Shared memory consistency protocol verification against weak memory models: refinement via model-checking*

Prosenjit Chatterjee, Hemanthkumar Sivaraj and Ganesh Gopalakrishnan
http://www.cs.utah.edu/formal_verification/cav02.html
{prosen| hemanth| ganesh}@cs.utah.edu

School of Computing, University of Utah

Abstract. *Weak shared memory consistency models,* especially those used by modern microprocessor families, are quite complex. The bus and/or directory-based protocols that help realize shared memory multiprocessors using these microprocessors are also exceedingly complex. Thus, the *correctness problem* – that all the executions generated by the multiprocessor for any given concurrent program are also allowed by the memory model – is a major challenge. In this paper, we present a formal approach to verify protocol implementation models against weak shared memory models through automatable *refinement checking* supported by a *model checker*. We define a taxonomy of weak shared memory models that includes most published commercial memory models, and detail how our approach applies over all these models. In our approach, the designer follows a prescribed procedure to build a highly simplified intermediate abstraction for the given implementation. The intermediate abstraction and the implementation are concurrently run using a model-checker, checking for refinement. The intermediate abstraction can be proved correct against the memory model specification using theorem proving. We have verified four different Alpha as well as Itanium memory model implementations[1] against their respective specifications. The results are encouraging in terms of the uniformity of the procedure, the high degree of automation, acceptable run-times, and empirically observed bug-hunting efficacy. The use of parallel model-checking, based on a version of the parallel Murφ model checker we have recently developed for the MPI library, has been essential to finish the search in a matter of a few hours.

1 Introduction

Modern weak shared memory consistency models [1–4] allow subtle reorderings among *load*s and *store*s falling into multiple storage classes to permit aggressive compiler optimizations, hide load latencies, and maintain I/O semantics as

* This work was supported by NSF Grants CCR-9987516 and CCR-0081406

[1] We designed both these protocols - one with a split-transaction bus and one with Scheurich's optimization - as there are no public domain Itanium protocols as far as we know.

well as legacy compatibility. Since these specifications are the basis for many generations of microprocessors, manufacturers are committed to formal methods for specifying them. Unfortunately, simple and intuitive formal specification methods that apply across a wide range of weak memory models are yet to be developed. In this paper, we address this problem and propose a parameterizable operational model that can cover a wide range of modern weak memory models. The bus and/or directory-based protocols that help realize shared memory multiprocessors using modern microprocessors are also exceedingly complex. Thus, in addition to the specification problem, formal verification of shared memory consistency protocols against weak shared memory models remains largely unsolved [2, 5]. Most reported successes have been either for far simpler memory models such as cache coherence or sequential consistency (e.g., [6–9]), or approaches that took advantage of existing architectural test program suites and created finite-state abstractions for them [10].

In this paper, we present a methodology that systematically applies to a wide spectrum of weak architectural memory consistency models. Basically, we instantiate our parameterizable operational model to obtain a finite-state approximation to the weak memory model of interest. We then take the finite-state model of the protocol under verification, and subject these models to 'lockstep' execution, to check for a simulation (refinement) relation: whether an event that can be fired on the interface of the implementation can be accepted by the specification. The execution happens within the explicit-state enumeration model-checker Murφ that we have recently ported to run on our in-house Network Testbed[2] using the MPI library. We demonstrate our results on four different Alpha as well as Itanium memory model implementations against their respective specifications. The results are encouraging in terms of reduced effort, potential for reuse of models, the degree of automation, run-times, and bug-hunting efficacy (it found many subtle coding errors). To the best of our knowledge, no other group has hitherto verified this variety of protocols against modern weak memory models.

Approach to verification: Instead of using a model-checker to verify a temporal logic formula, we use it to check for the existence of a refinement mapping between an implementation model and an abstract model. While operational models are well suited for this purpose, if they are non-deterministic, an inefficient backtracking search would be needed. To illustrate this difficulty, consider one source of internal non-determinism - namely *local bypassing* (otherwise known as *read forwarding*). Local bypassing allows a *load* to pick its value straight out of the store buffer, as opposed to allowing the store to post globally, and then reading the global store. Consider the operational-style specification of a memory model, such as the one illustrated in Figure 1(b) (the details of this figure are not important for our illustration). Consider one such model M_1, and make an *identical copy* calling it M_2. Let the external events of M_1 and M_2 (its alphabet) be *load* and *store*. Certainly we expect M_2 to refine M_1. However, both M_2 and M_1 can have different executions starting off with a common prefix, signifying non-determinism. For instance, if P1 runs program $store(a, 1)$; $load(a)$, and P2

[2] http://www.emulab.net

runs program $load(a)$, one execution of M_1 obtained by annotating *load*s with the returned values is
$\mathsf{Exec1} = P1 : store(a, 1);\ P1 : load(a, 1);\ P2 : load(a, 1)$
while an execution of the M_2 is
$\mathsf{Exec2} = P1 : store(a, 1);\ P1 : load(a, 1);\ P2 : load(a, \top)$,
where $\top$ is the initial value of a memory location. Note that Exec2 exercises the bypass option while Exec1 didn't do so. However such bypass events are invisible in high-level operational models that employ only the external *load*s and *store*s. Therefore, in the above example, even though the *load* values disagreed, we cannot throw an error, but must backtrack and search another internal execution path. However, by enriching the alphabet of our example to $\Sigma^{'} = \{store_{p1}(a, 1), store_g(a, 1), load(a, 1), load(a, \top\}$, where $store_{p1}$ refers to the store being visible to P1 and $store_g$ refers to the store being visible globally (these being internal events), Exec1 and Exec2 can be elaborated into Exec1' and Exec2', and these executions do not disagree on *load* after a common prefix:
$\mathsf{Exec1'} = P1 : store_{p1}(a, 1);\ P1 : load(a, 1);\ store_g(a, 1);\ P2 : load(a, 1)$,
$\mathsf{Exec2'} = P1 : store_{p1}(a, 1);\ P1 : load(a, 1);\ P2 : load(a, \top);\ store_g(a, 1)$.
In general, there are many sources to non-determinism than just local bypassing, and all of them must be determinized, essentially resulting in a situation where each *load* is associated with a *unique* past store event. In Section 6, we sketch how this approach can be applied to a wide range of weak memory models. In Section 2, we illustrate the visibility order approach on the Alpha memory model. The use of internal events in writing specifications is, fortunately, already accepted in practice (e.g., [2, 3, 11]). Most approaches in this area specify the so called *visibility order* of executions in terms of the enriched alphabet.
Creating Imp, Imp_{abs}, and the $Spec$ models: In our approach, verification is divided into two distinct phases. First, an intermediate abstraction Imp_{abs} which highly simplifies the implementation Imp is created, and Imp is verified against it through model-checking. Next, Imp_{abs} is verified against *Spec*, the visibility-order based specification of the memory model. We believe (as we will demonstrate) that Phase 1 can in itself be used as a very effective bug-hunting tool. Phase 2 can be conducted using theorem-proving, similar to [12], as detailed on our webpage. This paper is mostly about Phase 1. For a large class of implementations, Phase 2 does not vary, as the same Imp_{abs} results from all these implementations, thus permitting verification reuse. In fact, most Imp_{abs} models we end up creating are the same as *operational style* models, such as the UltraSparc operational model of [13] or the Itanium operational model [4]. We also expect the designer who creates Imp to be annotating it with internal events. However, since such annotations are designer assertions as to when (they think) the internal events are happening, it should be something a designer who has studied the visibility order *Spec* must be able to do.

The creation of Imp_{abs} is based on the following observation. Most protocol implementations possess two distinct partitions: a small *internal* partition containing the *load* and *store* buffers, and a *much larger external* partition. containing the *cache*, *main memory*, and the (multiple) buses and directories (see

Section 3 for an illustration). For a wide spectrum of memory models, Imp_{abs} is obtained by retaining the internal partition and replacing the external partition by a highly simplified operational structure which, in most cases, is a single-port memory (see Section 4 for an illustration). This approach also enables consistent annotation of the internal and external partitions of Imp and Imp_{abs} with events from the enriched alphabet. Another significant advantage is that we can *share* the internal partition during model-checking, explained as follows.

Why sharing the internal partition is possible: A specification state is a pair $\langle spec_int_part, spec_ext_part \rangle$ and an implementation state is a pair $\langle imp_int_part, imp_ext_part \rangle$. Typically, all the 'int_part's and 'ext_part's are bit-vectors of hundreds of bits. Let $\langle i_i, i_e \rangle$ be an implementation state and let $\langle s_i, s_e \rangle$ be the corresponding specification state (starting with the respective initial states). The state vector maintained during reachability is $\langle i_i, i_e, s_i, s_e \rangle$. We then select an eligible external event e from the enriched external alphabet and perform it on the implementation, advancing the verification state to $\langle i_i^{'}, i_e^{'}, s_i, s_e \rangle$. This state is *not* stored. If the same event e cannot be performed on the specification, an error-trace is generated; else, it is performed and the state is advanced to $\langle i_i^{'}, i_e^{'}, s_i^{'}, s_e^{'} \rangle$. Since we can retain the same internal partition, $i_i^{'}$ and $s_i^{'}$ are always the same - and so we can share their bits, reducing the state vector to $\langle i_i^{'}, i_e^{'}, s_e^{'} \rangle$. To sum up, since we do not multiply out states, the number of states generated during reachability is the *same* as the number of reachable states in the implementation model, and the state-vector size grows only by s_e.

Handling Protocols where the Temporal and Logical Orders differ: In many aggressive protocols, the *logical* order of events (the "explanation") is different from the *temporal* order in which the protocol performs these events. Consider an optimization described by Scheurich in the context of an invalidation-based directory protocol. In the unoptimized version, a store request sent to the directory causes invalidations to be sent to each read-shared copy. The store can proceed only after the invalidations have been performed. Under the optimization, read-sharers merely queue the invalidations, sending "fake" acknowledgements back to the directory, and perform the invalidations only later. Thus, even after a processor P_1 writes new data to the line, a *ld* from some other processor P_2 to the same cache line can read stale data. Thus, in the logical order, the new stores must be situated after the loads, even though in temporal order, the store is done before the loads on the stale lines. Such issues are not addressed in most prior work. The creation of the intermediate abstraction Imp_{abs} helps us partition our concerns [14]. Details appear on our webpage.

Handling protocols with large state-spaces: Shared memory consistency protocols can easily have several billions of states, with global dependencies caused by pointers and directory entries that defy compact representation using BDDs. We find the use of a parallel model-checker almost essential to make things practical. In some cases, we aborted runs that took more than 55 hours (due to thrashing) on sequential model-checkers on machines with 1GB memory, when they finished in less than a few hours on our parallel model-checker.

Summary of Results: We applied our method to an implementation of the

Alpha processor [15] that was modeled after a multiprocessor using the Compaq (DEC) Alpha 21264 microprocessor. The cache coherence protocol is a Gigaplane-like split transaction bus [16] protocol. We also verify an Alpha implementation with an underlying cache coherence protocol using multiple interleaved buses, modeled after the Sun $Ultra^{TM}$ $Enterprise^{TM}$ 10000 [17]. Both these implementations were verified with and without Scheurich's optimization. These four Alpha processor protocols finished in anywhere between 54 to 240 minutes on 16 processors, visiting up to 250 million states. The diameter of the reachability graph (indicating the highest degree of attainable parallelism if one cuts the graph along its diameter and distributes it) was in excess of 5,000. The highest numbers reported in [18] using their *original* Parallel $Mur\varphi$ on the Stanford FLASH as well as the SCI protocols were around 1 million states and a diameter of 50. While designer insight is required in selecting the external partition, the effort is not *case* specific, but instead *class* specific. As shown in Section 6, we can taxonomize memory models into four categories, and once and for all develop external partitions for each branch of the taxonomy. Designer insight is required in attaching events to the abstract model. The "final property" verified in our approach is quite simple: to reiterate, it is that the loads completing in the implementation and specification models return the same data. We therefore think that our method has the right ingredients for being scaled up towards considerably more aggressive protocols - including directory protocols.
Related Work: See [1] for a survey and [5] for a recent workshop. We showed how to port Collier's architectural testing work [19] to model-checking [10] and extend Collier's work to weak memory models [20]. In [21], event sequences generated by protocol implementations are verified by a much simpler trustworthy protocol processor. In [22, 23], shared memory consistency models are described in an operational style. In [6], sequential consistency verification, including parameterized model-checking is addressed. To our knowledge, we are the first to verify eight different protocols against two different weak memory models using a uniform approach. While we model "only" two processors, memory locations, as well as data values, we end-up getting trillions of transitions. We believe that before we can attempt parameterized verification, we must conquer the complexity of these "small" instance verifications. Weak memory models for Java are also under active study [24, 25].

2 Alpha memory model specification

A concurrent shared memory Alpha program is viewed as a set of sequences of instructions, one sequence per processor. Each sequence captures *program order* at that processor. An *execution* obtained by running the shared memory program is similar to the program itself, except that each *load(a)* now becomes *load(a,return_value)*. Every instruction in an execution can be decomposed into one or two[3] *events* (*local* and *global* in the latter case; in the former case, we shall use the words 'instruction' and 'event' synonymously). Each event t is

[3] In general, as discussed later, there could be more events.

defined as a tuple (p, l, o, a, d) where $p(t)$ is the processor in whose program t originates from, $l(t)$ is the label of instruction t in p's program, $o(t)$ is the event type (load/store/etc.), $a(t)$ is the memory address, and $d(t)$ the data. All instructions except *st* are decomposed into exactly one event. Each *st* is decomposed into a st_{local} and a st_{global}. An *execution* obeys Alpha memory model if all the memory events of the *execution* form at least one total order which obeys the *Per Processor Order* stipulated by the memory model, and the *Read Value* Rule stipulated by the memory model. In addition, the st_{global} events must form a total order. (Note that this total order may not respect program order of *st* instructions.) The *Read Value Rule* specifies the data value to be returned by the load events in an execution. The *Per Processor Order* respects both program order as well as data dependence. The fact that st_{global} operations form a single total order is modeled by generating only 'one copy' of a st_{global} event corresponding to each *st* instruction, and situating the st_{global} events in the total order '$\rightarrow$'. Since Alpha allows local bypassing, we split any store instruction t into two events t^{local} and t^{global} (and also create the corresponding tuples) where $o(t^{local})$=st_{local} and $o(t^{global})$=st_{global}. More specifically, an execution satisfies the Alpha memory model if there exists a logical total order '$\rightarrow$' of all the *ld*, st_{local} and st_{global} events and memory fence events present in the execution, such that '$\rightarrow$' satisfies the following clauses:

1. *Per Processor Order*: Let t_1 and t_2 be two events s.t $p(t_1) = p(t_2)$, $l(t_1) < l(t_2)$ (t_1 appears earlier in program order than t_2).
   (a) If $a(t_1) = a(t_2)$ and,
      i. $o(t_1) = st_{local}$, $o(t_2) = ld$, or
      ii. $o(t_1) = ld$, $o(t_2) = st_{local}$, or
      iii. $o(t_1) = st_{local}$, $o(t_2) = st_{local}$,
      iv. $o(t_1) = ld$, $o(t_2) = ld$, or
      v. $o(t_1) = st_{global}$, $o(t_2) = st_{global}$

      then $t_1 \rightarrow t_2$.
   (b) If there exists a fence(MB) instruction t_f s.t. $l(t_1) < l(t_f) < l(t_2)$ then $t_1 \rightarrow t_2$.
2. *Read Value*: This definition follows the style in which *Read Value* is defined in [15] for TSO. Formally, let t_1 be a load (*ld*) event. Then the data value of t_1 is the data value of the most recent local event, if present; if not, it is the most recent global store event (in the total order relation $\rightarrow$) to the same memory location as t_1. *i.e.*,
   (a) if
      i. $p(t_1) = p(t_2), a(t_1) = a(t_2)$, $t_2^{local} \rightarrow t_1 \rightarrow t_2^{global}$ and
      ii. there does not exist a *st* instruction t_3 s.t $p(t_1) = p(t_3)$, $a(t_1) = a(t_3)$ and $t_2^{local} \rightarrow t_3^{local} \rightarrow t_1$.

      then $d(t_1) = d(t_2^{local})$;
   (b) else if
      i. $a(t_1) = a(t_2)$, $t_2^{global} \rightarrow t_1$ and
      ii. there does not exist a *st* instruction t_3 s.t $a(t_1) = a(t_3)$ and $t_2^{global} \rightarrow t_3^{global} \rightarrow t_1$.

then $d(t_1) = d(t_2^{global})$;
(c) else, $d(t_1)$ is the "initial memory value" (taken to be $\top$ in our paper).

3 Alpha implementation model

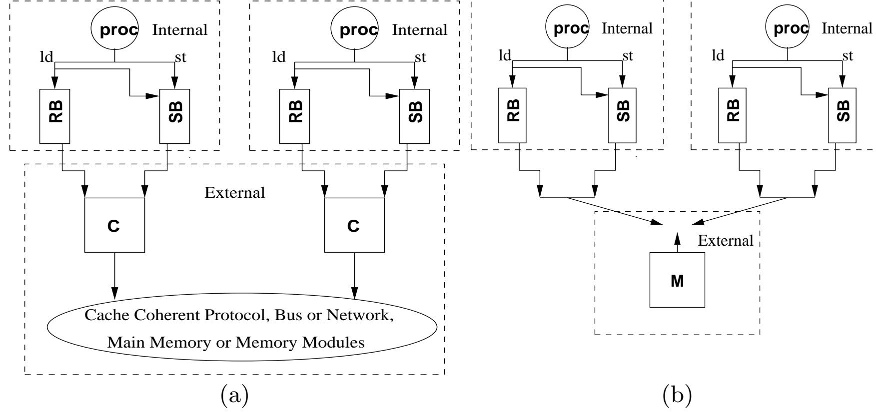


Fig. 1. (a) The Alpha Implementation model, and (b) The Alpha Intermediate abstraction

In the Alpha implementation of each processor is separated from its cache (situated in the external partition) with a coalescing re-order store buffer SB and a re-order read buffer RB (situated in the internal partition). Caches are kept coherent with a write-invalidate coherence protocol [11]. The data structure of caches is a two dimensional array C where, for event t, $C[p(t)][a(t)].a$ refers to data value of address $a(t)$ at processor $p(t)$, and $C[p(t)][a(t)].st$ refers to its address state (A-state)[4]. We begin with a brief explanation of our memory consistency protocol. This protocol is the same as the one used in [16] to describe a Gigaplane-like split-transaction bus. Memory blocks may be cached Invalid(I), Shared(S), or Exclusive(E). The A-state (address state) records how the block is cached and is used for responding to subsequent bus transactions. The protocol seeks to maintain the expected invariants (e.g, a block is Exclusive in at most one cache) and provides the usual coherent transactions: Get-Shared (GETS), Get-Exclusive (GETX), Upgrade (UPG, for upgrading the block from Shared to Exclusive) and Writeback (WB). As with the Gigaplane, coherence transactions immediately change the A-state, regardless of when the data arrives. If a processor issues a GETX transaction and then sees a GETS transaction for the same block by another processor, the processor's A-state for the block will go from Invalid to Exclusive to Shared, regardless of when it obtains the data. The processor issues all instructions in program order. Below, we specify exactly

[4] We overload the selectors ".a" and ".st" for notational brevity.

what happens when the processor issues one of these instructions.

1. *st*: A *st* instruction first gets issued to coalescing re-order buffer SB, completing the st_{local} event. Entries in SB are the size of cache lines. Stores to the same cache line are coalesced in the same entry and if two stores write to the same word, the corresponding entry will hold the value written by the store that was issued later. Entries are eventually deleted (flushed) from SB to the cache, although not necessarily in the order in which they were issued to the write buffer. Before deleting, the processor first makes sure there is no earlier issued *ld* instruction to the same address pending in RB (if any, those RB instructions must be completed before deleting that entry from SB). It then checks if the corresponding block's A-state is Exclusive(E). If not, the coherence protocol is invoked to change the A-state to E. Once in E state, the entry is deleted from SB and written into the cache atomically, thus completing the st_{global} event.

2. *ld*: To issue a *ld* instruction, the processor first checks in its SB for a *st* instruction to the same word. If there is one, the *ld* gets its value from it. If there is no such word, the processor buffers the *ld* in RB. In future, when an address is in E or S state, all *ld* entries to that same address in RB gets its data from cache and are then deleted from the buffer. *ld* entries to different words in RB can be deleted in any relative order. There is no overlap between the issuing of *ld*s and the flushing of *st*s to the same address once E state is obtained.

3. *MB*: Upon issuing a *MB* instruction, all entries in SB are flushed to the cache and all entries in RB are deleted after returning their values from cache, hence completing the corresponding *MB* event[5]. While flushing an entry from SB, the processor checks that there is no earlier issued *ld* instruction to the same address residing in RB. We call this entire process as $flush_{imp}$.

4 The Intermediate Abstraction

The Alpha abstract model retains the internal data partition of the implementation *without any changes*. However, the cache, the cache coherent protocol, bus and main memory in the implementation which belong to the external partition are all replaced by a single port main memory M in the abstract model. This replacement follows the rules of the thumb we have presented in Section 6 for dealing with memory models obeying write atomicity (as is the case with the Alpha model). We now take a look at how each of the instructions get implemented. As with the implementation, the processor issues all instructions in program order.

1. *st*: A *st* instruction first gets issued to SB just as in the implementation, completing the st_{local} event. At any time, an entry anywhere in SB can be deleted from the buffer and written to the single port memory M atomically, provided there is no earlier issued *ld* instruction to that address pending in RB. This completes the st_{global} event.

2. *ld*: Similarly, as in implementation, a *ld* instruction tries to hit SB and on

[5] Appropriate cache entries need to be in E state before flushing

a miss, it gets buffered in RB. However, any entry in RB can be deleted once it receives its data from M, both the steps being performed in one atomic step. Entries to same address get their data values from M at the same time.
3. *MB*: Upon issuing a MB instruction, all entries in SB are flushed to M and all entries in RB are deleted after returning their values from M. While flushing from SB the processor checks that there is no earlier issued load event to the same address residing in RB. We call this entire process as $flush_{abstract}$.

5 Model-checking based Refinement

The events $st_{local}, st_{global}, ld$ and MB have been defined for both the implementation and the abstract model. Every event of the implementation is composed

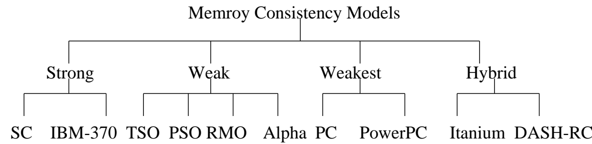
Event	Implementation	Operational Model
ld(t) $(SB_{p(t)} hit)$	read from $SB_{p(t)}$	read from $SB_{p(t)}$
ld(t) $(SB_{p(t)} miss)$	Issue to $RB_{p(t)}$; $C[p(t)][a(t)].st$=S or E and $d(t) \leftarrow C[p(t)][a(t)].a$	Issue to $RB_{p(t)}$; $d(t) \leftarrow M[a(t)]$
$st_{local}(t)$	Issue$(SB_{p(t)}, t)$	Issue$(SB_{p(t)}, t)$
$st_{global}(t)$	$C[p(t)][a(t)].st$=E and $C[p(t)][a(t)].a \leftarrow d(t)$	$M[a(t)] \leftarrow d(t)$
$MB(t)$	$flush_{imp}$	$flush_{abstract}$

Table 1. Completion steps of all events of implementation and abstract model

of multiple steps. However, in the abstract model each event except ld is composed of a single atomic step. For example, for a st_{global} event to complete, if the concerned address's A-state is Invalid, the processor will need to send a request on the bus to get an Exclusive copy. During this process many intermediate steps take place which include other processors and main memory reacting to the request. However, if a miss occurs while handling the st_{global} event in the abstract model, the entry in SB can be deleted and atomically written to single port memory.

Synchronization scheme: The discovery of the synchronization sequences between the implementation and the specification is the crux of our verification method. Table 1 provides an overview of the overall synchronization scheme. This table compares the completion steps of both the implementation and the abstract model, and highlights all synchronization points. Let us briefly elaborate the actions taken for ld entry in RB to complete. In the implementation,

[5] Here $d(t) \leftarrow C[p(t)].[a(t)].a$ refers to the load instruction t receiving its data from the updated cache entry



Memory Model	Splitting of store instructions	External Partition
Strong	store unsplit	single port memory
Weak	store split to local and global	single port memory
Weakest	store split to local and $(p+1)^6$globals	memory and re-order buffer per processor
Hybrid	store split to local and $(p+1)$ globals	memory and re-order buffer per processor

Fig. 2. (a) Memory model classes, and (b) Splitting of store and external partition for each class

coherence actions are first invoked to promote the cache line into an Exclusive or Shared state. Thereafter, the implementation receives data from the bus and at this point completes the ld event. At this point, the model-checker will immediately make the same event complete in the abstract model by simply returning the data from $M[a(t)]$ through the multiplexor switch. Synchronization happens if the same datum is returned. In general, the last step that completes any event in the implementation and the single step that completes the same event in the abstract model are performed atomically.

The synchronization scheme for instructions that may get buffered and get completed later are slightly more elaborate. Basically, synchronization must be performed both when the instruction is entered into the buffer and later when they complete. For example, since a ld instruction may miss the SB and hence

	Alpha Implementation			Itanium Implementation		
Cache Coherent Protocol	States ($\times 10^6$)	Transitions ($\times 10^6$)	Time (hrs)	States ($\times 10^6$)	Transitions ($\times 10^6$)	Time (hrs)
Split Trans. Bus	64.16	470.52	0.95	111.59	985.43	1.75
Split Trans. Bus with Scheurich's Opt.	251.92	1794.96	3.42	325.66	2769.77	4.80
Multiple Interleaved Buses	255.93	1820.38	3.65	773.27	2686.89	10.97
Multiple Interleaved Buses with Scheurich's Opt.	278.02	1946.67	3.90	927.31	3402.41	12.07

Table 2. Experimental Results

may not complete immediately, we will have to synchronize both the models when ld gets buffered, and finally synchronize again when the ld event completes. The synchronization of MB is accomplished indirectly, by the already existing synchronizations between the models at ld or st_{global}. This is because an MB completes when the above instructions occurring before it complete. Our experimental results are summarized in Table 2.

6 Creation of intermediate abstractions, Imp_{abs}

In our verification methodology, the abstract model always retains, without change, the internal partition of the implementation. However, the *external* partition is considerably simplified. Designer insight is required in the selection of a simplified external partition, as this depends on the memory model under examination. In this section we categorize memory models into four classes and show how a common external partition can be derived for memory models belonging to a particular memory model class, thus providing a systematic approach to deriving the abstract model. The four classes of memory models are as follows:

Strong: requires *Write atomicity* and does not allow local bypassing. (e.g. Sequential Consistency,IBM-370).

Weak: requires *Write atomicity* and allows local bypassing (e.g. Ultra Sparc TSO,PSO and RMO,Alpha)

Weakest: : does not require *Write atomicity* and allows local bypassing (e.g. PowerPC,PC).

Hybrid: : supports weak load and store instructions that come under memory model *Weakest* and also support strong load and store instructions that come under *Strong* or *Weak* memory model classes (e.g. Itanium, DASH-RC).

Depending upon the category a memory model falls under, we split a store instruction into one or more events. Load instructions for any memory model can always be treated as a single event. Here are a few examples of splitting events. In case of Sequential Consistency, we do not split even the stores as sequential consistency demands a single global total order of the loads and stores. For a weak memory model such as the Ultra Sparc TSO, we split the store instruction into two events, a local store event (which means that the store is only visible to the processor who issued it) and a global event (which means that the store event is visible to all processors). Since the *Weakest*[7] category of memory models lack write atomicity, we need to split stores into $p+1$ events, where p is number of processors, thus ending up with a local store event and p global events (global event i would mean that the store event is visible to processor i). Figure 2(b) summarizes these splitting decisions for various memory models. It also shows the nature of the external partition chosen for various memory models.

---

[6] p is number of processors

[7] Note that the abstract models for *weakest* memory models can also be used as abstract models for strong models. For example, the Lazy Caching protocol of Gerth [9] can be used as an abstract model for sequential consistency.

In case of *Strong* and *Weak* memory models, the external partition is just a single port memory M. The intuition behind having M is that both these classes of memory models require *Write Atomicity* and hence a store instruction should be visible to all processors instantaneously. *Weakest* and *Hybrid* memory models require more involved data structures where each processor i has its own memory M_i and also a re-ordering buffer that takes in incoming store instructions posted by different processors including itself from their SB. Store instructions residing in this buffer eventually get flushed to memory. The combination of M_i and a re-ordering buffer simulates a processor seeing store instructions at different times and different relative order as that of another processor. An algorithm that generates the correct external partition given a memory model has been designed. A remotely executable web-based tool is also available for experimenting with the operational models thus generated.

7 Conclusions

In this paper, we presented a uniform verification methodology that applies across a spectrum of architectural memory consistency models and handles a variety of consistency protocols. We report experiments involving eight different protocols and two different weak memory models. Our approach fits today's design flow where aggressive, performance oriented protocols are first designed by expert designers, and handed over to verification engineers. The verification engineer can follow a systematic method for deriving an abstract reference model. Our approach does not require special training to use, and can benefit from the use of multiple high-performance PCs to conduct parallel/distributed model checking, thereby covering large state spaces. In ongoing work, we are verifying directory based implementations for the Alpha and Itanium memory models. We are also working on several optimizations to speed-up model checking as well as exploring alternatives to model-checking.

References

1. Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.
2. http://research.microsoft.com/users/lamport/tla/wildfire-challenge.html.
3. Gil Neiger, 2001. http://www.cs.utah.edu/mpv/papers/neiger/fmcad2001.pdf.
4. Prosenjit Chatterjee and Ganesh Gopalakrishnan. Towards a formal model of shared memory consistency for intel itanium. In *ICCD*, pages 515–518, 2001.
5. Mpv: Workshop on specification and verification of shared memory systems, 2001. http://www.cs.utah.edu/mpv/.
6. Thomas Henzinger, Shaz Qadeer, and Sriram Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *CAV, LNCS 1633*, pages 301–315, 1999.
7. Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. Technical report, SRC, December 2001. Research Report 176.

8. Anne Condon and Alan J. Hu. Automatable verification of sequential consistency. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, July 2001.
9. Michael Merritt. Guest editorial: Special issue on shared memory systems. *Distributed Computing*, 12(12):55–56, 1999.
10. Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem, and Ganesh Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In *CAV, LNCS 1427*, pages 464–476, 1998.
11. D. Sorin et.al. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. Technical Report #1412, CS Department, U. Wisconsin, Madison, March 2000.
12. Seungjoon Park. *Computer Assisted Analysis of Multiprocessor Memory Systems.* PhD thesis, Stanford University, jun 1996. Department of Computer Science.
13. David L. Weaver and Tom Germond. *The SPARC Architecture Manual – Version 9.* P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.
14. Prosenjit Chatterjee. Formal specification and verification of memory consistency models of shared memory multiprocessors. Master's thesis, Univ Utah, School of Computing, 2002.
15. Anne Condon, Mark Hill, Manoj Plakal, and David Sorin. Using lamport clocks to reason about relaxed memory models. In *Proceedings of HPCA-5*, January 1999.
16. A.Singhal et.al. Gigaplane: A high performance bus for large smps. In *Proc. 4th Annual Symp on High Performance Interconnects, Stanford University*, pages 41–52, 1996.
17. The Ultra Enterprise 10000 Server,
http://www.sun.com/servers/highend/10000/
18. Ulrich Stern and David Dill. Parallelizing the Murϕ verifier. *Formal Methods in System Design*, 18(2):117–129, 2001. (Journal version of their CAV 1997 paper).
19. W. W. Collier. *Reasoning About Parallel Architectures.* Prentice-Hall, Englewood Cliffs, NJ, 1992.
20. Rajnish Ghughal and Ganesh Gopalakrishnann. Verification methods for weaker shared memory consistency models. In José Rolim et al. (Eds.), editor, *Proc. FMPPTA*, pages 985–992, May 2000. LNCS 1800.
21. Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Dynamic verification of cache coherence protocol. In *?*, June 2001. Workshop on Memory Performance Issues, in conjunction with ISCA.
22. David L. Dill, Seungjoon Park, and Andreas Nowatzyk. Formal specification of abstract memory models. In Gaetano Borriello and Carl Ebeling, editors, *Research on Integrated Systems*, pages 38–52. MIT Press, 1993.
23. P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching in TLA. *Distributed Computing*, 1997.
24. Jeremy Manson and William Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, June 2001.
25. Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Formalizing the java memory model for multithreaded program correctness and optimization. Technical Report UUCS-02-011, University of Utah, School of Computing, 2002. Also at http://www.cs.utah.edu/˜yyang/research.