Fast Complete Memory Consistency Verification *

Yunji Chen[1], Yi Lv[2], Weiwu Hu[1], Tianshi Chen[31], Haihua Shen[1], Pengyu Wang[1], Hong Pan[2]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, P. R. China

[2] Institute of Software, Chinese Academy of Sciences, Beijing 100190, P. R. China

[3] University of Science and Technology of China, Hefei, Anhui 230027, P. R. China

Abstract

The verification of an execution against memory consistency is known to be NP-hard. This paper proposes a novel fast memory consistency verification method by identifying a new natural partial order: time order. In multiprocessor systems with store atomicity, a time order restriction exists between two operations whose pending periods are disjoint: the former operation in time order must be observed by the latter operation. Based on the time order restriction, memory consistency verification is localized: for any operation, both inferring related orders and checking related cycles need to take into account only a bounded number of operations.

Our method has been implemented in a memory consistency verification tool for CMP (Chip Multi Processor), named LCHECK. The time complexity of the algorithm in LCHECK is $O(\mathrm{C}^p p^2 n^2)$ (where C is a constant, p is the number of processors and n is the number of operations) for soundly and completely checking, and $O(p^3 n)$ for soundly but incompletely checking. LCHECK has been integrated into both pre and post silicon verification platforms of the Godson-3 microprocessor, and many bugs of memory consistency and cache coherence were found with the help of LCHECK.

Keywords: memory consistency, cache coherence, verification, time order, pending period

*Corresponding author: Weiwu Hu (Email:hww@ict.ac.cn). This work is partially supported by the National Natural Science Foundation of China under Grants No.60721061, No.60833001, and No.60603049, the National High Technology Development 863 Program of China under Grants No.2007AA01Z147, No.2007AA01Z112, and No.2008AA110901, the National Grand Fundamental Research 973 Program of China under Grant No.2005CB321600, and the Beijing Natural Science Foundation under Grant No.4072024.

1 Introduction

Verifying a memory subsystem is a great challenge in the validation of CMPs. To share memory in multiprocessor systems, the memory subsystem must include many resources to support memory consistency and cache coherence. Therefore, memory consistency verification is an indispensable part of verifying the memory subsystem.

Researchers have found that the verification of an execution against memory consistency is NP-hard with respect to the number of memory operations [3, 8]. To cope with the problem in practice, there are two types of solutions: microarchitecture dependent methods which exploit the help of extra observability in the design to bring down the complexity [22, 25, 26], and microarchitecture independent methods which devise polynomial time algorithms that are sound but not necessarily complete [12, 23, 24, 29]. Generally speaking, microarchitecture dependent methods are difficult to generalize across different microarchitectures. Furthermore, they are hard to use in post-silicon verification, since the cost of providing extra observability in a real chip is high.

Microarchitecture independent methods are also far from perfect. Even the fastest one of them still requires time complexity of $O(pn^3)$ (p is the number of processors, n is the number of operations) [23], at the cost of losing completeness; if one wants to preserve completeness, time complexity further grows to $O((n/p)^p pn^3)$ [24]. Moreover, due to the restrictions of speed and other practical issues, length and memory access locations of test programs for microarchitecture independent methods are limited.

In this paper, we propose a fast and easy-to-generalize memory consistency verification method. We first introduce the concept of *time order* between operations by defining the *pending period* of an operation as the period from the enter time to the commit time of the operation. More precisely, an operation which has not entered any internal structures (e.g., instruction window, load store queue, write buffer and so on) must not be seen by any processor, and an operation which has been globally viewed by all processors cannot be affected by unentered operations. If the commit time of operation u is before the enter time of operation v (u and v have disjoint pending periods), we say u is before v in *time order*. Given the concept of time order, the time order restriction refers to the following fact: the former operation in time order must be observed by the latter operation in time order. In this paper, we demonstrate that in multiprocessor systems with store atomicity (most memory models have the store atomicity property [1]), the *time order restriction* does exist, though this concept does not appear in previous definitions of any memory consistency. With the time order restriction, memory consistency verification can be localized, since the fact that an operation cannot "live" forever implies that only a bounded number of operations are in the pending period of a memory operation. As a result, for any operation, both inferring related orders and checking related cycles need to take into account only a bounded number of operations.

Based on the theoretical investigations related to the time order restriction, we propose a memory consistency verification tool named LCHECK. LCHECK can verify a number of memory consistency models, including sequential consistency [30], processor consistency [10], weak consistency [6], and release consistency [7]. Although LCHECK requires the memory system to support store atomicity, it does not need certified store atomicity as a precondition, since it can find violations of both memory consistency and store atomicity. LCHECK supports both pre-silicon and post-silicon verification. For post-silicon verification, LCHECK only needs several additional software visible registers per processor to estimate the pending period of operations. The theoretical and empirical investigations in this paper show the following promising characteristics of LCHECK: the time complexity of LCHECK is $O(\mathrm{C}^p p^2 n^2)$ for soundly and completely checking (C is a constant), and $O(p^3 n)$ for soundly but incompletely checking. As a practical example, LCHECK is used to verify the Godson-3 microprocessor [17, 18], which is a CMP with 4-16 processor cores.

The main contributions of our work are summarized as follows: first, it is the first time that time order between operations is proposed for multiprocessor system; second, it is also the first time that a localized checking method is proposed for memory consistency verification; and third, the memory consistency verification tool proposed in this paper is fast, complete and easy-to-generalize, which is proven or validated by both theoretical and practical results.

The rest of the paper is organized as follows: Section 2 introduces related work; Section 3 theoretically analyzes memory consistency verification under time order restriction; Section 4 introduces LCHECK; Section 5 presents experiment results of LCHECK; Section 6 concludes the paper and discusses future work.

2 Related Work

Whether a multiprocess system conforms to a memory consistency model is notoriously hard to verify. Most formal verification approaches have focused on relatively small and manually constructed models, which cannot cope with the complexity of realistic modern multiprocessors [4, 11, 27, 32]. The most effective method widely accepted in industry is the dynamic verification approach, i.e., executing a large number of test programs and verifying them against a given memory consistency model using an ad hoc checker tool. Gibbons and Korach study some variations of the VSC (verifying sequential consistency) problem [8, 9]. They prove that if we know the read mapping which maps every read to the write sourcing its value, the obtained VSC-read problem is still NP-complete. If we know the write order for each memory location totally, the obtained VSC-write problem is also NP-complete. However, if we know both the read mapping and the total write order, the obtained VSC-conflict problem belongs to P.

Some previous works strongly rely on microarchitecture: they observe internal information from the design to reduce time complexity. In [25, 26], Meixner and Sorin carry out dynamic verification of sequential consistency and other memory consistency models by dynamically verifying a set of sub-invariants, which is based on the coherence epoch of cache lines [28]. However, to check the order of operations, their approach requires much detailed microarchitecture information, and needs to modify the functionality of the original design, including adding a new stage to the processor pipeline. Ludden *et al.* [22] and Victor *et al.* [31] use internal tools to verify IBM POWER series microprocessor systems by exploiting the additional observability present in simulation. Their methods heavily depend on their microarchitecture such as instruction-by-instruction checking and a memory-tracing mechanism, and the completeness or efficiency of their methods is not described.

Some previous works do not rely as much on microarchitecture: they focus on VSC-read like problems while no additional observability is required. In [5], Collier uses a dedicated test program to detect the violation of the memory system against a given memory consistency model. In [12], Hangal *et al.* propose a tool named TSOtool, which runs a pseudo-random generated test program with data races, and then checks the log file by performing cycle detection of the memory ordering relation graph. Till now, the time complexity of the TSOtool has been reduced to $O(pn^3)$, where p is the number of processors and n is the number of operations [23]. However, due to the lack of observability of total write order, the methods proposed in [12, 23, 29] are all incomplete. To resolve completeness, [24] improves the approach of [23] by adding a backtracking subroutine in the checker. It is a sound and complete algorithm with the time complexity of $O((n/p)^p pn^3)$. However, all microarchitecture independent methods should pay a price by their super-linear time complexity.

Our approach based on time order restriction is substantially different from the above works, although exploiting physical or logical time in memory consistency is not a new idea [19, 28, 13]. In this paper, time information in terms of time order and pending periods of operations is used to localize the relation among operations, which is a novel idea. This idea leads to a fast and easy-to-generalize method for memory consistency verification, which significantly reduces time complexity from $O((n/p)^p pn^3)$ to $O(\mathrm{C}^p p^2 n^2)$ for complete checking, and from $O(pn^3)$ to $O(p^3 n)$ for incomplete checking. Moreover, the hardware support needed by our method is trivial: several software visible registers per processor are sufficient to observe time order. The support has been implemented in the Godson-3 microprocessor.

3 Memory Consistency Verification Under Time Order Restriction

To verify an execution against memory consistency, the common method is to construct a directed graph representing orders in the execution, and then to check for cycles in the directed graph. In the previous works, constructing execution graphs requires multiple phases involving all operations in the execution [12, 23, 24, 29], resulting in the very high complexity of memory consistency verification. However, in multiprocessor systems with store atomicity, the lifetime of an operation is limited. Two operations with non-overlapping lifetimes should be ordered. We will show by theoretical analysis that this promising property can dramatically simplify the construction and the checking of the execution graph.

To carry out theoretical analysis, we first introduce the terminology and notations used in the rest of the paper. A *system* is a multiprocessor system of p processors that access a shared memory. Each processor can issue operations to access the memory through a series of read and write operations on the memory or to synchronize. O is the set of operations issued by all processors. The *temporary internal structure* of a system is the internal structure excluding cache and memory. Every operation *enters* the temporary internal structure of a processor in program order and is finally *globally viewed* by all processors. We use u or v (with subscript) to represent an operation, o (with subscript) to represent a memory operation, r (with subscript) to represent a read operation, w (with subscript) to represent a write operation, s (with subscript) to represent a sync operation. We denote the problem of memory consistency verification under time order restriction as *VMC-time*. Similar to [8], we refer to VMC-time with additional read mapping information as *VMC-time-read*.

3.1 Traditional Orders in Memory consistency

Operation orders are the basis of memory consistency verification. In this subsection, we briefly introduce traditional orders in memory consistency. First let us come to the definitions of program order, processor order and execution order, which are three well-known types of partial orders in multiprocessor systems.

Definition 1 ***Program Order****: Given two different operations u_1 and u_2 in the same processor, we say that u_1 is before u_2 in program order iff u_1 is before u_2 in the program. We denote this as $u_1 \xrightarrow{P} u_2$.*

Definition 2 ***Processor Order****: Given two different operations u_1 and u_2 in the same processor, we say that u_1 is before u_2 in processor order iff there is global agreement that u_1 is before u_2 for all processors. We denote this as $u_1 \xrightarrow{PO} u_2$.*

Many multiprocessor systems have cache subsystems, thus they maintain cache consistency, which can be defined as coherence order: all write operations to the same location are performed in some sequential order. When a multiprocessor system is required to be coherent, it makes sense to consider both write-before order and read-before order, which are called execution orders.

Definition 3 ***Execution Order****: We say that a write operation w is before operation u in execution order iff w is the latest write operation before u that accesses the same location as u. We denote this as $w \xrightarrow{E} u$. We say that a write operation w is after operation u in execution order iff w is the first write operation after u that accesses the same location as u. We denote this as $u \xrightarrow{E} w$.*

In addition, global order is another well-known partial order based on processor order and execution order. It is the transitive closure of processor order and execution order.

Definition 4 ***Global Order****: We say that operation u_1 is before operation u_2 in global order iff u_1 is before u_2 in processor order, or u_1 is before u_2 in execution order, or u_1 is before some operation u in global order and u is before u_2 in global order.*

$$(u_1 \xrightarrow{GO} u_2) \rightarrow \big((u_1 \xrightarrow{PO} u_2) \vee (u_1 \xrightarrow{E} u_2) \vee (\exists u \in O : u_1 \xrightarrow{GO} u \xrightarrow{GO} u_2)\big). \quad (1)$$

Most memory consistency models consist of different rules for processor order. When an operation u_1 is before u_2 in program order, this does not always mean that u_1 is before u_2 in processor order. For example, in a system supporting weak consistency, regular memory operations can execute out of order globally, except across synchronization points. In this paper, we concentrate on the Godson consistency model obeyed by the Godson-3 microprocessor [17, 18].

The restrictions on order in Godson consistency are: any load operation r is allowed to perform with respect to any other processors if all previous synchronization operations have been globally performed; any store operation w is allowed to perform with respect to any other processors if all previous operations have been globally performed; any synchronization operation s is allowed to perform with respect to any other processors if all previous operations have been globally performed. Formally, we have

$$(s \xrightarrow{P} r) \rightarrow (s \xrightarrow{PO} r),$$
$$(u \xrightarrow{P} w) \rightarrow (u \xrightarrow{PO} w),$$
$$(u \xrightarrow{P} s) \rightarrow (u \xrightarrow{PO} s).$$

An execution can be represented as a directed execution graph with processor order edges and execution order edges. Memory consistency verification is also equivalent to checking whether the execution graph is a DAG (directed acyclic graph) [20, 16].

3.2 Pending Period and Time Order

In this subsection, we infuse the concepts of pending period and time order into multiprocessor systems. In a von Neumann architecture, an operation must be fetched into the processor before it is executed, hence there is an enter time for any operation. Before an operation enters, it cannot affect other operations. An operation will leave all temporary internal structures of the processor in bounded time, otherwise there will be some deadlock or livelock, which can be detected by other tools. Thus there is a commit time for any operation. The perform time of an operation –i.e., the time when the operation is performed globally– must be between its enter time and its commit time.

The concrete definitions of enter time and commit time are architecture-dependent. The enter time of an operation can be the time when the operation is fetched, or when the operation is decoded. The commit time of store can be the time when the store operation writes its value into the data cache; the commit time of load can be the time when the load operation writes back to the register file. In any case, any unentered operation must observe the results of any committed operation, and any committed operation cannot be affected by any unentered operation.

The commit time can be treated as a relaxation of perform time: when an operation commits, it has been globally performed. The reason for using commit time instead of perform time is that, in the verification, the precise commit time can be easily obtained without considering the states of other processors, while precise perform time needs to consider the inner state of other processors, their caches and the network.

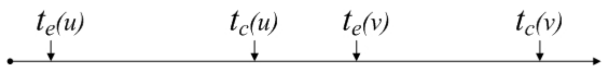


Figure 1. Time order between u and v

On the basis of enter time and commit time, we introduce the pending period of an operation.

Definition 5 ***Pending Period***: *The* ***pending period*** *of u is the period from $t_e(u)$ to $t_c(u)$. We say that an operation v is in the pending period of operation u iff the pending periods of the two operations are overlapped.*

In multiprocessor systems with store atomicity, whatever the precise definitions of enter time and commit time are, a partial order exists between two operations executing in disjoint pending periods. We call the partial order *time order*.

Definition 6 ***Time Order***: *If the commit time of operation u is before the enter time of operation v, we say that u is before v in time order. Formally,*

$$\big(t_c(u) < t_e(v)\big) \leftrightarrow (u \xrightarrow{T} v). \qquad (2)$$

Time order does not require that u and v are executed in the same processor or access the same location. It simply depends on their enter and commit times given by a global clock. According to the above definition of time order, if operation v is in the pending period of operation u, then $\neg(u \xrightarrow{T} v \vee v \xrightarrow{T} u)$ holds. As shown in Figure 1, if the pending periods of u and v do not overlap, then $u \xrightarrow{T} v$ holds.

According to the discussions above, given a global clock, we can obtain a total order of all time points (e.g., enter time and commit time) of operations. However, first we need to investigate the relation between partial orders of operations and total order of time points. The preconditions of our further investigations, which link partial orders of operations and total order of time points together, are presented below.
Preconditions: In a multiprocessor system with store atomicity, considering two operations u and v, the following preconditions are reasonable.

1. If u is before v in program order, the enter time of u is before the enter time of v:

$$(u \xrightarrow{P} v) \rightarrow \big(t_e(u) < t_e(v)\big);$$

2. If u is before v in processor order, the commit time of u is before the commit time of v:

$$(u \xrightarrow{PO} v) \rightarrow \big(t_c(u) < t_c(v)\big);$$

3. If u is before v in execution order, the commit time of u is before the commit time of v:

$$(u \xrightarrow{E} v) \rightarrow \big(t_c(u) < t_c(v)\big);$$

4. If u is before v in time order, the commit time of u is before the enter time of v:

$$(u \xrightarrow{T} v) \rightarrow \big(t_c(u) < t_e(v)\big).$$

Let us carry out a brief discussion related to the above preconditions. Precondition 1 is reasonable since any processor (even an out-of-order processor) must fetch operations in order. Precondition 2 is also reasonable since operations in the same processor should be globally viewed in processor order. For example, for processor consistency, any load operation is allowed to perform with respect to any other processor if all previous load operations have been globally performed. As a consequence, if load operation r_1 is before load operation r_2 in processor order, r_1 must be globally viewed earlier than r_2. The idea behind Precondition 3 is that in multiprocessor systems with store atomicity, memory operations to the same location is serialized. One operation cannot perform before an earlier conflicting operation is globally viewed. Precondition 4 can be derived from the definition of time order. From the above discussion we know that the four preconditions are reasonable regardless of the concrete definitions of enter time and commit time in different microarchitectures. Finally, although store atomicity is not mandatory to a memory model, most memory models support it [1], hence the store atomicity utilized in the preconditions above is reasonable.

Based on the preconditions, we can analyze the relation between global order and time order. The global order between two operations implies that one operation observes the result of the other operation, while the time order between two operations implies that one operation executes before the other operation. Intuitively, in a correct design, if there are both global order and time order between two operations, global order and time order should be consistent: the former operation in time order must be observed by the latter operation in time order, which is called *time order restriction* on memory consistency. In multiprocessor systems with store atomicity, time order restriction can be derived from our preconditions.

Theorem 1 (Time Order Restriction Theorem) *In a multiprocessor system with store atomicity, time order restriction holds, i.e., the former operation in time order must be observed by the latter operation in time order. Formally,*

$$(v \xrightarrow{T} u) \rightarrow \neg(u \xrightarrow{GO} v). \tag{3}$$

Proof. According to the definition of time order, the theorem is equivalent to $(u \xrightarrow{GO} v) \rightarrow \big(t_e(u) < t_c(v)\big)$. Since $(u \xrightarrow{PO} v) \rightarrow \big(t_c(u) < t_c(v)\big)$ and $(u \xrightarrow{E} v) \rightarrow \big(t_c(u) < t_c(v)\big)$ both hold, by transitivity of partial order we obtain that $(u \xrightarrow{GO} v) \rightarrow \big(t_c(u) < t_c(v)\big)$. Hence, $(u \xrightarrow{GO} v) \rightarrow \big(t_e(u) < t_c(v)\big)$ holds. $\square$

3.3 Correctness Rules of VMC-time

Time order sets up a new restriction for the order of memory operations: two operations unordered globally may have time order. It is obvious that the verifications of memory consistency with and without time order information are quite different. In this subsection, we provide some novel correctness rules of memory consistency verification under time order.

To present the correctness rules, we should introduce the so-called *time global order* first. As we know, the time order restriction theorem guarantees that there is no contradiction between time order and traditional orders. As a consequence, we can combine time order and global order together, forming a new partial order: the time global order, which is the transitive closure of time order, processor order and execution order. Denote time global order as "$\xrightarrow{TGO}$". On the basis of time global order, we can now build a new type of graph for an execution, including not only processor order edges and execution order edges but also time order edges. We call this new type of graph the *TGO graph*.

Let $\mathscr{C}$ be the set of all time global cycles including operation u in the TGO graph. Furthermore, let $\mathcal{C}$ be a cycle belonging to $\mathscr{C}$ ($\mathcal{C} \in \mathscr{C}$), such that u is an operation in $\mathcal{C}$ ($u \in \mathcal{C}$). Intuitively, for any operation u, there are three kinds of cycles containing u: 1) all operations of the cycle except u are not in the pending period of u; 2) some operations of the cycle are not in the pending period of u, while other operations are in the pending period of u; and 3) all operations of the cycle are in the pending period of u.

The purpose of adding time order into memory consistency verification is to localize relations between operations. The key of localization is to detect the first kind of cycles in a localized manner, since the first kind of cycles refers to global orders outside the pending period.

Lemma 1 *Given a time global order cycle $\mathcal{C}$ containing operation u, if all operations in $\mathcal{C}$ except u are before u in time order, there must be a write operation w in cycle $\mathcal{C}$, which*

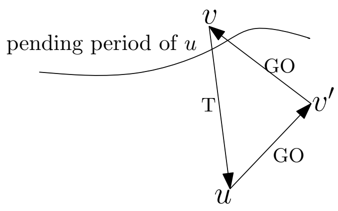


Figure 2. Violation of Rule 2: Orders of u, v and v' in a cycle

is after u in execution order. Formally,

$$\big(\forall v \in \mathcal{C} : (v \neq u) \rightarrow (v \xrightarrow{T} u)\big) \\ \rightarrow (\exists w \in \mathcal{C} : u \xrightarrow{E} w). \qquad (4)$$

Proof. Given that operation v' is the successor of u in cycle $\mathcal{C}$, there are three situations for us to consider: $u \xrightarrow{T} v'$, $u \xrightarrow{PO} v'$, and $u \xrightarrow{E} v'$. We know that all operations in $\mathcal{C}$ except u are before u in time order, therefore $u \xrightarrow{T} v'$ does not hold. Since $t_e(v')$ is before $t_e(u)$, according to Precondition 2, $u \xrightarrow{PO} v'$ does not hold. Hence, $u \xrightarrow{E} v'$ holds. Furthermore, if v' is a read operation, v' certainly cannot get the value of u from the future, and u cannot be before v' in execution order. Therefore v' is a write operation. Thus the theorem is proved. □

According to the previous theorem and lemma, we propose three correctness rules of VMC-time. Each correctness rule is related to one kind of cycle mentioned above.

Theorem 2 (Checking Rules Theorem) *There is no cycle in the TGO graph of the execution iff for any operation u of the execution, the following three correctness rules hold:*

Rule1: $\forall w \in O : (w \xrightarrow{T} u) \rightarrow \neg(u \xrightarrow{E} w);$
Rule2: $\forall v, v' \in O : (v \xrightarrow{T} u) \wedge (v' \xrightarrow{GO} v) \rightarrow \neg(u \xrightarrow{GO} v');$
Rule3: $\neg\Big(\exists \mathcal{C} \in \mathscr{C} : \big(\forall v \in \mathcal{C} : \neg(u \xrightarrow{T} v \vee v \xrightarrow{T} u)\big)\Big).$

Proof. "→". We assume that there is no cycle in the TGO graph of the execution. For Rule 1, given that a write operation w satisfies $w \xrightarrow{T} u$, $u \xrightarrow{E} w$ does not hold. Otherwise there will be a cycle $w \xrightarrow{T} u \xrightarrow{E} w$. For Rule 2, if operations u, v and v' satisfy $(v \xrightarrow{T} u) \wedge (v' \xrightarrow{GO} v)$, then $u \xrightarrow{GO} v'$ does not hold, otherwise there will be a cycle $v' \xrightarrow{GO} v \xrightarrow{T} u \xrightarrow{GO} v'$. Rule 3 is trivial: since there is no cycle in the whole graph, there certainly will be no cycle in the pending period of u. Hence "→" is proved.

"←". We use reduction to absurdity to prove it. Let us assume that Rules 1, 2 and 3 all hold, but there is a cycle $\mathcal{C}$. Let operation u be the last committed operation in the cycle. According to Rule 3, there must be some operation outside the pending period of u. We can travel $\mathcal{C}$ from u. Let v be the first operation before u in time order in traveling $\mathcal{C}$. Since u is the last committed operation in the cycle, $u \xrightarrow{T} v$ cannot hold. Instead, we have $v \xrightarrow{T} u$. If all operations except u in cycle $\mathcal{C}$ are before u in time order, according to Lemma 1, there must be some operation w such that $u \xrightarrow{E} w$, which contradicts Rule 1. Therefore there must be some operation in the pending period of u. Let v' be the precedent operation of v in $\mathcal{C}$. As shown in Figure 2, $u \xrightarrow{TGO} v'$ and $v' \xrightarrow{TGO} v$. Let the edge $a \xrightarrow{T} b$ be the first time order edge on the path from u to v in $\mathcal{C}$. According to the definition of time order, we obtain that $t_c(u) < t_c(a) < t_e(b) < t_c(b)$. However, since u is the operation committed last in the cycle, $t_c(u)$ cannot be before $t_c(b)$, and we reach a contradiction, thus there is no time order edge from u to v in $\mathcal{C}$. As a result, $u \xrightarrow{GO} v'$ and $v' \xrightarrow{GO} v$ hold. But $u \xrightarrow{GO} v' \xrightarrow{GO} v \xrightarrow{T} u$ contradicts Rule 2. Thus "←" is proved. □

In a real system, Rule 1 checks for the incorrect propagation of a write operation outside the pending period: a write operation does not correctly propagate to other processors because of bugs in the directory or network, and as a result, store atomicity is violated. To check Rule 1, we need to check whether the latest write before u in time order has propagated to u. Rule 2 focuses on ordering bugs between operations inside and outside of the pending period. To check Rule 2, we need to check all operations before u in global order to find cycles as shown in Figure 2. Rule 3 focuses on cycles inside the pending period.

3.4 Localization of Checking

In a multiprocessor system with store atomicity, a memory operation has been globally viewed when it is retired from internal structures of a single core. In practice, an operation cannot be unretired forever, otherwise there will be some deadlock or livelock. Therefore the number of operations in the pending period of one operation is bounded. Based on this property, we can localize memory consistency verification.

Theorem 3 (Localized Checking Theorem) *Under time order restriction, the checking for any operation needs to consider $O(p)$ operations, where p is the number of processors.*

Proof. Suppose that we are checking operation u. Rule 1 involves only a constant number of operations. To check Rule 2 and Rule 3 we need to travel through all operations in the pending period of u. Let T be the length of the pending period of u. If every processor can issue m instructions in one clock cycle, then the number of operations in the pending period of u is proportional to Tmp. T is independent of the number of operations n. Furthermore, since only few memory operations involve all processors, T can also be treated approximately independent of the number of processors p. Thus we can treat Tm as a constant C. As a result, to check operation u, we need to consider only Cp operations in a p-processor system. $\square$

The time order restriction theorem guarantees the soundness of memory consistency verification under time order restriction. The localized checking theorem ensures the completeness of checking rules for memory consistency verification under time order restriction. In the next section, we introduce a sound and complete memory consistency verification tool LCHECK which is based on the above theoretical work.

4 LCHECK

LCHECK is a verification tool for memory consistency based on time order restriction. It is used in the verification of the Godson-3 microprocessor.

There are four phases in the usage flow of LCHECK: test program generation phase, test program execution phase, execution graph construction phase, and cycle checking phase. In the test program generation phase, LCHECK generates a pseudo-random test program. In the test program execution phase, the generated test program is executed on the system to be verified. At run time, specified instructions save the logs of values received by each read operation into an internal RAM. A DMA engine transfers logs from the internal RAM to outside through IO such as HT (HyperTransport) link. In the execution graph construction phase, the LCHECK analyzer constructs the directed graph of the execution based on test programs and logs. In the cycle checking phase, the directed graph is checked to find cycles. The test program execution phase, execution graph construction phase, and cycle checking phase can be overlapped to check on the fly.

To reason about time order we need the enter times and commit times of all operations. However the precise time points of every operation are difficult to obtain in post-silicon verification. Fortunately, knowing the lower bound of enter time and the upper bound of commit time of each operations is enough for LCHECK to estimate time order, which is feasible in post-silicon verification. As shown in Figure 3, the upper bound of commit time of u is earlier than the lower bound of enter time of v, thus u is before v in time order. For any operation, the lower bound of its enter time can be the enter time of some operation before it in processor order, and the upper bound of its commit time can be the commit time of some operation after it in processor order. However, the lack of precision of enter and commit times slows down the verification, since the number of operations in the pending period of any operation increases.

In the Godson-3 microprocessor, there are two software visible registers per processor core: program_counter/enter_time of the last entered operation and program_counter/commit_time of the last committed operation. The test program reads values from these registers periodically, and writes them into internal RAM. Based on enter time and commit time of part of the operations, we can obtain the bounds of enter time and commit time of all operations.

4.1 Test Program Generation Phase

LCHECK uses a pseudo-random method to generate test programs. In the generated test programs, there are memory operations, synchronization operations, and arithmetic operations. There are some constraints on the generated test programs: Every write operation in the test programs has a certain value, and any two write operations to a same address have different write values. Following every read operation, there must be a dedicated instruction responsible for saving the value received by the read into internal RAM. Furthermore, for every 100 instructions, there is one dedicated instruction group responsible for reading enter time and commit time registers and saving them into internal RAM.

In the current version of LCHECK, branch operations are not supported, therefore there are no iteration in generated test programs. Thus the instruction flow of test programs is predefined. Nevertheless, we are currently working on supporting conditional branches in LCHECK test programs.

4.2 Test Program Execution Phase

The generated test program is executed on design under verification. At runtime the load values and the enter and commit times of operations are saved in the internal RAM. A DMA engine in the design can transfer the information from the internal RAM to outside through HT (HyperTrans-

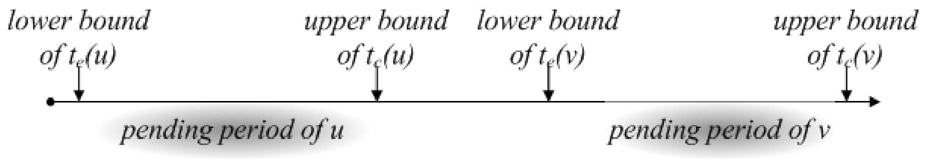


Figure 3. Upper bound and lower bound of u and v

port) link. Note that the DMA engine and internal RAM of the Godson-3 were designed for signal processing applications. LCHECK conveniently utilizes them for verification.

4.3 Execution Graph Construction Phase

LCHECK constructs a TGO graph for an execution, and checks the graph for cycles. Each node in the graph includes not only edge information, but also the lower bound of enter time and the upper bound of commit time of the operation. Since time order can be derived from bounds of time points, LCHECK does not explicitly construct time order edges. Thus we only need to add global order edges between operations without time order. The detailed edge rules in the basic LCHECK are listed below.

Processor Edge Rule: Add an edge between one operation and its predecessor in processor order based on memory consistency.

Execution Edge Rule: If read operation r gets a value from write operation w, then add an edge from w to r.

Observed Edge Rule: If read operation r gets a value from write operation w, and w' is the last write operation to the same address as r which precedes r in program order, then add an observed edge from w' to w.

Inferred Edge Rule 1: If w and r access the same address, read operation r gets a value from write operation w', and w' is in the pending period of w, and $w \xrightarrow{GO} r$ (inferring based on global order) or $w \xrightarrow{T} r$ (inferring based on time order), then add an inferred edge from w to w'.

Inferred Edge Rule 2: If w and w' access the same address, read operation r gets a value from write operation w, and w' is in the pending period of r, and $w \xrightarrow{GO} w'$ (inferring based on global order) or $w \xrightarrow{T} w'$ (inferring based on time order), then add an inferred edge from r to w'.

Processor edges can be obtained from the test program and memory consistency model. Moreover, execution edges are added per read operation: for a read operation r, load values of r can be known from logs, store values of each write operation are also determined at test program generation phase, thus we can find the write operation w corresponding to r and add an execution edge from w to r. Observed edges are derived from processor edges and execution edges based on observed edge rules.

Since the addition of edges by the processor edge rule, the execution edge rule and the observed edge rule has linear complexity, we only discuss the algorithm for inferred edge rules in this paper. In many memory consistency verification tools, inferring edges is the most time-consuming part [24, 29]. However, in LCHECK the complexity of inferring edges is linear because inferred edges are limited to be inside the pending period: if an inferred edge complies with time order, we do not need to infer it again; if an inferred edge contradicts time order, we can find the violation by Rule 1 (of Theorem 2) in the cycle checking phase (see bug example 1 in Section 5).

As it is shown in Algorithm 1, if an edge from u to v is added, function $infer\_edge$ is called to find other inferred edges. Based on time order, adding edges from u to v affects only global order relations starting from operations in the pending period of u. Therefore LCHECK considers all write operations w in the pending period of u (if u is a write operation, then u must also be considered). For all read operations r in the pending period of w, if r is after w in global order, r accesses the same address as w, and LCHECK has not inferred new edges based on $w \xrightarrow{GO} r$, LCHECK infers new edges based on $w \xrightarrow{GO} r$ with Inferred Edge Rule 1 and calls function $set\_inferred$ to record that we have tried to infer new edges based on $w \xrightarrow{GO} r$. For all write operations w' in the pending period of w, if w' is after w in global order, w' accesses the same address as w, and LCHECK has not inferred new edges based on $w \xrightarrow{GO} w'$, then LCHECK infers new edges baseds on $w \xrightarrow{GO} w'$ with Inferred Edge Rule 2 and calls function $set\_inferred$ to record that we have tried to infer new edges based on $w \xrightarrow{GO} w'$. If r is outside the pending period of w', we need to check only whether the inferred edge complies with time order.

4.4 Cycle Checking Phase

In the cycle checking phase, LCHECK checks for cycles in the graph. As shown in Algorithm 2, checking for cycles in LCHECK is based on the checking rules theorem (Theorem 2). For every operation u, we first check Rule 1 (of Theorem 2), i.e., we check whether the latest write operation w accessing the same address as u before u in time order propagates to u. If u gets a value from w' and

Algorithm 1. Algorithm for Inferring Edge

```
int infer_edge($u$, $v$) begin
  for(all write operations $w$ in the pending period of $u$) begin
    for(all read operations $r$ in the pending period of $w$) begin
      if($w \xrightarrow{GO} r$ and !inferred($w$,$r$) and address($w$)==address($r$)) begin
        let $w'$ = the write operation which $r$ gets a value from;
        if($w' \xrightarrow{T} w$) panic(); if($w \xrightarrow{T} w'$) continue;
        if(there is no edge from $w$ to $w'$) begin
          add_edge($w$, $w'$); infer_edge($w$, $w'$);
        end
        set_inferred($w$,$r$);
      end(**Inferred Edge Rule 1 Based on Global Order**)
    end
    for(all write operations $w'$ in the pending period of $w$) begin
      if($w \xrightarrow{GO} w'$ and !inferred($w$,$w'$) and address($w$)==address($w'$))
        for(all read operations $r$ which get value from $w$) begin
          if($w' \xrightarrow{T} r$) panic(); if($r \xrightarrow{T} w'$) continue;
          if(there is no edge from $r$ to $w'$) begin
            add_edge($r$, $w'$); infer_edge($r$, $w'$);
          end
        end
        set_inferred($w$,$w'$);
      end(**Inferred Edge Rule 2 Based on Global Order**)
    end
  end
end
```

$w \xrightarrow{GO} w' \vee w \xrightarrow{T} w'$, then there is a bug. Checking Rules 2 and 3 (of Theorem 2) can be achieved by traveling global order edges from u. If we travel to some operation before u in time order, Rule 2 is violated. If we travel back to u, Rule 3 is violated. If we travel to some operation v after u in time order, the related cycle will be considered in checking v. However, some edges cannot be found by inferred rules [24], thus the basic LCHECK is incomplete.

4.5 Complete LCHECK

To completely verify memory consistency, we need to confirm the total write order for each memory location to know all execution orders. However, some execution orders cannot be inferred [24]. For completeness, we implement backtracking on top of the basic incomplete LCHECK. For two conflict write operations without determined order, the complete LCHECK makes an arbitrary decision about the execution order of these two operations, and infers as many new edges as possible, then checks for cycles in the graph. If there is a violation, the complete LCHECK backtracks to the nearest arbitrary decision and tries the other branch of the decision. The backtracking algorithm is based on searching the frontier graph [8].

Algorithm 2. Algorithm for Checking for Cycles

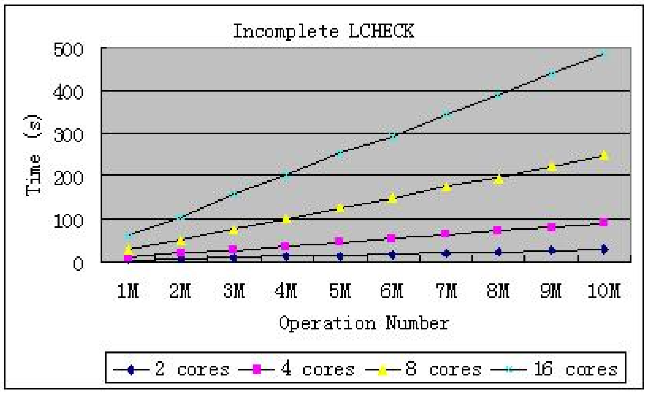
```
int check_cycle() begin
  for(all operation $u$) begin
    if($w \xrightarrow{GO} w' \vee w \xrightarrow{T} w'$) panic();
    travel all $v$ which have the edge starting from $u$
      if($v \xrightarrow{T} u \vee v == u$) panic();
  end
end
```

w is the latest entered write operation accessing the same address as u and before u in time order, w' is the write operation which u gets a value from.

4.6 Complexity of LCHECK

Since LCHECK adds inferred edges only between two operations with overlapping pending periods, there are at most $O(p)$ inferred edges related to one operation. As a result, there are less than $O(pn)$ inferred edges in all. Thus the function $infer\_edge$ can be called $O(pn)$ times and each time requires time complexity of $O(p^2)$. Hence, the time complexity of constructing a graph in LCHECK is $O(p^3n)$. Since there are only $O(pn)$ edges in the graph, the time complexity of checking for cycles in LCHECK is $O(pn)$. Therefore the time complexity of basic incomplete LCHECK is $O(p^3n)$.

The complete LCHECK needs to backtrack to traverse



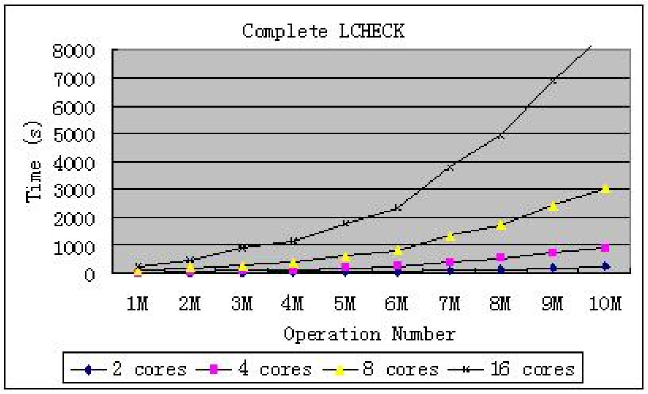


Figure 4. Analysis Time of LCHECK

the frontier graph. Under time order restriction, we can treat the entire execution as having n/Cp intervals, and each interval has $(Cp/p)^p$ frontiers. Thus there are $O((n/Cp) \times (Cp/p)^p)$ frontiers for an execution in all. Each backtracking step requires a time complexity of $O(p^3 n)$. As a result, the time complexity of complete LCHECK is $O((n/Cp) \times (Cp/p)^p \times p^3 n)$, i.e., $O(C^p p^2 n^2)$.

5 Experiments

Figure 4 shows time cost of both incomplete and complete LCHECK with different numbers of operations. The experimental environment is AMD athlon64 3200+ with 64GB memory. In Figure 4, the x-axis represents the number of operations, and the y-axis represents the running time of the checker in seconds. From experimental results, we can observe that the runtime of the incomplete LCHECK analyzer is linear with respect to the number of operations. Although the complete LCHECK is slower than the incomplete LCHECK, the complete LCHECK can still cope with a test program with 10 million memory operations. The reason is that, on each backtracking, only a few number of operations and order must be considered.

As part of the Godson project, LCHECK found many bugs in the Godson-3. Here we provide two examples. The first example, which is the first bug caught by LCHECK, is shown in Figure 5.a. There are 3 processors P_1, P_2, and P_3. w_1 is before w_2 in time order, and w_2 is before r_1 in time order. However, the invalidation of w_2 is not sent to P_3 correctly because of an error in the network. Thus r_1 gets a stale copy of A and reads the old value 1 written by w_1. Although this error seems very simple, many other memory consistency verification tools such as [24, 29] cannot find this error since there is no more information to derive the global order between w_1 and w_2. But LCHECK can detect the error. Rule 1 (of Theorem 2) is obeyed because r_1 gets its value from w_1, which is not the latest conflict write operation before r_1 in time order. Hence the error is caught by executing Line 3 of the program in Algorithm 2.

Another bug example is shown in Figure 5.b. There are 3 processors P_1, P_2, and P_3. The original value of memory address A is 0. w_1 is before r_2 in time order, and r_2 gets its value from w_2. According to Inferred Edge Rule 1, an inferred edge from w_1 to w_2 (dashed line in Figure 5.b) is added. According to the inferred edge, a cycle of $r_1 \to w_1 \to w_2 \to w_3 \to r_1$ violates Rule 3 (of Theorem 2), and is caught by executing Line 5 of the program in Algorithm 2. After careful debugging, we concluded that this bug is caused by an error in the load store queue, which makes w_3 precede w_2.

6 Conclusion and Future Work

The verification of memory consistency in general is known to be NP-hard. If we want to find an efficient and complete checking algorithm, the most effective solution is to look for some restriction on memory access orders. We have observed that there exists a natural restriction to most multiprocessor systems: the size of temporary internal structures of each processor is limited. As a consequence, at any moment only a bounded number of operations are executing. Inspired by this observation, we introduce the concepts of a pending period and a new natural partial order called time order. Based on the time order restriction, related operations are localized. Consequently, inferring related edges and checking related cycles are also localized for any operation. As a result, the time complexity of memory consistency verification can be significantly reduced.

On the basis of our theoretical investigations, we have introduced LCHECK, a memory consistency verification tool for CMP. LCHECK can verify a number of memory consistency models, including sequential consistency, processor consistency, weak consistency and release consistency. LCHECK requires the memory system to support

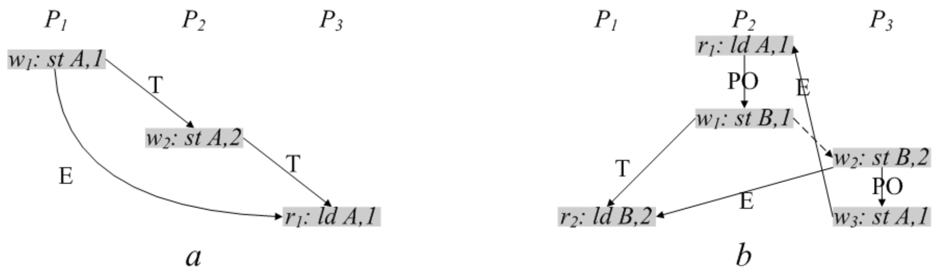


Figure 5. Bug Examples

store atomicity. However, LCHECK does not need certified store atomicity as a precondition, since it can find violations of both memory consistency and store atomicity. LCHECK only needs simple hardware support made of several software visible registers per processor, which have been implemented in an industrial CMP. To solve the VMC-time-read problem, LCHECK has the time complexity of $O(C^p p^2 n^2)$ for soundly and completely checking, and $O(p^3 n)$ for soundly but incompletely checking. Our method has been used in the validation of an industrial CMP, Godson-3, and it has found many bugs. Both theoretical and practical results demonstrate the effectiveness and efficiency of our approach.

However, LCHECK is not the end of the road for memory consistency verification tools relying on time order. There are still many research avenues to explore. First, we need a global clock to determine time order between operations. However, some multiprocessors may not have a global clock. How to establish the time order of operations without a physical global clock remains an open question. Second, LCHECK still needs some hardware support. Many processors have programmer-accessible event counters which can count the number of committed instructions and keep track of local time. Using these existing counters together with the knowledge of the instruction window size to find looser bounds on the enter and commit times seems attractive in post-silicon verification.

Acknowledgment

We are grateful to Michel Dubois and to the anonymous reviewers for their helpful comments and insightful suggestions. We also thank Timothy Pinkston for his help in improving the readability of the paper.

References

[1] Arvind and J. Maessen. "Memory Model = Instruction Reordering + Store Atomicity". In *Proceedings of the 33st International Symposium on Computer Architecture (ISCA'06)*, 2006.

[2] H. Cain and M. Lipasti. "Verifying Sequential Consistency Using Vector Clocks". In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, 2002.

[3] J. Cantin, M. Lipasti, and J. Smith. "The Complexity of Verifying Memory Coherence". In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures (SPAA'03)*, 2003.

[4] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan. "Shared Memory Consistency Protocol Verification Against Weak Memory Models: Refinement via Model-Checking". In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, 2002.

[5] W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, 1992.

[6] M. Dubois, C. Scheurich, and F. Briggs. "Memory Access Buffering in Multiprocessors". In *Proceedings of the 13rd International Symposium on Computer Architecture (ISCA'86)*, 1986.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multi Processors". In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA'90)*, 1990.

[8] P. Gibbons and E. Korach. "On Testing Cache-Coherent Shared Memories". In *Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures (SPAA'94)*, 1994.

[9] P. Gibbons and E. Korach. "Testing Shared Memories". *SIAM Journal on Computing*, Vol. 26, No. 4, pp. 1208-1244, 1997.

[10] J. Goodman. "Cache Consistency And Sequential Consistency ". *Technical Report No. 61*, SCI committee, 1989.

[11] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. "QB or not QB: An Efficient Execution Verification Tool for Memory Orderings". In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)*, 2004.

[12] S. Hangal, D. Vahia, C. Manovit, J. Lu, and S. Narayanan. "Tsotool: A Program for Verifying Memory Systems Using the Memory Consistency Model". In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA'04)*, 2004.

[13] M. Herlihy and J. Wing. "Linearizability: a Correctness Condition for Concurrent Object". *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, pp. 463-492, 1990.

[14] W. Hu, W. Shi, and Z. Tang. "Framework of Memory Consistency Models". *Journal of Computer Science and Technology*, Vol. 13, No. 2, pp. 110-124, 1998.

[15] W. Hu and P. Xia. "Out-of-order Execution in Sequentially Consistent Shared-memory Systems: Theory and Experiments". *Journal of Computer Science and Technology*, Vol. 13, No. 2, pp. 125-139, 1998.

[16] W. Hu. *Shared Memory Architecture*. Doctoral dissertation, 2001.

[17] W. Hu, J. Wang, X. Gao, and Y. Chen. "Microarchitecture of Godson-3 Multi-Core Processor". In *Proceedings of the 20th Hot Chips*, 2008.

[18] W. Hu, J. Wang, X. Gao, Y. Chen, and Q. Liu. "Microarchitecture of Godson-3 Multi-Core Processor". *IEEE Micro*, Vol. 29, No. 2, 2009.

[19] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". *Communications of the ACM*, Vol. 21, No. 7, pp. 558-565, 1978.

[20] A. Landin, E. Hagersten, and S. Haridi. "Race-free Interconnection Networks and Multiprocessor Consistency". In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA'91)*, 1991.

[21] S. Lu, J. Tucek, F. Qin, and Y. Zhou. "AVIO: detecting atomicity violations via access interleaving invariants". In *Proceedings of the 12nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, 2006.

[22] J. Ludden, W. Roesner, G. Heiling, J. Reysa, J. Jackson, B. Chu, M. Behm, J. Baumgartner, R. Peterson, J. Abdulhafiz, W. Bucy, J. Klaus, D. Klema, T. Le, F. Lewis, P. Milling, L. McConville, B. Nelson, V. Paruthi, T. Pouarz, A. Romonosky, J. Stuecheli, K. Thompson, D. Victor, and B. Wile. "Functional Verification of the POWER4 Microprocessor and POWER4 Multiprocessor Systems". *IBM Journal of Research and Development*, Vol. 46, No. 1, 2002.

[23] C. Manovit and S. Hangal. "Efficient Algorithms for Verifying Memory Consistency". In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architecure (SPAA'05)*, 2005.

[24] C. Manovit and S. Hangal. "Completely Verifying Memory Consistency of Test Program Executions". In *Proceedings of the 12nd International Symposium on High-Performance Computer Architecture (HPCA'06)*, 2006.

[25] A. Meixner and D. Sorin. "Dynamic Verification of Sequential Consistency". In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*, 2005.

[26] A. Meixner and D. Sorin. "Dynamic Verification of Memory Consistency in Cache-coherent Multithreaded Computer Architectures". In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, 2006.

[27] S. Park and D. Dill. "An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order)". In *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, 1995.

[28] M. Plakal, D. Sorin, A. Condon, and M. Hill. "Lamport Clocks: Verifying a Directory Cache-Coherence Protocol". In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA'98)*, 1998.

[29] A. Roy, S. Zeisset, C. Fleckenstein, and J. Huang. "Fast and Generalized Polynomial Time Memory Consistency Verification". In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, 2006.

[30] C. Scheurich and M. Dubois. "Correct Memory Operation of Cached-Based Multiprocessors". In *Proceedings of the 14th International Symposium on Computer Architecture (ISCA'87)*, 1987.

[31] D. Victor, J. Ludden, R. Peterson, B. Nelson, W. Sharp, J. Hsu, B. Chu, M. Behm, R. Gott, A. Romonosky, and S. Farago. "Functional Verification of the POWER5 Microprocessor and POWER5 Multiprocessor Systems". *IBM Journal of Research and Development*, Vol. 49, No. 4, pp. 541-553, 2005.

[32] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. "Nemos: a Framework for Axiomatic and Executable Specifications of Memory Consistency Models". In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.