Appears in the 15th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010)
Pittsburgh, PA, March, 2010

Specifying and Dynamically Verifying Address Translation-Aware Memory Consistency

Bogdan F. Romanescu
Department of ECE
Duke University
bfr2@ee.duke.edu

Alvin R. Lebeck
Department of Computer Science
Duke University
alvy@cs.duke.edu

Daniel J. Sorin
Department of ECE
Duke University
sorin@ee.duke.edu

Abstract

Computer systems with virtual memory are susceptible to design bugs and runtime faults in their address translation (AT) systems. Detecting bugs and faults requires a clear specification of correct behavior. To address this need, we develop a framework for AT-aware memory consistency models. We expand and divide memory consistency into the physical address memory consistency (PAMC) model that defines the behavior of operations on physical addresses and the virtual address memory consistency (VAMC) model that defines the behavior of operations on virtual addresses. As part of this expansion, we show what AT features are required to bridge the gap between PAMC and VAMC. Based on our AT-aware memory consistency specifications, we design efficient dynamic verification hardware that can detect violations of VAMC and thus detect the effects of design bugs and runtime faults, including most AT related bugs in published errata.

Categories and Subject Descriptors C.0 [Computer Systems Organization] GENERAL - Hardware/software interfaces, Systems specification methodology; B.8.1 [Performance and Reliability] Reliability, Testing, and Fault-Tolerance

General Terms Design, Reliability, Verification

Keywords memory consistency, virtual memory, address translation, dynamic verification

1. Introduction

The most important feature of a computer is correct execution. 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. In this work, we are concerned with detecting incorrect behavior—primarily due to design bugs but also due to physical faults—before it leads to a data corruption or crash.

To detect incorrect behavior, we must first precisely specify correctness. For a processor core, the architecture's ISA specifies the exact semantics of every instruction in the instruction set. The ISA also specifies the architecture's *memory consistency model* [2], 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 (SC), in 1979 [22], statically or dynamically verifying a memory system became easier. Lamport's key contribution to verification was a precise, microarchitecture-independent specification of correct memory system behavior.

Nevertheless, there are still many bugs in modern memory systems—we identified 21 address translation (AT) related bugs in published errata (a small subset of which is described in Table 1) [1, 4, 5, 16, 18, 19, 20], despite the relatively small size of the AT system—and we believe that one underlying cause of these bugs is a tendency to over-simplify memory consistency. *Memory consistency is not just one monolithic interface between the hardware and the software*, as illustrated in Figure 1; rather, *memory consistency is a set of interfaces between the hardware and various levels of software*, as illustrated in Figure 2. Although Adve and Gharachorloo previously explained the multi-level nature of memory consistency [2], this more comprehensive definition of memory consistency is not always adopted in the community. For example, when we teach computer architecture students about memory consistency models, we generally do not specify whether that model refers to virtual or physical addresses [14]. We often implicitly assume a one-to-one mapping from virtual to physical address.

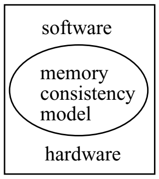

Figure 1. Address Translation-Oblivious Memory Consistency

In this paper, 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, memory barriers, etc.) on physical addresses and virtual addresses, respectively. As shown in Figure 2, PAMC and VAMC are important interfaces that support different layers of software. Correct PAMC is required for unmapped code to work correctly, and correct VAMC is required for mapped code to work correctly.

As part of this specification framework, we discuss and formalize the crucial role of address translation (AT) in supporting a VAMC model. VAMC has three AT-related aspects that fundamentally distinguish it from PAMC: 1) synonyms, 2) mapping and permission changes, and 3) load/store side effects. *Without correct AT, a system with virtual memory cannot enforce any VAMC model.* AT

ASPLOS '10 March 13-17, 2010, Pittsburgh, Pennsylvania, USA.

Table 1. A Small Sample of Published Address Translation Bugs

Chip	Bug	Effect
AMD Athlon64/Opteron [1]	TLB flush filter may cause coherency problem in multicore systems	unpredictable system failure (possible use of stale translations)
AMD Athlon64/Opteron [1]	INVLPG instruction with address prefix does not correctly invalidate the translation requested	unpredictable system behavior (use of stale translation)
Intel Core Duo [19]	One core is updating a page table entry while the other core is using the same translation entry may lead to unexpected behavior	unexpected processor behavior
Intel Core Duo [19]	Updating a PTE by changing R/W, U/S or P bits without TLB shootdown may cause unexpected processor behavior	unexpected processor behavior

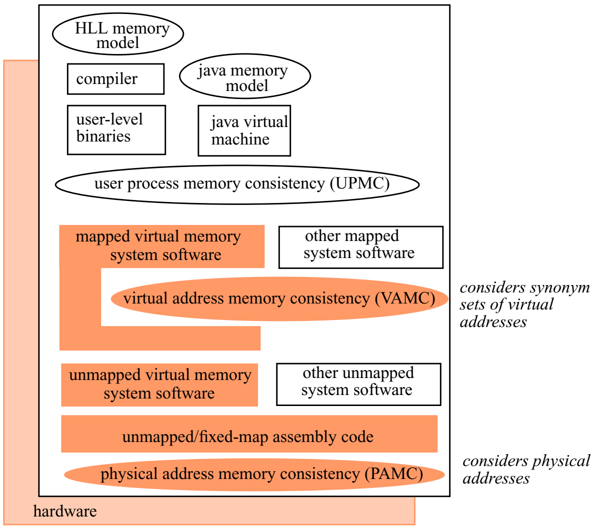
HLL memory model
compiler
java memory model
user-level binaries
java virtual machine
user process memory consistency (UPMC)
mapped virtual memory system software
other mapped system software
virtual address memory consistency (VAMC)
considers synonym sets of virtual addresses
unmapped virtual memory system software
other unmapped system software
unmapped/fixed-map assembly code
physical address memory consistency (PAMC)
considers physical addresses
hardware

Figure 2. Address Translation-Aware Memory Consistency. Shaded portions are focus of this paper.

encompasses both hardware and privileged software. A disproportionate fraction of the published bugs in shipped chips [1, 4, 5, 16, 18, 19, 20] involve AT hardware, and an AT bug caused a costly delay for AMD's recent quad-core Barcelona chip [41]. Furthermore, we show in this paper that writing privileged software for managing AT coherence can be quite complicated.

We develop an AT-aware specification of memory consistency not only to benefit design and verification teams, but also to enable architects to design runtime checkers that detect incorrect behaviors. Run-time checking of correctness is often referred to as *dynamic verification*, because the system verifies at runtime that specification is satisfied. For processor cores, specifications exist and architects have developed effective dynamic verification mechanisms for checking that they are operating correctly [7, 28]. Dynamic verification schemes also exist for AT-oblivious memory systems [10, 29, 30]. However, *to our knowledge no dynamic verification schemes include AT, and we believe the main reason for this is the lack of a specification of correctness that can be checked at runtime*. Our AT-aware memory consistency model specifications enable us to develop dynamic verification schemes to check AT system correctness.

In this work, we make two primary contributions:

- We develop a framework for precise, implementation-independent specification of AT-aware memory consistency. This framework facilitates hardware design verification, writing software that involves AT, and the development of dynamic verification schemes for memory consistency.
- We create an efficient dynamic verification scheme for AT systems that can detect errors due to design bugs and runtime faults. When combined with PAMC dynamic verification and timeouts, our AT dynamic verification scheme is comprehensive and can capture all 21 AT related bugs we identified.

In Section 2, we develop specifications of PAMC and VAMC and show that a combination of PAMC and the AT system is required to support VAMC. In Section 3, we present a framework for modeling AT systems. Then, in Section 4, we develop a scheme to dynamically verify AT (DVAT). Combining DVAT with an existing scheme for dynamic verification of PAMC [10, 29, 30] is sufficient for dynamic verification of VAMC. In Section 5, we experimentally evaluate our DVAT scheme. We compare our work to prior work in Section 6, and we conclude in Section 7.

2. Specifying AT-Aware Memory Consistency

A memory consistency specification provides two important functions: 1) it serves as a contract between the system and the programmer, and 2) it provides the formal framework necessary for verifying (statically or dynamically) correct memory system operation. In this section, we describe the multiple levels of memory consistency and explain how to adapt well-known existing consistency models to these levels. We focus primarily on two consistency levels, PAMC and VAMC. We present a VAMC specification and show how it differs from PAMC, describe how VAMC impacts both system programmers and verification, and discuss how AT bridges the gap between PAMC and VAMC.

2.1 Levels of Memory Consistency

A computer system presents memory interfaces (consistency models) at multiple levels, as shown in Figure 2. We position hardware below all levels, because it provides mechanisms that can be used to enforce consistency models at various levels (e.g., the processor provides in-order instruction commit). In this discussion, we consider four levels relevant to programmers. At each of these levels, 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 ISA, ABI, and API, but, for the purposes of this paper, we generally do not need to consider which interfaces belong in which specifications. We discuss these levels, starting at the lowest level:

Table 2. SC for PAMC. Loads and stores are to physical addresses. An “X” denotes an enforced ordering.

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

- *Physical address memory consistency* (PAMC): Unmapped software, including the boot code and part of the system software that manages AT, relies upon PAMC. Implementing PAMC is the hardware’s responsibility and, as such, is specified precisely in the architectural manual.
- V*irtual address memory consistency* (VAMC): VAMC is the level just above the PAMC. All mapped software (i.e., software that executes using virtual addresses) relies upon VAMC, including mapped system software. VAMC builds upon PAMC, and requires support from AT software and the hardware. One perhaps non-intuitive aspect of VAMC is that mapped virtual memory system software both relies upon VAMC and helps to support it.
- User process memory consistency (UPMC). UPMC may 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: At the highest level, user-level programmers see the consistency model specified by the high level language [3], such as the consistency models provided by C++ [9] or Java [25]. These models are supported by the compilers, runtime systems, and lower level consistency models.

Existing consistency models—such as sequential consistency (SC), processor consistency, weak ordering, release consistency, etc.—do not distinguish between virtual and physical addresses. Lamport’s original definition of SC is typical in that it specifies a total order of operations (loads and stores), but it does not specify whether the loads and stores are to virtual or physical addresses. Implicitly, most existing consistency models assume either unmapped software or software with a fixed one-to-one mapping from virtual to physical addresses. We refer to these consistency models as *AT-oblivious*. In this paper, we specifically focus on two interface levels, PAMC and VAMC, and the hardware and software involved in supporting them, as highlighted in Figure 2.

2.2 Specifying PAMC

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

In this paper, we specify consistency models (at all levels) using a table-based scheme like those of Hill et al. [15] and Arvind and Maessen [6]. The table specifies which program orderings are enforced by the consistency model. Thus, adaptations of SC and Weak Ordering for PAMC would be defined by Table 2 and Table 3, respectively. Under SC, all memory operations must appear to perform in program order; under Weak Ordering, memory operations are unordered.

Table 3. Weak Ordering for PAMC. Loads and stores are to physical addresses. An “X” denotes an enforced ordering. An “A” denotes an ordering that is enforced if the operations are to the same physical address. Empty entries denote no ordering.

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

Thread 1	**Thread 2**
Store VA1=1	
	Store VA2 = 2
	Load y=VA1 // y=1
Load x=VA2 // x=2	

Figure 3. Example of Synonym Problem. Assume sequential consistency and that VA1 and VA2 map to PA1. Assume that PA1 is initially zero.

Some consistency models have atomicity constraints that cannot be expressed with just a table (e.g., stores are atomic). We can specify these models by augmenting the table with a specification of atomicity requirements, as in prior work [6].

2.3 Specifying VAMC

Although adapting an AT-oblivious consistency model for PAMC is straightforward, there are three challenges when adapting an AT-oblivious consistency model for VAMC: 1) synonyms, 2) mapping and permission changes, and 3) load/store side effects.

2.3.1 Synonyms

The first challenge is the possible existence of *synonyms*, i.e., 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. SC requires a total order in which the value of a load equals the value of the most recent store to the same address. Unfortunately, naively applying SC 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 naive definition of VAMC that did not consider the level of indirection introduced by AT, 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 re-formulate AT-oblivious consistency models for VAMC by applying the model to *synonym sets of (virtual) addresses* rather than individual addresses. For example, we can define SC for VAMC as follows: there must exist 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*. Similar modifications can be made to adapt other AT-oblivious consistency models for VAMC.

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 ISA Code Snippets to Illustrate the Need to Consider MRF Ordering. Initially, VA1 is mapped to PA1, and the value of PA1 is A.

Impact on Programming. Programmers that utilize 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.

Impact on VAMC Verification. 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.

2.3.2 Mapping and Permission Changes

The second challenge is that 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 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)*.

Impact on Programming. The code snippet in the left-hand side of Figure 4, written for a system implementing the Power ISA, 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 does not guarantee *when* the tlbie instruction will be observed by Thread2 and 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 does not 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 is not sufficient for the programmer to reason about the behavior of this code; the programmer must also understand how MRFs are ordered. We show a corrected version of the code on the right-hand side of Figure 4. In this code, Thread1 executes a *tlbsync* instruction that is effectively a fence for the MRF. Specifically, the *tlbsync* guarantees that the tlbie instruction executed by Thread1 has been observed by other cores.[1]

Impact on VAMC Verification. Similar to the above programming example, 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.

2.3.3 Load/Store Side Effects

The third challenge in specifying VAMC is that loads and stores to virtual addresses have side effects. The AT system includes status bits (e.g., accessed and dirty bits) for each page table entry. These status bits (discussed in more detail in Section 3) are part of the architectural state, and the ordering of updates to those bits must thus be specified in VAMC. To achieve this we add two new operations to the specification tables: Ld-sb (load's impact on status bits) and St-sb (store's impact on status bits).

Impact on Programming. Consider the example in Figure 5. Without knowing how status updates are ordered, the OS cannot be

[1] In the Power ISA, Memory Barriers (sync instructions) only order normal loads and stores but not MRFs; other ISAs may use a single instruction for both of these purposes.

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

 /* 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. */
```

Figure 5. Code Snippet to Illustrate the Need to Consider Side Effects

Table 4. SC for VAMC. Loads and stores are to synonym sets of virtual addresses. An "X" denotes an enforced ordering.

		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

Table 5. Weak Ordering for VAMC. Loads and stores are to synonym sets of virtual addresses. "X" denotes an enforced ordering, "A" denotes an ordering that is enforced if operations are to the same synonym set, empty entries denote no ordering

		Operation 2					
		Ld	Ld-sb	St	St-sb	MemBar	MRF
Operation 1	Ld			A		X	X
	Ld-sb					X	X
	St	A		A		X	X
	St-sb					X	X
	Membar	X	X	X	X	X	X
	MRF	X	X	X	X	X	X

sure what state will be visible in these bits. It is 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 writeback was not necessary, resulting in data loss.

Impact on VAMC Verification. Without a precise specification of status bit ordering, verification could miss a situation analogous to the software example above. 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 lead to silent data corruption.

2.3.4 Putting It All Together

We present VAMC adaptations of SC and Weak Ordering in Table 4 and Table 5, respectively. Note that these specifications include MRFs and status bit updates, and that the loads and stores apply to synonym sets of virtual addresses (not individual virtual addresses). Also note that the weak ordering VAMC allows status bits to be reordered with respect to loads, stores, and other status bit updates. These specifications provide both a contract for programmers and enable development of techniques to verify correct memory system operation.

2.3.5 Alternative VAMC Models

The two VAMC models that we presented in the previous section are clearly not the only possibilities. For example, both of these adaptations strictly order MRFs, but other MRF orderings are possible. We are unaware of any current system that relaxes the ordering between MRFs that modify mappings and other memory operations, but at least one ISA (Power ISA) allows MRFs that upgrade permissions to be reordered with respect to certain memory operations. For example, an MRF that adds write permission to a region that currently only has read permission can be reordered with respect to loads since they are unaffected by the permission change [38]. However, we expect most VAMC models to order this type of MRF with respect to stores.

Another example of an alternative VAMC model is one in which all MRFs can be reordered unless an explicit fence-like instruction for MRFs is used, which could be a Memory Barrier (MemBar) or a dedicated instruction for ordering MRFs. Analogous to relaxed memory consistency models, software uses a serializing instruction, like the Power ISA's *tlbsync*, to enforce order when it wishes to have order, but the default situation allows a core to defer invalidations due to MRFs.

One important aspect of our future work will be to explore the wide range of possible VAMC models and the opportunities that they provide for performance optimizations.

2.4 Commercial Consistency Specifications

In Table 6, we compare the PAMC models and AT systems of six currently available commercial architectures. Note that there is a considerable diversity in PAMC models and hardware support for AT. For example, while all platforms implement TLB coherence, some architectures provide inter-processor interrupts for maintaining TLB coherence, whereas other architectures support TLB coherence by providing privileged instructions for invalidating TLB entries on other cores.

Fundamentally, an architecture cannot specify VAMC, because VAMC requires software support. The hardware only provides mechanisms to support virtual memory and thus, without requisite policies implemented in software, cannot guarantee VAMC (as demonstrated in Figure 4). Furthermore, we are unaware of any commercial consistency model that explicitly defines both PAMC and VAMC.

However, an architecture's PAMC and provided AT mechanisms can have some impact on VAMC, and usage information associated with an architecture can state what software should do to achieve a particular VAMC model (e.g., as part of the ABI). Some commercial architectures consider AT's impact on memory consistency to a limited extent. For example, SPARC v9 [40] assumes that a store to

Table 6. Address Translation in Commercial Architectures

		AT mechanisms		architecture's impact on VAMC	
ISA	**PAMC**[a]	**TLB management**	**TLB Coherence mechanisms**	**Invalidation Processing**	**Permissions consistency**
MIPS	sequential consistency	software	inter-processor interrupt (IPI)	immediate	strict
IA-32, Intel64	processor consistency	hardware	IPI	immediate	relaxed
IA-64	release consistency	hardware & software	IPI and global TLB invalidation	deferred	relaxed
AMD64	processor consistency	hardware	IPI	immediate	relaxed
SPARC	TSO, PSO, RMO	software	IPI (sent directly to MMU)	immediate	strict
Power ISA	weak consistency	hardware	IPI and global TLB invalidation	deferred	strict

a. These classifications are somewhat broad; all commercial consistency models have their particularities (e.g., AMD64 is most similar to processor consistency (PC) but not identical to how PC is defined in the literature).

one virtual address modifies the values of all other synonyms. Intel's IA-64 model [17] assumes a one-to-one mapping between virtual and physical addresses. In the rightmost two columns of Table 6 we list, for each architecture, its impact on two aspects of VAMC: (a) whether a TLB invalidation must be processed immediately or can be deferred and (b) whether translation permission bits must be strictly coherent. Thus the PAMC and AT mechanisms have an impact on the VAMC model that can be supported by a platform. For example, an architecture with relaxed permissions coherence might not be able to enforce some of the orderings in VAMC tables like Tables 4 and 5.

2.5 Gap Between PAMC and VAMC

As we stated in Section 1, one of our motivations in completely specifying memory consistency is to provide a definition of correct behavior that can be statically or dynamically verified. In this work, our goal is to dynamically verify VAMC. Schemes already exist for dynamic verification of PAMC [12, 29, 30], but no comparable scheme exists for VAMC. Because we have not 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 AT system. In the next section, we discuss AT systems and what they must provide in order to bridge the gap between PAMC and VAMC.

3. Specification of Address Translation

In this section, we present a framework for specifying AT systems. We discuss our assumptions about AT systems, present one specific, easy-to-understand AT model, and then describe a more general framework for specifying AT models.

3.1 AT Assumptions

We restrict our discussion to page-based AT systems and leave as future work issues due to differences with other virtual memory paradigms. A translation is a tuple <mapping (VP,PP), permissions, status>, where the mapping converts the virtual page (VP) to a physical page (PP). The PP, permissions, and status information are specified by the page table entry (PTE) defining the translation. The permission bits include whether the page is owned by the user or the kernel 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.

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

Figure 6. Pseudo-code for a Generic MRF

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 that we illustrate in Figure 6. Some of the activities in an MRF require complicated actions to be performed by the software or hardware. For example, delivering the TLB invalidations may require an inter-processor interrupt or a global TLB invalidation instruction that relies on hardware for distributing the invalidations.

3.2 A Provably Sufficient AT Model

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

3.2.1 AT_{SC}: A Sequential AT Model

AT_{SC} is a sequential *model* of an AT system. Because it is a model, it is a *logical* abstraction that encompasses the behaviors of a variety of possible *physical* implementations. We describe the three key aspects of this model:

- MRFs logically occur instantaneously and are thus totally ordered with respect to regular loads and stores and other AT operations.[2] 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

[2] 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(s) modified by the MRF. We leave the study of this model for future work.

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 re-executed.

- 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 of the model either by using physical caches or by using virtual caches with same index mapping of synonym set virtual addresses.

3.2.2 $\mathrm{PAMC_{SC}}$+$\mathrm{AT_{SC}} \rightarrow \mathrm{VAMC_{SC}}$

$\mathrm{PAMC_{SC}}$ specifies that all loads and stores using physical addresses are totally ordered. $\mathrm{AT_{SC}}$ specifies that a translation access occurs instantaneously and simultaneously with the load or store. Under $\mathrm{AT_{SC}}$, all MRFs are totally ordered with respect to each other and with respect to loads and stores. $\mathrm{AT_{SC}}$ also specifies that accesses to synonyms are ordered according to $\mathrm{PAMC_{SC}}$ (e.g., via the use of physical caches). Therefore, all loads and stores using virtual addresses are totally ordered. Finally, $\mathrm{AT_{SC}}$ specifies that status bit updates are performed simultaneously with the corresponding load or store, and thus status bit updates are totally ordered with respect to all other operations. Hence, $\mathrm{PAMC_{SC}}$ plus $\mathrm{AT_{SC}}$ results in $\mathrm{VAMC_{SC}}$, where ordering is enforced between all operations (see Table 4).

3.3 A Framework for Specifying AT Models

$\mathrm{AT_{SC}}$ is just one possible model for AT and thus one possible bridge from a PAMC model to a VAMC model. In this section, we present a framework for specifying AT models, including AT models that are more relaxed than the one presented in Section 3.2. A precisely specified AT model facilitates the verification of the AT system and, in turn, the verification of VAMC. We have not yet proved the sufficiency of AT models other than $\mathrm{AT_{SC}}$ (i.e., that they bridge any particular gap between a 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 (Section 3.3.1).
- Translations are "coherent" (Section 3.3.2). We put quotes around coherent, because we consider a range of definitions of coherence, depending on how reordered and lazy the propagation of updates is permitted to be. All systems of which we are aware maintain translation mapping coherence and coherence for permissions downgrades, either using software routines, an all-hardware protocol [35], or a hardware/software hybrid. Systems may or may not specify that status bits and/or permissions upgrades are also coherent. In the rest of this paper, without loss of generality, we assume that translations in their entirety are coherent.

3.3.1 Page Table Integrity

For AT to behave correctly, the contents of 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 root (or lowest level table) of the page table. The root of the address space is at a fixed physical address and uses a fixed mapping from 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 AT system, we divide page table integrity into two sub-invariants:

- [PT-SubInv1] The translations are correctly defined by the page table data structure. This sub-invariant is enforced by the privileged code that maintains the page table.
- [PT-SubInv2] The root of the page table is correct. This sub-invariant is enforced by hardware (as specified by PAMC), since the root has a fixed physical address.

3.3.2 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 is not new [38], although it has not previously been defined precisely, and there have been many different implementations of AT systems that provide coherence.

Because there are many possible VAMC models, there are also many possible definitions of translation coherence. The differences between these definitions of coherence are based on when translation updates must be made available to other cores (e.g., immediately or lazily) and whether updates may be reordered. We leave for future work a fully-parameterizable definition of translation coherence, and we now focus on a specific definition of coherence that is consistent with $\mathrm{AT_{SC}}$.

For $\mathrm{AT_{SC}}$, we specify the invariants that an AT system must maintain to provide translation coherence. *These invariants are independent of the protocol that is implemented to maintain these invariants.* We choose to specify the translation coherence invariants in a way that is similar to how cache coherence invariants were specified in Martin et al.'s Token Coherence [27] paper, with AT-specific differences highlighted. We have chosen to specify the invariants in terms of tokens, as is done in Token Coherence, in order to facilitate our specific scheme for dynamically verifying the invariants, as explained in Section 4.

We consider each translation to logically have a fixed number of tokens, T, associated with it.[3] T must be at least as great as the number of TLBs in the system. Tokens may reside in TLBs or in main memory. The following three sub-invariants are required:

- [Coherence-SubInv1] At any point in logical time [21], there exist exactly T tokens for each translation. This "conservation law" does not 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 (i.e., before releasing the lock) and making it visible. This invariant ensures that there is a single point in time at which the old (pre-modified) translation is no longer visible to any cores.

The first two sub-invariants are almost identical to those of Token Coherence (TC). The third sub-invariant, which is analogous to TC's invariant that a core needs all tokens to perform a store, is subtly different from TC, because an MRF is not an atomic write.

[3] 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.

In TC, 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.

As with normal cache coherence, there are many ways to implement AT_{SC} coherence such that it obeys these three sub-invariants. For example, instead of using explicit tokens, an AT system could use a snooping-like protocol with global invalidations or inter-processor interrupts for maintaining translation coherence.

4. Dynamic Verification of Address Translation

This section presents our method for dynamically verifying AT_{SC}—called *$DVAT_{SC}$*—which when used with existing methods to dynamically verify $PAMC_{SC}$ results in dynamically verifying $VAMC_{SC}$ per Section 3.2.2. To dynamically verify $PAMC_{SC}$ we leverage previous techniques [12, 29, 30]. *The contribution of this section is our simple method for dynamically verifying AT_{SC}* (e.g., MRF and load/store ordering), enabled by properly specifying VAMC.

4.1 System Model

Our baseline system is a cache-coherent multicore chip. Similar to modern processors, each core uses virtually-indexed, physically-tagged caches. Physical caches ensure a store's atomicity with respect to loads from the same synonym set.[4] Cores have hardware-managed TLBs, and updates to the status bits occur atomically in both the TLB and the page table when the corresponding load or store commits. The TLB coherence protocol is conservative and restricts parallelism. A core that performs an MRF locks the page table for the entire duration of the MRF, changes the PTE, flushes all TLBs (instead of invalidating only affected translations), and causes all other cores to spin after sending acknowledgments (instead of continuing immediately), waits for the acknowledgments from all other cores before continuing (instead of lazily collecting acknowledgments), and then signals that the other cores may continue.

We assume the existence of a checkpoint/recovery mechanism [33, 37] that can be invoked when $DVAT_{SC}$ detects an error. The ability to recover to a pre-error checkpoint enables us to take $DVAT_{SC}$'s operations off the critical path; an error can be detected somewhat lazily as long as a pre-error checkpoint still exists at the time of detection.

4.2 $DVAT_{SC}$ Overview

To dynamically verify AT_{SC}, we must dynamically verify both of its invariants: page table integrity and translation mapping coherence.

4.2.1 Checking Page Table Integrity

PT-SubInv1 is an invariant that is maintained by software. Fundamentally, there is no hardware solution that can completely check this invariant, because the hardware does not have semantic knowledge of what the software is trying to achieve. Hardware could be developed to perform some "sanity checks," but, fundamentally, software checking is required. One existing solution to this problem is self-checking code [8]; we defer further research into this area of reliable software engineering to future work.

To check that PT-SubInv2 is maintained, we can adopt any of the previously proposed dynamic verification schemes for PAMC [12, 29, 30].

4.2.2 Checking Translation Coherence

The focus of $DVAT_{SC}$ is the dynamic verification of the three translation coherence sub-invariants (Section 3.3.2). Because we have specified these sub-invariants in terms of tokens, we can dynamically verify the sub-invariants by adapting a scheme called TCSC [31] that was previously used to dynamically verify token-based cache coherence. TCSC's key insight is that cache coherence states can be represented with token counts that can be periodically checked; this same insight applies to translation coherence. Even though the specification of coherence is in terms of tokens, the coherence protocol implementation is unrestricted; the protocol simply needs to maintain the invariants. For example, Martin et al. showed that snooping and directory cache coherence protocols can be viewed as maintaining the token invariants [27]. Thus, $DVAT_{SC}$ is not architecturally visible, nor is it tied to any specific TLB coherence protocol.

Similar to TCSC, but for TLBs instead of normal caches, $DVAT_{SC}$ adds *explicit* tokens to the AT system.[5] Each translation has T tokens that are initially held by the translation's home memory and physically collocated with the translation's PTE. Because PTEs usually have some unused bits (e.g., 3 for IA-32 and 4 for the Power ISA), we can use these bits to store tokens. If we need more than the number of unused bits to hold T tokens, then we extend the memory block size to hold the extra bits. Because translations are dynamic and $DVAT_{SC}$ does not know a priori which blocks will hold PTEs, we must extend every memory block. A processor that brings a translation into its TLB acquires one token corresponding to the PTE defining the translation. This token is held in the corresponding TLB entry, which requires us to slightly enlarge every TLB entry. The token is relinquished by the processor and returned to the home memory once the translation is evicted from the TLB due to a replacement. In the case of a TLB invalidation, the token is sent to the core that requested the invalidation.

Each "node" in the system (in this paper, a node is a TLB or the memory) maintains a fixed-length signature of its token transfer history. This signature is a concise representation of the node's history of translation coherence events. Whenever a token is acquired or released, the signature is updated using a function that considers the physical address of the PTE to which the token corresponds and the logical time [21] of the transfer. Because extracting the translation mapping's virtual address from a TLB entry would require re-designing the TLB's CAM, the signature function operates on the PTE's physical address instead of its virtual-to-physical mapping. The PTE's physical address is a unique identifier for the translation. The challenge is that we now require that the SRAM portion of each TLB entry be expanded to hold the physical address of the PTE[6] (but this address does *not* need to be added to PTEs in the page table). Thus, *$signature_{new}$ = function ($signature_{old}$, PTE's physical address, logical time)*.

In a correctly operating AT_{SC} system, the exchanges of tokens will obey the three Coherence sub-invariants of AT_{SC} that we pre-

[4] Our DVAT implementation also applies to architectures with virtual caches, such as HyperSPARC, that require the OS to map synonym sets at the same cache indices. We are unaware of current high-performance processors implementing virtual caches that handle synonyms differently.

[5] The tokens used by $DVAT_{SC}$ are distinct from the tokens used by TCSC if TCSC is being used to check cache coherence.

[6] Having the PTE's physical address in the TLB entry is also useful for solving a problem we discuss in Section 4.3.

sented in Section 3.3.2. DVAT$_{SC}$ thus checks these three sub-invariants at runtime, in the following fashion:

Coherence-SubInv1. Periodically, the signatures of all nodes are aggregated at one central verification unit that can check whether the conservation of tokens has been maintained. Updating signatures and checking them are off the critical path, because we assume that we can recover to a pre-error checkpoint if an error is detected. The signature update function should be chosen so that it is easy to implement in hardware and avoids aliasing (i.e., hashing two different token event histories to the same signature) as best as possible. We use the same function as TCSC [31] because it achieves these goals, but other functions could be chosen. Any basis of logical time can be used as long as it respects causality, and thus we use a simple one based on loosely synchronized physical clocks, similar to one used in prior work [37]. It is critical for DVAT$_{SC}$ to consider the mapping (as represented by its PTE's physical address) and the time of the transfer in order to detect situations in which errors cause tokens to be sent for the wrong translations or tokens to be transferred at the wrong times.

Coherence-SubInv2. Checking this sub-invariant is straightforward. All that needs to be done is for each core to check that a token exists for a translation that it is accessing in its TLB. This check can be performed in parallel with the TLB access and thus does not impact performance.

Coherence-SubInv3. Checking this sub-invariant is similar to checking Coherence-SubInv2. In parallel with completing an MRF for a translation, a core checks that it has all *T* tokens for that translation.

4.3 Implementation Details

DVAT$_{SC}$ must address three challenges related to PTEs and token handling. The first issue is how to identify memory locations that contain PTEs. One simple option is to have the kernel mark pages that hold PTEs. Another option would be to monitor page table walks performed by the dedicated hardware; the first page table walk performed on a PTE marks the location accordingly and assigns it *T* tokens.

The second issue is determining where to send tokens when evicting a TLB entry to make room for a new translation (i.e., not in response to an invalidation). With a typical TLB, we would not be able to identify the home node for an evicted translation. However, because we already hold the physical address of the PTE in each TLB entry for other purposes (as explained in Section 4.2), we can easily identify the translation's home node.

The third problem is related to which tokens need to be sent to the initiator of a full TLB flush. Many ISAs, such as the PowerISA, specify that the ability to invalidate specific translations is an optional feature for implementations, and thus implementations without this feature rely on full flushes of TLBs. As a consequence, a core that is requested to flush its TLB is unlikely to know which translations, if any, are actually being modified by the MRF that triggered the flush. One solution to this situation is for the core to send its tokens for all of its TLB entries to the initiator of the flush. The initiator keeps the tokens it wants (i.e., tokens for the translations it is modifying) and forwards the rest of them to their home nodes.

4.4 Checking Liveness

If the AT system behaves safely (i.e., does not behave incorrectly) but fails to make forward progress (e.g., because a node refuses to invalidate a translation that is required by another node), then DVAT$_{SC}$ will not detect this situation. Fortunately, timeout mechanisms are a simple approach for detecting liveness problems, and we have added such timeouts to our DVAT$_{SC}$ implementation.

Table 7. Target System Parameters

Parameter	Value
cores	2, 4, 8, 16 in-order, scalar cores
L1D/L1I caches	128KB, 4-way, 3-cycle hit latency
L2 cache	4MB, 4-way, 6-cycle hit latency
memory	4GB, 160-cycle hit latency
TLBs	1 I-TLB and 1 D-TLB per core; each TLB has 128-entries and is 4-way
cache coherence	MOSI broadcast snooping
network	broadcast tree
DVAT$_{SC}$ tokens	each PTE has T = 2C tokens
DVAT$_{SC}$ signature	64 bits

Table 8. Scientific Benchmarks

benchmark	description
knary	spawn tree of threads
mm	dense matrix multiplication
lu	LU factorization of dense matrix
msort	mergesort of integers
barnes-hut	N-body simulation

5. Evaluation of DVAT$_{SC}$

We now evaluate DVAT$_{SC}$'s error detection ability, performance impact, and hardware cost.

5.1 Methodology

System Model and Simulator. Because AT involves system software, we must use full-system simulation in our experiments. We use Simics [24] for functional simulation of an IA-32 multiprocessor augmented with a TLB module (for controlling TLB behavior and fault injection) and GEMS [26] for timing simulation. The operating system is Fedora Core 5 (kernel 2.6.15).[7] Our target system, described in Table 7, is one particular implementation that satisfies the system model presented in Section 4.1. Because our target system conforms to the IA-32 architecture, TLB management and page walks are performed in hardware and inter-processor interrupts are used to communicate translation invalidations. The interrupt handler at the invalidated node performs the invalidation.

Benchmarks. We evaluate DVAT$_{SC}$ using several scientific benchmarks and one microbenchmark. The five scientific workloads, described briefly in Table 8, were developed as part of the Hood user-level threads library (http://www.cs.utexas.edu/users/hood). We wrote the microbenchmark specifically to stress DVAT$_{SC}$'s error coverage, which is difficult to do with typical benchmarks. This microbenchmark has two threads that continuously map and

[7] We initially used RedHat 7.3 (kernel 2.4.18), but it had a bug that led to incorrect TLB invalidations. Reassuringly, our DVAT implementation detected the resulting errors, but the bug precluded us from using this operating system for our experiments.

remap a shared memory region, thus forcing TLB coherence events to occur.

Error Injection. We injected errors into the AT system, many that correspond to published bugs, including: corrupted, lost, or erroneously delayed TLB coherence messages; TLB corruptions; TLB invalidations that are acknowledged but not applied properly (e.g., flushes that do not flush all TLB entries); and errors in DVAT$_{SC}$ hardware itself. These fault injection experiments mimicked the behavior of real processor bugs, since identically modeling these bugs is impossible for an academic study. Because our simulation infrastructure accurately models the orderings of translation accesses with respect to MRFs, we can accurately evaluate DVAT$_{SC}$'s error detection coverage.

5.2 Error Detection Ability

Prior work has already shown how to comprehensively detect errors in PAMC [12, 29, 30]. Thus we focus on the ability of DVAT$_{SC}$ to detect errors in AT$_{SC}$. We can evaluate its error coverage both empirically and analytically.

Empirical Evaluation. When DVAT$_{SC}$ is combined with PAMC verification (e.g., TCSC) and timeouts, it detects the errors that mimic all 21 published AT bugs. The four bugs in Table 1 are detected when they violate the following Coherence Sub-invariants, respectively: 1 or 2 (the bug violates both sub-invariants and will be detected by the checker for whichever sub-invariant it violates first), 1 or 2, 3, and 3.

Analytical Evaluation. Like TCSC, DVAT$_{SC}$ detects all single errors (and many multiple-error scenarios) that lead to violations of safety and that are not masked by signature aliasing. This error coverage was mathematically proved and experimentally confirmed for TCSC [31]. With a 64-bit signature size and a reasonable algorithm for computing signature updates, the probability of aliasing approaches 2^{-64}. We have performed some error injection experiments to corroborate this result, but the number of experiments necessary to draw conclusions about such an extremely unlikely event is prohibitive.

5.3 Performance Impact

Checking PAMC has been shown to have little performance impact [12, 29, 30]. The rest of DVAT$_{SC}$'s actions are off the critical path, because we use checkpoint/recovery to handle a detected error. The only way in which DVAT$_{SC}$ can impact performance is by increasing interconnection network congestion due to token exchanges, sending the physical address of a PTE along with the translation, and the periodic aggregation of signatures at a central verifier. DVAT$_{SC}$ aggregates and checks signatures at fixed intervals of logical time; in our experiments, we use an interval length of 10,000 snooping coherence transactions, because this interval corresponds to our checkpointing interval.

In Figure 7, we plot the average link utilization in the interconnection network, both with and without DVAT$_{SC}$. For each benchmark data point, we plot the highest overhead observed across 100 runs that are perturbed to have slightly different timings, to avoid underestimating utilization due to a particularly fortuitous timing. We observe that, for all benchmarks and all numbers of cores, the increase in utilization due to DVAT$_{SC}$ is quite small. DVAT$_{SC}$'s bandwidth overhead has low impact on network congestion.

Results (not shown due to space constraints) show that the performance impact of DVAT$_{SC}$'s extra bandwidth consumption is less than 2%.

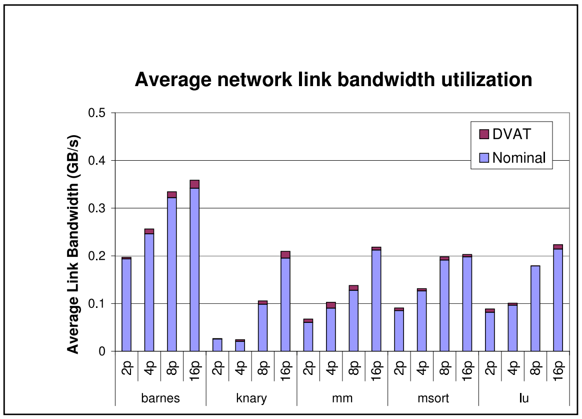

Figure 7. DVAT$_{SC}$'s Bandwidth Overhead

5.4 Hardware Cost

DVAT$_{SC}$ has four hardware costs: the hardware required to dynamically verify PAMC (shown in prior work [12, 29, 30] to be small), the storage for tokens, the extension to each TLB entry to hold the address of the PTE, the hardware to hold and update signatures (shown in TCSC [31] to be small), and the small amount of logic for checking the Coherence sub-invariants. The most significant hardware cost is the storage for tokens. For a system with C cores and 2 TLBs per core (I-TLB and D-TLB), DVAT$_{SC}$ adds 2C tokens to each PTE, thus requiring $\log_2 2C$ bits. For systems with few cores, these bits are likely to fit in the unused bits of the PTE. For systems with many cores, one way to reduce the token storage cost is to extend the coherence sub-invariants to the coarser granularity of a memory block (instead of a PTE), i.e., associate *T* tokens with a memory block. For a 128-core system with 8 PTEs per memory block, we can keep the storage cost to only 11 bits per block (minus those bits that can be fit into unused PTE bits). The overhead is thus only 4.3% and 2.1% for 32 and 64 byte blocks, respectively.

As with any error detection mechanism, DVAT$_{SC}$ benefits from the existence of a checkpoint/recovery mechanism [33, 37] to recover from detected errors. The cost of checkpoint/recovery depends on the specific implementation.

6. Related Work

We discuss prior work in specifying and dynamically verifying correctness, as well as ad-hoc detection of design bugs.

6.1 Specification and Dynamic Verification

We divide this prior work based on which part of the system it considers.

Memory Systems. Meixner and Sorin [29, 30] and Chen et al. [12] dynamically verify AT-oblivious memory consistency models. These schemes apply directly to PAMC, and they can be applied to VAMC if one assumes a one-to-one mapping from VA to PA (i.e., no synonyms). Similarly, Chen et al. [13] dynamically verify the consistency of AT-oblivious transactional memory systems. Cain and Lipasti also developed algorithms for checking AT-oblivious memory consistency [10], but they did not pursue a full implementation. Other work has developed checkers for AT-oblivious cache coherence, which is a necessary sub-invariant of AT-oblivious memory consistency [11, 31]. Our work differs from this prior work by considering address translation.

Processor Cores. The ISA specifies the correct behavior of the processor core, including the exact semantics of every instruction, exception, interrupt, etc. The first dynamic verification scheme for processor cores is DIVA [7]. The insight behind DIVA is that we can check a complicated, superscalar core with a simple, statically verifiable core that has the same ISA. The checker core is so simple that its design can be statically verified (e.g., using a model checker) and thus it detects all design bugs in the superscalar core. Another approach to specification and verification is Argus [28]. Argus is based on the observation that a core's behavior can be verified by checking the correctness of three tasks: control flow, dataflow, and computation. The Argus-1 implementation uses checkers for each of these tasks to dynamically verify the core. Other work by Reddy and Rotenberg [34] has specified *microarchitectural* invariants that can be dynamically verified. These invariants are necessary but not sufficient for correctness (as defined by the ISA). Our work differs from Reddy and Rotenberg by considering architectural correctness.

6.2 Ad-Hoc Bug Detection

Rather than formally specify correctness and then dynamically verify it, another option is for the system to look for known buggy states or anomalies that might indicate that a bug has been exercised. Wagner et al. [39] use a pattern matching technique to detect when the system is in a known buggy state. Work by Narayanasamy et al. [32] and Sarangi et al. [36] proposes to detect design bugs by monitoring a certain subset of processor signals for potential anomalies. If a bug is detected, the authors propose patching it with a piece of programmable hardware. Li et al. [23] take a similar approach to detecting errors (due to physical faults, but the same approach applies to hardware design bugs), but instead of observing hardware anomalies they detect anomalies at the software level. Our work differs from this work in anomaly detection by formally specifying correctness and dynamically verifying that specification, rather than observing an ad-hoc set of signals.

7. Conclusions

We have developed a framework for specifying a system's memory consistency at two important levels: PAMC and VAMC. We also analyzed how the AT system bridges the gap between PAMC and VAMC. Having a thorough, multi-level 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, including all 21 AT related bugs we identified in published errata.

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

Acknowledgments

This work has been supported in part by the Semiconductor Research Corporation under contract 2009-HJ-1881 and the National Science Foundation under 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.

References

[1] Advanced Micro Devices. Revision Guide for AMD Athlon64 and AMD Opteron Processors. Publication 25759, Revision 3.59, Sept. 2006.

[2] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[3] S. V. Adve, V. S. Pai, and P. Ranganathan. Recent Advances in Memory Consistency Models for Hardware Shared Memory Systems. *Proc. of the IEEE*, 87(3):445–455, Mar. 1999.

[4] AMD. Revision Guide for AMD Family 10h Processors. Technical Report 41322, Sept. 2008.

[5] AMD. Revision Guide for AMD Family 11h Processors. Technical Report 41788, July 2008.

[6] Arvind and J.-W. Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *Proc. of the 33rd Annual Int'l Symposium on Computer Architecture*, June 2006.

[7] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. of the 32nd Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[8] M. Blum and S. Kannan. Designing Programs that Check Their Work. In *ACM Symposium on Theory of Computing*, pages 86–97, May 1989.

[9] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proc. of the Conference on Programming Language Design and Implementation*, June 2008.

[10] H. W. Cain and M. H. Lipasti. Verifying Sequential Consistency Using Vector Clocks. In *Revue in conjunction with Symposium on Parallel Algorithms and Architectures*, Aug. 2002.

[11] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Dynamic Verification of Cache Coherence Protocols. In *Workshop on Memory Performance Issues*, June 2001.

[12] K. Chen, S. Malik, and P. Patra. Runtime Validation of Memory Ordering Using Constraint Graph Checking. In *Proc. of the Thirteenth Int'l Symposium on High-Performance Computer Architecture*, Feb. 2008.

[13] K. Chen, S. Malik, and P. Patra. Runtime Validation of Transactional Memory Systems. In *Proc. of the Int'l Symposium on Quality Electronic Design*, Mar. 2008.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[15] M. D. Hill, A. E. Condon, M. Plakal, and D. J. Sorin. A System-Level Specification Framework for I/O Architectures. In *Proc. of the Eleventh ACM Symposium on Parallel Algorithms and Architectures*, June 1999.

[16] IBM. IBM PowerPC 750FX and 750FL RISC Microprocessor Errata List DD2.X, version 1.3, Feb. 2006.

[17] Intel Corporation. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, Oct. 2002.

[18] Intel Corporation. Intel Pentium 4 Processor Specification Update. Document Number 249199-065, June 2006.

[19] Intel Corporation. Intel Core Duo Processor and Intel Core Solo Processor on 65nm Process Specification Update. Technical Report 309222-016, Feb. 2007.

[20] Intel Corporation. Intel Core2 Extreme Quad-Core Processor QX6000 Seuqnce and Intel Core2 Quad Processor Q6000 Seuqnece Specification Update. Technical Report 315593-021, Feb. 2008.

[21] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[22] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, Sept. 1979.

[23] M.-L. Li et al. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Proc. of the Thirteenth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[24] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[25] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proc. of the 32nd Symposium on Principles of Programming Languages*, Jan. 2005.

[26] M. M. Martin et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.

[27] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proc. of the 30th Annual Int'l Symposium on Computer Architecture*, June 2003.

[28] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proc. of the 40th Annual Int'l Symposium on Microarchitecture*, Dec. 2007.

[29] A. Meixner and D. J. Sorin. Dynamic Verification of Sequential Consistency. In *Proc. of the 32nd Annual Int'l Symposium on Computer Architecture*, June 2005.

[30] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, June 2006.

[31] A. Meixner and D. J. Sorin. Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures. In *Proc. of the Twelfth Int'l Symposium on High-Performance Computer Architecture*, Feb. 2007.

[32] S. Narayanasamy, B. Carneal, and B. Calder. Patching Processor Design Errors. In *Proc. of the Int'l Conference on Computer Design*, Oct. 2006.

[33] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proc. 29th Annual Int'l Symposium on Computer Architecture*, May 2002.

[34] V. K. Reddy and E. Rotenberg. Coverage of a Microarchitecture-level Fault Check Regimen in a Superscalar Processor. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, June 2008.

[35] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy. UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All. In *Proc. Fifteenth Int'l Symposium on High-Performance Computer Architecture*, Jan. 2010.

[36] S. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware. In *Proc. of the 39th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, Dec. 2006.

[37] D. J. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proc. of the 29th Annual Int'l Symposium on Computer Architecture*, May 2002.

[38] P. J. Teller. Translation-Lookaside Buffer Consistency. *IEEE Computer*, 23(6):26–36, June 1990.

[39] I. Wagner, V. Bertacco, and T. Austin. Shielding Against Design Flaws with Field Repairable Control Logic. In *Proc. of the Design Automation Conference*, July 2006.

[40] D. L. Weaver and T. Germond, eds. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[41] A. Wolfe. AMD's Quad-Core Barcelona Bug Revealed. *InformationWeek*, December 11 2007.