Continual Hashing for Efficient Fine-grain State Inconsistency Detection

Jae W. Lee, Myron King
Computer Science and Artificial Intelligence Laboratory (CSAIL)
Massachusetts Institute of Technology
Cambridge, MA 02139
{leejw, mdk}@csail.mit.edu

Krste Asanović
Computer Science Division, EECS Department
University of California at Berkeley
Berkeley, CA 94720-1776
krste@eecs.berkeley.edu

Abstract

Transaction-level modeling (TLM) allows a designer to save functional verification effort during the modular refinement of an SoC by reusing the prior implementation of a module as a golden *model for state inconsistency detection. One problem in simulation-based verification is the performance and bandwidth overhead of state dump and comparison between two models. In this paper, we propose an efficient fine-grain state inconsistency detection technique that checks the consistency of two states of arbitrary size at sub-transaction (tick) granularity using* incremental hashes*. At each tick, the hash generates a signature of the entire state, which can be efficiently updated and compared. We evaluate the proposed signature scheme with a FIR filter and a Vorbis decoder and show that very fine-grain state consistency checking is feasible. The hash signature checking increases execution time of Bluespec RTL simulation by 1.2 % for the FIR filter and by 2.2 % for the Vorbis decoder while correctly detecting any injected state inconsistency.*

1 Introduction

Transaction-level modeling (TLM) is a promising strategy for designing complex systems-on-a-chip (SoCs) [10]. System designers separate communication (channels) from computation (units or modules) so that they can incrementally refine each module toward synthesizable RTL, while maintaining the same interfaces to neighboring modules. TLM also helps hardware and software developers interact with each other in the very early phases of design, reducing total system development time.

During the modular refinement process in TLM, designers can save effort for functional verification of a unit in at least two ways. First, they can reuse the verification environment surrounding the prior implementation of the unit [3]. The difference in the level of abstraction between two implementations of a unit is cleanly isolated by transaction-based interfaces. Hence, designers can verify the transaction-level behavior without writing new testbenches for the implementation under test. In addition, mixed-level transactional co-simulation can reduce simulation time. Second, designers can use the more abstract implementation as a "golden" model to check internal state consistency between the refined unit and the golden model at various sub-transaction checkpoints as they both execute a transaction. Once a commonality of data structures between the two implementations is identified, designers can simply install monitor points to dump out internal data of interest for consistency checking [8].

A transaction consists of a series of smaller operations, or *ticks* in our terminology. Pinpointing a tick that puts the candidate design into an inconsistent state against the golden model is invaluable for debugging, as otherwise designers may have to manually trace back through many ticks to determine the root cause of a bug. A simple dump-and-diff technique does not work well for such fine-grain state inconsistency detection because the performance and/or bandwidth overhead is proportional to the size of the checked state. Less frequent (coarse-grain) checking or checking only certain critical blocks reduces overhead but degrades temporal precision or coverage.

In this paper, we present an efficient fine-grain state inconsistency detection technique, which can check the con-

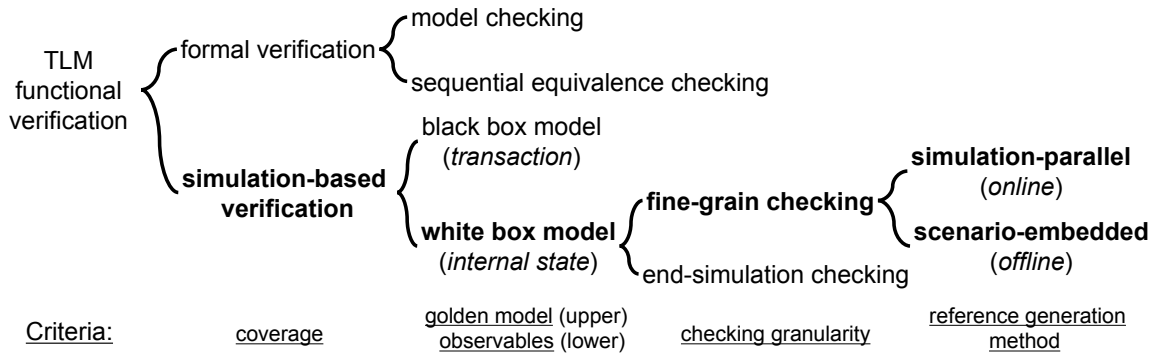


Figure 1. A taxonomy of TLM functional verification approaches. Our proposed approach is highlighted in boldface.

sistency of two states of arbitrary size at frequent checkpoints with reasonable simulation slowdown. Our technique minimizes the bandwidth overhead by having each implementation send out a hash signature as a compact summary of its entire state. We rely on *incremental hash functions* [5] to reduce performance overhead, since only a fraction of state elements are modified at each tick in most cases.

2 Related Work

Figure 1 shows a taxonomy of approaches for TLM functional verification, with our proposed approach highlighted in boldface.

The first criterion is the *coverage* of the verification technique. Formal verification aims to prove the correctness of a model, providing an exhaustive check for bugs. There are review papers that cover recent advances and research challenges in formal verification from the TLM perspective [4, 10, 13]. Formal verification methods are further divided into two classes according to how the golden reference is generated against which the design under test (DUT) is verified. In model checking, the designer is responsible for writing assertions to describe properties to be verified. In sequential equivalence checking, the golden model is a model at a higher level of abstraction. There are multiple notions of equivalence such as combinational equivalence, cycle-accurate equivalence, transaction equivalence, pipelined equivalence and stream-based equivalence [13]. However, formal verification approaches do not scale very well in general, so that they are not always applicable in practice.

Simulation-based approaches are generally not exhaustive, but can be applied to large designs. In many cases, both directed and constrained random tests are deployed to meet coverage goals. Wen et al. [7] discuss a systematic way to justify test patterns generated automatically. To ease RTL debugging, Hsu et al. [17] extract a behavioral model out of an RTL description combined with a simulation trace and provide an interface to query, trace and assign a value to an arbitrary node.

Simulation-based approaches are divided into black box and white box models, depending on what the verification process can observe. In the black box model, as in [3], only I/O behaviors (transactions) of the DUT can be observed outside. In the white box model, as in [8], peeking inside the DUT is also allowed to verify the consistency of internal states between models. Depending on how often consistency is checked, the white box model can be further divided into fine-grain and coarse-grain state checking. Finally, there are two methods to compare the internal states [10]: *simulation-parallel* comparison, where two models run in parallel as the golden model generates the golden state sequence (online), *scenario-embedded* comparison, where the golden state sequence is generated in advance and embedded in the test input a priori (offline).

Brier et al. [8] describe a verification methodology where they dump out all the intermediate values in two versions (C and Verilog) of a resize module for fine-grain state consistency checking, but this has high overheads in both bandwidth and storage space. We propose a technique for low-overhead state consistency checking without compromising the coverage or precision of inconsistency detection.

3 Fine-grain State Inconsistency Detection

In this section, we first introduce common terms and concepts, then describe a proposed functional verification framework, and finally compare three alternative approaches for state comparison.

3.1 Background

In comparing the internal states of two implementations of a module having the same transactional behavior, we assume a common specification of the module describing the

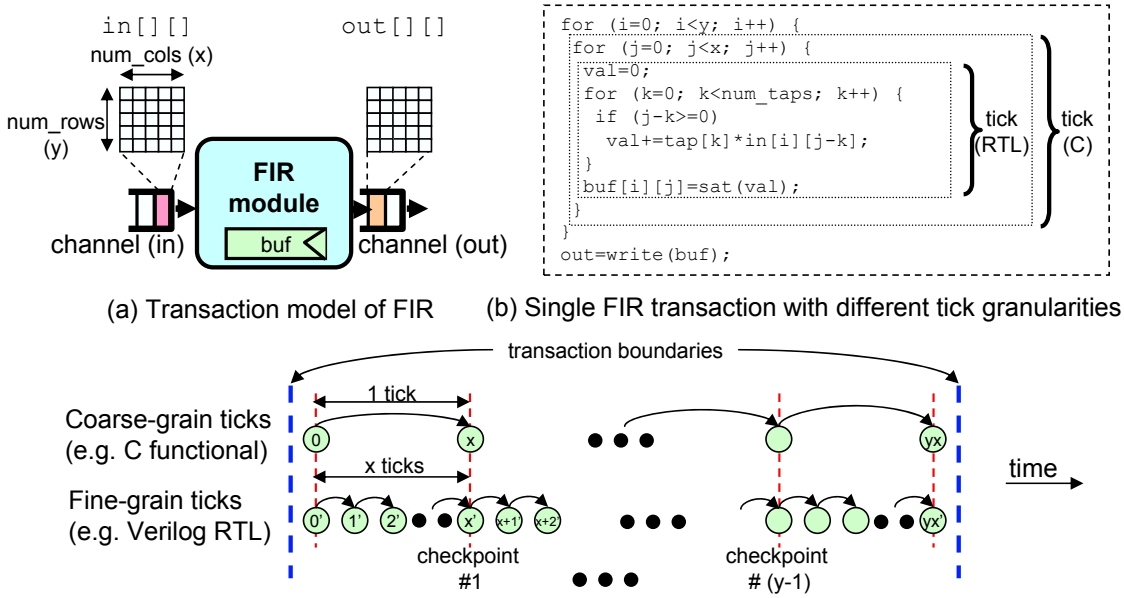


Figure 2. Transactional view of FIR filter module.

set of state elements that an outside block can observe or mutate. We call this externally-visible state the *architectural* state (AS). Each implementation may have a unique set of state elements not visible from outside, which constitute the implementation's *microarchitectural* state (μAS). In the rest of the paper, we focus only on verification of the architectural state (*AS*).

Ghenassia and Clouard define a *transaction* in TLM as "data transfer or synchronization between two modules at an instant determined by the hardware/software system specification" [10]. A module firing off a transaction usually steps through one or more unit operations, or *ticks*, which possibly put the module into an intermediate architectural state.

Figure 2 illustrates a transactional view of an FIR filter module which is similar to part of the resizer module in [8]. This module takes an input frame (`in[][]`) of `num_cols(x)` by `num_rows(y)` pixels, filters it and outputs the result through a channel. Two implementations are presented to illustrate different granularities of ticks. The implementation with coarse-grain ticks (e.g. C functional model) processes one row per tick while the implementation with fine-grain ticks (e.g. Verilog RTL) processes one pixel per tick. The common architectural state in this case is the input buffer (`in[][]`) and the output buffer (`out[][]`), whose intermediate states are shadowed by `buf[][]`. Note that for fine-grain transactions (or coarse-grain ticks) a tick might be equivalent to a transaction.

The figure also shows *checkpoints* where the two implementations are supposed to be in the same (intermediate) architectural state. Our technique is capable of detecting a state inconsistency at a checkpoint, which usually occur much more frequently than transactions.

Although we discuss our technique in the context of TLM, it can be applied to any pair of models for which we can identify common internal state and checkpoints.

3.2 Proposed Verification Framework

Procedure 1 State consistency checking (at n-th checkpoint)

Require: $AS_A[0] = AS_B[0]$
1: $State\_Info_A[n] = dump\_state\_info(AS_A[n])$
2: $State\_Info_B[n] = dump\_state\_info(AS_B[n])$
3: **if** $State\_Info_A[n] == State\_Info_B[n]$ **then**
4: **return** *STATE_CONSISTENT*
5: **else** // inconsistent state_info
6: **if** $AS_A[n] \neq AS_B[n]$ **then**
7: **return** *STATE_INCONSISTENT_ERROR*
8: **else**
9: **return** *STATE_INFO_ERROR*
10: **end if**
11: **end if**

Procedure 1 describes the state consistency checking for two implementations of a module, A and B, performed at every checkpoint. The two implementations do not have to execute at the same time. A stream of golden state information generated and stored offline can be reused for future implementations of a module.

Require: We assume that there exist a finite *aligning* sequence for each implementation that puts both imple-

mentations into the same initial architectural state [14]. We also assume that non-determinism in simulation is tightly controlled. Each instance of execution must preserve the order of checkpoints and the architectural state must be regenerated identically.

Lines 1-2: Both implementations dump out information on their current architectural state. This information could be the full architectural state (*AS*), a description of changes since the previous checkpoint (ΔAS), or a signature (*SIG*) for the state or its changes.

Lines 6-9: If state information mismatches, the checker performs a full-state checking. If the full states do not match, it returns STATE_INCONSISTENT_ERROR. Otherwise, it returns STATE_INFO_ERROR, which means that the state information was different for two identical architectural states.

The precision of state inconsistency detection possible in this framework is limited in temporal granularity by the model with the coarser-granularity tick, and limited in coverage to the common exposed architectural state.

Procedure 2 An example two-pass debugging procedure using two implementations of a unit, A and B

```
1:  for all (test, input) in a test suite do
2:    START :
3:    OUT_A = execute_A(test, input)
4:    OUT_B = execute_B(test, input)
5:    if OUT_A ≠ OUT_B then
6:      repeat // pass 1
7:        coarse_grain_check(A, B, test, input)
8:      until STATE_INCONSISTENCY detected
9:      rollback_to_previous_checkpoint()
10:     repeat // pass 2
11:       fine_grain_check(A, B, test, input)
12:     until STATE_INCONSISTENCY detected
13:     fix_bugs(A, B)
14:     goto START
15:   end if
16: end for
17: return PASSED_ALL_TESTS
```

Procedure 2 illustrates an example debugging procedure using our proposed verification framework for a given test suite. To minimize the performance overhead caused by hash calculation, this example performs two passes: coarse-grain checking followed by fine-grain checking.

3.3 Approaches for State Comparison

We consider three options for state information (*State_Info*$[n]$ in Procedure 1) as shown in Figure 3:

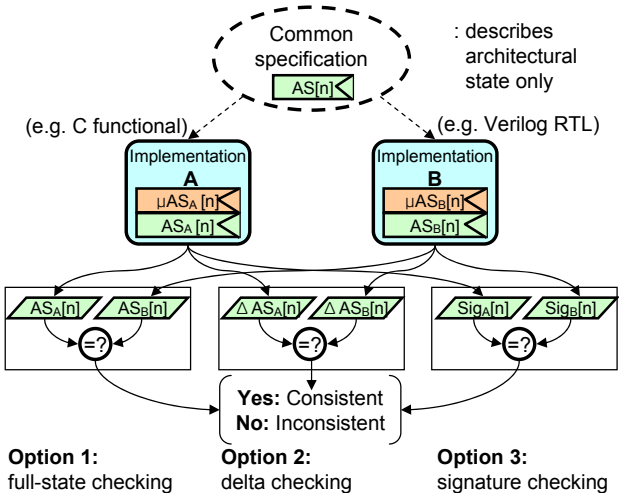


Figure 3. Three possible implementations of state consistency checking. At n-th checkpoint, each implementation dumps out the entire architectural state ($AS[n]$) for Option 1, the changes in the architectural state ($\Delta AS[n]$) since the previous checkpoint for Option 2 and a signature ($Sig[n]$) for Option 3, respectively. The checker can detect a state inconsistency by comparing these dumped outputs.

3.3.1 Full State Checking (Option 1)

Fine-grain full-state checking is only feasible when the size of the architectural state is small. Both the bandwidth requirement for dumping state information and the computation requirement for comparing it is $O(sizeof(AS[n]))$.

One might argue for checking critical words only instead of checking the full architectural state. However, we advocate full coverage of architectural state because it is not always possible to identify critical words. For example, in the FIR filter in Figure 2 all pixels have the same criticality. This is also true for large memory structures, such as caches.

3.3.2 Delta Checking (Option 2)

Delta checking, which uses the changes since the previous checkpoint, is better suited to fine-grain checking. For example, each module could dump out a list of (`block_id`, `new_data`) pairs for state elements being modified. The bandwidth and computation overhead is $O(sizeof(\Delta AS[n]))$ which is much smaller than $O(sizeof(AS[n]))$ in general.

However, we point out two issues with simple delta checking. First, it is possibly bandwidth inefficient (and storage inefficient if used offline) if the number of changes between checkpoints is large. Second, it is not clear how to implement this scheme efficiently in a hardware platform

(e.g. FPGA). It must either have a queue to log the changes since the previous checkpoint, or have a dirty bit associated with each block and walk through the touched blocks to send out the modified data.

To address these issues, we introduce a third option: hash signature checking.

3.3.3 Hash Signature Checking (Option 3)

The hash signature reduces the bandwidth overhead to a fixed constant cost, at the cost of additional computation to recalculate the hash. We propose the use of incremental hashes to generate signatures of the entire state to minimize performance overhead [5]. An incremental hash is designed such that if a change between two states is small, it is possible to quickly update the hash of the new state from the hash of the previous state rather than recomputing the new hash from scratch. Incremental hashes have been used for various applications including virus protection, memory integrity checking, and broadcast networks [15]. To the best of the authors' knowledge, this paper is the first work that applies incremental hashing for efficient state consistency checking for functional verification.

4 Implementation Issues

In this section, we address several implementation issues of the incremental hash-based functional verification.

4.1 Incremental Hash Function Design

The simple incremental hash we use to summarize the architectural state is as follows:

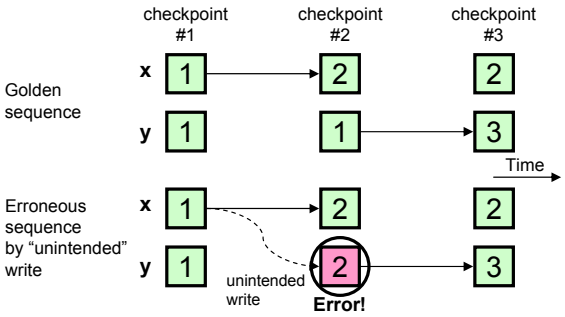
$$h = \otimes_{i=1}^{n} f(block\_id_i, data_i)$$

where n is the total number of blocks, $\otimes$ is XOR operator, and f is a block hash function. A pseudorandom function was used for f in the original literature [5] but we replace it with a simpler hash function (e.g., multiplicative hash) to reduce performance overhead. Because we are not coping with a security application and an active adversary who can modify the hash based on knowledge about the system, we leave out data randomization, pair block chaining, and random permutation [5].

With this hash, we can readily calculate the new hash (h') from the old hash (h) after replacing old_data_i with new_data_i:

$$h' = h \otimes f(block\_id_i, old\_data_i) \otimes f(block\_id_i, new\_data_i)$$

Each signature update costs two $\otimes$ and two f operations. One might argue for an alternative hash update scheme ($\hat{h}$) which recalculates the new hash with new_data_i only, but not with old_data_i, to save one $\otimes$ and one f operations. This scheme was used in [12] for dynamic detection of a processor's soft errors by CRC hashing the stream of committed values to register file.



	@ checkpoint #1	@ checkpoint #2	@ checkpoint #3
h'_{golden}	$f(x,1) \otimes f(y,1)$	$f(y,1) \otimes f(x,2)$	$f(x,2) \otimes f(y,3)$
h'_{error}	$f(x,1) \otimes f(y,1)$	$f(y,1) \otimes f(x,2)$	$f(y,1) \otimes f(x,2) \otimes f(y,2) \otimes f(y,3)$
$\hat{h}_{golden}$	$f(x,1) \otimes f(y,1)$	$f(x,1) \otimes f(y,1) \otimes f(x,2)$	$f(x,1) \otimes f(y,1) \otimes f(x,2) \otimes f(y,3)$
$\hat{h}_{error}$	$f(x,1) \otimes f(y,1)$	$f(x,1) \otimes f(y,1) \otimes f(x,2)$	$f(x,1) \otimes f(y,1) \otimes f(x,2) \otimes f(y,3)$

Figure 4. Example of error caused by unintended write with a module having two blocks (x and y). This error is detected by hashing both old and new values of a block (h') but not by hashing just new values ($\hat{h}$).

However, the alternative hash function $\hat{h}$ has weaker error detection capability than h' and cannot detect errors caused by so-called "unintended" writes. Some errors are caused by "intended" writes where a designer updates the hash signature correctly but has written a wrong value to the actual storage of data. Others are caused by unintended writes where a designer has written a value to a storage that he is not supposed to write, so that the hash fails to keep track of all write operations actually happened.

Figure 4 illustrates how a designer can detect an error caused by an unintended write using h' while he fails to detect it using $\hat{h}$. This kind of errors are commonly observed in the address decoder of a memory block. There are two memory elements, denoted by x and y, in the module. Then we can easily find that $\hat{h}_{golden} = \hat{h}_{error}$ (i.e., error detection failure) at Checkpoint #3 while $h'_{golden} \neq h'_{error}$ (i.e., error detection success). Note that at Checkpoint #2 neither of the hash signatures (h'_{error} and $\hat{h}_{error}$) can capture the actual state of memory (i.e., $x = y = 2$) correctly because the integrated hash logic updates its signature based on (*addr*, *data*) pair available at the input of the address decoder. h' detects this error at the next write to the errorneous memory block (i.e., y).

Block hash design (f) is another implementation issue and crucial to minimize performance overhead and false negatives (i.e., undetected inconsistencies). In choosing a block hash function, a user can exploit efficient implementations available on the simulation platform. For example, modern FPGAs have efficient hash implementations available as IPs such as MD5, SHA-1 [11]. On the other hand, we can use simple multiplicative hashes on software platforms.

For evaluation in Section 5, we use the following simple multiplicative hash to hash block i:

$$f(block\_id_i, data_i) = \\ ((\otimes_{j=1}^{m} word_j) \otimes block\_id_i) * GOLDEN\_RATIO$$

where m is the number of words in $data_i$, *GOLDEN_RATIO* is the golden ratio (0.61803399...) minimizing collisions in a multiplicative hash [9]. We need to XOR *block_id* because we would not be able to differentiate two writes of the same value to different blocks otherwise.

Choosing block granularity is another design decision, where a block is the unit of hash calculation. Finer-grain blocks allow a single block hash ($f(block\_id_i, data_i)$) to be calculated more quickly but updates more hashes per tick. A rule of thumb is to use the amount of data typically modified in a single tick as the block size.

4.2 Hash Integration to DUT

In this paper, our evaluation is all performed on a software simulation platform with C/C++ reference models and RTL models written in the Bluespec HDL [1]. Hash integration is done manually by inserting `sig_update` and `sig_check` function calls written in C at every function and rule that updates the architectural state. Bluespec allows designers to import functions written in C for simulation.

Ideally, we envision this integration to be highly stylized and automatically performed by a CAD tool chain using a user-provided specification because manual code transformation is an error-prone process. The specification could contain the list of architectural state elements of interest with block size and ID, hash function to be used, checkpoint granularity, and so on. In hardware emulation platforms (e.g., FPGAs), it makes sense to synthesize the hash logic along with the DUT and expose an interface for the designer to access the signature value and control signature generation.

4.3 Extracting Architectural States

Although we confine ourselves to detecting an inconsistency between common architectural state, extracting this is not always trivial. For example, to verify a pipelined processor using its functional model, Burch et al. [6] introduce a flush operation to put the pipelined processor into a state that can be compared against the state of the functional model. In some cases, a designer may not be able to statically extract the architectural state. For example, a processor model having a renamed physical register file has to extract the architectural state of the register file dynamically. Here, a user might have to provide a custom post-processing function to perform the extraction along with other parameters. This will impose additional performance overhead and add to verification complexity. Therefore, in the TLM approach, maintaining correspondence between internal states is a desirable practice for verification purpose [8, 10].

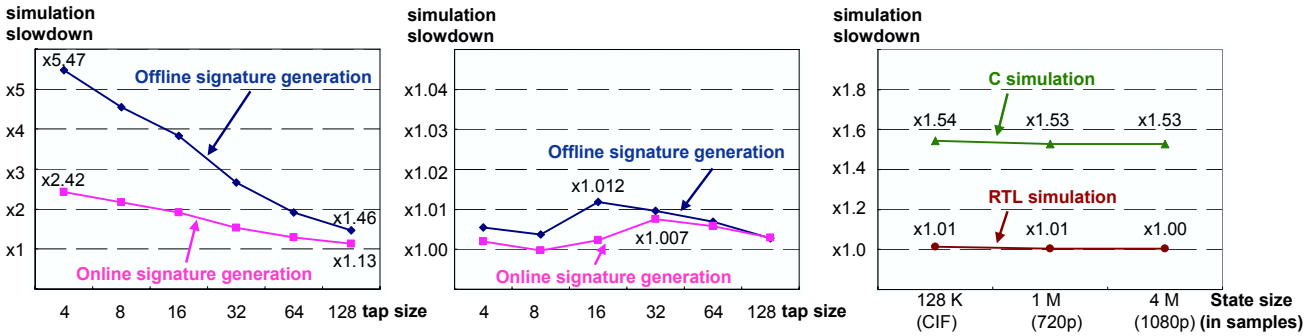
5 Evaluation

We evaluate the error detection capability and performance overhead of the incremental hash technique using both a FIR filter and a Vorbis decoder. For both designs, the reference model is written in C and the RTL is written in Bluespec HDL [1]. We have implemented a portable C library for signature generation and checking using the incremental hash discussed previously. Functional correctness of the library has been verified by injecting a random mismatch between the golden state and the state under test. This library is linked with C reference models as well as Bluespec RTL models. We use Bluesim, a cycle-accurate simulator for Bluespec [1], to measure the performance of RTL simulation. We have not detected any hash collision with the default 32-bit hash, but the library supports 64-bit hash to be used when hash collision (i.e., false negatives or undetected inconsistencies) is a concern.

5.1 FIR Filter

Although simple, the FIR filter shown in Figure 2 provides an interesting evaluation example. The FIR filter has a set of parameter knobs we can exploit to evaluate the performance overhead of our proposed hash scheme. By changing `tap_size`, we can easily change the payload versus hash calculation ratio. As `tap_size` increases, the FIR filter does more FIR computation per hash, thereby reducing the relative overhead of hash calculation. By changing the frame size, we can obtain an arbitrary size of architectural state. In addition, we can easily break down a transaction into multiple ticks at various granularities.

Simulation slowdown is a primary concern in adopting continual hashing and we consider two cases: online and offline signature generation. In the online case, the golden model and the design under test generate signatures dynamically as both models execute in parallel (the golden model slightly runs ahead of the design under test to generate golden signatures). In the offline case, we execute the



(a) Performance overhead (C model) (b) Performance overhead (RTL model) (c) Performance overhead (varying state size)

Figure 5. Performance evaluation of continual hashing in terms of simulation slowdown. We calculate and check the signature at every tick (i.e., for every sample) with 1 mega-sample frames. (a) shows the performance overhead caused by hash calculation and state inconsistency checking for reference C model and (b) for Bluespec RTL model. (c) shows that the performance overhead does not depend on the total state size as expected. For (c) `tap_size` is 32 and the online scheme is used.

model under test only and use the generated signatures offline from the golden model. Because of the file I/O overhead (reading in and writing out golden signatures) we expect the online scheme to perform better. Note that we only measure the simulation speed of the RTL but not that of the high-level reference model. This gives us an optimistic result when running two models in parallel, but we believe this is acceptable because the total simulation time is usually limited by the slower simulator.

Figure 5 (a) and (b) show the slowdown of simulation speed for both the C and the RTL models as we vary `tap_size` in FIR filter. In a typical use case, we generate a golden signature sequence using a fast C model to verify an RTL implementation of the same unit. In this use case, Figure 5 (a) measures the signature generation cost, and Figure 5 (b) measures the signature checking cost of the DUT. For *signature generation*, the range of slowdown is $\times$1.46-$\times$5.47 for offline and $\times$1.13-$\times$2.42 for online as we change `tap_size` from 4 through 128. The incremental hash achieves a dramatic improvement (about 2000$\times$ faster) in signature generation compared to recalculating a hash for the architectural state from scratch at every tick. We also observe that the most of performance overhead for the offline scheme is due to file I/O, which underlines the importance of bandwidth (storage) efficiency of the signatures.

On the other hand, there is only negligible performance overhead for *signature checking* with the RTL model as shown in Figure 5 (b). The maximum observed slowdown is 1.2 % for the offline scheme when `tap_size` is 16. The difference in simulation slowdown between C and RTL models is attributed to the difference in the level of abstraction. The hash calculation overhead is relatively much smaller in detailed RTL simulation than in abstract C simulation. The non-monotonic simulation slowdown, as `tap_size` increases, is likely to be caused by an artifact of caching behavior.

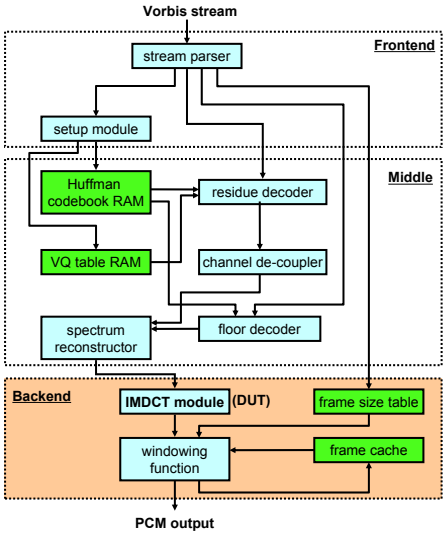


Figure 6. Functional modules of Vorbis decoder. Memory and computation modules are colored differently.

Finally, Figure 5 (c) illustrates that the performance overhead does not depend on the size of the entire state. In other words, our proposed scheme is scalable to the state size in terms of performance overhead. We use three different frames whose sizes are 128K, 1M and 4M samples, respectively. These frames roughly correspond to YUV frames of CIF, 720p HDTV and 1080p HDTV in terms of their size.

5.2 Vorbis Decoder

For more realistic evaluation of the proposed scheme, we use the backend of a Vorbis decoder implementation. Vorbis is an open-source, patent-free lossy audio codec that is generally comparable to MP3. At a high level, the Vorbis

decoding process can be divided into three sections shown in Figure 6. Among the three sections, the backend is the most computation-intensive. Within the backend, we are particularly interested in the Inverse Modified Discrete Cosine Transform (IMDCT) block, which dominates the backend in terms of hardware complexity and computation requirement. For the rest of this section, we assume that the DUT is the IMDCT block.

The golden reference model (Tremor) is written in C and available in [2]. The RTL implements the algorithm described in [16] where the kernel takes 8 steps to complete the IMDCT operation of an audio frame. There is a 4-KB shared memory block where the intermediate values after each step are stored. In our experiment, we verify the consistency of the shared memory block, which is the biggest state element within the DUT, at three sub-transaction checkpoints after Step 3, 4 and 8 in [16]. Note that we use an offline scheme–a sequence of golden signatures is generated in advance and read in by file I/O for signature checking. Given that the reference model executes about 64× faster than the RTL implementation, it is likely that the online scheme (i.e., parallel simulation of two models) will add only a small fraction of execution time to the RTL simulation.

According to our simulations, hash signature checking increases execution time of RTL simulation by only 2.2% while correctly detecting any injected state inconsistency.

6 Conclusion

In this paper, we proposed continual hashing to allow a designer to detect state inconsistency between two models at a fine-grain tick granularity. We presented an incremental hash based on a simple multiplicative block hash to minimize the performance overhead. Our evaluation showed that deploying fine-grain state inconsistency checking is feasible. Total bug detection time can be reduced even further by multi-pass consistency checking, i.e., using coarse-grain sweeping followed by fine-grain sweeping.

The hash signature is a compact summary of not only the architectural state but also the state path taken at any point in time, and can be used for other applications. For example, a stream of hash signatures can be used to replay a bug to confirm that the same sequence of state transitions is followed.

7 Acknowledgements

We thank Nirav Dave for helpful discussion in an early phase of this work. This work was partly funded by Nokia Inc. and NSF Award CCF-0541164.

References

[1] `http://www.bluespec.com`.

[2] `http://xiph.org`.

[3] A. Bruce et al. Maintaining consistency between SystemC and RTL system designs. In *DAC '06: Proceedings of the 43rd Design Automation Conference*, pages 85–89, New York, NY, USA, 2006. ACM Press.

[4] A. J. Hu. High-level vs. RTL combinational equivalence: An Introduction. In *24th International Conference on Computer Design (ICCD 2006), San Jose, CA, USA*, 2006.

[5] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *STOC '95: Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 45–56, New York, NY, USA, 1995. ACM Press.

[6] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 68–80, London, UK, 1994. Springer-Verlag.

[7] C. H.-P. Wen et al. Simulation-based functional test justification using a boolean data miner. In *24th International Conference on Computer Design (ICCD 2006), San Jose, CA, USA*, 2006.

[8] D. Brier and R. S. Mitra. Use of C/C++ models for architecture exploration and verification of DSPs. In *DAC '06: Proceedings of the 43rd Design Automation Conference*, pages 79–84, New York, NY, USA, 2006. ACM Press.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 3 (2nd ed.): Sorting and Searching*. Addison Wesley Longman Publishing, Redwood City, CA, USA, 1998.

[10] Frank Ghenassia et al. *Transaction-level Modeling with SystemC*. Springer, Dordrecht, The Netherlands, 2005.

[11] Helion Technology Limited. Helion IP Core Products - Authentication cores. `http://www.heliontech.com`.

[12] J. C. Smolens et al. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, New York, NY, USA, 2004. ACM Press.

[13] A. Koelbl, Y. Lu, and A. Mathur. Embedded tutorial: Formal equivalence checking between system-level models and RTL. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design*, pages 965–971, Washington, DC, USA, 2005.

[14] M. N. Mneimneh and K. A. Sakallah. Principles of sequential-equivalence verification. *IEEE Des. Test*, 22(3):248–257, 2005.

[15] R. C.-W. Phan and D. Wagner. Security considerations for incremental hash functions based on pair block chaining. *Computers & Security*, 25(2):131–136, 2006.

[16] T. Sporer, K. Brandenburg, and B. Edler. The use of multirate filter banks for coding of high quality digital audio. In *In Proceedings of the 6th European Signal Processing Conference*, pages 211–214, 1992.

[17] Y.-C. Hsu et al. Advanced techniques for RTL debugging. In *DAC '03: Proceedings of the 40th Conference on Design Automation*, pages 362–367, New York, NY, USA, 2003.