TSO_ATOMICITY: Efficient Hardware Primitive for TSO-Preserving Region Optimizations

Cheng Wang Youfeng Wu

Programming Systems Lab
Microprocessor and Programming Research / Intel Labs
{cheng.c.wang, youfeng.wu}@intel.com

Abstract

Program optimizations based on data dependences may not preserve the memory consistency in the programs. Previous works leverage a hardware *ATOMICITY* primitive to restrict the thread interleaving for preserving *sequential consistency* in region optimizations. However, ATOMICITY primitive is over restrictive on the thread interleaving for optimizing real-world applications developed with the popular *Total-Store-Ordering* (TSO) memory consistency, which is weaker than sequential consistency. In this paper, we present a novel hardware *TSO_ATOMICITY* primitive, which has less restriction on the thread interleaving than ATOMICITY primitive to permit more efficient program execution than ATOMICITY primitive, but can still preserve TSO memory consistency in all region optimizations. Furthermore, TSO_ATOMICITY primitive requires similar architecture support as ATOMICITY primitive and can be implemented with only slight change to the existing ATOMICITY primitive implementation. Our experimental results show that in a start-of-art dynamic binary optimization system on a large set of workloads, ATOMICITY primitive can only improve the performance by 4% on average. TSO_ATOMICITY primitive can reduce the overhead associated with ATOMICITY primitive and improve the performance by 12% on average.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Processors – code generation, optimization.

General Terms: Performance

Keywords: Atomicity; Dynamic Optimization; Memory Consistency

1. Introduction

Program optimizations must preserve the program memory consistency. Traditional optimizations based on data dependences may not preserve the memory consistency in the programs. For example, in the program example shown in Figure 1 (a), *Sequential Consistency* (SC) [22] requires memory operations in a thread to execute in the order specified by its program order. So if load m_1 executes after m_4 (i.e. the result *r1 == 1*), as shown by the arrow, then m_2 must execute after m_3 (i.e. the result *r2 == 1*). Therefore, the result *r1 == 1* and *r2 == 0* should never be produced by the program. Programs under SC can use that causality property for synchronization between threads (e.g. in spinlocks) such that the condition *r1 == 1* will guarantee that load m_2 in Thread 1 reads the store data of m_3 in Thread 2. However, since there is no data dependence between m_1 and m_2, optimizations on Thread 1 based on data dependences may reorder them to Thread 1', (e.g. vectorization optimizations may need that reordering to merge load m_2 with a previous load for vector load). As the result, the program in Figure 1 (b) can produce the result, *r1 == 1* and *r2 == 0*, with the execution order shown by the arrows in the figure, which may result in incorrect synchronizations (i.e. the optimization in Figure 1 (b) is a *non-SC-preserving optimization*).

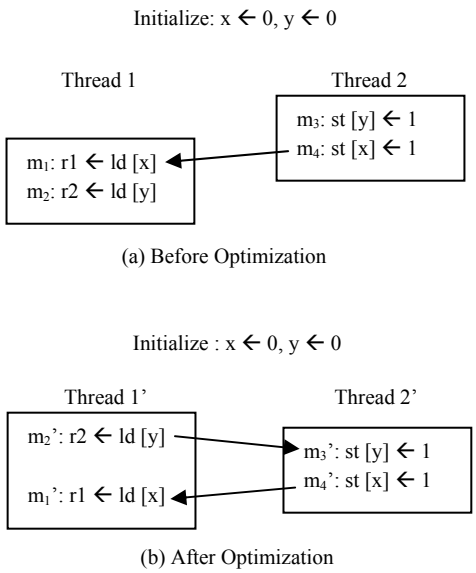

Figure 1: An Non-SC-Preserving and Non-TSO-Preserving Optimization Example

Previous works [2][14][27][30] show that we can leverage the hardware *ATOMICITY primitive* [3][19][21][26][31] to restrict the thread interleaving for preserving SC in region optimizations. For example, we can group m_1 and m_2 in Thread 1 of Figure 1 (a) into a *region* with ATOMICITY primitive such that the hardware will always execute the whole region atomically. Then after optimizing thread 1 to thread 1', the thread interleaving shown in

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

Figure 1 (b) cannot happen (i.e. the execution of thread 2' cannot interleave between the atomic execution of m_2' and m_1'), and thus SC is preserved in optimizations within the region (i.e. the optimization in Figure 1 (b) becomes an *SC-preserving region optimization* with ATOMICITY primitive).

SC is inefficient for program execution, and is not adopted in practical processor designs. Popular processors like X86 and SPARC deployed a relaxed *Total-Store-Ordering* (TSO) [28][37] memory consistency, which can achieve significantly more efficient program execution than SC. Previous works [13][36] show that majority optimizations, including the one shown in Figure 1, cannot preserve TSO. By restricting the optimizations to be TSO-preserving, 60% of the memory optimization benefit will be lost on average [36]. So in this paper, we study the efficient hardware primitive for preserving the popularly deployed TSO in region optimizations.

We can leverage the existing hardware ATOMICITY primitive to restrict the thread interleaving for preserving TSO in region optimizations. However, TSO has weaker consistency than SC such that ATOMICITY primitive for preserving SC is over-restrictive than necessary for preserving TSO, which leads to less efficient program execution. For example, TSO allows later loads to pass earlier stores in the execution, which is not allowed by SC. So the optimization of reordering m_1 and m_2 in Figure 2 preserves TSO, but not SC. Executing the optimized code region in Thread 1' of Figure 2 (b) atomically (under ATOMICITY primitive) would unnecessarily prevent the thread interleaving shown by the arrows in the figure.

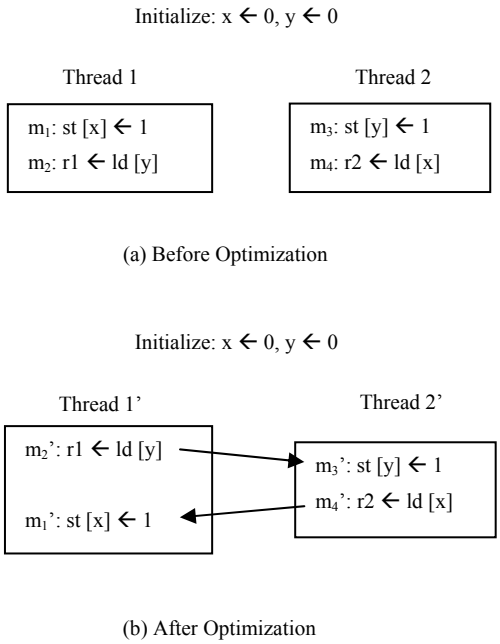

Figure 2: A TSO-Preserving but Non-SC-Preserving Optimization Example

In this paper, we present a novel hardware *TSO_ATOMICITY* primitive, which has weaker restriction on the thread interleaving than ATOMICITY primitive to permit more efficient program execution than ATOMICITY primitive, but can still preserve TSO memory consistency in all region optimizations. For clarity, we refer to the existing ATOMICITY primitive for preserving SC *SC_ATOMICITY* primitive in the rest of the paper. We can use Figure 3 to illustrate the relationship between the memory consistency and the restriction on the thread interleaving for preserving the memory consistency in region optimizations. The stronger memory consistency requires stronger restriction on the thread interleaving for preserving the memory consistency in region optimizations. So the dark area in Figure 3 covers the points that can preserve the memory consistency in region optimizations. For SC, SC_ATOMICITY (short name SC_A) can preserve the memory consistency in region optimizations (i.e. the intersection of SC and SC_A, denoted SC+SC_A, falls in the dark area). For TSO, a weaker restriction on the thread interleaving, TSO_ATOMICITY (short name TSO_A), is enough to preserve the memory consistency in region optimizations (i.e. the intersection of TSO and TSO_A, denoted TSO+TSO_A, falls in the dark area). SC_A can also preserve TSO in region optimizations (i.e. the intersection of TSO and SC_A, denoted TSO+SC_A, falls in the dark area), but is over-restrictive (thus less efficient). TSO_A is too weak to preserve SC in region optimizations (i.e. the intersection of SC and TSO_A, denoted SC+TSO_A, stays out of the dark area).

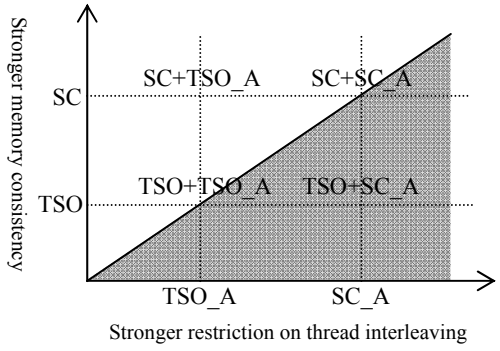

Figure 3: Relationship between Memory Consistency and the Restriction on the Thread Interleaving for Preserving the Memory Consistency in Region Optimizations

We show that TSO_ATOMICITY primitive requires similar architecture support as SC_ATOMICITY primitive and can be implemented with only slight change to the existing SC_ATOMICITY primitive implementation. Our experimental results show that in a state-of-art dynamic binary optimization system with a large set of industrial strength workloads, SC_ATOMICITY primitive can only improve the performance by 4%. TSO_ATOMICITY primitive can reduce the overhead associated with SC_ATOMICITY primitive and improve the performance by 12%.

2. Backgrounds on Memory Consistency Models

2.1 Sequential Consistency

Lamport [22] defines *Sequential Consistency* (SC) as follows. "A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program". In this paper, we formally model SC and define:

[Definition 1] A *program* is a set of *memory operations* (i.e. *stores* and *loads*) with a partial *program order*, which partitions all the memory operations into *threads* such that the program

order is a total order for memory operations within a thread and there is no program order for memory operations across threads.

We use *s* to denote a store, *l* to denote a load and *m* to denote a load or a store. We use *loc(s)* and *loc(l)* to denote the memory location of a store *s* and a load *l* respectively. We use *L* to denote the set of loads in a program and *S* to denote the set of stores in a program. We use $m_1 <_P m_2$ to denote that m_1 precedes m_2 in the program order $<_P$. As an example, Figure 1 (a) shows a program with thread 1: $\{ m_1, m_2 \}$ and thread 2: $\{ m_3, m_4 \}$ such that $m_1 <_P m_2$ and $m_3 <_P m_4$. We model the execution order of memory operations by a memory order and define:

[Definition 2] A *memory order* is a partial order of the memory operations.

We use $m_1 <_M m_2$ to denote that m_1 precedes m_2 in the memory order $<_M$. So formally we get:

[SC-Memory-Order] A memory order under SC is constrained to be a *total order* satisfying: $m_1 <_P m_2 \Rightarrow m_1 <_M m_2$.

As examples, for the program shown in Figure 1 (a), $m_3 <_M m_4 <_M m_1 <_M m_2$ is a memory order under SC, but $m_2 <_M m_3 <_M m_4 <_M m_1$ is not a memory order under SC, as it violates $m_1 <_P m_2 \Rightarrow m_1 <_M m_2$.

We model the program execution result by a program execution witness and define:

[Definition 3] A *program execution witness* (or in short a *witness*) is a mapping $W: L \rightarrow S \cup \{\perp\}$ such that $W(l) = s$ means that load *l* reads the data of store *s* and $W(l) = \perp$ means that load *l* reads the initial memory data at the entry of the program execution.

In order for the program execution witness to also model the final memory data at the exit of the program execution, we assume that there are loads at the exit of the program execution that read all the memory data. So the final memory data at the exit of the program execution will be witnessed.

A program may have different memory orders in different runs. Different memory orders may get different program execution witnesses. Under SC, each load will witness the most recent store in the memory order. So the program execution witness under SC can be derived from the memory order with the following formula:

[SC-Witness-Formula] $W(l) = latest_{<_M} \{ s \mid (loc(s) = loc(l)) \wedge (s <_M l) \}$, where $latest_{<_M}$ gets the latest store in the memory order $<_M$ on a set of stores with $latest_{<_M} \varnothing = \perp$.

As an example in Figure 1 (b), we can derive the witness, $W(m_2') = \perp$ (i.e. *r2 == 0*) and $W(m_1') = m_4'$ (i.e. *r1 == 1*), from the memory order $m_2' <_M m_3' <_M m_4' <_M m_1'$.

2.2 Total-Store-Ordering

Although SC makes it easy for verifying program correctness with SC-Memory-Order and SC-Witness-Formula, it is inefficient for the program execution, as memory operations from the same thread cannot be reordered in a memory order. Specifically, an earlier store that misses the cache may unnecessarily stall a later load execution, even if there is no data dependence between them. Figure 4 (a) shows an example execution trace under SC. Store m_3 misses the cache, which cannot write the data to the cache to become globally visible through the cache coherence protocol until the cache miss is resolved. All the later memory operations are stalled until the cache miss is resolved, which can take significant amount of time in modern processor executions. It is possible to reduce the pipeline stalls due to the cache miss through hardware speculation techniques [17][32]. But that increases hardware cost, complexity and power consumption.

Popular processors like X86 and SPARC deployed the relaxed *Total-Store-Ordering* (TSO) memory consistency, which is more efficient for the program execution. Figure 4 (b) shows an example execution trace under TSO. Store m_3 can write the store data into an internal *store-queue* waiting for the store miss to be resolved and then write to the cache. Later store m_4 can also append the store data into the store-queue and write to the cache after store m_3. However, later loads m_5 and m_6 can execute without waiting for the cache miss of store m_3 to be resolved. As the result, the execution shown in Figure 4 (b) will have the memory order, $m_1 <_P m_2 <_P m_5 <_P m_6 <_P m_3 <_P m_4$, in which later loads m_5 and m_6 execute before earlier stores m_3 and m_4 write to cache. Note that load m_5 reads the same memory location as store m_3. However, load m_5 can still execute by getting the forwarded data of store m_3 from the store-queue without waiting for the cache miss of store m_3 to be resolved. Compared to the program execution under SC, the program execution under TSO clearly is much more efficient (faster).

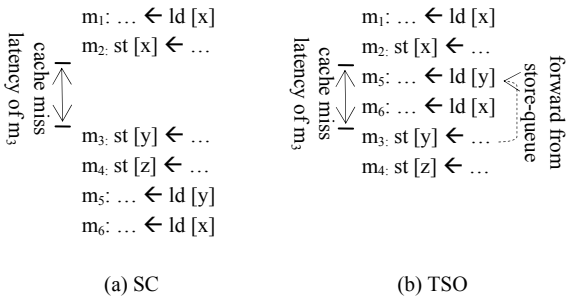

Figure 4: Example Execution Trace in SC and TSO

Except for the reordering between earlier stores and later loads, no other reordering is allowed by TSO. So formally we get:

[TSO-Memory-Order] A memory order under TSO is constrained to be a *total order on all stores* in the program satisfying:

(1) $l_1 <_P l_2 \Rightarrow l_1 <_M l_2$
(2) $s_1 <_P s_2 \Rightarrow s_1 <_M s_2$
(3) $l <_P s \Rightarrow l <_M s$

Due to the store forwarding, a load under TSO will witness the most recent store in the memory order *or in the program order* (i.e. the forwarded data). So the program execution witness under TSO can be derived from the memory order with the following formula:

[TSO-Witness-Formula] $W(l) = latest_{<_M} \{ s \mid (loc(s) = loc(l)) \wedge ((s <_M l) \vee (s <_P l)) \}$

As an example, load m_5 in Figure 4 (b) will witness store m_3, which precedes load m_5 in the program order, even though store m_3 does not precede load m_5 in the memory order.

We should note that under SC-Memory-Order, SC-Witness-Formula is mathematically equivalent to TSO-Witness-Formula, as the stores that precede a load in the program order must precede the load in the memory order. So in the rest paper, we simply treat SC and TSO as using the same TSO-Witness-Formula to derive the witness.

2.3 SC_ATOMICITY

There are many existing works [3][19][21][26][31] on hardware transactional memory, which provides an *ATOMICITY* primitive (called *SC_ATOMICITY* primitive in this paper) for a region execution. Formally, we define:

[Definition 4] A *region R* is a set of memory operations in a thread consecutive in the program order. Formally: $m_1 \subseteq R$, $m_2 \subseteq R$, $m_1 <_P m <_P m_2 \Rightarrow m \in R$.

In the figures used in the paper, we simply use a rectangle to represent a region. As an example, Thread 1 in Figure 1 (b) contains a region $\{ m_2', m_1' \}$.

When SC_ATOMICITY primitive is applied on a region, e.g. enclosing the region with a *begin_atomic* marker and an *end_atomic* marker, the hardware will always execute all memory operations in the region atomically. So formally, SC_ATOMICITY primitive enforces the following memory order constraint on a region:

[SC_ATOMICITY-Constraint] $m_1 \subseteq R$, $m_2 \subseteq R$, $m_1 <_M m <_M m_2 \Rightarrow m \in R$

SC_ATOMICITY-Constraint restricts that no memory operation outside a region *R* (including both memory operations in other threads, and memory operations in the same thread but not in region *R*) can interleave in the memory order between any two memory operations in region *R*. Depending on the memory consistency in the program, SC or TSO, the program with SC_ATOMICITY primitive (short name SC_A) on regions can have two corresponding memory consistency models: SC+SC_A and TSO+SC_A. The memory order under SC+SC_A will satisfy SC-Memory-Order in addition to SC_ATOMICITY-Constraint on regions. Similarly, memory order under TSO+SC_A will satisfy TSO-Memory-Order in addition to SC_ATOMICITY-Constraint on regions. As an example, for the program shown in Figure 1 (b), $m_2' <_M m_3' <_M m_4' <_M m_1'$ is a memory order under SC and TSO, but not a memory order under SC+SC_A or TSO+SC_A. As the result, $W(m_2') = \perp$ (i.e. *r2 == 0*) and $W(m_1') = m_4'$ (i.e. *r1 == 1*) is a witness under SC and TSO, but not a witness under SC+SC_A or TSO+SC_A.

For efficiency, existing hardware transactional memory techniques [3][19][21][26][31] implement SC_ATOMICITY primitive optimistically by speculatively executing the region with atomicity violation detection via the snooping mechanism in the cache coherence protocol. For atomicity violation detection, cache lines are extended with speculative read bits and speculative write bits (i.e. a speculative cache) to mark the loads and stores executed in a region respectively. From TSO-Witness-Formula, only the memory order between *conflicted* memory operations (i.e. two memory operations that access the same memory location and at least one of them is a store) will affect the program execution witness. If during the region execution, there is no memory access from other concurrent threads conflicted with the loads and stores in the region as detected by the snooping on read bits and write bits, the region will be witnessed to execute atomically and can be committed at the end of the region execution by atomically clearing all the read bits and write bits in the speculative cache. Otherwise, the execution is rolled back to a checkpoint at the entry of the region and re-executed.

Due to SC_ATOMICITY-Constraint, TSO with SC_ATOMICITY primitive (i.e. TSO+SC_A) suffers certain inefficiency as SC as it restricts any reordering across the region boundary. For example in Figure 5, store m_3 in region *R1* misses the cache and region *R1* cannot commit until the cache miss is resolved. Then the execution of memory operations after region *R1* has to be stalled until region *R1* commits, which has the same inefficiency as SC in Figure 4 (a).

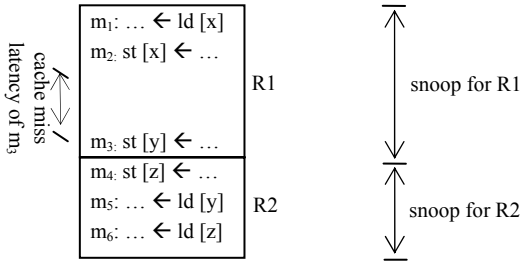

Figure 5: TSO Execution with SC_ATOMICITY Primitive

Existing hardware speculation techniques [17] may alleviate the performance impact of TSO+SC_A. For example, out-of-order processors may execute loads after region *R1* speculatively within an instruction scheduling window without waiting for region *R1* to be committed (i.e. in-window load speculation). However, this increases the power consumption and is not adopted for power-efficient processors (such as Atom [39]). Moreover, even with in-window load speculation, the retirement of loads after *R1* still needs to be stalled until region *R1* commits. As we show in our experiments later on a dynamic binary optimization system, the stall of retirement due to SC_ATOMICITY primitive can impact 8% performance on average. More aggressive hardware speculation techniques such as multi-version speculative cache [14] may further reduce the stalls on retirement, but that would significantly increase the hardware complexity and power.

3. TSO_ATOMICITY

Instead of improving performance of TSO+SC_A through more aggressive speculations, we relax the constraints of SC_ATOMICITY primitive and develop a novel TSO_ATOMICITY primitive. Formally, TSO_ATOMICITY primitive enforces the following memory order constraints on a region:

[TSO_ATOMICITY-Constraint]

(1) $l_1 \subseteq R$, $l_2 \subseteq R$, $l_1 <_M m <_M l_2 \Rightarrow m \in R$
(2) $s_1 \subseteq R$, $s_2 \subseteq R$, $s_1 <_M m <_M s_2 \Rightarrow m \in R$
(3) $s \subseteq R$, $l \subseteq R$, $s <_M m <_M l \Rightarrow m \in R$

So unlike SC_ATOMICITY which enforces atomic memory order among all memory operations in a region, TSO_ATOMICITY *does not constraint the atomic memory order between earlier loads and later stores in a region*. Depending on the memory consistency in the program, SC or TSO, the program with TSO_ATOMICITY primitive (short name TSO_A) on regions can have two corresponding memory consistency models, SC+TSO_A and TSO+TSO_A, which satisfy SC-Memory-Order and TSO-Memory-Order respectively in addition to TSO_ATOMICITY-Constraint on regions. As an example, for the program shown in Figure 2 (b), $m_2' <_M m_3' <_M m_4' <_M m_1'$ is not a memory order under SC+SC_A or TSO+SC_A, but is a memory order under SC+TSO_A and TSO+TSO_A. Consequently, $W(m_2') = \perp$ (i.e. *r1 == 0*) and $W(m_4') = \perp$ (i.e. *r2*

== *0*) is not a witness under SC+SC_A or TSO+SC_A, but is a witness under SC+TSO_A and TSO+TSO_A.

TSO+TSO_A can be implemented with only slight change to the existing implementation for TSO+SC_A and can achieve the efficient overlapped region execution as shown in Figure 6 (a). Store m_3 and m_4 in region *R1* write the store data into the store-queue waiting for the cache miss of store m_3 to be resolved, just like regular TSO implementation. TSO_ATOMICITY primitive allows loads after region *R1* to execute without waiting for the cache miss of store m_3 to be resolved. TSO_ATOMICITY primitive also separates the snooping on read bits from that on write bits. Figure 6 (b) shows the snooping periods for TSO_ATOMICITY primitive. Before the execution of instructions after *R1*, we stop the snooping on read bits of region *R1* by atomically clearing all the read bits in the speculative cache such that they can be used to mark loads executed in region *R2* for snooping. However, we still keep the snooping on write bits of *R1* until the cache miss for store m_3 is resolved. This will not affect the overlapped execution of *R2* because stores in *R2* will be appended into the store-queue waiting for the cache miss of store m_3 to be resolved (see TSO implementation in section 2.2), and stores in store-queue will not need to set write bits until the store data is written from the store-queue to the cache, which becomes globally visible. We also keep the register checkpoint for region *R1* until the end of the snooping on write bits of *R1* in case that the snooping may detect conflicts and roll back region *R1*. After all the store data of region *R1* are written from the store-queue to the speculative cache, we commit region *R1* by atomically clearing all the write bits in the speculative cache. Since read bits are cleared earlier before the region commits, TSO_ATOMICITY primitive not only reduces stalls across region boundary to achieve the same efficiency as the regular TSO execution in Figure 4 (b), but also reduces the potential rollback due to the conflicts on read bits (i.e. the case that the thread interleaving is allowed by TSO_ATOMICITY primitive but not by SC_ATOMICITY primitive as shown in Figure 2 (b)).

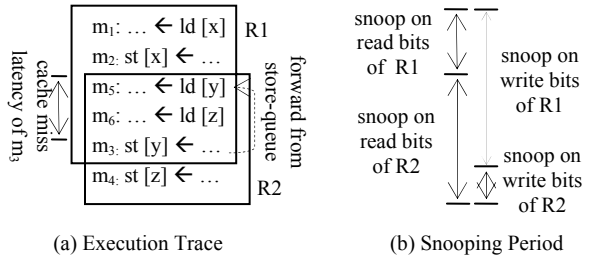

Figure 6: TSO Execution with TSO_ATOMICITY Primitive

Since TSO_ATOMICITY primitive atomically clear all the read bits after all the loads in the region are executed and atomically clear all the write bits after all the stores in the region are written to the cache and become globally visible, all the loads in the region will be witnessed to execute atomically (i.e. TSO_ATOMICITY-Constraint (1)) and all the stores in the region will witnessed to execute atomically (i.e. TSO_ATOMICITY-Constraint (2)). Since TSO_ATOMICITY primitive always clear write bits later than read bits, TSO_ATOMICITY primitive will only constraint atomic memory order between earlier stores and later loads in a region (i.e. TSO_ATOMICITY-Constraint (3)), but not between earlier loads and later stores in a region.

4. Semantic Comparison

We next compare semantics of different memory consistencies by their program execution witnesses, and define:

[Definition 5] Memory consistency *X* is *equal or stronger* than memory consistency *Y*, if for any program, a program execution witness under consistency *X* is a program execution witness under consistency *Y*.

We use $X \subseteq Y$ to denote that memory consistency *X* is equal or stronger than memory consistency *Y*. Since all the consistencies discussed in previous sections use the same TSO-Witness-Formula to derive the witness from the memory order, we can compare memory consistencies by comparing their memory order constraints. So by definitions, we can get the relationship between different memory consistencies shown in Figure 7.

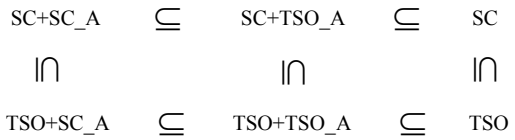

Figure 7: Relationship between Different Memory Consistencies

4.1 Comparison with Database Isolations

SC_ATOMICITY has been used in database systems to achieve different isolations for transactions [6]. Considering a program composed of regions and treat each region as a transaction, SC+SC_A can achieve the serializable isolation. Many database systems also implemented the more efficient snapshot isolation [6], which only guarantees the following two conditions:
Condition 1: All reads made in a transaction will see a consistent snapshot of the database.
Condition 2: The transaction will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot (i.e. no lost update).

Just like that TSO+TSO_A is more efficient for program execution than SC+SC_A, snapshot isolation is more efficient for transaction execution in database system than serializable isolation. So, similar to TSO+TSO_A, the snapshot isolation allows program in Figure 2 (a) to produce the result *r1 == 0* and *r2 == 0*, which is not allowed by serializable isolation. However, TSO+TSO_A can only guarantee condition 1 of snapshot isolation, but not condition 2. So TSO+TSO_A is even weaker (thus potentially more efficient) than snapshot isolation. As an example, for the program shown in Figure 8, TSO+TSO_A allows the program to produce the result $x == 1$ (with memory order $m_1 <_M m_3 <_M m_2 <_M m_4$), i.e. one of the updates to *x* is lost, which is not allowed by snapshot isolation.

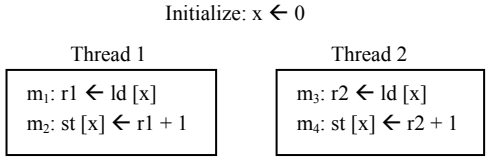

Figure 8: A Program Example for Comparing TSO+TSO_A with Snapshot Isolation

5. Preserve Memory Consistency in Optimization

We now show that, even though TSO_ATOMICITY primitive relaxes SC_ATOMICITY primitive, it can still preserve TSO in all program optimizations. Similar to [36], we model a program optimization as a transformation from the original program to the optimized program such that the optimized program represents the original program for execution, as shown in Figure 9. Each load *l* in the original program is mapped to a load *l'* in the optimized program by OPT_L such that *l'* represents *l* to read data. Each store *s'* in the optimized program is mapped to a store *s* in the original program by OPT_S such that store *s'* represents *s* to write data. We also denote $OPT_S(\perp)=\perp$ so that the optimized program has the same initial data at the entry of the program as the original program. As the results, a program execution witness *W'* of the optimized program will represent a program execution witness *W* of the original program such that $W = OPT_S \circ W' \circ OPT_L$. As an example, by reordering m_1 and m_2 in thread 1 of Figure 1 (a) to thread 1' of Figure 1 (b), we get an optimization such that OPT_L maps m_1 and m_2 to m_1' and m_2' respectively, and OPT_S maps m_3' and m_4' to m_3 and m_4 respectively. Then the execution witness of the optimized program $W'(m_1')= m_4'$ (i.e. *r1 == 1*) and $W'(m_2')= \perp$ (i.e. *r2 == 0*) will represent the execution witness of the original program $W(m_1)= m_4$ (i.e. *r1 == 1*) and $W(m_2)= \perp$ (i.e. *r2 == 0*).

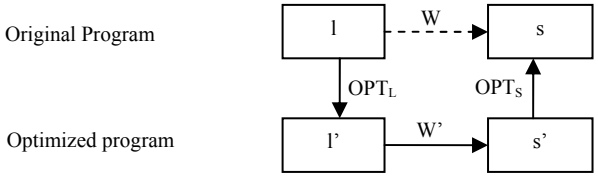

Figure 9: Modeling Optimization

Formally, we define:

[Definition 6] An *optimization* is a transformation from an original program with store set *S* and load set *L* to the optimized program with store set *S'* and load set *L'* through two mappings: OPT_L: *L* → *L'* and OPT_S: *S'* → *S* with $OPT_S(\perp)=\perp$.

OPT_L and OPT_S can model not only memory reordering, but also general optimizations. For example, the redundant load elimination shown in Figure 10 can be modeled as the optimization with $OPT_L(m_1) = OPT_L(m_2) = m_3$.

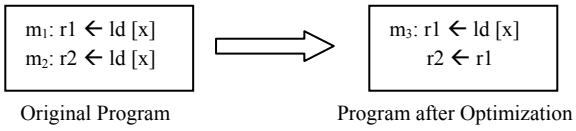

Figure 10: Redundant Load Elimination Example

To preserve the memory consistency, each witness of the optimized program must represent a witness of the original program. So formally, we define:

[Definition 7] An optimization *preserves program execution witness under memory consistency X (*or in short*, preserves memory consistency X*), if for each witness *W'* of the optimized program under consistency *X*, there is a witness *W* of the original program under consistency *X* such that $W = OPT_S \circ W' \circ OPT_L$.

As an example optimization that do not preserve the memory consistency, the memory order of the program execution in Figure 1 (b) will produce the witness $W'(m_1')= m_4'$ (i.e. *r1 == 1*) and $W'(m_2')= \perp$ (i.e. *r2 == 0*) under SC and TSO. However, no memory order of the original program in Figure 1 (a) would produce the witness $W(m_1) = m_4$ (i.e. *r1 == 1*) and $W(m_2) = \perp$ (i.e. *r2 == 0*) under SC or TSO. So the reordering of m_1 and m_2 in thread 1 of Figure 1 (a) to thread 1' of Figure 1 (b) does not preserve either SC or TSO.

Running the same program under different consistencies may have different memory orders and different memory orders may produce different program execution witnesses. However, we can prove:

[Single-Thread-Witness] Given a single-thread program, the program execution witnesses under all the memory consistencies discussed in previous sections are unique and same, which can be derived with formula: $W(l) = latest_{<_P} \{ s \mid (loc(s) = loc(l)) \wedge (s <_P l) \}$, where $latest_{<_P}$ gets the latest store in the program order $<_P$ on a set of stores with $latest_{<_P} \varnothing = \perp$.

Since all the other memory consistencies discussed in previous sections are equal or stronger than TSO, we can prove Single-Thread-Witness by simply verifying that for a single-thread program, TSO-Witness-Formula with TSO-Memory-Order is mathematically equivalent to the formula in Single-Thread-Witness.

So the program execution witness for a single-thread program is independent of the memory consistency. Optimizations on a single-thread program only need to preserve Single-Thread-Witness based on data dependences for preserving the memory consistency. For multi-thread program, however, this may not be true, as shown by the optimization example in Figure 1. But we can prove:

[Optimization-Theorem] Under SC+SC_A, TSO+SC_A or TSO+TSO_A, an optimization within a region preserves the respective memory consistency, if by treating the region as a single-thread program, the optimization preserves Single-Thread-Witness.

Optimization-Theorem allows us to apply all the traditional data-dependence based optimizations within the region boundary while still preserve the memory consistency under SC+SC_A, TSO+SC_A and TSO+TSO_A. So to optimize a region and preserve SC, we can mark the region with SC_ATOMICITY primitive. The resulting optimized program will preserve SC+SC_A (i.e. preserve SC because SC+SC_A is stronger than SC). Similarly, both SC_ATOMICITY primitive and TSO_ATOMICITY primitive can be applied on a region to preserve TSO in the region optimizations. However, TSO_ATOMICITY primitive cannot be used on a region to preserve SC in the region optimizations, as Optimization-Theorem does not cover SC+TSO_A. Figure 2 shows an example such that an optimization on a region with TSO_ATOMICITY primitive does not preserve SC, as the result *r1 == 0, r2 == 0* is allowed by the optimized program under SC+TSO_A, but not allowed by the original program under SC.

To prove Optimization-Theorem under SC+SC_A, we only need to show that given any memory order $<'_M$ of the optimized program under SC+SC_A with witness *W'*, we can find a memory order $<_M$ of the original program under SC+SC_A with witness *W* such that $W = OPT_S \circ W' \circ OPT_L$. To achieve that, we construct the memory order $<_M$ from $<'_M$ such that within threads it is consistent with the program order of the original program, and across threads it is consistent with $<'_M$ under the mapping of

OPT_L and OPT_S. As an example, take a look at the optimization from the program in Figure 2 (a) to the program in Figure 11. Assume the execution of the optimized program gives the memory order $m_3' <'_M m_4' <'_M m_2' <'_M m_1'$ with witness $W'(m_2') = m_3'$ (i.e. r1 == 1) and $W'(m_4') = \perp$ (i.e. r2 == 0), as shown in Figure 11. For the original program, we first construct the memory order $m_1 <_M m_2$ and $m_3 <_M m_4$ to be consistent with the program order $m_1 <_P m_2$ and $m_3 <_P m_4$ within threads. Across threads, we construct the memory order $m_3 <_M m_2$, $m_4 <_M m_2$, $m_3 <_M m_1$ and $m_4 <_M m_1$ to be consistent with the memory order $m_3' <'_M m_2'$, $m_4' <'_M m_2'$, $m_3' <'_M m_1'$ and $m_4' <'_M m_1'$ under the mapping: $m_1 = OPT_S(m_1')$, $m_2' = OPT_L(m_2)$, $m_3 = OPT_S(m_3')$ and $m_4' = OPT_L(m_4)$. The resulting memory order $m_3 <_M m_4 <_M m_1 <_M m_2$ is a memory order of the original program under SC+SC_A with the witness $W(m_2) = m_3$ (i.e. r1 == 1) and $W(m_4) = \perp$ (i.e. r2 == 0). We can verify that, for any memory $<'_M$ of the optimized program under SC+SC_A, the memory order $<_M$ constructed in this way must be a memory order of the original program under SC+SC_A due to SC_ATOMICITY-Constraint. Moreover, the witness W produced by the constructed memory order $<_M$ must satisfy $W = OPT_S \circ W' \circ OPT_L$. This is because within the threads, the optimization preserves Single-Thread-Witness and across threads, $<_M$ is consistent with $<'_M$ under the mapping of OPT_L and OPT_S. Note that as discussed in section 2.1, the program execution witness models the final memory data at the exit of the program execution. So by treating the region as a single thread program, the optimization that preserves Single-Thread-Witness will ensure the correct region live-out stores.

Without SC_ATOMICITY-Constraint of $<'_M$, we may not be able to construct the memory order $<_M$. For example, in the optimization from Figure 2 (a) to Figure 2 (b), the memory order $m_2' <'_M m_3' <'_M m_4' <'_M m_1'$ in Figure 2 (b) violates SC_ATOMICITY-Constraint. Given $<'_M$, if we construct $m_1 <_M m_2$ and $m_3 <_M m_4$ to be consistent with the program order within threads, and across threads $m_2 <_M m_3$, $m_2 <_M m_4$, $m_3 <_M m_1$ and $m_4 <_M m_1$ to be consistent with $<'_M$. The resulting $<_M$ will contain a cycle $m_1 <_M m_2 <_M m_3 <_M m_4 <_M m_1$, which is not a valid memory order.

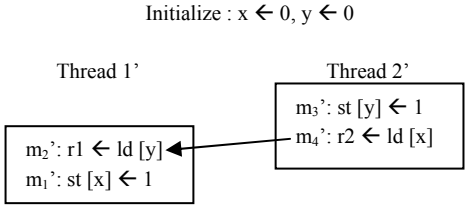

Figure 11: After Optimization

The Optimization-Theorem under TSO+SC_A can be proved in the same way. The Optimization-Theorem under TSO+TSO_A can be proved similarly except that within threads we do not construct memory order $<_M$ *between an earlier store and a later load* to be consistent with the program order, which is not required by TSO-Memory-Order. As an example, for the optimization from Figure 2 (a) to Figure 2 (b), given the memory order $<'_M$ under TSO+TSO_A with witness $W'(m_2') = \perp$ and $W'(m_4') = \perp$, as shown in Figure 2 (b), we will only construct the memory order: $m_2 <_M m_3$, $m_2 <_M m_4$, $m_3 <_M m_1$ and $m_4 <_M m_1$ (note no $m_1 <_M m_2$ or $m_3 <_M m_4$) with witness $W(m_2) = \perp$ and $W(m_4) = \perp$. Then there is no memory order cycle as shown in the previous paragraph, even though TSO_ATOMICITY does not constraint atomic memory order between earlier loads and later stores in a region. We can verify that the memory order $<_M$ constructed in this way must always be a memory order under TSO+TSO_A with the witness W satisfying $W = OPT_S \circ W' \circ OPT_L$.

We should note that in all above discussions, we do not consider branches in the region. By enumerating all paths, it is straightforward to extend the above discussions to regions with branches but without loops. There is an issue to group codes with loops into a region with SC_ATOMICITY primitive or TSO_ATOMCITY primitive, as it may introduce deadlocks due to rollbacks [7], which do not exist in the original program without SC_ATOMICITY primitive or TSO_ATOMCITY primitive. So in loop region optimizations with SC_ATOMICITY primitive or TSO_ATOMICITY primitive, we only speculatively group the code into regions for the optimizations. In case of region rollback, we will discard the optimizations on the region and run the original non-optimized code without SC_ATOMICITY primitive or TSO_ATOMICITY primitive.

6. Experiments

6.1 Hardware Implementation

To evaluate and compare SC_ATOMICITY primitive and TSO_ATOMICITY primitive, we implemented them on a cycle-accurate in-house simulator that faithfully models an out-of-order processor for IA32 microarchitecture. Table 1 shows the microarchitecture parameters for the processor.

Table 1: Simulator Parameters

Components	*Configurations*
Issue Width	4 way
Scheduling Window	54 entries
ROB	168 entries
Load Queue	64 entries
Store Queue	40 entries
L1 I-cache	32K, 4 way, 64 byte line, 1 cycle hit latency
L1 D-cache	32K, 8 way, 64 byte line, 4 cycles hit latency
L2 Unified Cache	256K, 8 way, 64 byte line, 12 cycles hit latency
L3 Uncore Cache	4M, 16 way
Memory Latency	200 Cycles
Core	4

Our out-of-order processor implements TSO. Stores can retire with store data in a store-queue waiting for in-order write-back to the cache (i.e. senior stores). Later instructions can retire with earlier store data in the store-queue. The processor also implements in-window load speculation [17] such that loads can be executed speculatively out-of-order within an instruction schedule window. To maintain the memory order on loads, the speculatively executed loads keep snooping for conflicts in a load queue until loads retire.

To support SC_ATOMICITY primitive or TSO_ATOMICITY primitive, L1 D-cache is extended with speculative read bits and speculative write bits. Loads in a region will set the read bits at load retirement and stores in a region will set write bits at write-back from store-queue to the cache. After all the instructions in a region retire, SC_ATOMICITY primitive cannot commit the region until all the data in the store-queue are drained and written back to cache. That stalls the instructions after the region from retiring, potentially for many cycles if stores miss the last level cache. TSO_ATOMICITY primitive does not stall the instructions after the region from retiring. Instead, it just

clears all the read bits and then later instructions can retire normally. When the last store data of the region is written back to cache, the whole region can commit by clearing all the write bits and releasing the register checkpoint for the region.

The register check-pointing for SC_ATOMICITY primitive and TSO_ATOMICITY primitive is implemented lazily through a Copy-On-Write (COW) mechanism at the instruction retirement using a COW queue with 144 entries. We support overlapped execution of at most four regions in TSO_ATOMICITY primitive.

6.2 Dynamic Binary Optimization

To evaluate the optimization benefit with SC_ATOMICITY primitive and TSO_ATOMICITY primitive, we implemented a dynamic binary optimization system in the simulator that takes a program compiled for the IA32 architecture and generates micro-operation (uop) code. The dynamic optimizer dynamically profiles the execution and constructs frames as regions for optimization. A frame is a superblock with all internal branches converted to asserts [30]. Frame regions are widely used in dynamic binary optimization [29][30][33] due to their low runtime optimization overheads. Table 2 lists the optimizations used in the experiments.

Table 2: List of Optimizations

Non-Memory Optimizations	Memory Optimizations
Register-Forward-Propagation	Redundant-Load-Elimination
Register-Backward-Propagation	Store-Load-Forwarding
Identical-Logic-Operation	Dead-Load-Elimination
Condition-Propagation	Dead-Store-Elimination
Assert-Elimination	Store-Fusion
Redundant-Operation-Elimination	Load-Fusion
Dead-Code-Elimination	
Constant-Folding-Propagation	
ALU-Fusion	

Dynamic binary optimization on IA32 micro-operations must preserve TSO for all the programs as the optimizer has no source-level memory consistency information. The optimizer also has no knowledge on whether the optimized code is running in a single-thread application or a multi-thread application. Due to that, we apply SC_ATOMICITY primitive or TSO_ATOMICITY primitive for all the optimized regions.

The benefit of TSO_ATOMICITY primitive over SC_ATOMICITY primitive comes from two facts. First, TSO_ATOMICITY primitive does not stall the retirement of instructions in later regions waiting for the store miss of instructions in earlier regions to be resolved. Second, TSO_ATOMICITY primitive clears read bits earlier to reduce region rollbacks. The stall of retirement across region boundary has high performance impact on small regions due to their frequent crossing of region boundary. The region rollback has high performance impact on large regions due to their high possibility of conflict and large rollback overhead. Frame regions typically have small region sizes (average ~20 instructions, as compared to the ROB size 168). So we focus on evaluating the performance impact of the stall of retirement across region boundary in this paper. The performance impact of region rollback (or squash) due to thread interleaving is negligible in our experiments (Note that our dynamic binary optimization does not optimize regions across the locked instructions so there is no region rollback due to the synchronization through locked instructions).

The stall of retirement across region boundary impacts both single-thread executions and multi-thread executions. The concurrent multi-thread executions make it unstable to evaluate the performance impact due to the non-determinism in the thread synchronizations. So we collect performance data only on single-thread executions. This methodology is similar to other studies on region optimizations with SC_ATOMICITY primitive [27].

We collect performance data on around 400 snippets from about 200 programs in nine categories, as shown in Table 3. Some applications that are represented by other programs are omitted (e.g. tonto in SPEC2006 FP is omitted since it behaves similar to some other programs). This wide range of application coverage is essential to properly evaluate the proposed mechanisms under the complex interaction among original code, the dynamic optimizing compiler, uop code, and out-of-order processors.

Table 3: Applications Used for Evaluation

Application Categories	# of Applications (and examples)	# of snippet
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

6.3 Performance Evaluation

Figure 12 shows the performance gain from frame optimizations. The baseline performance is for the run without binary optimization. The left bar (non-mem-opt) shows the speedup from binary optimization with only non-memory optimizations listed in the first column of Table 2. Since non-memory optimizations never affect the memory consistency, we do not need SC_ATOMICITY primitive or TSO_ATOMICITY primitive support. Overall, we get about 6.7% performance improvements with the non-memory optimizations. The middle bar (opt with SC_ATOMICITY) shows the speedup with SC_ATOMICITY primitive support to preserve memory consistency for optimizing frames. So all the optimizations listed in Table 2 are applied to the frames. Due to the stall overheads of SC_ATOMICITY primitive across the region boundary, we get only 4% performance improvement over the baseline, even with the additional memory optimizations. The right bar (opt with TSO-ATOMICITY) shows the speedup with TSO-ATOMICITY primitive support. Overall, we get 12% performance improvement over the baseline without dynamic binary optimization, an 8% better than the performance with SC_ATOMICITY primitive support. For certain categories (e.g. CE), the performance difference between SC_ATOMICITY primitive and TSO_ATOMICITY primitive can be as high as 15%. Due to the high stall overhead associated with

SC_ATOMICITY primitives, many prior works do not rely on SC_ATOMICITY to preserve memory consistency during binary optimization. They either work only on single core [20] or restrict certain memory optimizations (e.g. no optimization of stores) [33].

It may be interesting to note that the average performance with SC_ATOMICITY is actually a little lower than that with only non-memory optimizations without SC_ATOMICITY support. The main reasons are 1) SC_ATOMICITY delays the retirements of instructions after the regions, and 2) the regions (frames) is relative small, and the optimization benefit under SC_ATOMICITY cannot fully amortize the overhead at region boundary. We report more detailed data on the impacts of region size and the delay at region boundary in the next subsections.

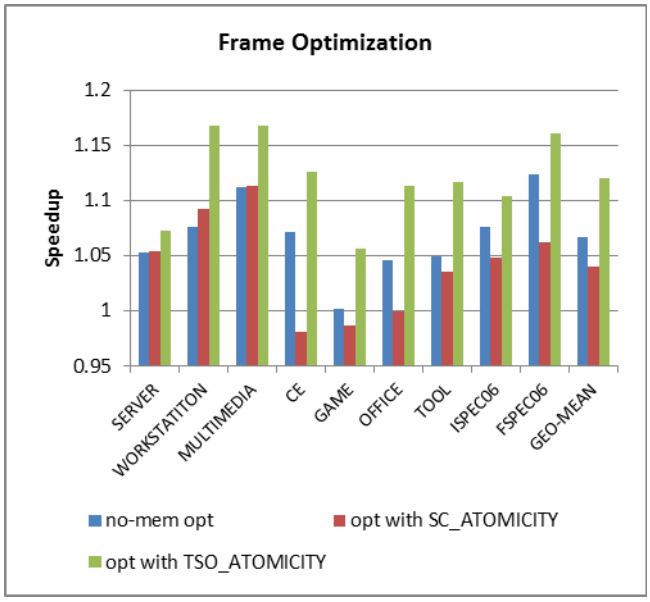

Figure 12: Speedup in Frame Optimization

6.4 Sensitivity to Region Size

Since the performance impact of SC_ATOMICITY primitive and TSO_ATOMICITY primitive is sensitive to the region/frame size, for evaluating the performance impact of larger regions, we extended the dynamic binary optimizer to aggressively build multi-frame regions (including DAG and Loop regions) by connecting paths between frames. Figure 13 shows the average dynamic region sizes of single-frame regions (~20 instructions) and multi-frame regions (~70 instructions). Supporting atomic regions beyond frames has associated HW and SW costs. For example, [11] recommends a two-level atomicity scheme which requires both gated store buffer and speculative cache support, [10] proposes a conditional commit extension to dynamically extend atomic scope, and [1] further elects to make L2 cache, rather than L1 cache, speculative in order to accommodate large atomic regions. So our general multi-frame region really stretches the performance evaluation for large regions than practically deployed.

Figure 14 shows the performance gain from our multi-frame region optimization. Due to the larger region and relatively lower across region overheads, we get more optimization benefit. Overall, we get 15.9% and 20.4% performance improve from SC_ATOMICITY primitive and TSO_ATOMICITY primitive supports, respectively. So even with the large regions, TSO_ATOMICITY primitive still can perform 4.5% better than SC_ATOMICITY primitive.

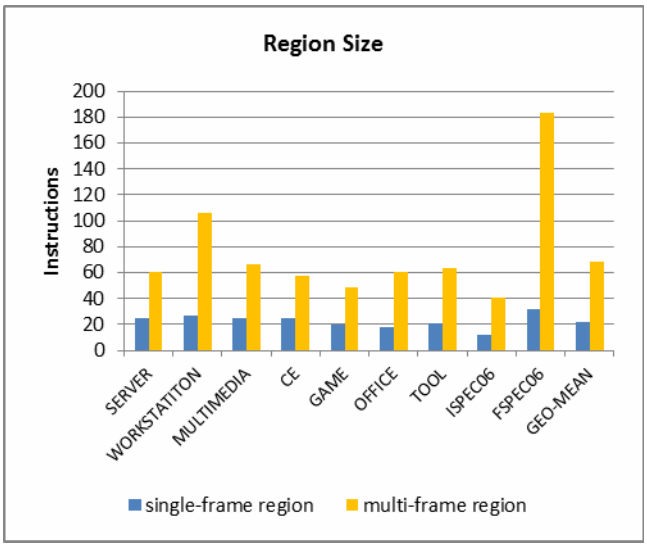

Figure 13: Dynamic Region Instruction Count

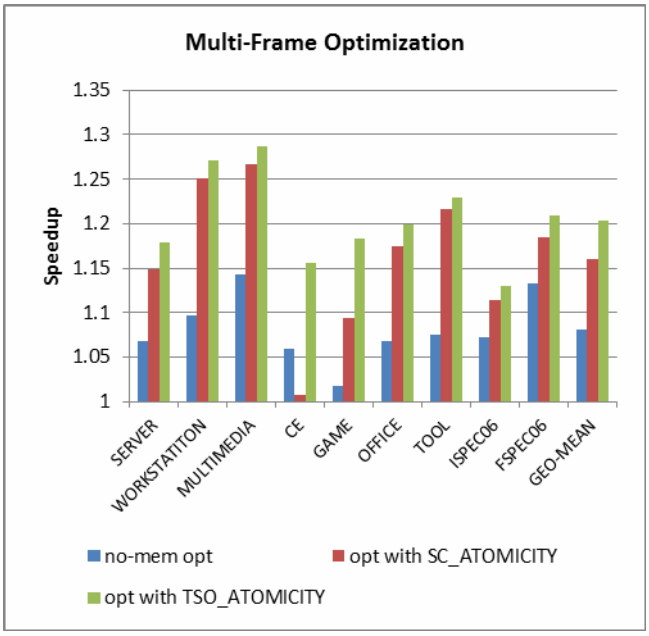

Figure 14: Speedup in Multi-Frame Optimization

Figure 15 shows the average stall cycles per region commit with SC_ATOMICITY primitive for multi-frame regions. Among all the categories, CE and GAME have long stall cycles at region commit, which lead to the big performance differences between TSO_ATOMICITY primitive and SC_ATOMICITY primitive (see Figure 14). FSPEC06 also has long stall cycles. But its larger region size (see Figure 13) amortizes most of the overhead. Note that, for the out-of-order processor, even when the retirement stage is stalled, later instructions can still flow from fetch stage to the reorder buffer and be dispatched for execution. So many stall cycles can be hidden by the out-of-order execution, until the reorder buffer is full. If we were modeling an in-order processor (e.g. an in-order Atom or ARM processor), we would

expect the 7 cycles of average stall at region commit to cause more serious performance loss than that shown in Figure 14.

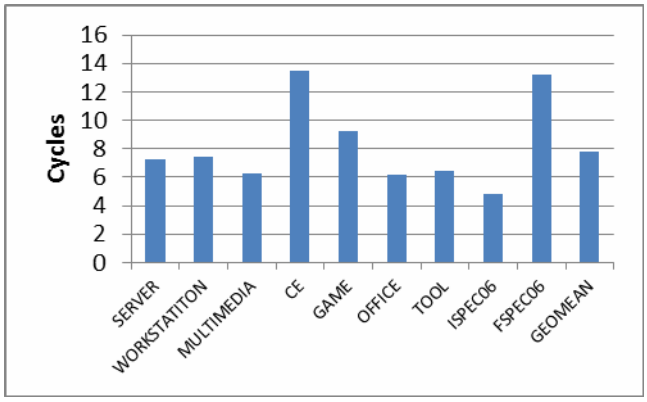

Figure 15: Stall Cycles per Region Commit

7. Related Works

SC_ATOMICITY primitive has been widely studied in transactional memory techniques [3][19][21][26][31] to synchronize threads without locks. Recently, Intel announced a SC_ATOMICITY primitive (called TSX for Transactional Synchronization Extension) for the new Haswell microarchitecture [40]. Recent research work [2] shows that SC_ATOMICITY primitive can be used to preserve sequential consistency in program optimization. Architectural techniques [8][38] also leverage SC_ATOMICITY primitive to reorder memory operations for performance improvement. Our result suggests that TSX won't be able to efficiently support dynamic binary optimizations of frame or superblocks [20], unless TSX is extended to support TSO_ATOMICITY.

Previous works [12][25][36] show that SC and TSO impose considerable memory consistency constraints to the program optimization. Recent work [34] shows that it requires thread-local and shared read-only information in the program in order to achieve efficient SC implementation. Due to that, high-level program language like C [9] and Java [24] typically adopt the relaxed memory consistency models [16][18] to avoid the restrictions on the program optimization. Without high-level language memory consistency information, binary optimization, however, has to obey the processor memory consistency.

There are many research papers and systems on dynamic binary translation (DBT), e.g. Dynamo [4], IA32-EL [5], DynamoRIO [12], Transmeta [20], DAISY [15], Pin [23], Replay[29], Parrot [33], HDTrans [35]. All these works show performance improvement through dynamic binary optimization with little touch on the issues of preserving memory consistency in optimization.

8. Conclusion and Future Works

In this paper, we introduce a novel TSO_ATOMICITY primitive, which can preserve TSO memory consistency in region optimizations efficiently. Our experimental results show that TSO_ATOMICITY primitive can significantly improve the performance over the existing SC_ATOMICITY primitive.

TSO_ATOMICITY primitive introduced in this paper can support not only binary program optimization, but also microarchitecture optimizations. For example, due to the store order constraint, current microarchitecture implementing TSO cannot combine two stores to the same cache line across other stores to reduce the store port pressure. We may leverage TSO_ATOMICITY primitive to combine stores in the store-queue. Furthermore, we believe that additional variances of atomicity primitives can be introduced to efficiently preserve other memory consistencies, e.g. PSO [37], in region optimization.

References

[1] R. Agarwal, J. Torrellas, "FlexBulk: intelligently forming atomic blocks in blocked-execution multiprocessors to minimize squashes", ISCA 2011

[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] C. S. Ananian and K. Asanovic and B. C. Kuszmaul, C. E. Leiserson and S. Lie, "Unbounded Transactional Memory", HPCA 2005.

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

[5] 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.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil, "A Critique of ANSI SQL Isolation Levels", *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data.*

[7] C. Blundell, E. C. Levwis and M. K. Martin, "Deconstructing Transactions: The Subtleties of ATOMICITY", WDDD 2005.

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

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

[10] E. Borin, Y. Wu, M. Breternitz Jr., C. Wang, "LAR-CC: Large atomic regions with conditional commits," CGO 2011.

[11] 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.

[12] D. Bruening, T. Garnett and S. Amarasinghe, "An infrastructure for adaptive Dynamic Optimization", CGO 2003.

[13] S. Burckhardt, M. Musuvathi and V. Singh. "Verifying local transformations on relaxed memory models", CC 2010.

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

[15] K. Ebcioglu , E. R. Altman, "DAISY: dynamic compilation for 100% architectural compatibility", ISCA 1997.

[16] G. Gao, V. Sarkar, "Location Consistency-A New Memory Model and Cache Consistency Protocol", IEEE Trans. Computers, 2000.

[17] K. Gharachorloo, A. Gupta and J. Hennessy, “Two Techniques to Enhance the Performance of Memory Consistency Models”, ICPP 1991.

[18] K. Gharachorloo , D. Lenoski , J. Laudon , P. Gibbons , A. Gupta and J. Hennessy, “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors”, ISCA 1990.

[19] L. Hammond, V. Wong, M. Chen. B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, H. Wijaya; C. Kozyrakis and K. Olukotun, “Transactional Memory Coherence and Consistency”, ISCA 2004.

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

[21] 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.

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

[23] 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.

[24] J. Manson, W. Pugh , S. V. Adve, “The Java memory model”, POPL 2005.

[25] D. Marino, A. Singh, T. Millstein, M. Musuvathi and S. Narayanasamy, “A Case for SC-Preserving Compiler”, PLDI 2011.

[26] K. E. Moore and J. Bobba and M. J. Moravan and M. D. Hill and D. A. Wood, “LogTM: Log-based Transactional Memory”, HPCA 2006.

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

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

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

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

[31] R. Rajwar and M. Herlihy and K. Lai, “Virtualizing Transactional Memory”, ISCA 2005.

[32] P. Ranganathan, V. Pai and S. Adve, “Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models”, SPAA 1997.

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

[34] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, M. Musuvathi, “End-to-End Sequential Consistency”, ISCA 2012.

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

[36] C. Wang, Y. Wu, “Modeling and Performance Evaluation of TSO-Preserving Binary Optimization”, PACT 2011.

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

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

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

[40] “Intel® Architecture Instruction Set Extensions Programming Reference”, February 2012