Efficiently and Completely Verifying Synchronized Consistency Models*

Yi Lv[1], Luming Sun[1], Xiaochun Ye[2], Dongrui Fan[2], and Peng Wu[1]

[1] State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, China
[2] State Key Laboratory of Computer Architecture,
Institute of Computing Technology, Chinese Academy of Sciences, China

Abstract. The physical time order information can help verifying the memory model of a multiprocessor system rather efficiently. But we find that this time order based approach is limited to the sequential consistency model. For most relaxed memory models, an incompatible time order may possibly result in a false negative verdict. In this paper, we extend the original time order based approach to synchronized consistency models, and propose an active frontier approach to rule out such false verdicts based on a reasonably relaxed time order. Our approach can be applied to most known memory models, especially to those with non-atomic write operations, while nevertheless retaining the efficiency of the original time order based approach. We implement our approach in a Memory Order Dynamic Verifier (MODV). A case study with an industrial Godson-T many-core processor demonstrates the effectiveness and efficiency of our approach. Several bugs of the design of this processor are also found by MODV.

1 Introduction

With the increasingly aggressive development of hardware optimization technologies, most multi-core processors support relaxed memory models for the sake of high performance. Synchronized consistency models, such as release consistency [1] and scope consistency [2], were usually deployed in software Distributed Shared Memory (DSM) systems. Recently, these models have been implemented at the hardware level by many-core systems [3] and network-on-chip based multi-core systems [4,5]. These systems allow out-of-order executions of the memory access operations within lock protected code sections. Such relaxation would trigger more nondeterministic executions dramatically, hence making it more difficult to verify these relaxed memory models. The verification problem of synchronized consistency models has been rarely studied so far due to its high complexity.

* This work is partially supported by the National Natural Science Foundation of China under Grants No.61272135, No.61100015, No.61204047, No.61100069, and No.61161130530.

F. Cassez and J.-F. Raskin (Eds.): ATVA 2014, LNCS 8837, pp. 264–280, 2014.

A common way to verify the memory model of a multiprocessor system is by running concurrent test programs on the system and then checking whether their executions comply with the memory model under concern. Test programs can be pre-specified or generated randomly. A directed constraint graph can be constructed on the memory access operations in an execution. The edges of the graph represent the order between these operations as permitted by the memory model under concern. In this way, a cycle in the graph would mean a violation of the memory model under concern.

The problem of verifying an execution against a memory model is NP-complete in general [6]. It has been shown that for a constant number of processors, it can take just linear time (in the number of operations) to solve this problem with the aid of the pending period information of operations [7,8]. The pending period of an operation is the interval between the time when the operation is *issued* and the time when the operation is *committed.* Intuitively, two operations in an execution can be ordered in the physical time if one of them is committed before the other one is issued.

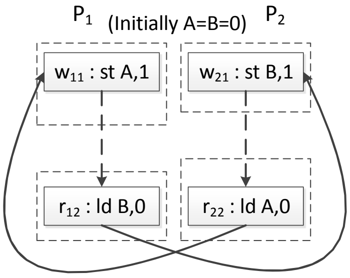

Fig. 1. A False Cycle

However, we find that this time order based approach does not even apply to the total store order (TSO) memory model [9] because it implicitly assumes that the time order along an execution is compatible with the memory model under concern. This underlying assumption does not apply to the TSO/x86 memory model. Fig. 1 shows a typical execution on an x86 microprocessor [10]. In this execution, the write operation w_{11} (respectively, w_{21}) writes the value 1 to the memory address A (respectively, B) on processor P_1 (respectively, P_2). But the read operation r_{12} (respectively, r_{22}) still reads the initial value 0 at the memory address B (respectively, A). In Fig. 1, the solid edges represent the TSO edges, while the dashed boxes indicates the pending periods of the operations. These dashed boxes are not overlapped, hence inducing the time order edges (represented by the dashed edges in Fig. 1). The original time order based approach in [7,8] would treat the cycle of edges in Fig. 1 as a violation of the TSO/x86 memory model. This would also happen to synchronized consistency models, which neither guarantee write operations to be atomic. Therefore, the original time order based approach is limited to the sequential consistency (SC) memory model.

In this paper, we extend the original time order based approach to synchronized consistency models. Given an execution, the synchronization operations accessing the same locks are mutually exclusive to each other under synchronized consistency models. Therefore, these operations need to be executed in a sequentially consistent way. This also applies to the write operations accessing the same addresses. Our approach aims to find total orders between these synchronization operations and between these write operations, in order to justify the execution against the synchronized consistency model under concern. To avoid the above false negative results by the original time order based approach, we relax the notion of time order so that the write operations in an execution can be ordered approximately in the time when these operations are globally visible to all processors. In this way, the relaxed time order along the execution is compatible with the relaxed memory models that do not guarantee write operations to be atomic.

We then propose an active frontier approach to deal with synchronized consistency models based on the notion of relaxed time order. Those operations that should be executed sequentially are identified in separate and composed into active frontiers based on their relaxed pending period information. Our approach is proved to be sound and complete, in the sense that it can indeed find the necessary total orders as a witness if the given execution complies with the synchronized consistency model under concern, and vice versa. As far as we know, our approach provides the first efficient solution for the verification problem of synchronized consistency models.

A precise implementation of our approach would require extra dedicated hardware support for retrieving the time information of executions. For the sake of generality and cost-effectiveness, we implement an over-approximation of our approach in a Memory Order Dynamic Verifier (MODV). As the main case study, we use this tool to verify the memory model of Godson-T, a many-core architecture of industrial size. Memory accesses inside any region were assumed to be coherent for Godson-T. MODV finds that such coherence is actually not guaranteed for regions with multiple locks. This ambiguity has been confirmed by the designers of Godson-T and corrected in its programming manual based on the results of our work. This case study shows that MODV can handle hundreds of thousands of operations on 16 cores in minutes.

2 Related Work

We refer to [11] and [12] for a survey on memory consistency models. Synchronized consistency models such as release consistency [1], entry consistency [13], scope consistency [2] and location consistency [14] have also been uniformly defined in [12] in an axiomatic style.

An empirical approach was presented in [15] to generate litmus tests automatically for multi-core processors. Formal verification techniques have been applied to verify concurrent programs for memory models. To name a few, [16] used the explicit model checker Murφ for operational memory models; while [17] used a SAT solver for axiomatic memory models. [18] presented a verification approach for store buffer safety by non-intrusively monitoring the sequential consistent executions of concurrent programs. However, these techniques still suffer from the scalability issue.

Dynamic analysis has gained more attention for the verification problem of memory consistency models. It can be broadly classified into two categories: hardware-assisted and software-based methods. In hardware-assisted methods, the runtime information such as read mapping and write order can be directly collected through auxiliary hardware. Consequently, efficient verification algorithms can be developed with the time complexity of $O(n)$, where n is the number of the operations in the given execution [19,20,21]. However, this advantage is often offset by extra design effort and silicon area consumption, as well as performance loss, on the hardware level.

On the contrary, software-based methods avoid such nontrivial hardware support by deriving the runtime information from the given execution. The first software-based method was the frontier graph method presented in [6] for the SC memory model. Its time complexity is $O(n^p)$, where p is the number of processors. A sound but incomplete algorithm for the TSO memory model was first proposed in [22] with the time complexity of $O(n^5)$. This algorithm was extended in [23] based on the concept of vector clocks, with the time complexity reduced to $O(pn^3)$. The vector clocks in [23] is computed out by splitting the given execution into virtual SC processors. Another more efficient implementation of [22] was presented in [24] with the time complexity of $O(n^4)$. Furthermore, a backtracking algorithm was proposed in [25] to make the software-based methods complete. The time complexity of this backtracking algorithm is $O(n^p/p^p \times pn^3)$.

The most closely related work to ours are [7,8], where the additional pending period information was exploited for the sake of efficiency. But their approaches are sound only for the SC memory model and may report false negative results for the memory models that do not guarantee write operations to be atomic.

3 Synchronized Consistency Models

In this section, we introduce the memory orders of synchronized consistency models [12]. Herein, we consider four types of operations: read, write, acquire and release. Suppose a multiprocessor system consists of $p \geq 1$ processors with a shared memory. Let A, B denote a memory address, and l denote a lock.

A read operation r in the form of "$ld\ A, i$" reads the value i from the memory address A, while a write operation w in the form of "$st\ B, i$" writes the value i to the memory address B.[1] Let $add(r)$ and $val(r)$ be the memory address that r accesses and the value that r reads, respectively. Similarly, let $addr(w)$ and $val(w)$ be the memory address that w accesses and the value that w writes, respectively. Read and write operations are referred to as memory operations in this paper.

An acquire operation s^a in the form of "$acq\ l$" acquires the lock l, while a release operation s^r in the form of "$rel\ l$" releases it. Let $lock(s^a)$ and $lock(s^r)$ be the locks that s^a and s^r access, respectively. Acquire and release operations are referred to as synchronization operations, denoted s, in this paper. These operations can be used together to implement other atomic synchronization operations, such as barrier operations.

Let u, v denote an operation in general, and $\mathbb{O}$ be the set of all operations. An execution of the system is a tuple $\sigma = (\sigma_1, \ldots, \sigma_p)$, where $\sigma_i = u_{i,1} \ldots u_{i,n_i}$ is a finite sequence of operations on the i-th processor with $1 \leq i \leq p, n_i \geq 1$.

[1] Without loss of generality, we assume that all written values are different.

On each σ_i, acquire and release operations should appear in pairs for the same locks. A fragment of σ_i from an acquire operation s^a to its accompanying release operation s^r constitutes a *synchronization session*, denoted $S = (s^a, s^r)$. Similarly, let $lock(S)$ be the lock that protects the synchronization session S, i.e., $lock(S) = lock(s^a) = lock(s^r)$.

An execution of the system is obtained typically by running a concurrent test program on the system. In an execution σ, two operations u and v of the same processor constitute a *program order* pair, denoted $u \xrightarrow{P} v$, if u is executed before v as dictated by the program. We use the notion of *constraint function* [24] to specify the program order that must be abided under a memory model. A constraint function $cf : \mathbb{O} \times \mathbb{O} \rightarrow Boolean$ of a memory model is defined such that $cf(u, v) = true$ if u must be executed before v under the memory model. In weak consistency, two write operations w_1 and w_2 with $addr(w_1) = addr(w_2)$ must be executed in their program order. In release consistency, an acquire operation s^a must be executed before any read operation r such that $s^a \xrightarrow{P} r$; while in scope consistency, this only happens when s^a and r belong to the same synchronization session.

With the above notations, we now define the axioms of memory orders of synchronized consistency models. A synchronized consistency model with its constraint function cf requires the following partial orders to be satisfied by any execution σ of the system:

Writes-to Order. A write operation w and a read operation r of two different processors constitute a *writes-to order* pair, denoted $w \xrightarrow{Wt} r$, if r reads the value that w writes, i.e., $val(r) = val(w)$.

Local Order. Two operations u and v of the same processor constitute a *local order* pair, denoted $u \xrightarrow{L} v$, if $u \xrightarrow{P} v$ and one of the following two conditions holds:

- $cf(u, v) = true$ if either u or v is a memory operation;
- u and v are both synchronization operations with $lock(u) = lock(v)$.

Synchronization Order. Given two synchronization sessions $S = (s^a, s^r)$ and $S' = (s'^a, s'^r)$ with $lock(S) = lock(S')$, S and S' must be mutually exclusive to each other. This can be formally defined as $(s^r \xrightarrow{Syn} s'^a) \oplus (s'^r \xrightarrow{Syn} s^a)$, where $\oplus$ is the exclusive disjunction operator. Consequently, the synchronized sessions protected by the same lock should be able to be serialized in a total synchronization order.

Coherence Order. Given two write operations w_1 and w_2 with $addr(w_1) = addr(w_2)$, w_1 and w_2 should be able to be serialized. This can be formally defined as $(w_1 \xrightarrow{Co} w_2) \oplus (w_2 \xrightarrow{Co} w_1)$. Then, a read operation r and a write operation w with $addr(r) = addr(w)$ constitute an *inferred* coherence order pair, denoted $r \xrightarrow{Co} w$, if there is a write operation w' such that $w' \xrightarrow{Co} w$ and $val(w') = val(r)$.

This axiom of coherence order was referred to as *write atomicity* in [11], *coherence* in [12,26,27] and *store atomicity* in [7]. Similarly, a total coherence order should exist between the write operations that access the same memory address. In this paper, we include this axiom for the generality of our approach. It is not supported by all synchronized consistency models. Whenever this axiom is included, a local order pair $w \xrightarrow{L} w'$ should hold for any write operations w and w' such that $w \xrightarrow{P} w'$ and $addr(w) = addr(w')$.

Global Order. The transitive closure of the above orders is referred to as global order in this paper. Two operations u and v constitute a *global order* pair, denoted $u \xrightarrow{G} v$, if ($u \xrightarrow{Wt} v$), or ($u \xrightarrow{L} v$), or ($u \xrightarrow{Syn} v$), or ($u \xrightarrow{Co} v$), or there exists an operation u' along the execution such that $u \xrightarrow{G} u'$ and $u' \xrightarrow{G} v$.

Herein, the axiomatic definitions of *Local Order* and *Synchronization Order* are similar to those in [12]. Alternatively, synchronized consistency models can be defined by a "view" method, where each processor has its own view of memory orders of operations [12]. This method has been applied to characterize POWER processors [27]. The memory orders defined in this section can be easily transformed as a linear view order for each processor over all of its operations, together with all the write and synchronization operations of the other processors. In this case, all the processors would share the same view of inter-processor writes-to, synchronization and coherence orders.

4 Baseline Algorithm

Given an execution of a multiprocessor system and a synchronized consistency model with its constraint function, we aim to develop an algorithm that can decide whether the execution complies with the synchronized consistency model. In this section we propose a baseline algorithm for this purpose with an extended notion of frontier.

As in [22,23,24,25], we model the given execution as a constraint graph (V, E), where V is a finite set of nodes representing the operations in the given execution, and $E \subseteq V \times V$ is a finite set of edges representing the ordered pairs of these operations. For brevity, we refer to the operations and the corresponding nodes by the same notation. Then, for two operations u and v, $(u, v) \in E$ if $u \xrightarrow{G} v$.

For the orders defined in Section 3, the corresponding edges can be categorized into two classes: static and dynamic edges. The writes-to and local order edges are static in the sense that these edges are fixed in the constraint graph and can be determined directly by the given execution. On the contrary, the synchronization and coherence order edges have to be constructed tentatively in order to establish the necessary total synchronization and coherence orders.

We extend the notion of frontier [6] to present the search routine for the dynamic edges that can fit in certain total synchronization and coherence orders. For an execution $\sigma = (\sigma_1, \ldots, \sigma_p)$, let $addr(\sigma)$ and $lock(\sigma)$ be the set of the addresses and locks accessed in σ, respectively. Let $\sigma_i|_A$ be the projection of σ_i on the write operations accessing the address $A \in addr(\sigma)$, and $\sigma_i|_l$ be the projection of σ_i on the synchronization operations accessing the lock $l \in lock(\sigma)$.

Without loss of generality, let A_j and l_k range over the addresses in $\mathit{addr}(\sigma)$ and the locks in $\mathit{lock}(\sigma)$, respectively, with $1 \leq j \leq |\mathit{addr}(\sigma)|$ and $1 \leq k \leq |\mathit{lock}(\sigma)|$. Then, a *frontier* is a tuple $f = (w_{11}, \ldots, w_{p|\mathit{addr}(\sigma)|}, s_{11}, \ldots, s_{p|\mathit{lock}(\sigma)|})$, where w_{ij} is a write operation on the i-th processor with $\mathit{addr}(w_{ij}) = A_j$ and s_{ik} is a synchronization operation on the i-th processor with $\mathit{lock}(s_{ik}) = l_k$ for $1 \leq i \leq p, 1 \leq j \leq |\mathit{addr}(\sigma)|, 1 \leq k \leq |\mathit{lock}(\sigma)|$.

Intuitively, in a frontier f, there is one and only one write operation on each processor that accesses each memory address, as well as one and only one synchronization operation on each processor that accesses each lock. A next frontier $f' = f\{u'/u\}$ results from f by replacing u in f with u' such that u and u' belong to the same i-th processor (for some $1 \leq i \leq p$) and u' is the follow-up operation of u on $\sigma_i|_{\mathit{addr}(u)}$ (if u is a write operation) or $\sigma_i|_{\mathit{lock}(u)}$ (if u is a synchronization operation). Then, u' is referred to as the *active* operation of f'. Especially, we attach the beginning operation $\bot$ before the first operation of each $\sigma_i|_{A_j}$ and $\sigma_i|_{l_k}$, and the ending operation $\top$ after the last operation of each $\sigma_i|_{A_j}$ and $\sigma_i|_{l_k}$. The beginning frontier (denoted $f_\bot$) and the ending frontier (denoted $f_\top$) are the ones consisting of $p(|\mathit{addr}(\sigma)| + |\mathit{lock}(\sigma)|)$ beginning and ending operations, respectively.

The baseline algorithm is shown in Algorithm 1. In this algorithm, the static edges are added first and then checked for a possible cycle (Lines 1-2). It can be seen that the constraint graph is acyclic at Line 4. Then, dynamic edges are searched for through a recursive function `ExploreFrontier` (Line 5).

Algorithm 1. Baseline Algorithm

```
  Input: an execution and the constraint function of a memory model
  Output: true if no cycle has been detected, and false otherwise
1 Add writes-to and local order edges;
2 if the above static edges result in a cycle then
3     return false;
4 $f_0 \leftarrow$ the beginning frontier;
5 $sat \leftarrow$ ExploreFrontier($f_0$);
6 return $sat$;
```

The function `ExploreFrontier`, shown in Function 2, explores all the possible frontiers in a depth-first manner. At Line 8, a synchronization order edge is added tentatively between the two latest visited synchronization sessions accessing $\mathit{lock}(u')$; while at Line 12, a coherence order edge is added tentatively between the two latest visited write operations accessing $\mathit{addr}(u')$, together with the coherence order edges inferred from it. Then, the newly added dynamic edges are checked for a possible cycle in the current constraint graph (Line 13). Such a cycle would invalidate the newly added dynamic edges. Hence, if a cycle is detected, the newly added dynamic edges are then removed (Line 14). If all the next frontiers of f have been explored without achieving an acyclic constraint graph, then the function `ExploreFrontier` returns back to its caller with the negative result at Line 19. If this means to return to Algorithm 1, then there is no way to establish a total synchronization order and a total coherence order over the given execution.

Function 2. ExploreFrontier(f)

```
   Input: a frontier $f$
   Output: true if no cycle has been detected, and false otherwise
 1 if $f$ is the ending frontier then
 2     return true;
 3 $res \leftarrow$ false;
 4 for each next frontier $f'$ of $f$ do
 5     $u' \leftarrow$ the active operation of $f'$;
 6     switch ($u'$) do
 7         case $u'$ is an acquire operation
 8             Add the edge $s^r \xrightarrow{Syn} u'$, where $s^r$ is the last active release
               operation with $lock(s^r) = lock(u')$;
 9         case $u'$ is a write operation
10             $w \leftarrow$ the last active write operation with $addr(w) = addr(u')$;
11             for each $r$ such that $val(r) = val(w)$ do
12                 Add the edges $w \xrightarrow{Co} u'$ and $r \xrightarrow{Co} u'$;
13     if FindPath($u', u'$) then
14         Remove the newly added edge(s);
15     else
16         $res \leftarrow$ ExploreFrontier($f'$);
17         if $res$ then
18             break;
19 return $res$;
```

If the ending frontier is eventually reached, then the function `ExploreFrontier` returns directly the positive result (Line 2), which will be carried over to Algorithm 1 through Line 18. In this case, the necessary total synchronization and coherence orders have just been established for the given execution.

As shown in Function 3, we implement the cycle checking function `FindPath`(u, v) in a straightforward way for the baseline algorithm. It is meant to find a path from u to v in the current constraint graph. A cycle passing through an operation u' can then be detected by calling this function with (u', u').

Function 3. FindPath(u, v)

```
Input: operations $u$ and $v$
Output: true if there is a path from $u$ to $v$, and false otherwise
for each $v'$ such that $u \xrightarrow{G} v'$ do
    if $v' = v$ or FindPath($v', v$) then
        return true;
return false;
```

It can be seen that the baseline algorithm is sound and complete for synchronized consistency models, in the sense that it returns false if and only if the given execution does not satisfy the memory model under concern. This can be proved in the similar way as in [25]. But the baseline algorithm would scale poorly because of the combinatorial explosion of the number of frontiers to be explored. Suppose the given execution σ contains n operations on p processors. Then, the baseline algorithm needs to explore at most $O(n^{p(|addr(\sigma)|+|lock(\sigma)|)})$ frontiers. Each time a frontier is confronted, it takes at most $O(n)$ time to check if the newly added dynamic edge(s) would cause a cycle. Moreover, it takes at most $O(n^2)$ time to check whether static edges may result in a cycle. Hence, the worst time complexity is $O(n^2 + n^{p(|addr(\sigma)|+|lock(\sigma)|)+1})$ in total.

5 Exploiting Time Order Information

Apparently the baseline algorithm can not deal with large executions efficiently. In this section we first recall and relax the definition of time order for synchronized consistency models. Then, we present an improvement of the baseline algorithm by taking into account the relaxed time order of the given execution.

In a multiprocessor system with a unique global physical clock, an operation can neither affect others before being issued (namely, entering the instruction window of a processor); nor can be affected after having been committed (namely, having retired from the instruction window of the processor). For an operation u, let $t_e(u)$ and $t_c(u)$ denote the *enter time* when u is issued and the *commit time* when u is committed, respectively. Obviously, $t_e(u) < t_c(u)$ for any operation u. The *pending period* of the operation u is the time interval $[t_e(u), t_c(u)]$. Then, two operations with disjoint pending periods can be ordered in physical time. This can be formalized as the *time order* T such that $u \xrightarrow{T} v$ if $t_c(u) \leq t_e(v)$, otherwise $u \overset{T}{\not\to} v$.

The notion of time order defines a natural order between the operations along the given execution. The time order edges can be determined implicitly by checking the enter and commit time of the related operations.

However, as shown in Fig. 1, the time order is not naturally compatible with the global order in general. According to the definitions in Section 3, the two solid edges in Fig. 1 are actually coherence order edges, which are inferred from the fact that r_{22} and r_{12} read the initial value 0 of A and B, respectively. Then, w_{11} (respectively, w_{21}) is committed when it writes to the internal write buffer of the processor P_1 (respectively, P_2). At this moment, w_{11} (respectively w_{21}) has not been performed globally. Hence, the values written by w_{11} and w_{21} are not yet visible to all the processors.

Let $t_p(u)$ denote the *performed time* when the operation u is performed globally and is visible to all processors. A read operation is performed globally when it fetches a value from the specified memory address, while a write operation is performed globally when it stores the specified value to the main memory (or the L2 cache for a multi-core processor). A synchronization operation is performed globally when it gets the access to the specified lock. Hence, it can be seen that all but non-atomic write operations would take effect before being committed. Obviously, $t_e(u) < t_p(u)$ for any operation u.

If the time order can be rectified by replacing the commit time of an operation with its performed time, the cycle in Fig. 1 can then be eliminated, as shown in Fig. 2 (where the dashed boxes surrounding the write operations are enlarged to indicate their expanded pending periods).

However, the performed time of a write operation can not be observed directly from the given execution. We choose to approximate it based on the pending period information of the related read operations and its follow-up operations.

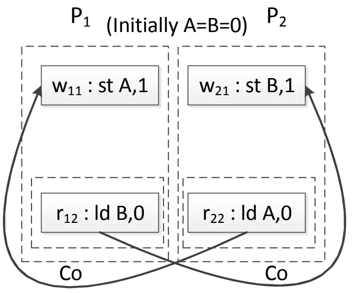

Fig. 2. False Cycle is Eliminated

Definition 1 (Relaxed Time Order). *The* relaxed commit time *of an operation u, denoted $t_{rc}(u)$, is defined as follows:*

- *if u is a read or synchronization operation, $t_{rc}(u) = t_c(u)$;*
- *if u is a write operation, $t_{rc}(u) = \min_{v \in N(u)} t_{rc}(v)$ if $N(u) \neq \emptyset$, where $N(u) = \{v \mid u \xrightarrow{Wt} v$, or $u \xrightarrow{L} v\}$; otherwise, $t_{rc}(u) = t_\infty$, where t_∞ is a sufficiently large time constant such that any operation in the given execution will be performed by then.*

Accordingly, the relaxed pending period *of the operation u is the time interval $[t_e(u),\ t_{rc}(u)]$. Any operations u and v constitute a* relaxed time order *pair, denoted $u \xrightarrow{RT} v$, if $t_{rc}(u) \leq t_e(v)$; otherwise, $u \overset{RT}{\not\longrightarrow} v$.*

The following lemma shows that the relaxed pending period of an operation u covers its performed time, i.e., $t_e(u) < t_p(u) \leq t_{rc}(u)$. The details of its proof can be found in the technical report [28].

Lemma 1. *$t_p(u) \leq t_{rc}(u)$ for any operation u.*

Moreover, it is generally accepted that a multiprocessor system should be designed to be able to guarantee certain physical time constraints under its memory model [7,8]. Definition 2 summarizes the time constraints for the implementation mechanisms of multiprocessor systems.

Definition 2 (Preconditions of Time Order). *For any operations u and v:*

1. *If $u \xrightarrow{P} v$, then u is issued no later than v, i.e., $t_e(u) \leq t_e(v)$.*
2. *If $u \xrightarrow{G} v$, then u is performed no later than v, i.e., $t_p(u) \leq t_p(v)$.*

These preconditions are defined following the same principles as in the original time order based approach [7,8]. For a read operations r and a write operation w, if $w \xrightarrow{Wt} r$, then r can only fetch the value $val(w)$ after w stores it into the main memory. Hence, $t_p(w) \leq t_p(r)$. Similarly, synchronization operations accessing the same locks, as well as write operations accessing the same memory addresses, should also be managed in a serializable manner. If a multiprocessor system supports a synchronized consistency model, then any execution of the system should satisfy the synchronized consistency model without violating these preconditions. The following theorem shows that the relaxed time order is compatible with the global order under the preconditions in Definition 2.

Theorem 1. *For any operations u and v, $u \xrightarrow{G} v$ implies $v \not\xrightarrow{RT} u$.*

Proof. If $u \xrightarrow{G} v$, then $t_p(u) \leq t_p(v)$ by Definition 2. Since $t_p(v) \leq t_{rc}(v)$ (by Lemma 1) and $t_e(u) < t_p(u)$, we have $t_e(u) < t_{rc}(v)$, i.e., $v \not\xrightarrow{RT} u$. □

We now present the final algorithm that can take advantages of the relaxed time order. In addition to the given execution and the constraint function of the memory model under concern, the time information of the execution is required as part of the input to the final algorithm. This time information will be preprocessed by the final algorithm to compute the relaxed pending periods of the operations. Then, the final algorithm proceeds as the baseline algorithm, except replacing the function `ExploreFrontier` of the baseline algorithm with the function `ExploreActiveFrontier`, shown in Function 4.

Function 4. ExploreActiveFrontier(f)

```
Input: a frontier $f$
Output: true if no cycle has been detected, and false otherwise
if $f$ is the ending frontier then
    return true;
$res \leftarrow$ false;
for each next active frontier $f'$ of $f$ do
    $u' \leftarrow$ the active operation of $f'$;
    switch ($u'$) do
        case $u'$ is an acquire operation
            Add the edge $s^r \xrightarrow{Syn} u'$, where $s^r$ is the last active release
            operation with $lock(s^r) = lock(u')$;
        case $u'$ is a write operation
            $w \leftarrow$ the last active write operation with $addr(w) = addr(u')$;
            for each $r$ such that $val(r) = val(w)$ do
                if $u' \xrightarrow{RT} r$ then
                    return $res$;
            for each $r$ such that $val(r) = val(w)$ and $u' \not\xrightarrow{RT} r$ do
                Add the edges $w \xrightarrow{Co} u'$ and $r \xrightarrow{Co} u'$;
    if FindTimedPath($u', u'$) then
        Remove the newly added edges;
    else
        $res \leftarrow$ ExploreActiveFrontier($f'$);
        if $res$ then
            break;
return $res$;
```

At Line 4 of Function 4, only active frontiers need to be explored. Given an execution σ, the *active period of a write operation* w on the i-th processor is the time interval $[t_e(w), t_{rc}(w')]$, where w' is the follow-up write operation of w in $\sigma_i|_{addr(w)}$; while the *active period of a synchronization operation* s on the i-th processor is the time interval $[t_e(s), t_{rc}(s')]$, where s' is the follow-up synchronization operation of s in $\sigma_i|_{lock(s)}$. Then, a frontier f is *active* if each operation in f is in the active period of each other operation in f. The notion of active frontier is inspired by the notion of feasible frontier in [8]. But [8] concerns only the SC memory model and assumes the pending periods of two consecutive operations on the same processor are always overlapped.

In this way, the frontiers that are not active under the physical time can be ignored without missing any chance to establish the correctness of the given execution. At Line 13 of Function 4, a cycle is detected with $r \xrightarrow{Co} u'$ and $u' \xrightarrow{RT} r$. This is contrary to Theorem 1, which directly means a violation of the given memory model under the preconditions in Definition 2. At Line 16 of Function 4, a new cycle checking function `FindTimedPath` is called to check for a possible cycle in the current constraint graph under the relaxed time order.

The function `FindTimedPath` (u, v), shown in Function 5, only needs to examine the operations within the relaxed pending period of the operation v. For any operation v' such that $u \xrightarrow{G} v'$, if it is committed before the relaxed pending period of the operation v, then there exists a relaxed time order edge from v' to v, i.e., $v' \xrightarrow{RT} v$. Thus, a timed path $u \xrightarrow{G} v' \xrightarrow{RT} v$ is resulting from the current constraint graph (at Line 3 of Function 5). If $u = v$, this path constitutes a cycle that invalidates the newly added dynamic edges. In this way, the subsequent global order edges from v' need not to be further checked. For an operation v' issued after the relaxed pending period of the operation v, the global order edge $u \xrightarrow{G} v'$ would be considered as a time order edge for later cycle checking.

Function 5. FindTimedPath(u, v)

```
   Input: operations $u$ and $v$
   Output: true if there is a path backing to $u$ from $v$, and false otherwise
1  for each $v'$ such that $u \xrightarrow{G} v'$ and $v \not\xrightarrow{RT} v'$ do
2      if $v' = v$ or $v' \xrightarrow{RT} v$ or FindTimedPath($v', v$) then
3          return true;
4  return false;
```

Since the relaxed time order is compatible with the global order, it can be seen that the final algorithm is sound and complete, as stated in the following theorem. The details of its proof can be found in the technical report [28].

Theorem 2 (Soundness and Completeness of the Final Algorithm). *The final algorithm presented in this section returns false if and only if the given execution does not satisfy the given synchronized consistency model under the preconditions in Definition 2.*

Time Complexity. Suppose in the relaxed pending period of an operation, there are C operations running on each processor. C is usually a hardware-dependant constant [7]. Then, at most $O(nC^{p(|addr(\sigma)|+|lock(\sigma)|)-1})$ active frontiers need to be explored. Similarly, when an active frontier is confronted, it would only take $O(pC)$ time to check for a possible cycle within the relaxed pending period of the latest active operation. So the upper bound of the time complexity of active frontier traversal is $O(npC^{p(|addr(\sigma)|+|lock(\sigma)|)})$. Furthermore, it would take at most $O(n^2)$ time to relax the pending periods of write operations. Recall that it would also take at most $O(n^2)$ time to check whether static edges may cause a cycle. Hence, the worst time complexity of this final algorithm is $O(2n^2 + npC^{p(|addr(\sigma)|+|lock(\sigma)|)})$ in total. Obviously, the final algorithm would scale much better with large executions than the baseline algorithm.

6 Experimental Results

It can be seen that a precise implementation of the final algorithm would closely depend on the time information of executions. However, it requires extra hardware support with specific internal registers to retrieve the enter time and commit time of each operation. Similar to [8], we uses the general performance counter sampling mechanism to over-approximate the pending period information of operations. Hence, the soundness of the final algorithm is preserved under this approximation. We have developed a Memory Order Dynamic Verifier (MODV) to implement our algorithms.[2] Through combining different constraint functions and axiomatic rules of memory orders, MODV can support various memory models, including SC, TSO/x86 and typical synchronized consistency models.

Performance counters have been supported by most industrial processors. In a multiprocessor system, the values of performance counters can be scanned out from its internal registers through certain debug interface. The pending period information of each operation can be computed out through scanning performance counters periodically, though the actual time order information may be lost partially during the consecutive scans. The tighter the performance counter scan period is, the more precise the pending period information obtained can be. But, a tight scan period would exert too much pressure on the system performance. The performance counter scan period is set to be 600 cycles per scan in the following experiments.

As the main case study, we use MODV to verify the memory model of the Godson-T many-core architecture [3]. Godson-T is a many-core processor with 64 homogeneous processing cores, each of which has a 16KB private instruction cache and a 32KB local memory. Moreover, a dedicated synchronization manager provides architectural support for mutual exclusion and barrier synchronization. The memory model of Godson-T is a variant of scope consistency. Godson-T uses a region-based cache coherence (RCC) protocol to support large-scale parallelism. A region is exactly a synchronization session defined in this paper.

[2] MODV is available at `http://lcs.ios.ac.cn/~lvyi/MODV/index.html`

MODV has found several bugs in the design of Godson-T. One of them is related to Godson-T's memory model. Memory accesses inside any region were assumed to be coherent for Godson-T. But actually this is not guaranteed for regions with multiple locks. MODV finds this bug through an execution shown in Fig. 3 (with simplification for clarity). In this execution, $w_{13} \xrightarrow{Wt} r_{22}$ because r_{22} reads the value of w_{13}; Similarly, $w_{12} \xrightarrow{Wt} r_{33}$. Then, since $w_{12} \xrightarrow{L} w_{13}$, an inferred coherence order edge exists between r_{33} and w_{13}, i.e., $r_{33} \xrightarrow{Co} w_{13}$. Moreover, $r_{22} \xrightarrow{RT} r_{33}$ because $t_{rc}(r_{22}) < t_e(r_{33})$. Hence, the cycle $w_{13} \xrightarrow{Wt} r_{22} \xrightarrow{RT} r_{33} \xrightarrow{Co} w_{13}$ is detected.

The reason of this cycle is as follows. When a processor writes a value into a memory address in a region, it first stores the value into its internal cache, and then writes through into the memory (L2 cache) immediately. If a processor reads a memory address for the first time in a region, it first invalidates its cache, and then reads the value from the memory directly. For the subsequent read operations to the same memory address in the same region, it will read the value from its cache. Therefore, r_{32} and r_{33} reads the value 1 from the memory and from P_3's cache, respectively. In the meanwhile, the values of the same memory address at the memory and at P_2's cache are both 2. Hence, the memory system of Godson-T is not cache coherent for regions with multiple locks.

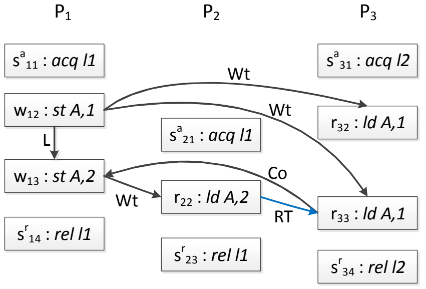

Fig. 3. A Bug of Godson-T

We then illustrate the performance of our algorithms with large scale test programs. All the experiments have been carried out on a Linux server with four 8-core 2.4GHz Intel Xeon processors and 48GB memory. To validate synchronization and coherence orders together, we randomly generate concurrent test programs with 60% load instructions and 30% store instructions for 2 different addresses, and 10% synchronization instructions for one lock. Branch instructions are not used in these programs.

Fig. 4 shows the average performance of the baseline and final algorithms for up to 100K operations on 2 cores. It can be seen that the final algorithm performs much better than the baseline algorithm when the number of operations increases. As a matter of fact, for no less than 4K operations on no less than 4 cores, the baseline algorithm often cannot return within 8 hours. Fig. 5 shows the average performance of the final algorithm on 2, 4, 8 and 16 cores. It can be seen that with the aid of the relaxed time order, the final algorithm also scales well with the increasing numbers of cores.

The fluctuations in Fig. 5 are because the information derived from consecutive scans is an over-approximation of the pending periods of the operations. The lost time order information would result in extra backtracking during the

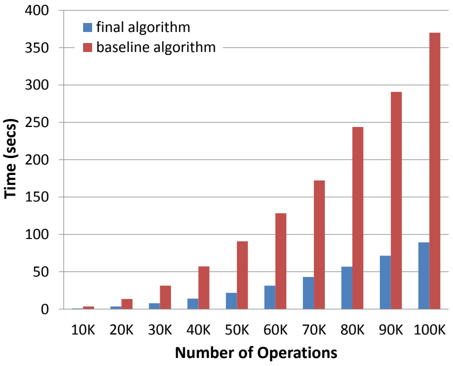

Fig. 4. Comparison of Results

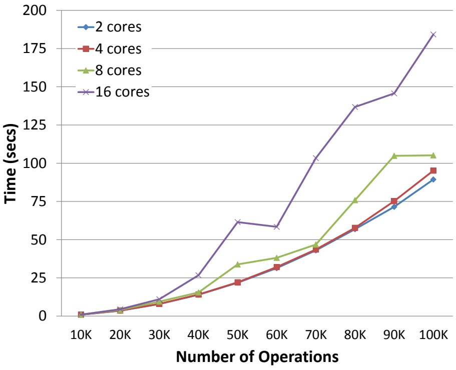

Fig. 5. Performance Test

exploration of active frontiers. This makes the time consumption of MODV fluctuate, especially for more than 4 cores.

7 Conclusion

We present in this paper a relaxed time order based active frontier approach for verifying synchronized consistency models. The original notion of frontier is expanded with the memory addresses and the locks accessed along an execution. Then, we integrate this extended frontier approach with the pending period information of operations. The notion of active frontier is introduced to reduce the number of frontiers to be explored and the number of operations to be examined for cycle checking. In literature, the notion of time order has not yet been widely appreciated due to its incompatibility issue. Our approach addresses this issue by relaxing the time order of the given execution in a conservative way. On one hand, our approach is sound in the sense that it would not produce false negative results for memory models with non-atomic write operations. On the other hand, our approach is also complete in the sense that it can guarantee to detect a cycle if the given execution does not comply with the memory model under concern.

Without loss of generality and cost-effectiveness, we have implemented an over-approximation of our approach in a verification tool MODV. The tool preserves the soundness of our approach, and can be easily customized to support various memory models with user-defined constraint functions and user-selected memory orders. We have used MODV to verify the memory model of the Godson-T many-core processor, and found that Godson-T does not support the coherence order for regions with multiple locks. This bug has been confirmed by the designers of Godson-T. Its programming manual has been revised based on the results of our work. This case study shows that our approach is very efficient in practice for detecting subtle bugs in multiprocessor systems.

Our approach exploits the advantages of time order for verifying a wider range of memory models. As the future work, we will investigate the memory models of the POWER and ARM architectures, where write operations are also not guaranteed to be atomic.

References

1. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors. In: Baer, J., Snyder, L., Goodman, J.R. (eds.) ISCA 1990, pp. 15–26. ACM (1990)
2. Iftode, L., Singh, J.P., Li, K.: Scope consistency: A bridge between release consistency and entry consistency. Theory of Computing Systems 31(4), 451–473 (1998)
3. Fan, D., Zhang, H., Wang, D., Ye, X., Song, F., Li, G., Sun, N.: Godson-T: An efficient many-core processor exploring thread-level parallelism. IEEE Micro 32(2), 38–47 (2012)
4. Naeem, A., Jantsch, A., Lu, Z.: Scalability analysis of memory consistency models in NoC-based distributed shared memory SoCs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32(5), 760–773 (2013)
5. Hansson, A., Goossens, K., Bekooij, M., Huisken, J.: CoMPSoC: A template for composable and predictable multi-processor system on chips. ACM Trans. Des. Autom. Electron. Syst. 14(1), 2:1–2:24 (2009)
6. Gibbons, P.B., Korach, E.: On testing cache-coherent shared memories. In: SPAA 1994, pp. 177–188. ACM (1994)
7. Chen, Y., Lv, Y., Hu, W., Chen, T., Shen, H., Wang, P., Pan, H.: Fast complete memory consistency verification. In: HPCA 2009, pp. 381–392. IEEE Computer Society (2009)
8. Hu, W., Chen, Y., Chen, T., Qian, C., Li, L.: Linear time memory consistency verification. IEEE Transactions on Computers 61(4), 502–516 (2012)
9. Sindhu, P., Frailong, J.M., Cekleov, M.: Formal specification of memory models. In: Dubois, M., Thakkar, S. (eds.) Scalable Shared Memory Multiprocessors, US, pp. 25–41. Springer, Heidelberg (1992)
10. Sorin, D.J., Hill, M.D., Wood, D.A.: A primer on memory consistency and cache coherence. Synthesis Lectures on Computer Architecture 6(3), 1–212 (2011)
11. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29, 66–76 (1996)
12. Robert, C.S., Gary, J.N.: A unified theory of shared memory consistency. J. ACM 51(5), 800–849 (2004)
13. Bershad, B., Zekauskas, M., Sawdon, W.: The Midway distributed shared memory system. In: COMPCON 1993. Digest of Papers, pp. 528–537 (1993)
14. Gao, G., Sarkar, V.: Location consistency: a new memory model and cache consistency protocol. IEEE Transactions on Computers 49(8), 798–813 (2000)
15. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Litmus: Running tests against hardware. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 41–44. Springer, Heidelberg (2011)
16. Park, S., Dill, D.L.: An executable specification, analyzer and verifier for RMO (relaxed memory order). In: SPAA 1995, pp. 34–41. ACM (1995)
17. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: a framework for axiomatic and executable specifications of memory consistency models. In: IPDPS 2004, pp. 31–40. IEEE Computer Society (2004)

18. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
19. Meixner, A., Sorin, D.: Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. IEEE Transactions on Dependable and Secure Computing 6(1), 18–31 (2009)
20. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: Detecting atomicity violations via access-interleaving invariants. IEEE Micro 27(1), 26–35 (2007)
21. DeOrio, A., Wagner, I., Bertacco, V.: Dacota: Post-silicon validation of the memory subsystem in multi-core designs. In: HPCA 2009, pp. 405–416. IEEE Computer Society (2009)
22. Hangal, S., Vahia, D., Manovit, C., Lu, J.Y.J.: TSOtool: A program for verifying memory systems using the memory consistency model. In: ISCA 2004, pp. 114–123. IEEE Computer Society (2004)
23. Manovit, C., Hangal, S.: Efficient algorithms for verifying memory consistency. In: SPAA 2005, pp. 245–252. ACM (2005)
24. Roy, A., Zeisset, S., Fleckenstein, C., Huang, J.: Fast and generalized polynomial time memory consistency verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 503–516. Springer, Heidelberg (2006)
25. Manovit, C., Hangal, S.: Completely verifying memory consistency of test program executions. In: HPCA 2006, pp. 166–175. IEEE Computer Society (2006)
26. Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Nardelli, F.Z.: The semantics of POWER and ARM multiprocessor machine code. In: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, pp. 13–24. ACM (2009)
27. Mador-Haim, S., et al.: An axiomatic memory model for POWER multiprocessors. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 495–512. Springer, Heidelberg (2012)
28. Lv, Y., Sun, L., Ye, X., Fan, D., Wu, P.: Efficiently and completely verifying synchronized consistency models. Technical Report ISCAS-SKLCS-14-07, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences (2014), `http://lcs.ios.ac.cn/~lvyi/MODV/files/ISCAS-SKLCS-14-07.pdf`