Architectural Trace-Based Functional Coverage for Multiprocessor Verification

Biruk Mammo[†], Jim Larimer[‡], Matthew Morgan[‡], Dave Fan[‡], Eric Hennenhoefer[‡] and Valeria Bertacco[†]
[†]University of Michigan
{birukw,valeria}@umich.edu
[‡]ARM Limited
{first.last}@arm.com

Abstract—**Functional coverage plays a pivotal role in assuring the quality of input stimuli used in the verification of modern digital designs. For an out-of-order multi-processor design, simulation of a detailed model of the design is often required to observe relevant design behaviors for functional coverage. However, since such a model is not available during the early phases of test development, verification teams are forced to wait until much later in the verification process to evaluate the quality of their test cases. Even then, the quality of the tests can be assured only on one specific design implementation - an undesirable characteristic for test and regression suites that are meant to be used across multiple generations and/or implementations of an architecture. This work addresses this issue by presenting a novel, implementation-independent, execution trace-based, coverage collection solution. Our solution enables the early evaluation of multi-processor tests using a high-level model of a design. In addition, it can be deployed with detailed design models, if desired, for further analysis alongside implementation-specific coverage models.**

I. Introduction

Simulation-based functional coverage metrics are widely deployed in the verification of processor designs. They provide insights into the effectiveness of test stimuli in exercising the critical aspects of a design. Such insights, in turn, enable the development of high quality tests that can help meet the verification goals faster.

It is desirable for industrial verification frameworks to be usable across multiple implementations of an architecture. Framework re-use minimizes the verification effort and shortens the time to market. Among other things, implementation-independent coverage models are required to enable the creation of high quality, re-usable test cases. Developing such coverage models is quite challenging, especially for shared memory multi-processor architectures that define weakly ordered memory consistency models, such as ARM [1] and PowerPC [2]. In order to adequately verify weakly ordered designs, it is necessary to simulate several test cases that target the memory interactions specified by the architecture. Certain design behaviors that need to be monitored for evaluating these test cases, such as the reordering of memory operations and the propagation of barriers, are unfortunately implementation-dependent [3]. In addition, some of these design behaviors are not visible at the architectural level, regardless of the level of abstraction of the simulated design model. This limits the amount of coverage data that can be collected at an architectural level.

model	example	accuracy	portability	availability
architectural	ISS	low	high	early
performance	gem5 [4]	medium	low	later
implementation	RTL	high	low	latest

TABLE I: **Properties of classes of design abstractions**. A design under verification can be described at various abstraction levels, ranging from instruction set simulators (ISS), to performance models and behavioral register transfer level (RTL) descriptions. Coverage based on these abstractions varies in accuracy, portability, and how soon in the verification process it can be measured.

Coverage models based on high-level design models are attractive because they are available early in the verification process and can simulate faster, allowing for quick turn-around times when evaluating test cases. However, high-level design abstractions do not model the complex micro-architectural interactions that produce the desired design behaviors, further inhibiting coverage collection. Table I summarizes the relative properties of the different design abstractions available for use with multi-processor coverage models. Note that an instruction set simulator (ISS), which is usually the earliest model available, allows for the most portable, yet least accurate multi-processor coverage.

We propose an architectural coverage collection mechanism that provides early feedback on multi-processor test quality using high-level design models. As verification progresses, it can be deployed with more detailed models of the design, allowing for further evaluation and refinement of test cases. Such a mechanism enables the development of good quality, re-usable test cases faster than otherwise possible, potentially speeding up the verification process.

Contributions

In this work, we present a novel methodology for a flexible architectural coverage collector that can detect weakly ordered multi-processor design behaviors. Our methodology tracks memory access patterns and control dependencies visible in multi-processor execution traces to identify memory ordering behaviors. It can operate on architectural events collected from any type of design model, enabling it to be used with an ISS to guide test creation early in the verification process. Even though it is impossible to accurately detect all weak ordering behaviors using only architectural traces, our methodology utilizes the available information to provide useful feedback that was otherwise unavailable.

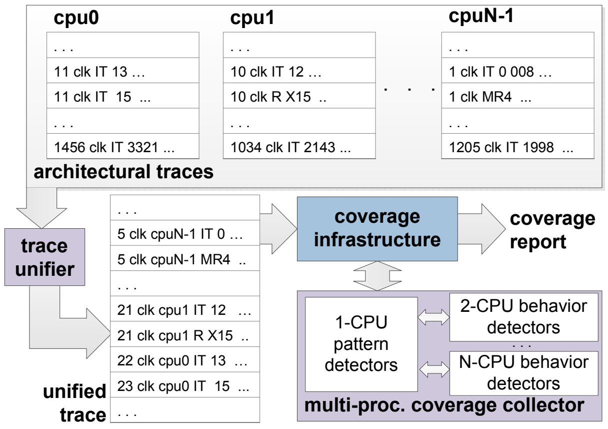


Fig. 1: **High-level overview of our methodology.** Each core logs relevant architectural events and produces execution traces in the tarmac trace format [5]. These are then serialized into a unified trace which is parsed by the coverage infrastructure. The multi-processor coverage collector contains coverage models that detect interesting multi-processing behaviors in the parsed trace events.

Our coverage collection mechanism, shown in Figure 1, is built upon an industrial SystemVerilog coverage infrastructure. We collect multi-processor execution traces from simulation and strip out implementation specific details, generating unified event traces for use with our mechanism. The goal of the coverage infrastructure is to parse execution traces and populate data structures; these data structures are then used by plug-in coverage models. Our multi-processor coverage model is composed of single- and multi-processor behavior detector modules, designed to scale to an arbitrary number of processors and behaviors. The multi-processor behavior detector modules utilize patterns already identified by single-processor detector modules. In the end, the coverage infrastructure collects the coverage statistics and generates a report for the user.

II. Background

In a typical processor verification effort, test programs are used to create sequences of a design's activities and behaviors that must be checked to meet the verification goals. Throughout the development, different design models become available, usually more and more detailed as time progresses. Coverage statistics measured during the simulation of these design models are used to evaluate how well the test programs create the desired design behaviors [6]. This feedback, in turn, allows for improvements in test quality; constrained-random test generators, if available, can be fine-tuned and new directed test programs can be created. In settings where test programs are developed independently from design implementations, it is desirable to have a functional coverage collection mechanism that is i) available early to enable the refinement of test programs before actual testing on an implementation begins, and is ii) portable to allow the tests and coverage models to be re-used across different design generations or implementations.

For single-processor designs, functional coverage models based on architectural execution traces meet these desirable traits. Architectural execution traces contain dynamic, non-speculative, instruction instances and their associated register and memory updates. These traces can be collected early on by simulating test programs on an ISS, and later on from behavioral register transfer level (RTL) simulations. Since all single-processor variants of an instruction set architecture (ISA) must guarantee a unique correct execution flow of a program, their architectural traces must be identical. Note that any discrepancy between the architectural traces indicates a functional error - a property commonly used to detect bugs.

However, for shared memory multi-processors, architectural traces collected from different models / implementations are not guaranteed to be identical. Micro-architectural features that vary among models / implementations, such as caches, affect the order in which different cores observe memory accesses. Multiple correct program executions, each with a different interleaving of memory accesses observed by the cores, are legal in this context. The discrepancy is even more pronounced in designs that implement weak memory models, as they are allowed to reorder certain memory operations more freely, unless restricted by order enforcing instructions [3]. It is possible for individual cores to observe memory interactions in different order, making it extra challenging to provide architectural trace-based functional coverage for weakly ordered multi-processor designs.

In addition to the discrepancy between traces and the inability to serialize architectural events, certain design behaviors may only be observable at the micro-architectural level. For example, in an implementation of the ARM ISA [1], barriers, atomic accesses through exclusive loads and stores, and cache and TLB maintenance instructions may all trigger micro-architectural events that affect multiple cores. As illustrated in Figure 2, not all of these events are readily visible architecturally: i) a micro-architectural (μ-arch) event may be completely hidden, ii) others may manifest out-of-order, iii) a μ-arch event may have a delayed manifestation, iv) some may manifest coalesced, and v) non-concurrent μ-arch events may manifest concurrently.

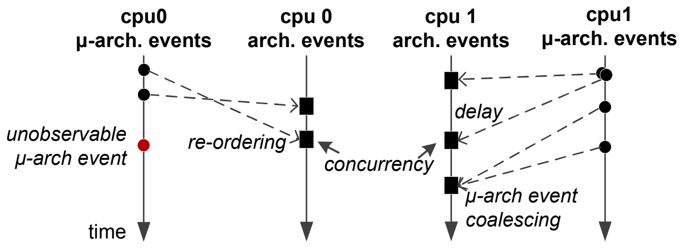


Fig. 2: **Event visibility**. Not all micro-architectural events have observable architectural effects. Those that do, may appear out of order, concurrently or coalesced with other micro-architectural events or at a far later time than their actual occurrence.

III. Related Work

Functional coverage metrics [6], [7] are widely used in the verification of processors, providing feedback on the (un)tested aspects of a design. This feedback is often used by verification engineers to direct testing, either by crafting tests or tuning random test generators [8] to "plug" coverage holes. Several solutions, under the name *coverage-directed test generation (CDG)*, have been proposed to automate the feedback process. These solutions mainly focus on ways to transform the coverage information into constraints / biases / directives for the random test generator. These include model checking ([9], [10]) and machine-learning ([11], [12], [13], [14]) based approaches. All of these, however, are geared towards single processors and do not present effective ways for collecting coverage for shared memory multi-processors.

Solutions for generating multi-processor tests have focused on exercising cache coherence state transitions by monitoring state coverage [15], [16], or by introducing memory access collisions in independent, randomly generated instruction streams [17]. This latter approach is used in commercial test generators [18]. Small tests of specially crafted instruction sequences, known as litmus tests, are widely used in the verification of weakly ordered systems [19], [20], [21], [22]. Random generators focusing on tests for memory consistency have also been proposed [23]. Such tests can benefit from a coverage model like ours.

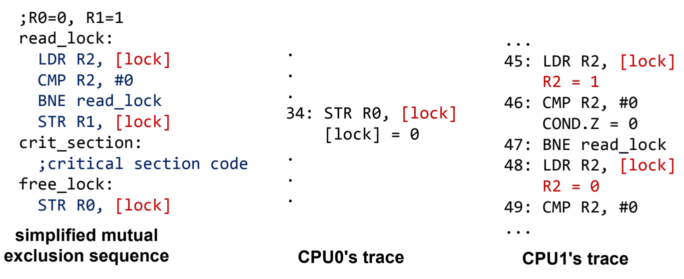
Partial orders of multi-processor events, constructed from memory accesses and data dependencies, have been used to reason about weakly consistent memory systems [24], [21] and verify their implementations [25], [26], [27], [28]. Our solution uses a similar approach to establish ordering relationships between observed multi-processor events, adapted to the constraints of architectural coverage collection.

IV. Methodology

Our coverage collection mechanism operates on execution traces collected during simulation. Our evaluation framework consists of an ISS and a behavioral RTL description for an ARM processor design. The two models are instrumented to generate execution traces in the tarmac trace format [5]. We only consider the architecturally-visible parts of the execution trace, namely the dynamic, non-speculative instruction traces, register updates and memory accesses. A single instruction and its corresponding state updates form a *trace event*. For the ISS we use, it is possible to obtain a single execution trace file containing an ordered stream of trace events from all cores being simulated. Since the ISS does not model micro-architectural timing, each dynamic instruction is simulated atomically in one cycle. For performance reasons, the ISS simulates instruction streams in batches. It schedules batches from multiple cores in a round-robin fashion. Our RTL simulations produce multiple execution trace files, one for each core. In both cases, a simulation cycle count is associated with each trace event.

Since our solution must be independent of the design model used to generate the traces, we have to abstract away the model specific details. To this end, we first "flatten" the execution traces that we obtain for each simulation into a unified trace containing a pseudo-ordered sequence of trace events. Program order (instruction count) is used to infer the order among intra-core events. Moreover, additional information from the simulator, such as the simulation cycle counts, are used as timing hints to order inter-core events. For trace-events from multiple cores that occur concurrently in the original RTL traces, we simply assign order in a round-robin fashion.

The flattening process eliminates any model-specific trace artifact but does not guarantee a correct logical ordering among trace events from multiple cores. Instead of relying on the synthetic order that we create in the unified trace, our coverage models infer inter-core ordering more reliably from shared memory access patterns and the outcome of branches dependent on loads to shared memory addresses. To illustrate with an example, consider the two processor, simplified mutual exclusion instruction sequence shown in Figure 3. In the unified trace generated from two processors running this instruction sequence, memory accesses to a common address by the two processors point us to the lock address. Multiple dynamic instances of a load issued to the lock address by CPU1, always followed by a compare and a branch dependent on the loaded value, identify CPU1's spin-loop. This pattern is broken when CPU1 gets out of its spin-loop, which can only happen after CPU0 writes a 0 to the lock address. Therefore, this property allows our system to logically order CPU0's write to the lock address (trace event 34) in-between two of CPU1's reads from the lock address (trace events 45 and 48). Note that when searching for patterns, events that occur close to each other in time are likely to be located closely to each other in the unified trace. Moreover, due to implementation constraints such as number of cores and sizes of instruction windows in the cores, the lag between the occurrence of micro-architectural events and their manifestation at the architectural level is bounded. These observations enable our coverage collection mechanism to limit the search for interesting patterns to a bounded interval window of events, which a user can configure based on the design characteristics.



```
;R0=0, R1=1
read_lock:
  LDR R2, [lock]
  CMP R2, #0
  BNE read_lock
  STR R1, [lock]
crit_section:
  ;critical section code
free_lock:
  STR R0, [lock]
```

```
.
.
.
34: STR R0, [lock]
    [lock] = 0
.
.
.
```

```
...
45: LDR R2, [lock]
    R2 = 1
46: CMP R2, #0
    COND.Z = 0
47: BNE read_lock
48: LDR R2, [lock]
    R2 = 0
49: CMP R2, #0
...
```

Fig. 3: **Ordering inference example.** Both processors execute the simplified mutual exclusion code shown on the left side. When CPU0 leaves its critical section, its 34th dynamic instruction releases the lock. Values loaded by CPU1's 45th and 48th dynamic instructions allow us to infer that CPU1 observed CPU0's store somewhere in between the two loads. *Note that this simplified code sequence does not guarantee mutual exclusion in weakly ordered systems.*

Our goal is to detect weak ordering behaviors that manifest at the architectural level. Our implementation is based on test sequences specially designed to trigger such relevant behaviors [22]. We analyze these test sequences to identify how the behaviors manifest and implement flexible and portable mechanisms to detect them. This is achieved by first identifying single-processor patterns from the tests, and then modelling multi-processor behaviors as ordered instances of patterns from multiple processors. For the mutual exclusion instruction sequence in Figure 3, for example, we can identify three patterns: *wait on lock*, *acquire lock*, and *release lock*. Manifestations of two-processor mutual exclusion behavior are then composed of a sequence of CPU0's and CPU1's single-processor patterns. For example, one manifestation of the behavior *"CPU1 attempts to acquire the lock while CPU0 is in its critical section"*, can be captured by the following sequence: $cpu0.wait \rightarrow cpu0.acquire \rightarrow cpu1.wait \rightarrow cpu1.wait$. We allow for a bounded number of trace-events, as specified by the user-configured interval window, to appear in between two consecutive patterns in a behavior.

We implement single-processor pattern detectors and multi-processor behavior detectors using the SystemVerilog `sequence` construct [29]. The `sequence` construct is part of the SystemVerilog assertion library and provides a robust regular expression-like syntax for describing sequences of events. We set unique flags for every pattern that we detect and save relevant information, such as a lock address, which are later used by the multi-processor behavior detectors. We define a library of macros and functions to simplify the expression of new patterns and behaviors in our coverage model, to a degree where the process can be automated. Our coverage infrastructure collects functional coverage using the SystemVerilog `covergroup` construct [29], which allows us to encapsulate our coverage model. Our multi-processor `covergroup` samples *flag bitmasks* constructed from the flags set by our multi-processor behavior detectors. `coverpoint`s inside our `covergroup` accumulate statistics for sampled flags, which are reported to the user at the end of simulation.

V. Preliminary Results

We conducted experiments to test the behavior detection capability of our coverage collection mechanism. We developed detectors for several of the behaviors specified in [22]. These behaviors are defined as multi-processor instruction / event sequences with explicitly specified address and data dependencies. We developed a test suite of ARM assembly programs for a total of 19 variants of these behaviors. A summary of the behaviors we tested is given in Table II. Detailed discussions of most of these variants, complete with ARM assembly sequences, can be found in [22].

We ran our test suite on the ISS and behavioral RTL models and collected execution traces. We had three sets of traces, from i) the ISS configured with a large batch-size of 150 - 200 instructions, ii) the ISS configured with a batch-size of 1 instruction, and iii) the RTL model. The large batch-size ISS traces represent the most unfavorable inputs for our solution.

behavior	**variants**	**no. of cores**
Simple Weakly Consistent Ordering	1	2
Message Passing	8	2-3
Causal Consistency	2	3
Post Store before Polling for Ack.	2	2
WFE and WFI and Barriers	1	2
Acquiring and Releasing a Lock	4	2
Mailbox to Send an Interrupt	1	2

TABLE II: **Summary of behaviors we tested**.

The execution traces for each simulation were "flattened" and then processed by our coverage collection mechanism, as described in Section IV. For each set of traces, we tweaked the interval windows of our coverage models based on batch-size for the ISS traces and design characteristics for the RTL traces. For all three sets of traces, our coverage models were able to detect the occurrences of all the behaviors we tested.

VI. Conclusions and Future Work

We presented a flexible multi-processor architectural coverage collection mechanism that can be used with different models of an architecture, from architectural to register-transfer level. We have shown that our solution can be used to detect the occurrence of relevant weak ordering behaviors that have architectural manifestations. Even though our solution sacrifices accuracy for implementation-independence, it enables the development of high quality and flexible regression suites much sooner than currently possible. Future work will attempt to improve the quality of results by collecting and integrating coverage from multiple simulations with different implementation-specific configuration settings. By analyzing results for a wide range of configurations, better insights could be obtained on the portability of test cases.

References

[1] ARM Ltd., "ARM architecture reference manual ARMv7-A and ARMv7-R edition," 2012. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html

[2] IBM, "Power ISA version 2.06 revision B," July 2010. [Online]. Available: https://www.power.org/documentation/power-isa-version-2-06-revision-b/

[3] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, 2011.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[5] ARM Ltd., "Fast models tarmac trace user guide," May 2012. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0532d/index.html

[6] A. Piziali, "Coverage-driven verification," in *Functional Verification Coverage Measurement and Analysis*. Springer US, 2004.

[7] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *Design & Test of Computers, IEEE*, vol. 18, no. 4, jul/aug 2001.

[8] Obsidian Software Inc., "Raven: Product datasheet." [Online]. Available: http://www.obsidiansoft.com/pdf/Datasheet.pdf

[9] P. Mishra and N. Dutt, "Graph-based functional test program generation for pipelined processors," in *Proc. DATE*, 2004.

[10] P. Mishra and N. Dutt, "Functional coverage driven test generation for validation of pipelined processors," in *Proc. DATE*, 2005.

[11] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proc. DAC*, 2003.

[12] M. Bose, J. Shin, E. Rudnick, T. Dukes, and M. Abadir, “A genetic approach to automatic bias generation for biased random instruction generation,” in *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, 2001.

[13] H. Shen, W. Wei, Y. Chen, B. Chen, and Q. Guo, “Coverage directed test generation: Godson experience,” in *Proc. ATS*, 2008.

[14] C. Ioannides, G. Barrett, and K. Eder, “Introducing xcs to coverage directed test generation,” 2011.

[15] I. Wagner and V. Bertacco, “Mcjammer: adaptive verification for multi-core designs,” in *Proc. DATE*, 2008.

[16] X. Qin and P. Mishra, “Automated generation of directed tests for transition coverage in cache coherence protocols,” in *Proc. DATE*, 2012.

[17] A. Adir and G. Shurek, “Generating concurrent test-programs with collisions for multi-processor verification,” 2002.

[18] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, “Genesys-pro: innovations in test program generation for functional processor verification,” *Design & Test of Computers, IEEE*, vol. 21, no. 2, mar-apr 2004.

[19] S. Mador-Haim, R. Alur, and M. M. K. Martin, “Generating litmus tests for contrasting memory consistency models,” in *Proc. CAV*, 2010.

[20] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, “Litmus: running tests against hardware,” in *Proc. TACAS*, 2011.

[21] N. Chong and S. Ishtiaq, “Reasoning about the ARM weakly consistent memory model,” in *Proc. MSPC*, 2008.

[22] ARM Ltd., “Barrier litmus tests and cookbook,” November 2009. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/index.html

[23] E. Rambo, O. Henschel, and L. dos Santos, “Automatic generation of memory consistency tests for chip multiprocessing,” in *Proc. ICECS*, 2011.

[24] H. Cain, M. Lipasti, and R. Nair, “Constraint graph analysis of multi-threaded programs,” in *Proc. PACT*, 2003.

[25] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, “Tsotool: A program for verifying memory systems using the memory consistency model,” in *Proc. ISCA*, 2004.

[26] A. Meixner and D. Sorin, “Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures,” in *Proc. DSN*, 2006.

[27] K. Chen, S. Malik, and P. Patra, “Runtime validation of memory ordering using constraint graph checking,” in *Proc. HPCA*, 2008.

[28] A. DeOrio, I. Wagner, and V. Bertacco, “Dacota: Post-silicon validation of the memory subsystem in multi-core designs,” in *Proc. HPCA*, 2009.

[29] Accelera Organization, Inc., “SystemVerilog 3.1a language reference manual,” 2004. [Online]. Available: http://www.eda.org/sv/SystemVerilog_3.1a.pdf