforced by HW atomicity supports incidentally imposes stronger consistency than necessary, and the stronger consistency leads to inefficient region commit operations. For example, processors implementing TSO allow later load operations to pass earlier independent store operations in the global memory order. That relaxed memory order reduces unnecessary pipeline stalls on load operations due to cache misses of earlier store operations. Due to that, popular processors like X86 [28] and SPARC [26] implements TSO instead of consequential consistency. However, the HW atomicity support needs to provide the strict global memory order for memory operations across the region boundary, which may introduce additional pipeline stalls at the region boundary as compared to the execution without atomicity. Previous works [14][21][23] have shown that in hot-spot based dynamic binary optimization, the average region size is usually small. So the strict memory order across region boundary can cause significant performance loss. As we will see in our experiments, the small region size makes the overheads across region boundary a big performance concern to dynamic binary optimization.

In this paper, we introduce the HW support for a novel TSO-atomicity, which provides weaker consistency than atomicity for more efficient region commit operations, but can still enforce TSO for aggressive optimizations. Based on the fact that TSO

allows later load operations to pass earlier independent store operations, TSO-atomicity provides atomicity for load operations and store operations separately. So instead of providing the atomicity for all the memory operations in the region, TSO-atomicity only provides atomicity among all load operations and atomicity among all store operations. There is no atomicity between load operations and store operations. Instead, later load operations can pass earlier independent store operations in the global memory order, just like that later load operations can pass earlier independent store operations in the global memory order in TSO. Also, earlier stores forward data to later dependent loads to respect the data dependences within the thread, just like the store-buffer forwarding supported in TSO processors [28]. That reduces unnecessary pipeline stalls on load operations in later regions due to cache misses of store operations in earlier regions. Figure 1 shows the logical view of a region execution with TSO-atomicity. All the live-in load operations occur atomically at time A and all the live-out store operations occur atomically at time B. So logically, we can view the region execution as first reading all the live-in data atomically at time A, then performing all the computation in isolation (with aggressive optimizations) from other concurrent thread execution and at last, writing back all the live-out data atomically at time B.

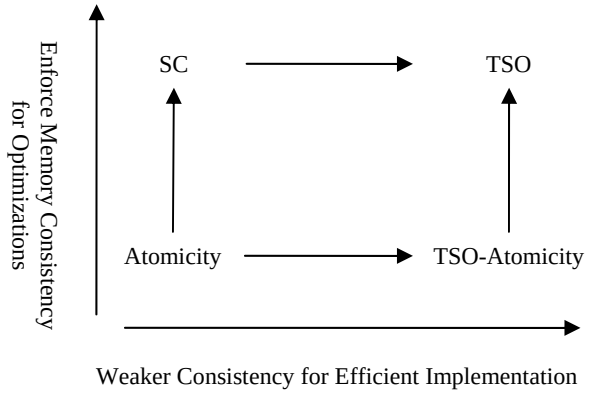

Figure 2: Relationship among SC, TSO, Atomicity and TSO-Atomicity

Interestingly, the TSO-atomicity with separated atomicity among load operations and store operations still can enforce TSO for optimizations. Since given the live-in data, no optimization can affect the live-out data generated by the computation. Consequently, no optimization will affect the region execution in TSO-atomicity. Figure 2 shows the relationship among SC, TSO, atomicity and TSO-atomicity.

We show in Section 3 that TSO-atomicity can be easily implemented by slight modifications to the existing HW atomicity implementation, but more efficiently. We developed a dynamic binary optimization infrastructure for X86 processors (supporting TSO model) to evaluate the performance benefit. We show in our experiments that, dynamic binary optimizations without atomicity or TSO-atomicity supports can improve performance by only 6.7% and 8.1% in small frame-level optimization (around 20 instructions per frame) and large region-level optimization (around 70 instructions per region) respectively. With atomicity supports, region-level binary optimization can improve the performance further by 7.8%. However, frame-level binary optimization will actually see 2.7% performance loss due to the overheads associated with the frame commit for atomicity. TSO-atomicity supports can avoid the overheads in commit for atomicity and improve the performance by 5.3% and 12.3% to a total 12% and 20.4% for frame-level and region-level binary optimization respectively.

The major contributions of the paper are summarized as follows:

- To support aggressive optimizations, e.g. reordering loads and stores, we introduced a novel TSO-atomicity, which provides weaker consistency than atomicity for more efficient commit operation, but can still enforce TSO memory order.
- We showed how TSO-atomicity can be implemented efficiently by slight modifications to the existing HW atomicity implementation.
- We developed a dynamic binary optimization infrastructure and provide experimental data to qualify the performance benefits of TSO-atomicity supports over atomicity, in both frame-level and region-level optimizations in the dynamic binary optimization infrastructure.

The rest of the paper is organized as follows: We first highlight some differences between atomicity and TSO-atomicity in Section 2. We discuss the HW implementations for SC, TSO, atomicity and TSO-atomicity in Section 3. We present our experimental results in Section 4. Finally, we discuss the related work in Section 5, and conclude the paper and point out the future directions in Section 6.

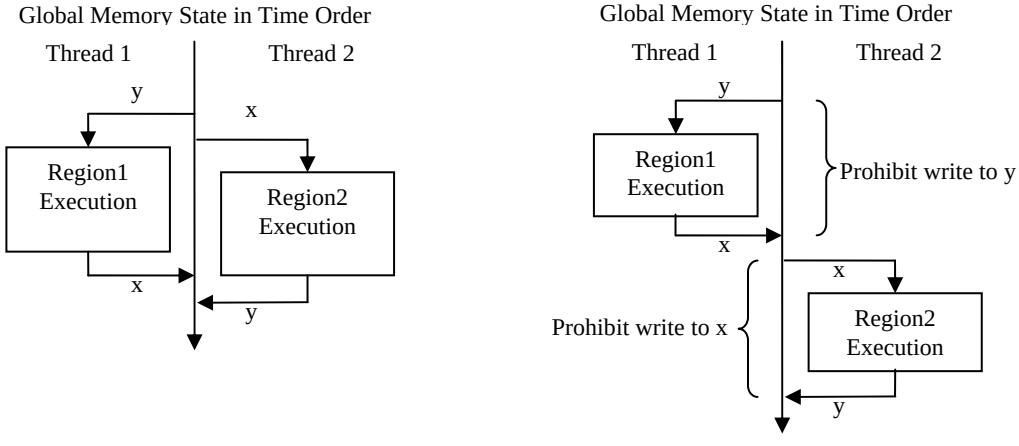

Figure 3: Comparison between TSO-Atomicity and Atomicity

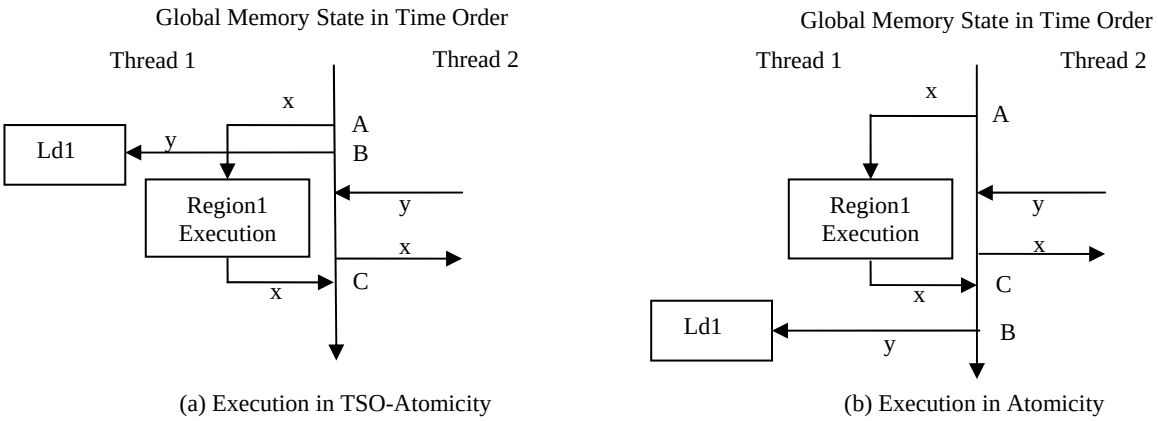

Figure 4: Comparison between TSO-Atomicity and Atomicity

2. Differences between Atomicity and TSO-Atomicity

In this section, we highlight some differences between atomicity and TSO-atomicity. TSO-atomicity provides weaker consistency than atomicity, which makes it more efficient to implement on TSO processors than atomicity. Unlike atomicity, TSO-atomicity allows the overlapped execution of regions in concurrent threads, even if they are not serializable. Figure 3 (a) shows the overlapped execution for region1 in thread 1 and region2 in thread 2. Due to the interleaving accesses to data x and y, the executions of the two regions are not serializable, but are allowed in TSO-atomicity because TSO allows that interleaving memory accesses. Atomicity imposes unnecessary consistency constraints in this example, which restricts the execution of the two regions to be serializable. Specifically, atomicity does not allow the write to x in region1 between the read of x and write of y in region2. So in execution with atomicity, region2 may be aborted and rolled back. Then the final execution for the two regions will be as shown in Figure 3 (b). The relaxed consistency in TSO-atomicity eliminates the unnecessary rollbacks for atomicity in this example.

Moreover, within the same thread, TSO-atomicity allows the overlap of region execution with later loads. Figure 4 (a) shows an example. Here Ld1 is a load succeeding region1 in program order, which is allowed to pass the write of x in region1 in global memory order because TSO allows that. Atomicity imposes an unnecessary consistency constraint in this example, which enforces a strict global memory order across the region boundary. Otherwise, the write to y and the read to x in thread 2 as shown in Figure 4 (a) can observe the reordered write of x and read of y in thread 1. So in execution with atomicity, the execution of region1 cannot overlap with Ld1 and the final execution will be as shown in Figure 4 (b). The relaxed consistency constraint in TSO-atomicity may eliminate the unnecessary stalls on Ld1 in this example.

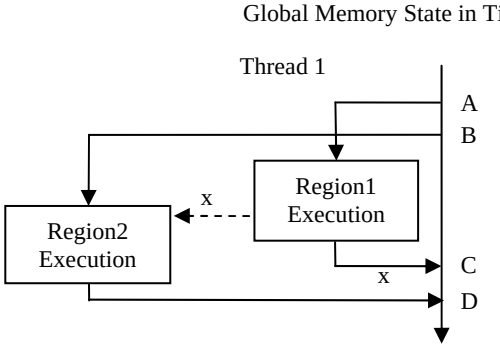

Figure 5: Overlap of Execution within Thread

Furthermore, TSO-atomicity allows the executions of regions in the same thread to overlap. In case that the live-out data of the earlier region contain the live-in data of the later region, TSO-atomicity still allows the execution of the two regions to overlap, as long as the earlier region can forward the live-out data to the later region to respect the data dependences within the thread. Figure 5 shows two consecutive regions in thread 1, region1 and region2, whose executions overlap. If x is live-out from region1 and live-in to region2, region1 needs to forward data x to region2, as denoted by the dotted line.

3. HW Implementations

3.1 SC and TSO Implementation

Sequential Consistency (SC) model [15] requires serialization of the program execution, i.e. all executed memory operations must occur sequentially in a global memory order consistent with the original program order. Although it is easy for verifying program correctness, SC model is inefficient to implement in hardware, as no memory operations from the same processor can be reordered in the global memory order. Specifically, store instructions are not allowed to retire until the store data are globally visible. That may unnecessarily stall a load from retirement due to the cache miss of a previous store, even if we know there is no dependence between them. Figure 6 (a) shows an execution in SC. To show the stalls on stores due to store cache miss, we use "store execute" to indicate that the store starts to execute and "store retire" to indicate that the store retires after the store data becomes globally visible. In order for the store data to be globally visible through the cache coherence protocol, the store miss has to be resolved. So the gap between "store execute" and "store retire" shows the store miss latency. In this example, store st3 cannot retire until its cache miss has been resolved, which can take significant time in modern processors.

TSO memory model is more efficient for program execution. A store operation can write the store data into a store-buffer and retire without waiting for the store data to be globally visible (i.e. without waiting for the store miss to be resolved). Thus the later

loads can also retire without waiting for the cache misses of all the previous stores to be resolved. As the result, later loads may occur before earlier stores in the global memory order. Figure 6 (b) shows the program execution in TSO. Here we use "store retire" to indicate that the store retires. After a store retires, the store data may stay in store-buffer waiting for the store miss to be resolved. We use "store write" to indicate that the store data is written from store-buffer to the cache after the store miss is resolved and the store data becomes globally visible. In this example, store st3 can retire before it writes data to the cache. Load ld4 can also retire before the store data of st3 is written from store-buffer to the cache. Store st5 can retire with its store data appended to the store-buffer. In case that a later load ld6 reads the store data of st3, ld6 can get forwarded data of st3 from the store-buffer and retire, before st3 miss is resolved (i.e. store-buffer forwarding). Compared to the execution in SC model, program execution in TSO model clearly is much more efficient.

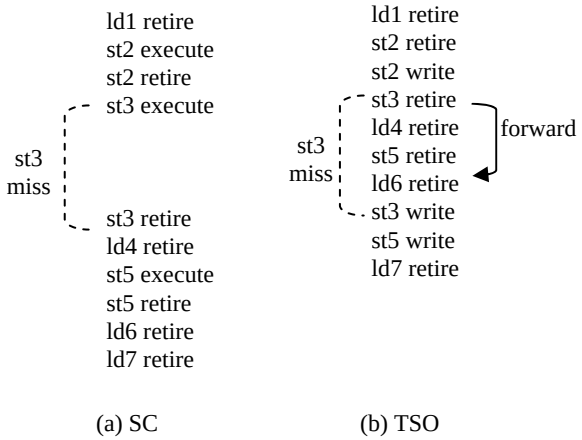

Figure 6: Execution in SC and TSO

3.2 Atomicity Implementation

We consider a typical HW atomicity support which extends the data cache with speculative read and write bits (a speculative cache) to buffer all the store data in the region, and leverage the cache coherence protocol to snoop all the conflicting memory operations in other concurrent threads and check them against the speculative data marked with the speculative read/write bits in the speculative cache. If there is no conflict memory access detected at the end of the region execution, the whole region commits atomically by clearing all the speculative read/write bits in the speculative cache. In case a memory conflict is detected, the region will be completely rolled back by invalidating all the buffered speculative store data and clearing all the speculative read/write bits in the speculative cache. HW may also need to checkpoint the registers at the region entry for region rollback.

The HW atomicity implementation has some inherent inefficiency on processors implementing TSO. To support snooping for a speculative load, HW implementation for atomicity needs to set the speculative read bit to the cache line when the load retires. To support snooping for a speculative store, the HW implementation for atomicity needs to set the speculative write bit to the cache line when the store data is written from the store-buffer to the cache (i.e. after the cache miss is resolved and the ownership for the cache line in cache coherence protocol is acquired). Then for atomicity, we need to drain all the "senior" store data in the region from the store-buffer to the cache before the region can commit. Since a store cannot be drained to the cache until its cache miss is resolved, the draining may stall the load instructions after the region from retirement for many cycles (i.e. waiting for all the pending store misses in the region to be resolved).

Figure 7 (a) shows the region execution in atomicity. Assume region1 contains instructions ld1– st3 and instructions ld4 – ld7 follow region1. Also assume that store st3 misses cache at the end of region1 execution. In this case, we need to wait for the store miss to be resolved before region1 can commit (see Region1 commit in the figure). That will stall all the instructions ld4-ld7 from retirement waiting for the store miss to be resolved, just like the stall of instructions from retirement waiting for the store miss to be resolve in the execution in SC as shown in Figure 6 (a). Note that for atomicity, all the loads and stores in the region are snooped to detect memory conflict (see Region1 snoop in the figure) during the region execution until the region commits.

To prevent the later instructions from stalls on retirement during the draining of store-buffer at the region commit, we may deploy additional HW structures/buffers [5]. However, this increases the HW cost and is more complex than our TSO-Atomicity implementation discussed in the following section.

3.3 TSO-Atomicity Implementation

TSO-atomicity can be implemented by only slight modifications to the atomicity implementation. Instead of atomically commit-

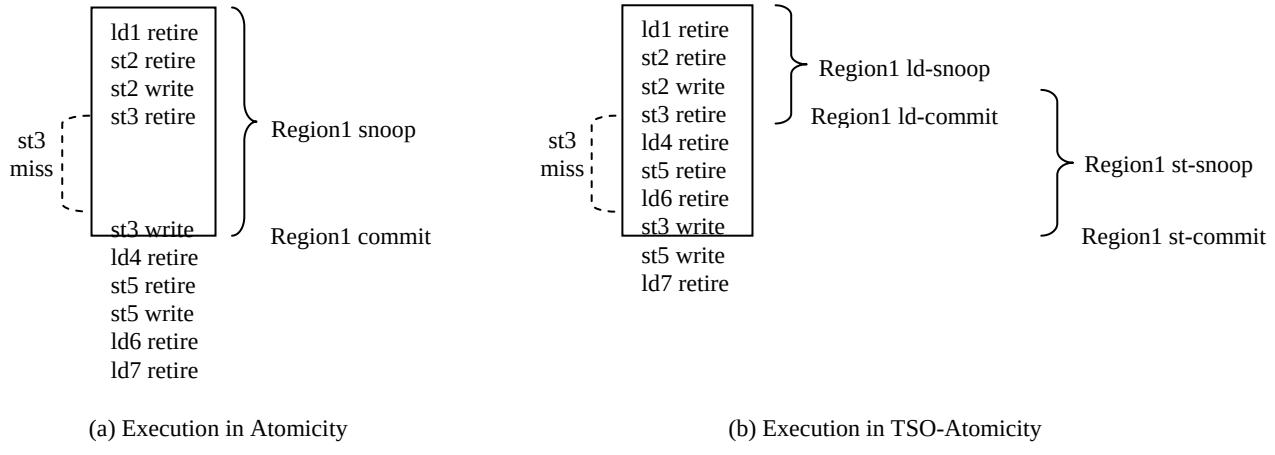

Figure 7: Snoop and Commit in Atomicity and TSO-Atomicity

Application Categories	Number of Applications (and examples)	#traces
SERVER	14 (e.g. specweb, google_query, oracle_kernel, tpce)	46
WORKSTATION	39 (e.g. eclipse, autocad, verilog, bioinformatics, fluent)	82
MULTIMEDIA	34 (e.g. adobe, sysmark, mobilemark, renderman, cinebench)	73
CONSUMER ELECTRONIC (CE)	26 (e.g. pcmark, h264, mpeg2, real_video, mp3)	44
GAME	20 (e.g. doom3, quake4, battlefield2, splintercell3, ut2004)	31
OFFICE	15 (e.g. excel, access, word, outlook, powerpoint, flash)	34
PRODUCTIVITY TOOL (TOOL)	22 (e.g. xml-parser, winzip, virus-scan, winrar, openssl)	33
SPEC CPU2006 INT (ISPEC06)	9 (all except gobmk, sjeng, xalancbmk)	25
SPEC CPU2006 FP (FSPEC06)	16 (all except tonto)	33

Table 1: Applications

ting the whole region after all the data in store-buffer are drained to cache, we allow two separate commits: a *load-commit* and a *store-commit*. Load-commit atomically commits all the load operations in the region when the last instruction in the region retires. After load-commit, all the speculative read bits in the speculative cache are cleared (i.e. stop snooping on loads). Store-commit atomically commits all the store operations in the region after all stores are written from store buffer to the cache. After store-commit, all the speculative write bits in the speculative cache are cleared (i.e. stop snooping on stores). Between load-commit and store-commit, we keep the register checkpoint for the region in case that the whole region may be rolled back, e.g. due to some conflicts on stores.

TSO-atomicity can prevent the instructions after the current region from stalls on retirement. Since the region performs load-commit when all the instructions in the current region are retired, load instructions after the region are allowed to retire once all the loads in the region are committed without waiting for the draining of store-buffer, and stores after the region can retire as long as the store buffer has space. For back-to-back regions, after all the loads in the current region are committed, all the speculative read bits in the speculative cache are cleared so that they can be used by the retired load operations from the following region. The retired stores in the following region are just appended to the store-buffer. The following region is also allowed to perform load-commit without waiting for the stores in current region are committed.

Figure 7 (b) shows the region execution in TSO-atomicity. Region1 has two separate commits: load-commit (see Region1 ld-commit in the figure) and store-commit (see Region1 st-commit in the figure) and, two corresponding periods of snooping for load operations (see Region1 ld-snoop in the figure) and store operations (see Region1 st-snoop in the figure). After st3 retires, we perform load-commit for region1 and stop the snooping on loads in region1 by clearing all the speculative read bits in the speculative cache. After that, instructions ld4-ld7 are allowed to retire without waiting for st3's cache miss to be resolved. After all the store data in region1 are written from the store-buffer to the cache (i.e. store miss for st3 is resolved), we perform store-commit for region1 and stop the snooping on stores in region1 by clearing all the speculative write bits in speculative cache. Consequently, TSO-atomicity does not enforce the strict global memory order across region boundary and can operate more efficiently than atomicity, just like that TSO can be implemented more efficiently than SC model as shown in Figure 6. Since the region commits load operations earlier, TSO-atomicity may also potentially reduce the region aborts due to memory conflicts on the load operations, if the conflicts happen between the load-commit and the store-commit (i.e. allow the execution shown in Figure 3(a)).

We should note that TSO-atomicity is weaker than atomicity. We can view the atomicity implementation as a special case of TSO-atomicity implementation such that the load-commit and store-commit occur at the same commit point. Unlike atomicity, TSO-atomicity cannot ensure the strict memory order across memory fence/barrier instructions or lock/atomic instructions. Due to that, TSO-atomicity does not allow optimizations to cross these synchronization instructions, which is not attempted in hot-spot based dynamic binary optimizations anyway [23].

4. Experiments

We implemented a dynamic binary optimization infrastructure to evaluate the performance effects of the memory model on binary optimizations, using a product-quality execution-driven timing-accurate simulator targeting future generation of 4-wide out-of-order X86 processors. The base line performance of the simulator is consistent to high-end 4-wide out-of-order processor with IPC around 1.5. So we only show the relative performance data in our experiments.

The simulator front-end translates the X86 instructions into internal operations and the simulator back-end only executes the internal operations. Our dynamic binary optimization works on the internal operations and the optimized code is buffered internally for re-execution to reduce front-end fetch/decoding overheads. This also allows the optimizer to use more internal registers than the X86 ISA allows and leverage ISA invisible HW features such as fusion [13] and assert [21] in the optimized code for performance improvement.

Our dynamic binary optimization performs not only frame-level binary optimizations similar to replay [20] and Parrot [23], but also large region-level binary optimizations similar to [7]. A frame is a superblock with all internal branches converted to asserts [21], and frame-level binary optimizations are applied on them. Large regions are built on top of frames based on branch profile information, and both frame-level and region-level binary optimizations are applied on them. We form frames similar to Parrot [23] and connect frames into regions similar to IA32EL [4]. The optimization overhead for the frames is modeled as done in HW as in Parrot [23].

Our simulator supports both atomicity and TSO-atomicity. These two supports differ mainly in their commit operations. A atomicity frame/region commit stalls late instructions from retire until the store buffer is fully drained. An TSO-atomicity frame/region commits all loads as soon as all the instructions in the frame/region are retired. After that, instructions after the frame/region can retire without waiting for the draining of store buffer. When the last store data of the frame/region is written from store-buffer to the speculative cache, the whole frame/region is committed.

We measure the performance for the following three configurations:

- *No mem opt*: perform only register-based and branch related optimizations without HW atomicity/TSO-atomicity support

- *Opt with atomicity*: perform all optimizations with HW atomicity support
- *Opt with TSO-atomicity*: perform all optimizations with HW TSO-atomicity support

We measure performance benefits for around 400 traces covering about 200 real world applications in nine categories shown in Table 1. Each trace is selected from a representative piece of

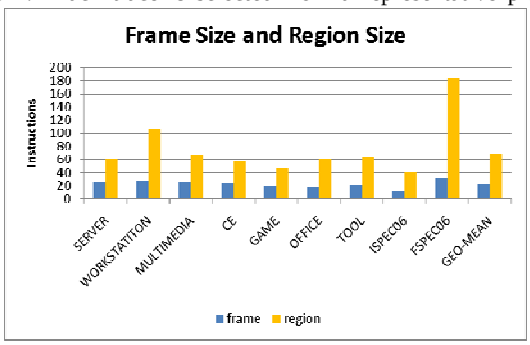

Figure 8: Dynamic Size

code in an application. We follow the common approach that collects performance data after the simulator warms up microarchitectural state.

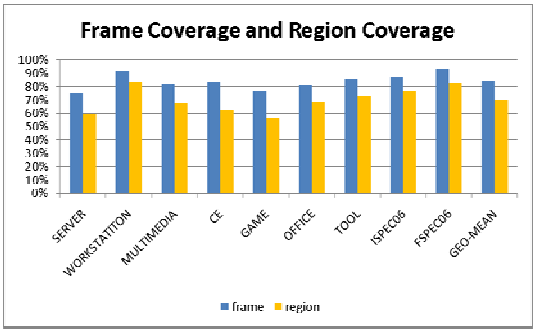

Figure 9: Dynamic Coverage

4.1 Experimental Results

Figure 8 and Figure 9 shows the dynamic size and coverage for frame and region, weighted by the execution frequency, for each category. On average, each frame contains about 20 instructions and each region dynamically run about 70 instructions. On average, the frames cover about 84% of all the instructions executed.

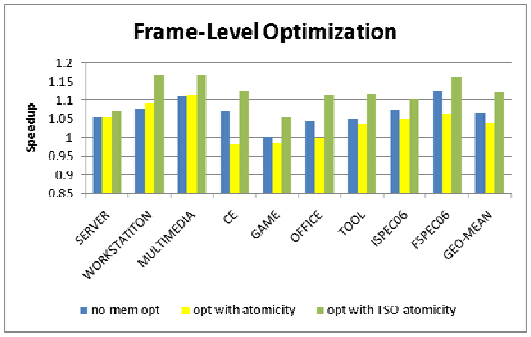

Figure 10: Performance on Frame-Level Optimization

Only instructions inside frames are optimized for frame-level binary optimizations. On average, the region covers about 70% of all the instructions executed. Only instruction insides regions are optimized for region-level optimizations. We do not include single frame that cannot connect with other frames for region level optimizations.

Figure 10 and Figure 11 show the performance gain from the frame and region level binary optimization. The baseline performance is for the run without binary optimizations. The left bar shows the speedup from binary optimization without optimization on memory operations (no mem opt). We get about 6.7% and 8.1% performance improvement for frame-level optimization and region-level optimization respectively. The middle bar shows the speedup with optimizations on memory operations through atomicity support (opt with atomicity). In this case, the performance includes the overhead of draining the store-buffer at commit. Overall, with the additional memory optimizations, we get 4% and 15.9% performance improvement over the baseline from frame-level optimization and region-level optimization respectively. This represents a 2% loss from non-memory optimizations in frame-level optimization due to the overheads associated with commit, although we get 7.8% grain over non-memory operations in region-level optimization. The right bar shows the speedup of binary optimizations with memory operations through TSO-atomicity support (opt with TSO-atomicity), which does not suffer from the overhead of store-buffer draining. Overall, we get 12% and 20.4% performance improvement over the baseline through dynamic binary optimizations, a 5.3% and 12.3% gain over non-memory optimizations, about 8% and 4.5% better than the performance with atomicity support for frame-level optimization and region-level optimization respectively.

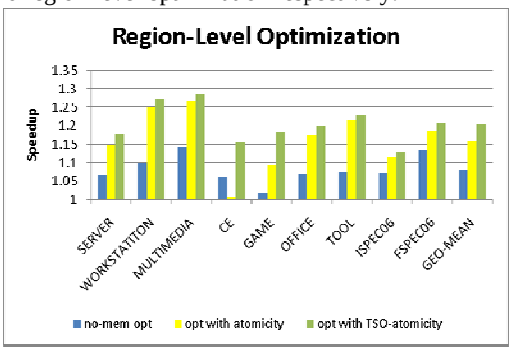

Figure 11: Performance on Region-Level Optimization

5. Related Works

Memory model has been actively studied for many years. Adve and Gharachorloo [1] give an introduction to the foundations of memory consistency and Collier [10] provides a good reference on the subject for computer architecture. Sequential consistency [15] is well-studied although it is less efficient to implement in hardware. SPARC [26] architectures implement a Total Store Ordering (TSO) memory model and Owens et al. [19] proposed a similar Total Store Ordering for X86 architectures (or X86-TSO). TSO and X86-TSO differ from each other only in certain architecture specific features (e.g. memory fence instructions, instructions with LOCK prefix, non-cacheable I/O memory instructions, etc). So in this paper, we treat TSO and X86-TSO interchangeably, and focus on the more efficient and commercially widely used X86-TSO memory model.

There are many research papers and systems on dynamic binary translation (DBT), e.g. Dynamo [3], IA32-EL [4], Dynamo-RIO [8], Transmeta [14], Pin [16], Replay [20], Parrot [23], HDTrans [24] and StarDBT [25]. All these works show performance improvement through dynamic binary optimization with

little touch on the correctness and effectiveness related to memory model issues.

Transactional memory techniques [11] [12] leverage atomicity support to avoid locks in parallel programming. Static and dynamic optimization techniques [2] [14] [18] leverage atomicity support for aggressive optimization. Architectural techniques [5] [9] [27] leverage atomicity support to relax memory ordering constraint for performance improvement. All these previous works follow the same atomicity constraint for region commit and thus suffer from region commit overhead, although some of them use additional HW buffer to alleviate region commit overhead [5]. Our TSO-atomicity region, on the other hand, relaxes the atomicity constraint without any additional HW buffer to achieve efficient region commits.

Techniques such as proposed in [5] [27] support TSO memory model in chunk-based atomic execution. They take advantage TSO model to use more efficient criteria to initialize the chunk than those used in SC model. Once a chunk is formed, however, they always commit the whole chunk atomically without further taking advantage of the TSO model to reduce chunk commit overhead. TSO-atomicity can help these techniques to reduce the chunk commit overhead.

6. Conclusion and Future Works

In this paper, we show that the implementation for atomicity has an inherent inefficiency in TSO model and propose the TSO-atomicity, which relaxes the consistency requirement in atomicity for more efficient implementation in TSO processor, but still enforces TSO for optimizations. We show in our experiments that TSO-atomicity supports can improve the performance by 8% and 4.5% over atomicity support for frame-level and region-level dynamic binary optimization respectively.

Our work can be expanded in several directions. First, weaker consistency usually leads to more efficient implementation in hardware. In this paper, we studied TSO-atomicity, which has a weaker consistency and more efficient implementation than atomicity. It is an interesting topic to research whether or not TSO-atomicity is the weakest consistency model to enforce TSO for optimization. Also, by giving up some optimization opportunities, we wonder if there is some weaker consistency with more efficient implementation than TSO-atomicity to achieve better overall performance. At last, for processors implementing other relaxed memory models, it is an open question whether there are weaker but more efficient consistency model than TSO-atomicity to support optimizations.

References

[1] S. V. Adve and K. Gharachorloo, “Shared Memory Consistency Model: A Tutorial”, IEEE Computer, pages 66-76, Dec. 1996.

[2] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J. Lee, X. Fang, S. Midkiff, S and D. Wong, “BulkCompiler: high-performance sequential consistency through cooperative compiler and hardware support”, MICRO 2009.

[3] V. Bala, E. Duesterwald and S. Banerjia, “Dynamo: A transparent runtime optimization system”, PLDI 2000.

[4] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skalesky, Y. Wang and Y. Zemach, “IA-32 Execution Layer: A Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium®-based Systems”, MICRO 2003.

[5] C. Blundell, M. M. Martin and T. F. Wenisch, “InvisiFence: performance-transparent memory ordering in conventional multiprocessors”. ISCA 2009.

[6] H. Boehm and S. V. Adve, “Foundations of the C++ concurrency memory model”, PLDI 2008.

[7] E. Borin, Y. Wu, C. Wang, W. Liu, M. Breternitz, S. Hu, E. Natanzon, S. Rotem and R. Rosner, “TAO: two-level atomicity for dynamic binary optimizations”, CGO 2010.

[8] D. Bruening, T. Garnett and S. Amarasinghe, “An infrastructure for adaptive Dynamic Optimization”, CGO 2003.

[9] L. Ceze, J. Tuck, P. Montesinos and J. Torrellas, “BulkSC: bulk enforcement of sequential consistency”, ISCA 2007.

[10] W. W. Collier, “Reasoning about parallel architectures”, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[11] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis and K. Olukotun, “Transactional Memory Coherence and Consistency (atomicity)”, ISCA 2004.

[12] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural support for lock-free data structures”, In Proceedings of *the 20th annual International Symposium on Computer Architecture* (ISCA) 1993.

[13] S. Hu, I. Kim, M. H. Lipasti, J. Smith, “An approach for implementing efficient superscalar CISC processors”, ISCA 2006.

[14] K. Krewell, “Transmeta Gets More Efficeon”, *Microprocessor report*. v.17, October, 2003.

[15] L. Lamport, “How to Make a Multiprocessor Compute That Correctly Executes Multiprocess Programs”, *IEEE Transactions on Computers*, 1979.

[16] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”, PLDI 2005.

[17] J. Manson, W. Pugh , S. V. Adve, “The Java memory model”, In Proceedings of *the 32nd atomicityM SIGPLAN-SIGatomicityT symposium on Principles of Programming Languages* (POPL), 2005.

[18] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan and C. Zilles, “Hardware atomicity for reliable software speculation”, ISCA 2007.

[19] S. Owens, S. Sarkar and P. Sewell, “A Better X86 Memory Model: X86-TSO”, *Theorem Proving in Higher Order Logics, 22nd International Conference* (TPHOLs), 2009.

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

[21] S. Patel, T. Tung, S. Bose and M. Crum, “Increasing the size of atomic instruction blocks using control flow assertions”, MICRO 2000.

[22] R. Rajwar, J. R. Goodman, “Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution”, MICRO 2001.

[23] R. Rosner, Y. Almog, Y, M. Moffie, N. Schwartz and A. Mendelson, “Power Awareness through Selective Dynamically Optimized Frames”, ISCA 2004.

[24] S. Sridhar, J. S. Shapiro, E. Northup and P. Bungale, “HDTrans: An Open Source, Low-Level Dynamic Instrumentation System”, VEE 2006.

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

[26] D. L. Weaver and T. Germond, editors, “The SPARC architecture Manual (Version 9)”, Prentice-Hall, 1994.

[27] T. F. Wenisch, A. Ailamaki, B. Falsafi and A. Moshovos. “Mechanisms for store-wait-free multiprocessors”, ISCA 2007.

[28] “Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A”, Order Number: 253668032US.

[29] J.E. Smith, S. Sastry, T. Heil, T.M. Bezenek, “Achieving high performance via co-designed virtual machines”, Innovative Architecture for Future Generation High-Performance Processors and Systems, 1998, pp 77 – 84

[30] Y. Wu, S. Hu, E. Borin, C. Wang, “A HW/SW Co-designed Heterogeneous Virtual Machine for Energy-Efficient General Purpose Computing,” CGO 2011