Technical Report Duke-ECE-2009-1, May 2009

Proving the Completeness of the Composition of Two Dynamic Verification Techniques

Michael E. Bauer, Albert Meixner, and Daniel J. Sorin
Duke University

Abstract

There has been a significant amount of recent research in low-cost mechanisms for detecting errors in computer execution that are due to hardware faults. One exciting, low-cost approach to error detection is dynamic verification (sometimes also called online testing or runtime invariant checking). The idea is for the hardware to dynamically (i.e., at runtime) check whether certain necessary invariants are being maintained. In this paper, we prove that a combination of two previously developed dynamic verification schemes—the Argus scheme for processor cores and the dynamic verification of memory consistency (DVMC) scheme for the memory system—provides complete error detection for a multi-core processor chip. Both Argus and DVMC have been proved to be complete for their respective portions of the chip, but it is not obvious that the composition of these two DV schemes is complete. We show that the composition of these two DV schemes detects all possible errors, and thus we show that the interface between Argus and DVMC does not have any gaps through which errors could slip undetected. This proof provides chip designers with a formal guarantee of error detection capability, and such guarantees are often required if a chip is to meet a desired reliability or availability goal.

1. Introduction

There has been a significant amount of recent research in low-cost mechanisms for detecting errors in computer execution that are due to hardware faults. Error detection is important because it is the first step in fault tolerance—a system cannot recover from an error it does not detect—but providing error detection cannot incur large costs for commodity computer hardware. The traditional approach of macro-scale redundancy, such as dual modular redundancy (DMR), is not cost-effective in this market.

One exciting, low-cost approach to error detection is *dynamic verification* (sometimes also called online testing or runtime invariant checking). The idea is for the hardware to dynamically (i.e., at runtime) check whether certain necessary invariants are being maintained. By checking invariants rather than components, dynamic verification (DV) offers two primary advantages over the typical approach

of composing multiple per-component error detection mechanisms. First, DV is often less expensive, in terms of hardware. Second, DV often facilitates formal evaluation of its error coverage.

DV has been developed in several contexts. The pioneering work in DV was DIVA [3]. DIVA dynamically verified a large, complex superscalar processor core by adding a small, simple, but functionally equivalent checker core to the superscalar core's commit pipeline stage. The checker core is provably equivalent to the superscalar core in the absence of errors and, thus, discrepancies between their states reveal errors.

Since DIVA, other DV schemes have been developed for various parts of the computer. There have been schemes to dynamically verify the core, including dynamic dataflow verification [meixner:ddfv:pact:2007] and Argus [8]. Other schemes have been developed to dynamically verify the memory system, including schemes that dynamically verify cache coherence [5], the memory consistency model [4, chen:dvmc:hpca:2008, 9, 10], and transactional memory consistency [chen:dvtm:isqed:2008]. Our goal is to prove that a combination of dynamic verification schemes can provide complete error detection for a multicore processor chip. Such a proof would provide chip designers with a formal guarantee of error detection capability, and such guarantees are often required if a chip is to meet a desired reliability or availability goal.

In this paper, we focus on the composition of two particular DV schemes: the Argus DV scheme for processor cores [8] and dynamic verification of memory consistency (DVMC) for memory systems [9, 10]. Both Argus and DVMC have been proved to be complete for their respective portions of the chip, but it is not obvious that the composition of these two DV schemes is *complete*. We want to show that the composition of these two DV schemes detects all possible errors, and thus we must show that the interface between Argus and DVMC does not have any gaps through which errors could slip undetected.

The rest of this paper is as follows. In Section 2, we provide background on Argus and DVMC. In Section 3, we describe our system model and the assumptions we make. In Section 4, we present our proof of completeness. In Section 5, we draw conclusions from this work.

2. Background on the Two DV Techniques

In this section we discuss Argus and DVMC. Because those two DV schemes have already been published, we only provide high-level overviews here.

2.1. Argus

Meixner et al. developed Argus as a DV scheme for processor cores [8]. It is tailored for simple, in-order cores, but the idea applies to any von Neumann core. The key idea is that the behavior of a von Neumann core consists of only four tasks: selecting instructions to execute (control flow), performing the computation for each instruction, routing the results of producer instructions to the inputs of consumer instructions (dataflow), and interacting with memory.[1] An implementation of Argus thus must include four checkers: Control Flow Checker, Computation Checker, Dataflow Checker, and Memory Checker.

Meixner et al. proved that a core with an ideal checker for each task would detect all possible errors within the core (with a few exceptions we mention at the end of this section). Although ideal checkers are not always feasible or cost-effective, Meixner et al. also proved that at least one specific set of existing checkers was equivalent to ideal checkers, except for certain implementation-specific restrictions. Most importantly, several of the checkers use fixed-length signatures, which can be shown to be equivalent to ideal checkers only in the absence of signature aliasing. In an actual checker implementation, there is always a non-zero probability of aliasing, which can cause errors to go undetected.

Argus is not just a theoretical framework. Meixner et al. designed a prototype at the logic gate level and showed that it did indeed detect injected errors in a simple processor core (the OpenRISC 1200 [7]).

Limitations. Argus ignores the issues of address translation (from virtual to physical), exceptions/interrupts, and I/O; errors in these activities may go undetected by Argus. Argus does not apply to exotic, non-von Neumann machines, such as dataflow computers [2]. Although Argus ensures the correctness of a uniprocessor memory system—in the context of a single core, each load will obtain the value of the most recent store to the same address—it makes no such guarantee for multiprocessor memory systems.

2.2. Dynamic Verification of Memory Consistency (DVMC)

The memory system of a multicore chip includes caches, memory, coherence controllers, and the interconnection network. The memory system is large and complicated, but it fortunately has a formal specification of its correct behavior: the memory consistency model [1]. The memory consistency model specifies the legal software-visible orderings of the loads and stores performed by the threads

1. The core's interactions with memory include computing memory addresses and communicating addresses and data to the cache.

of a given process. Sequential consistency (SC), for example, specifies that there should appear to be a total order of all loads and stores, such that each load obtains the value of the most recent (in the total order) store to the same address [6]. Because memory consistency defines correctness, a mechanism that dynamically verifies memory consistency also completely detects errors in the memory system.

Meixner and Sorin developed Dynamic Verification of Memory Consistency (DVMC) to provide error detection for the memory system [9, 10]. DVMC works by dynamically verifying three sub-invariants which the authors proved to be sufficient for memory consistency. An implementation of DVMC includes a hardware checker for each of these invariants.

- Uniprocessor Ordering: Every core should behave like a single-core system unless a shared memory location is accessed by another core.
- Allowable Reordering: A subset of memory operations may be allowed to perform globally in an order different from program order, but only when permitted by the specific memory consistency model.
- Cache Coherence: A memory system is coherent if all processors observe the same history of values for a given memory location. DVMC further requires that the system observes the Single-Writer/Multiple-Reader (SWMR) property. This requirement is stronger than coherence, but virtually all coherence protocols use SWMR to ensure coherence.

Many memory consistency models exist, and the DVMC framework is parameterizable—by choosing which reorderings are detected as violations by the Allowable Reordering Checker—such that it can be applied to all commercially existing consistency models, including the consistency models specified for computers made by Intel (including IA-32 and IA-64), AMD, IBM, and Sun.

Meixner and Sorin proved that a DVMC implementation with ideal checkers for each invariant completely detects errors in the memory system. They developed ideal checkers for the first two invariants, but there have not yet been any practical schemes for ideally checking the Cache Coherence invariant. The most efficient checker for this invariant is called Token Coherence Signature Checking (TCSC) [meixner:tcsc:hpca:2007], and it uses fixed-length signatures to represent histories of coherence events. Similar to the case for Argus, DVMC implementations may encounter some signature aliasing and thus have a non-zero probability of not detecting an error.

Limitations. DVMC does not consider address translation or uncacheable memory accesses (including memory-mapped I/O). DVMC does not apply to memory systems that do not provide hardware for

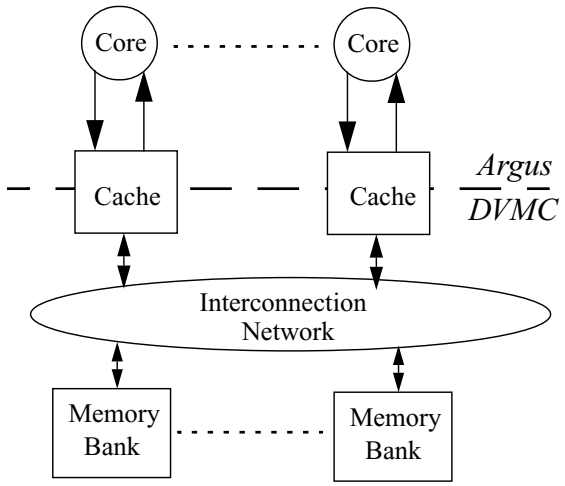

Figure 1. Multicore System Model

cache-coherent shared memory (e.g., Cray supercomputers or pure message-passing machines), but such systems comprise an extremely small fraction of commodity computers.

2.3. Interface Between Argus and DVMC

The interface between Argus and DVMC corresponds to the interface between the cores and the memory system, as illustrated in Figure 1. This interface exists at the highest level of the cache hierarchy (i.e., the L1 cache). Each core issues loads and stores to the memory system, oblivious of the implemented memory hierarchy, and expects to receive data in response to loads. In a single-core processor, each load returns the value of the most recent store by that core to the same address. In a multicore, a load may also return a value that was stored by another core, depending on the specific memory consistency model.

3. System Model and Assumptions

Our model of a multicore processor consists of the following components:

- some number of single-threaded cores, *C*, with von Neumann architectures,
- a shared memory system consisting of one main memory (perhaps implemented with multiple banks) and one cache associated with each core, and
- an interconnection network that connects all of the cores.

System Model Assumptions. We assume that the memory system uses a single-writer, multiple-reader (SWMR) cache coherence protocol. All addresses in the memory system are physical (no

memory address translation or TLBs). We could add specialized hardware to the system in order to implement any memory consistency model including relaxed models (e.g., by adding write buffers that support performing writes out of order). In this paper, however, we use a minimal processor model in order to limit the number of assumptions regarding the hardware. We do not consider I/O operations.

Without loss of generality, we assume that a program has *C* threads, each of which runs on its own single-threaded core (i.e., there is exactly one thread per core). We do not consider multiprogrammed workloads or context switches.

Definition. We define a *memory access* to be either a read or write request for an address issued by a core to its cache.

Definition. We define the value of a cache block at address A to be *current* if the value is equivalent to the most recent store performed to address A.

4. Proof of Completeness

The main idea behind this proof is to recognize that the caches represent the only place where Argus and DVMC overlap. We will now demonstrate that by using the caches as the interface between the two checking systems we can comprehensively check (i.e., detect all errors in) our model of a multicore.

Theorem. The combination of Argus and DVMC, *given ideal checkers for both of them*, provides comprehensive error detection for a multicore processor (under the assumptions in Section 3).

We prove this theorem by using induction on memory accesses. We begin by selecting one thread of the *C* executing on the multicore processor, and we use induction on the memory accesses issued to the caches. The induction step will be proved in two steps, with the first step making use of three lemmas to simplify the process.

Base Case: We assume that the initial configuration of the multicore processor is checked using a checksum. We now consider the execution of program P up until P's first memory request. Using the control flow, data flow, and computational checking invariants provided by Argus, the processor checks the execution of P until the first memory access.

Induction Case: We assume that we check the execution up to the n^{th} memory access. We now want to show that we check the n^{th} memory access and the execution up until the $(n+1)^{th}$ memory access.

Step 1: Show that we check the n^{th} memory access.

We do this in three parts:

1. Show that we check the core's specifications of memory accesses to its cache (Lemma 1).
2. Show that we check that the values in the caches are current (Lemma 2).
3. Show that we check the interactions between the core and memory system through the cache (Lemma 3).

Lemma 1. Checking the execution of a program between memory accesses is sufficient for checking that memory accesses are issued correctly.

Proof. The induction case ensures that the execution up to the n^{th} memory access is checked. A combination of three of Argus's four checkers (data flow, control flow, computation) check that the n^{th} memory access is issued correctly. That is, we check that all of the fields in the memory access instruction are correct and will be sent to the cache correctly.

Lemma 2. Checking SWMR cache coherence is sufficient for checking that all cache values are current.

Proof. The SWMR cache coherence protocol enforces the policy that only one processor will ever write to an address at a time, and this invariant is checked by DVMC. Therefore, whenever a write occurs, DVMC checks that the write does indeed overwrite all other copies of that value within the system. We thus check that any read occurring after a write will see the current value of that address.[2]

Lemma 3. Given Lemma 1 and Lemma 2, the combination of DVMC's Uniprocessor Ordering Checker and Allowable Reordering Checker is sufficient for checking that memory accesses are performed correctly for the given memory consistency model.

Proof. We demonstrated in Lemma 1 that we check all memory accesses issued to the cache. We know from Lemma 2 that we check that the values in the cache are current. We therefore know that Argus's Memory Checker will check that the value returned from the cache is current (and is the value corresponding to the requested address) and that the memory access commits correctly. However, we

2. It might appear that a physical error could corrupt a cache block and make its value non-current. However, checking cache coherence is sufficient to detect this scenario. DVMC's Cache Coherence Checker uses an error detecting code on cache blocks to detect this violation of the Cache Coherence invariant.

have yet to demonstrate that we check that the n^{th} memory access will perform correctly *with respect to all other memory accesses*.

This proof has three facets. First, DVMC's Uniprocessor Ordering Checker checks that the n^{th} memory access will perform correctly with respect to all other memory accesses to the same address being issued from core *C*. Second, we know from Lemma 2 that we check that the n^{th} memory access will perform correctly with respect to all other memory accesses to the same address that are *not* issued by core *C*. Third, the Allowable Reordering Checker checks that the n^{th} memory access performs correctly with respect to all other memory accesses that are not to the same address as the n^{th} memory access. We can therefore conclude that the n^{th} memory access will perform correctly with respect to all other memory accesses. We have thus demonstrated that the n^{th} memory access both commits and performs correctly.

Step 2: Show that we check the execution from after the n^{th} to before the $(n+1)^{th}$ memory access.

We have shown that we check the n^{th} memory access. We now employ the previous argument that three of Argus's four checkers (data flow, control flow, and computation) check the execution between the n^{th} and $(n+1)^{th}$ memory accesses. We have therefore shown that execution from before the n^{th} to before the $(n+1)^{th}$ memory access is checked and thereby have satisfied the inductive hypothesis.

We have shown that we check that a program P executes correctly on the multicore processor. Because our choice of programs was arbitrary, we have therefore shown that all programs on the multicore are checked.

End Proof.

5. Conclusions

In this paper, we have proved that the combination of two previously developed dynamic verification schemes can provide complete error detection for a multicore processor (excluding aspects such as address translation and I/O). This proof provides an important guarantee to architects who are considering the use of dynamic verification. Without such a proof, the risk of undetectable errors is likely to be too great, and the architects are likely to use more traditional error detection schemes, even if those schemes have greater hardware and power costs.

This proof does not consider certain aspects of systems—including address translation and I/O—and our plans for the future involve developing dynamic verification schemes for these aspects. Fur-

thermore, once these schemes have been developed, we plan to prove that composing them with Argus and DVMC provides complete error detection for the entire multicore chip.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant CCR-0444516. We thank Anita Lungu for feedback on this work.

References

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

[2] K. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[3] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[4] 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.

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

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

[7] D. Lampret. OpenRISC 1200 IP Core Specification, Rev. 0.7. http://www.opencores.org, Sept. 2001.

[8] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 210–222, Dec. 2007.

[9] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 73–82, June 2006.

[10] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. *IEEE Transactions on Dependable and Secure Computing*, 5(2), April-June 2008.