Address Translation Aware Memory Consistency

Computer systems with virtual memory are susceptible to design bugs and runtime faults in their address translation systems. Detecting bugs and faults requires a clear specification of correct behavior. A new framework for address translation aware memory consistency models addresses this need.

Bogdan F. Romanescu
Alvin R. Lebeck
Daniel J. Sorin
Duke University

We expect computers to function correctly, despite potential problems like design bugs and physical faults. The consequences of incorrect functionality include silent data corruptions and crashes. The goal of dependable computing is to reduce the probabilities of these events at the lowest cost in terms of performance loss, hardware, power consumption, and design and verification time. Here, we focus on detecting incorrect behavior caused by design bugs or physical faults before it leads to a data corruption or crash.

To detect incorrect behavior, we must first precisely specify what constitutes correct behavior. For a processor core, the instruction set architecture (ISA) specifies the exact semantics of every instruction in the instruction set. The ISA also specifies the architecture's memory consistency model,[1] which defines the legal software-visible orderings of loads and stores performed by multiple threads. After Lamport introduced the first memory consistency model (sequential consistency[2]), it became easier to statically or dynamically verify a memory system. Lamport's key contribution to verification was a precise, microarchitecture-independent specification of correct memory system behavior.

Nevertheless, modern memory systems still have many bugs. We identified 21 bugs related to address translation in published errata (see Table 1 for a small subset of these).[3-5] And we believe that one underlying cause of these bugs is a tendency to oversimplify memory consistency. Memory consistency isn't just one monolithic interface between the hardware and the software, as Figure 1 shows. Rather, it's a set of interfaces between the hardware and various levels of software, as in Figure 2. Although Adve and Gharachorloo explained memory consistency's multilevel nature,[1] architects don't always adopt this more comprehensive definition of memory consistency. For example, when we teach students about memory consistency models, we generally don't specify whether that model refers to virtual or physical addresses.

In this article, we develop a framework for specifying two critical levels of memory consistency: the physical address memory consistency (PAMC) model and the virtual address memory consistency (VAMC) model, which define the behavior of operations (loads, stores, and memory barriers) on physical and virtual addresses, respectively. As part of this specification framework, we discuss and formalize address translation's crucial

0272-1732/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

Table 1. A small sample of published address translation bugs.

Chip	Bug	Effect
AMD Athlon64/Opteron[6]	TLB (translation look-aside buffer) flush filter can cause coherency problem in multicore systems.	Unpredictable system failure (possible use of stale translations).
	INVLPG instruction with address prefix doesn't correctly invalidate the translation requested.	Unpredictable system behavior (possible use of stale translations).
Intel Core Duo[7]	One core updates a page table entry while the other core uses the same translation entry, which could lead to unexpected behavior.	Unexpected processor behavior.
	Updating a PTE (page table entry) by changing read/write, U/S or P bits without TLB shootdown could cause unexpected processor behavior.	Unexpected processor behavior.

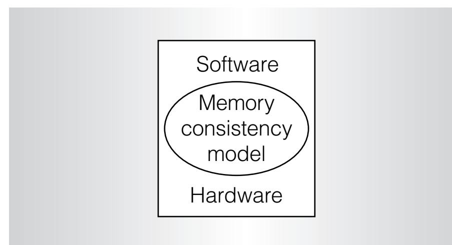

Figure 1. Address translation oblivious memory consistency. In this simplistic view of consistency, there is a single interface between software and hardware.

role in supporting a VAMC model. Without correct address translation, a system with virtual memory can't enforce any VAMC model. Furthermore, we develop address translation-aware specifications of memory consistency not only to benefit design and verification teams, but also to enable architects to design runtime checkers that detect incorrect behaviors.

Specifying address translation aware memory consistency

A memory consistency specification provides two important functions: it serves as a contract between the system and the programmer, and it provides the formal framework necessary for verifying correct memory system operation, either statistically or dynamically.

Levels of memory consistency

A computer system presents memory interfaces (consistency models) at multiple levels, as Figure 2 shows. We position hardware below all levels, because it provides mechanisms that we can use to enforce consistency models at various levels (for example, the processor provides in-order instruction commit). We consider four levels relevant to programmers. At each level, the consistency model defines the legal orderings of the memory operations available at that level. These consistency models are necessary interfaces that are included in the specifications of the instruction set architecture (ISA), application binary interface (ABI), and application programming interface (API). Here, we discuss these levels, starting at the lowest level.

- PAMC: Unmapped software, including the boot code and part of the system software that manages address translation, relies on PAMC. Implementing PAMC is the hardware's responsibility and, as such, is specified precisely in the architectural manual.
- VAMC: All mapped software relies on VAMC, including mapped system software. VAMC builds upon PAMC and requires support from address translation software and the hardware.
- User process memory consistency (UPMC): UPMC could be identical to VAMC, or it could differ, as in the case of software transactional memory or software distributed shared memory.
- High-level language consistency: User-level programmers see the consistency model specified by the high-level

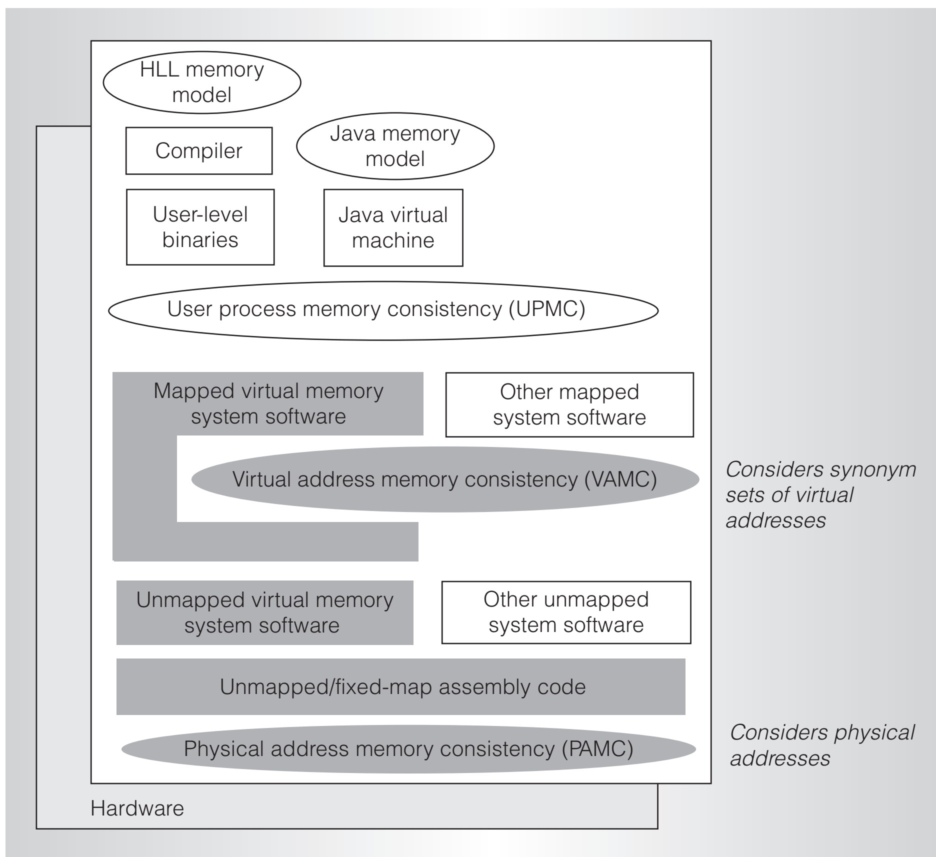

Figure 2. Address translation aware memory consistency. In this richer view of consistency, there are multiple interfaces for supporting different types of software. This article focuses on the shaded portions of this figure.

language, such as Java's consistency model.[8] These high-level language consistency models are supported by the compilers, runtime systems, and lower level consistency models.

Existing consistency models—such as sequential consistency, processor consistency, weak ordering, and release consistency—don't distinguish between virtual and physical addresses. Lamport's original definition of sequential consistency is typical in that it specifies a total order of operations (loads and stores). But it doesn't specify whether the loads and stores are to virtual or physical addresses. We refer to these consistency models as being *address translation oblivious*.

We specifically focus on PAMC and VAMC and the hardware and software involved in supporting them.

Specifying PAMC

We can fairly easily adapt an address translation oblivious consistency model as PAMC's specification. For example, the PAMC model could be sequential consistency, in which case the interface would specify that there must exist a total order of all loads and stores to physical addresses that respects each thread's program order, and that each load's value is equal to the value of the most recent store to that physical address in the total order.

We specify consistency models using a table-based scheme like those of Hill et al.[9] and Arvind and Maessen.[10] The table specifies which program orderings are enforced by the consistency model. Thus, Tables 2 and 3 define adaptations of sequential consistency and weak ordering for PAMC, respectively. Some consistency models have atomicity

Table 2. Sequential consistency for physical address memory consistency (PAMC).

Op 1 \ Op 2	Load	Store
Load	X	X
Store	X	X

*Loads and stores are to physical addresses. An "X" denotes an enforced ordering.

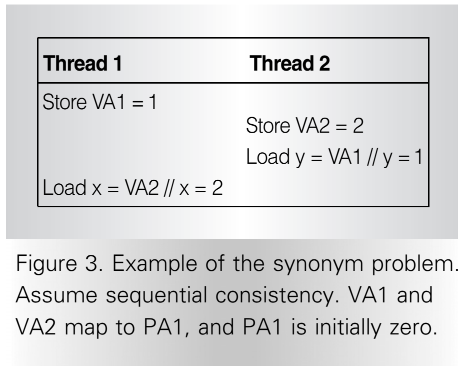

Figure 3. Example of the synonym problem. Assume sequential consistency. VA1 and VA2 map to PA1, and PA1 is initially zero.

Table 3. Weak ordering for PAMC.

Op 1 \ Op 2	Load	Store	MemoryBarrier
Load		A	X
Store	A	A	X
MemoryBarrier	X	X	X

*Loads and stores are to physical addresses. An "X" denotes an enforced ordering; an "A" denotes an ordering that's enforced if the operations are to the same physical address. Empty entries denote no ordering.

constraints that can't be expressed with just a table (for example, stores are atomic). We can specify these models by augmenting the table with a specification of atomicity requirements, as previous researchers have done.[10]

Specifying VAMC

Although adapting an address translation oblivious consistency model for PAMC is straightforward, three challenges arise when adapting an address translation oblivious consistency model for VAMC: synonyms, mapping and permission changes, and load and store side effects.

Synonyms. Synonyms are multiple virtual addresses (VAs) that map to the same physical address (PA). Consider the example in Figure 3, in which VA1 and VA2 map to PA1. Sequential consistency requires a total order in which a load's value equals the value of the most recent store to the same address. Unfortunately, naively applying sequential consistency at the VAMC level allows an execution in which $x = 2$ and $y = 1$. The programmer expects that the loads in both threads will be assigned the value of the most recent update to PA1. However, a definition of VAMC that didn't consider address translation would allow x to receive the most recent value of VA2 and y to receive the most recent value of VA1, without considering that they both map to PA1. To overcome this challenge, we reformulate address translation oblivious consistency models for VAMC by applying the model to *synonym sets of (virtual) addresses* rather than individual addresses. For example, we define sequential consistency for VAMC as follows: there must be a total order of all loads and stores to virtual addresses that respects program order and in which each load gets the value of the most recent store to *any virtual address in the same virtual address synonym set.* We can make similar modifications to adapt other address translation oblivious consistency models for VAMC.

Programmers that use synonyms generally expect ordering to be maintained between accesses to synonymous virtual addresses. Incorporating synonyms explicitly in the consistency model enables programmers to reason about the ordering of accesses to virtual addresses.

Explicitly stating the ordering constraints of synonyms is necessary for verification. An error in the address translation hardware could result in a violation of ordering among synonyms that might not be detected without the formal specification.

Mapping and permission changes. There is a richer set of memory operations at the VAMC level than at the PAMC level. User-level and system-level programmers at the VAMC interface are provided with OS software routines to map and remap

Buggy Code

Thread1	Thread2

MRF {map VA1 to PA2;
    tlbie VA1; // invalidate
            // translation
            // (VA1→PA1)
}	

sync; // memory barrier for
      // regular memory ops	

Store VA2 = B	
sync	while (VA2!=B) {spin}
	sync
	Store VA1 = C
	sync
	Store VA2 = D
while (VA2 != D) {spin}	
sync	
Load VA1 // can get C or A	

Correct Code

Thread1	Thread2

MRF {map VA1 to PA2;
    tlbie VA1; // invalidate
            // translation
            // (VA1→PA1)
}	

tlbsync // fence for MRF	

sync; // memory barrier for
      // regular memory ops	

Store VA2 = B	
sync	while (VA2!=B) {spin}
	sync
	Store VA1 = C
	sync
	Store VA2 = D
while (VA2 != D) {spin}	
sync	
Load VA1 // can only get C	

Figure 4. Power instruction set architecture (ISA) code snippets illustrate the need to consider the ordering of map/remap functions (MRF). Initially, VA1 maps to PA1, and the value of PA1 is A.

or change permissions on virtual memory regions, such as the `mk_pte()` (make new page table entry) or `pte_mkread()` (make page table entry readable) functions in Linux 2.6. We call these software routines map/remap functions (MRFs).

The code snippet in the left side of Figure 4 is written for a system implementing the Power ISA. The code snippet illustrates the need to consider MRFs and their ordering. We expect that the load by Thread1 should return the value C written by Thread2, because that appears to be the value of the most recent write (in causal order, according to the Power ISA's weak-ordered memory model). However, this code snippet doesn't guarantee when the `tlbie` (translation look-aside buffer invalidate entry) instruction will be observed by Thread2—thus, Thread2 could continue to operate with the old translation of VA1 to PA1. Therefore, Thread2's store to VA1 could modify PA1. When Thread1 performs its load to VA1, it could access PA2 and thus obtain B's old value.

The problem with the code is that it doesn't guarantee that the invalidation generated by the `tlbie` instruction will execute on Thread2's core before Thread2's store to VA1 accesses its translation in its TLB. Understanding only the PAMC model isn't sufficient for the programmer to reason about this code's behavior. The programmer must also understand how MRFs are ordered. We show a corrected version of the code on the right side of Figure 4. In this code, Thread1 executes a `tlbsync` (translation look-aside buffer synchronize) instruction that's effectively a fence for the MRF. Specifically, the `tlbsync` guarantees that other cores have observed the `tlbie` instruction executed by Thread1.

A runtime hardware error or design bug could cause a TLB invalidation to be dropped or delayed, resulting in TLB incoherence. A formal specification of MRF orderings is required to develop proper verification techniques, and PAMC is insufficient for this purpose.

Load/store side effects. The third challenge in specifying VAMC is that loads and stores to virtual addresses have side effects. The address translation system includes status

```
Store VA1=1;// VA1 maps to PA1
Load VA2;// VA2 maps to the page table entry of VA1
```

Figure 5. Code snippet illustrating the need to consider side effects. The load is used by the VM system to determine if the page mapped by VA1 needs to be written back to secondary storage. If reordered, a Dirty bit set by the store could be missed and the page incorrectly not written back.

Table 4. Sequential consistency for VAMC.

		Operation 2				
		Ld	Ld-sb	St	St-sb	MRF
Operation 1	Ld	X	X	X	X	X
	Ld-sb	X	X	X	X	X
	St	X	X	X	X	X
	St-sb	X	X	X	X	X
	MRF	X	X	X	X	X

*Loads and stores are to synonym sets of virtual addresses. An "X" denotes an enforced ordering.

bits (such as Accessed and Dirty bits) for each page table entry. These status bits are part of the architectural state. Thus, the ordering of updates to those bits must be specified in VAMC. We add two new operations to the specification tables: `Ld-sb` and `St-sb` (load and store impact on status bits).

Consider the example in Figure 5. Without knowing how status updates are ordered, the operating system can't be sure what state will be visible in these bits. It's possible that the load of the PTE (page table entry) occurs before the first store's Dirty bit update. The OS could incorrectly determine that a write-back was unnecessary, resulting in data loss.

Without a precise specification of status bit ordering, verification could miss a situation analogous to the software example we've discussed (see Figure 5). A physical fault could lead to an error in the ordering of setting a status bit, and this error could be overlooked by dynamic verification hardware and could lead to silent data corruption.

Putting it all together. Table 4 presents a VAMC adaptation of sequential consistency. This specification includes MRFs and status bit updates, and loads and stores apply to synonym sets of virtual addresses (not individual virtual addresses). This specification provides both a contract for programmers and enables development of techniques to verify correct memory system operation.

Gap between PAMC and VAMC

One of our research areas is dynamic verification of VAMC (for more information, see the "Dynamic verification of VAMC" sidebar). Because we haven't yet discovered an efficient scheme for direct dynamic verification of VAMC, we instead dynamically verify VAMC by dynamically verifying PAMC (using an existing scheme) as well as the gap between PAMC and VAMC. This gap is the address translation system.

Specification of address translation

We present a framework for specifying address translation systems.

Address translation assumptions

We restrict our discussion to page-based address translation systems. A translation is a tuple <mapping (VP, PP), permissions,

status>, where the mapping converts the virtual page to a physical page. The physical page, permissions, and status information are specified by the page table entry (PTE) defining the translation. The permission bits include whether the user or kernel owns the page and whether the page is readable, writeable, or executable. The status bits denote whether the page has been accessed or is dirty. The status bits are atomically updated in the TLB and in the page table in memory. In an architecture with hardware-managed TLBs, the hardware is responsible for eventually updating the status bits. If the TLBs are software-managed, then status bit updates occur in exception handlers.

To create, modify, or delete a translation or to modify a translation's permission bits, the kernel performs an MRF. An MRF typically has four activities (see Figure 6). Some of the activities in an MRF require the software or hardware to perform complicated actions. For example, delivering the TLB invalidations could require an interprocessor interrupt or a global TLB invalidation instruction that relies on hardware for distributing the invalidations.

```
generic MRF{
        acquire page table lock(s);
        create/modify the translation;
        send TLB invalidations to other cores;
        release page table lock(s);
}
```

Figure 6. Pseudocode for a generic map/remap function (MRF).

A provably sufficient address translation model

We present a model of an address translation system that, when combined with $PAMC_{SC}$ (see Table 2), is provably sufficient for providing $VAMC_{SC}$ (see Table 4). This address translation model, which we call AT_{SC}, is similar to current Linux platforms. AT_{SC} is restrictive and conservative, but it's also realistic.

AT_{SC}: A sequential address translation model. AT_{SC} is a sequential model of an address translation system. It's a logical abstraction that encompasses the behaviors of various possible physical implementations. This model has three key aspects:

- MRFs logically occur instantaneously and are thus totally ordered with respect to regular loads and stores and other address translation operations. (Recent operating systems, such as Linux 2.4.16-2.6.24, relax this AT_{SC} constraint by instead postponing memory accesses that depend on the translation modified by the MRF.) Linux enforces this aspect of the model using locks.
- A load or store logically occurs instantaneously and simultaneously with its corresponding translation access (accessing the mapping, permissions, and status) and possible status bit updates. A core can adhere to this aspect of the model in many ways, such as by snooping TLB invalidations between when a load or store executes and when it commits. A snoop hit forces the load or store to be squashed and reexecuted.
- A store atomically updates all the values in the synonym set cached by the core executing the store, and a coherence invalidation atomically invalidates all of the values in the synonym set cached by the core receiving the invalidation. To our knowledge, current systems adhere to this aspect either by using physical caches or by using virtual caches with the same index mapping of synonym set virtual addresses.

We now show that AT_{SC} is the bridge between $PAMC_{SC}$ and $VAMC_{SC}$.

$PAMC_{SC}+AT_{SC}\rightarrow VAMC_{SC}$. $PAMC_{SC}$ specifies that all loads and stores using physical addresses are totally ordered. AT_{SC} specifies that a translation access occurs instantaneously and simultaneously with the load or store. Under AT_{SC}, all MRFs are totally ordered with respect to each other and with respect to loads and stores. AT_{SC} also specifies that accesses to synonyms are ordered according to $PAMC_{SC}$ (for example, via the use of physical caches). Therefore, all loads and stores using virtual addresses are totally ordered. Finally, AT_{SC} specifies that status bit

Dynamic verification of VAMC

One of our primary goals in specifying virtual address memory consistency (VAMC) was to be able to dynamically verify it. Prior work developed schemes for dynamic verification of PAMC,[1,2] and we initially attempted to develop an analogous scheme for VAMC. Due to implementation costs, we changed our approach. Rather than directly verify VAMC, we factored it into its constituent parts, PAMC and address translation, and dynamically verified them instead. Because schemes already exist for PAMC, we focused on dynamic verification of address translation.

To dynamically verify AT_{SC} (a sequential model of an address translation system)—which we call $DVAT_{SC}$—we must dynamically verify both AT_{SC} invariants: page table integrity and translation mapping coherence.

- *Checking page table integrity.* PT-SubInv1 is an invariant that's maintained by software. Fundamentally, no hardware solution can completely check this invariant, because the hardware doesn't have semantic knowledge of the software's goal. One existing solution to this problem is self-checking code;[3] we defer further research in this area to future work. To check that PT-SubInv2 is maintained, we can adopt any previously proposed dynamic verification scheme for PAMC.
- *Checking translation coherence.* $DVAT_{SC}$ aims to dynamically verify the three translation coherence subinvariants. Because we've specified these subinvariants in terms of tokens, we can dynamically verify them by adapting the TCSC scheme,[4] which was previously used to dynamically verify token-based cache coherence. TCSC's key insight is that cache coherence states can be logically represented with token counts that can be periodically checked; this same insight applies to translation coherence. In a correctly operating AT_{SC} system, the exchanges of logical tokens will obey AT_{SC}'s three coherence subinvariants. $DVAT_{SC}$ therefore checks these three subinvariants at runtime.

We've implemented and experimentally evaluated $DVAT_{SC}$. Error injection results show that $DVAT_{SC}$ detects the errors that mimic all 21 published address translation bugs. Performance results show that the performance impact of $DVAT_{SC}$ is less than 2 percent.

References

1. K. Chen, S. Malik, and P. Patra, ''Runtime Validation of Memory Ordering Using Constraint Graph Checking,'' *Proc. 13th Int'l Symp. High-Performance Computer Architecture,* IEEE Press, 2008, pp. 415-426.
2. A. Meixner and D.J. Sorin, ''Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures,'' *Proc. Int'l Conf. Dependable Systems and Networks,* IEEE Press, 2006, pp. 73-82.
3. M. Blum and S. Kannan, ''Designing Programs that Check Their Work,'' *Proc. 21st Ann. ACM Symp. Theory of Computing,* ACM Press, 1989, pp. 86-97.
4. A. Meixner and D.J. Sorin, ''Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures,'' *Proc. 12th Int'l Symp. High-Performance Computer Architecture,* IEEE Press, 2007, pp. 145-156.

updates are performed simultaneously with the corresponding load or store, and thus status bit updates are totally ordered with respect to all other operations. Thus, $PAMC_{SC}$ plus AT_{SC} results in $VAMC_{SC}$, where ordering is enforced between all operations.

A framework for specifying address translation models

AT_{SC} is just one possible model for address translation and thus one possible bridge from a PAMC model to a VAMC model. We present a framework for specifying address translation models. A precisely specified address translation model helps verify the address translation system and, in turn, VAMC. We haven't yet proved the sufficiency of address translation models other than AT_{SC} (that is, that they bridge any particular gap between PAMC and VAMC); we leave such proofs for future work. Our framework consists of two invariants that are enforced by a combination of hardware and privileged software: the page table is correct, and translations are coherent.

Page table integrity. The page table must contain the correct translations. The page table is simply a data structure in memory that we can reason about in two parts. One part is the page table's root (or lowest level table). The page table's root is at a fixed physical address and uses a fixed mapping from the virtual to physical address. The second part is dynamically mapped and thus relies on address translation.

To more clearly distinguish how hardware and software collaborate in the address translation system, we divide page table integrity into two subinvariants.

PT-SubInv1 requires that the page table data structure correctly defines the translations. This subinvariant is enforced by the privileged code that maintains the page table. PT-SubInv2 requires that the page table's root is correct. This subinvariant is enforced by hardware (as specified by PAMC) because the root has a fixed physical address.

Translation coherence. Translation coherence is similar but not identical to cache coherence for regular memory. All cached copies of a translation (in TLBs) should be coherent with respect to the page table. The notion of TLB coherence isn't new,[11] although it hasn't yet been defined precisely.

We choose to specify the translation coherence invariants in a way that's similar to how Martin et al. specified cache coherence invariants for Token Coherence.[12] (The abstract tokens that we consider here are independent of any tokens used for the purposes of implementing either regular cache coherence or translation coherence.) We consider each translation to *logically* have a fixed number of tokens, T, associated with it. T must be at least as great as the number of TLBs in the system. Tokens can reside in TLBs or in main memory. The following three subinvariants are required:

- Coherence-SubInv1: At any point in logical time, exactly T tokens exist for each translation. This conservation law doesn't permit a token to be created, destroyed, or converted into a token for another translation.
- Coherence-SubInv2: A core that accesses a translation (to perform a load or store) must have at least one token for that translation.
- Coherence-SubInv3: A core that performs an MRF to a translation must have all T tokens for that translation before completing the MRF (that is, before releasing the lock) and making it visible. This invariant ensures that there's a single point in time at which the old (pre-modified) translation is no longer visible to any cores.

The first two subinvariants are almost identical to those of Token Coherence.[12] The third subinvariant (which is analogous to Token Coherence's invariant that a core needs all tokens to perform a store) is subtly different from Token Coherence, because an MRF usually isn't an atomic write. In Token Coherence, a core must hold all tokens throughout the entire lifetime of the store, but an MRF only requires the core to hold all tokens before releasing the lock.

Having a thorough, multilevel specification of consistency enables programmers, designers, and design verifiers to more easily reason about the memory system's correctness. Furthermore, it facilitates the development of comprehensive dynamic verification techniques that can, at runtime, detect errors due to design bugs and physical faults.

This work represents an initial exploration of this research area. We foresee further research into VAMC models and address translation systems, as well into relationships between them. One future avenue of research is to explore address translation models that are more relaxed than AT_{SC}, yet still provably sufficient for bridging gaps between specific PAMC and VAMC models. We anticipate that address translation can be made more scalable if it's less conservative, but more relaxed designs are viable only if designers and verifiers can convince themselves that they're correct. Our framework for specifying VAMC enables these explorations. MICRO

Acknowledgments

This work was supported in part by Semiconductor Research Corporation contract 2009-HJ-1881 and the National Science Foundation grant CCR-0444516. For their helpful feedback on this work, we thank Anne Bracy, Dave Christie, Landon Cox, Stephan Diestelhorst, Anita Lungu, Milo Martin, and Albert Meixner. We thank Trey Cain and Cathy May for help in understanding issues specific to the Power ISA. An earlier version of this paper was published at ASPLOS 2010.[13]

References

1. S.V. Adve and K. Gharachorloo, ''Shared Memory Consistency Models: A Tutorial,'' *Computer,* vol. 29, no. 12, 1996, pp. 66-76.
2. L. Lamport, ''How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs,'' *IEEE Trans. Computers,* vol. 28, no. 9, 1979, pp. 690-691.
3. ''Revision Guide for AMD Family 10h Processors,'' tech. report 41322, Advanced Micro Devices, 2008.
4. IBM, ''IBM PowerPC 750FX and 750FL RISC Microprocessor Errata List DD2.X, version 1.3,'' Feb. 2006.
5. ''Intel Core2 Extreme Quad-Core Processor QX6000 Sequence and Intel Core2 Quad Processor Q6000 Sequence Specification Update,'' tech. report 315593-021, Intel, 2008.
6. Advanced Micro Devices, ''Revision Guide for AMD Athlon64 and AMD Opteron Processors,'' Publication 25759, rev. 3.59, 2006.
7. ''Intel Core Duo Processor and Intel Core Solo Processor on 65nm Process Specification Update,'' tech. report 309222-016, Intel, 2007.
8. J. Manson, W. Pugh, and S.V. Adve, ''The Java Memory Model,'' *Proc. 32nd ACM Sigplan-Sigact Symp. Principles of Programming,* ACM Press, 2005, pp. 378-391.
9. M.D. Hill et al., ''A System-Level Specification Framework for I/O Architectures,'' *Proc. 11th Ann. ACM Symp. Parallel Algorithms and Architectures,* ACM Press, 1999, pp. 138-147.
10. A. Arvind and J.-W. Maessen, ''Memory Model = Instruction Reordering + Store Atomicity,'' *Proc. 33rd Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, 2006, pp. 29-40.
11. P.J. Teller, ''Translation-Lookaside Buffer Consistency,'' *Computer,* vol. 23, no. 6, 1990, pp. 26-36.
12. M.M.K. Martin, M.D. Hill, and D.A. Wood, ''Token Coherence: Decoupling Performance and Correctness,'' *Proc. 30th Ann. Int'l Symp. Computer Architecture,* ACM Press, 2003, pp. 182-193.
13. B.F. Romanescu, A.R. Lebeck, and D.J. Sorin, ''Specifying and Dynamically Verifying Address Translation-Aware Memory Consistency,'' *Proc. 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, 2010, pp. 323-334.

Bogdan F. Romanescu is a member of the Online Services Division at Microsoft. His research interests include memory consistency, memory coherence, and scalable communication protocols in large-scale systems. He has a PhD in electrical and computer engineering from Duke University.

Alvin R. Lebeck is a professor in the Departments of Electrical and Computer Engineering and of Computer Science at Duke University. His interests include architectures for emerging nanotechnologies, multicore processors, memory systems, and energy-efficient computing. He has a PhD in computer science from the University of Wisconsin, Madison. He's a senior member of IEEE and a member of the ACM.

Daniel J. Sorin is an associate professor in the Departments of Electrical and Computer Engineering and of Computer Science at Duke University. His research interests include computer architecture, fault tolerance, memory systems, and design for verifiability. He has a PhD in electrical engineering from the University of Wisconsin, Madison. He's a senior member of IEEE and the ACM.

Direct questions and comments to Daniel J. Sorin, Duke Univ., Box 90291, Durham, NC 27708-0291; sorin@ee.duke.edu.

cn *Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.*