ArChIVED: High Performance Validation of Microprocessors Using Event Digests

Chang-Hong Hsu†, Debapriya Chatterjee†, Ronny Morad‡, Raviv Gal‡ and Valeria Bertacco†

†University of Michigan Ann Arbor, MI
{hsuch,dchatt,valeria}@umich.edu

‡IBM Research Lab
Haifa, Israel
{morad,ravivg}@il.ibm.com

ABSTRACT

Simulation-based techniques play a key role in validating the functional correctness of microprocessor designs. A common approach for microprocessors (called instruction-by-instruction, or IBI checking) consists of running a RTL and an architectural simulation in lock-step, while comparing processor architectural state at each instruction retirement. This solution, however, cannot be deployed on long regression tests, because of the limited performance of RTL simulators.

Acceleration platforms have the performance power to overcome this issue, but are not amenable to the deployment of an IBI checking methodology. Indeed, validation on these platforms require logging activity on-platform and then checking it against a golden model off-platform. Unfortunately, an IBI checking approach following this paradigm entails a large slowdown of the acceleration platform, because of the sizeable amount of data that must be transferred off platform for comparison against a golden model.

In this work we propose a sequence-by-sequence (SBS) checking approach that is efficient and practical for acceleration platforms. Our SBS checking solution validates the test execution over sequences of instructions (instead of individual ones), thus greatly reducing the amount of data transferred for off-platform checking. We found experimentally that SBS checking delivers the same bug-detection accuracy as traditional IBI checking, while reducing the amount of traced data by >90%.

1. INTRODUCTION

Design verification has become increasingly challenging due to shrinking transistor sizes with each technology node, which has allowed designers to fit more transistors in the same chip area, and thus to develop more complex micro-architectural features with each generation. This increase in complexity has greatly increased the state space of microprocessor designs, and has resulted in a significant increase in associated verification effort. In this context, simulation-based validation continue to be the primary mode of verification in the industry. In this methodology, the correctness of the design under verification (DUV) is checked by examining simulation results created from executing a large collection of long test regression suites on different abstraction levels of the DUV. Usually, billions of simulation cycles are executed for each new revision of the microprocessor under development. To achieve sufficient simulation coverage for these long regressions in a reasonable amount of time, the performance of the simulator plays a key role. Software-based simulation tools are most prevalent, but unfortunately their performance is not even close to being adequate (1-10 cycles per second on a full-chip design) to obtain acceptable validation coverage for modern microprocessor designs.

This crucial requirement for simulator performance has led functional verification engineers to transition from software-based simulation solution, towards acceleration platforms that can meet the ever growing verification performance requirements. These platforms achieve orders of magnitude higher performance over software-based solutions by using specialized hardware components for logic simulation.

This boosted simulation performance, however, comes at the cost of reduced checking and debugging capability. While the microprocessor design can be mapped onto the platforms, these platforms are designed to only simulate synthesizable logic descriptions, leading to challenges in integrating the software checkers onto the same platform. To overcome this issue, one often needs to rely on the communication between the host and the platform to transfer the data to be checked off the platform. This approach, however, also has limitations; specifically, the limited input/output (I/O) pins and off-platform bandwidth of the platform greatly constrain the observability of internal signals. Another drawback is related to the fact that most checkers designed for microprocessor validation execute in lock-step, which frequently suspends the simulation, eroding the performance advantage of hardware platforms. These issues, unfortunately, render this off-platform checking approach infeasible.

Another verification technique considers adapting software checkers to acceleration platforms by following a "log and then check" approach, which utilizes recording mechanisms of the platforms. In this approach, only a relevant subset of a design's signals/events are recorded during simulation. The recorded data is then checked offline for consistent behavior. However, as the number of recorded signals increases, simulation performance degrade quickly. This rapid degradation implies that, when crafting a design checking solution for acceleration platforms, we need to ensure that the recording bit-rate is minimal. The need of minimizing the recording bit-rate leads to major challenges in acceleration-platform checkers: how can we ensure the same quality of checking as with software-based simulation, while collecting only minimal information?

In this work, we target a common family of checking schemes, called instruction-by-instruction (IBI) checking, which is able to identify any architectural state deviation in a design's behavior from the golden reference at the architectural state and provides bug localization capability. Even though the deployment of this solution is quite straightforward in software-based simulation, creating an equivalent scheme for acceleration platforms is challenging. First, the checking functionality is usually too complex to be implemented in hardware, and it is further complicated by re-orderings in architectural state updates due to the micro-architectural implementation. To address this issue, we must resort to a "'log and then check" approach. Second, the recording rate necessary to gather information for IBI checking (*i.e.*, all updates to architectural state) is too high to sustain the performance advantage of acceleration. These two challenges require novel alternative methods to attain the objective of IBI checking, namely, validation of the architectural state updates with respect to a golden model by recording minimal information.

1.1 Contributions

In this work, we introduce a novel sequence-by-sequence checking scheme that checks the validity of the accumulated updates on the architectural state of sequences of instructions. This approach drastically reduces the volume of recorded data, while still being capable of discerning most types of discrepancies in the architectural state with high probability. We achieve this goal by constructing a digest of the architectural events over a sequence of instructions. Our digest-based solution has the following features:

1. Minimal average recording bit-rate.
2. Error-detection ratio comparable to fine-grain IBI checking.
3. Digests can be realized with a small logic footprint.

We observe that, even for digests resulting from long (>10,000) instruction sequences, sensitivity to architectural state discrepancies is not diminished when using appropriate compression schemes.

2. BACKGROUND AND RELATED WORK

In this section we first discuss relevant background for hardware-accelerated simulation platforms; then we introduce prior checking solutions developed for design validation on such platforms and finally delineate some of the challenges faced by any solution attempting to perform architectural checking of microprocessor designs on such platforms.

2.1 Acceleration background

Acceleration [6] platforms are becoming increasingly vital to design validation. Design simulation acceleration platforms leveraging special-purpose hardware (whose computational units are optimized for the simulation of a single logic gate or a small block of gates) has been developed and utilized over past decades [1, 10]. One of the very first such systems, the Yorktown Simulation Engine [7], was developed at IBM and consisted of an array of special-purpose processors. Modern acceleration platforms are typically composed of large arrays of customized processing elements dedicated to simulate logic gates in a concurrent fashion. The design under verification (DUV) must be synthesized into a structural logic description to map the design to the execution platform's components. Cadence Palladium [4], IBM AWAN [6] are examples of such platforms. Simulation performance of these platforms are typically between 10 kHz to 1MHz.

Acceleration platforms experience slowdown when increasing the amount of simulated logic; hence it is important to ensure that the footprint of any additional logic necessary to enable checking is minimal. As mentioned earlier, acceleration platforms allow recording and subsequent transfer of a subset of the design's signal values, but this process slows down the simulation, thus nullifying the key advantage of acceleration. Degradation of acceleration performance is usually proportional to the number of traced signals. The exact relationship between performance degradation and recording rate depends on the specific architecture of the acceleration platform. This is due to the fact that the underlying architecture of the acceleration platform records the values of the signals marked for observation in each cycle and stores them in internal memory. To do so, the simulation must be temporarily suspended whenever the memory becomes full, so that the content can be transferred via a low bandwidth channel to a connected host machine. The more frequently the transfer takes place, the higher the associated performance penalty. Thus, the lower the number of traced bits, the longer it takes to exhaust the internal memory resources and the longer the intervals of uninterrupted simulation. As a result, limiting the recording bit rate is critical for a solution to be practical on acceleration platforms. Emulation platforms exhibit similar trade-offs, hence the same considerations apply.

2.2 Checking in acceleration

While acceleration and emulation platforms can provide high-performance simulation of synthesized structural descriptions of a design, they cannot accommodate the associated high-level software testbenches and checking environments necessary for fine grain validation. Hence, acceleration and emulation platforms are typically utilized to perform coarse grain validation, such as comparing final outputs of a simulated design model and the design running on the acceleration platform. Moreover, lockstep execution of software checkers on a host paired with the design simulated on an accelerator is not tenable, since it unacceptably degrades overall performance to an unacceptable level. It is, therefore, critical to adapt checkers to acceleration platforms to fully leverage high-performance simulation for verification and debugging. Current industry methodologies on this front have focused on limiting the number of synchronizations between the host running the checkers and the accelerator by: i) accumulating short and frequent interactions between the design and the testbench into longer and infrequent transactions [11, 12], by ii) recording the values of critical design signals during simulation on-platform and off-loading the log at the end to check for consistency with a software checker [5], or by iii) synthesizing some of the checkers into hardware for simulation alongside the design [3].

2.3 Architectural checking on acceleration

Architectural checking, *i.e.*, instruction-by-instruction (IBI) checking is a dominant technique for microprocessor validation. In IBI checking, a test regression is executed on the simulated microprocessor, while updates to the architectural state of the simulated microprocessor design are compared against those of a software-based golden architectural model, one instruction at a time. IBI checking involves monitoring two types of updates on the architectural state: namely instruction completion (IC) and associated architectural register updates (RU). This technique provides the advantage of instantaneously detecting any deviation between the behavior of the DUV and that of the golden reference, thus facilitating bug localization. For decades, software-based IBI checking [13] has been among the most effective solutions for microprocessor validation and it is currently widely deployed in the industry. As design complexity increases however, software simulation simply cannot meet the ever-increasing demands for greater simulation speed and shorter times-to-market. To expedite this process, hardware-accelerated simulation platforms [8] have become predominant.

Nevertheless, the higher simulation speed comes at the cost of reduced observability of the architectural states. To gather IC and RU event data, it is now necessary to instrument the design to be simulated. For an IC-event, we require the program counter value to correspond to a non-speculative instruction completion. For RU events, we must monitor non-speculative architectural register update events, along with their updated values. Hence, a straightforward solution would be to record all such events and then compare them against the golden model. Some additional logic would also be necessary to output the collected information, so that it can be brought off-platform. A recent solution in this space is [5], where the authors propose to decouple event-tracing from checking efforts. The proposed method detects any divergence between architectural states in the accelerated simulation and in the software-based golden model. It accomplishes this goal by recording a compressed "abstraction" of each instruction footprints, and so it transfers data off-platform at a much lower rate than if it were to transfer the raw data recording. Unfortunately, even with this compression technique, the latency overhead on the accelerator is still higher than what is typically tolerable ($\sim$50% slowdown in the worst case) in high-performance validation flows. For any such solution to be successful, two major challenges must be overcome:

Handling the lack of event correlation: For modern microprocessor designs in the industry, recorded IC and RU events lack close time correlation due to microarchitectural implementation. This problem is reported in [5] as well. Because of this lag, the trace obtained from acceleration platforms only consists of a sequence of IC and RU events, although the RU events corresponding to an IC event are neither grouped together nor ordered with respect to the IC event. In fact, an RU event may appear in the trace a few positions before or after its corresponding IC event. Even so, it can be guaranteed that under correct execution, all RU events corresponding to an IC event will appear within a bounded number of cycles from the IC event. This upper bound can also be considered in terms of a conservative number of IC events following the associated IC event; we treat this value as a parameter k for our experiments. Note that, all IC events occur in program order. Also, even though the RU events are re-ordered with respect to their corresponding IC events, if we consider a certain architectural register, all the updates to that particular register appear in program order in the trace. Our digest computation scheme takes advantage of this fact. Any checking scheme operating on a trace of this type must find a match between any logged event and the expected events predicted by the golden model. A failure to match an expected RU event within k IC events of the associated instruction will be considered a missing RU event.

Reducing the amount of traced data: The recording rate necessary to trace all the IC and RU events, along with the associated program counter and register values, is prohibitive for acceleration platforms. Hence, a straightforward tracing approach would erode away acceleration performance advantage, in which case, it becomes compulsory to perform on-platform compression using additional logic. Previous attempts to tackle this problem [5] record all events but only compress the values associated with each of these events (such as updated register value) using checksum schemes. Our solution achieves much higher compression density than what is achievable by merely compressing data values.

3. SEQUENCE-BY-SEQUENCE CHECKING

In this section we introduce our novel sequence-by-sequence (SBS) checking solution for architectural state validation. First, we present a high-level overview of our solution followed by an in-depth presentation of checking flow and methods. We also detail the additional tracing logic that is necessary to collect event data from the design during its simulation on the acceleration platform.

3.1 Sequence-by-sequence checking overview

The classic approach of recording all events and then comparing each event against the golden model is fundamentally limited in terms of recording bit-rate reduction. That is because, in this context, the only way to achieve reduction is to compress the data values associated with each event and compare <event,compressed value> pairs. Clearly, the recording rate attained by this technique is limited by the event generation rate. In contrast, our solution maintains a record of cumulative changes for each architectural resource (in our case, architectural register values and completed instruction addresses) over a simulation interval of several clock cycles, and compares it against the golden model after accruing a large number of events. In this context, the average data generation rate is no longer related to event generation rate; rather, it is amortized over the length of the simulation interval. This is the key insight of our solution.

Our solution use a two-phase approach: acceleration run and a post-simulation checking. Figure 1 presents a high-level overview of the solution. During the simulation phase on the accelerator platform, architectural event digests are computed for each interval of simulation, called an epoch. Low-overhead additional logic can be

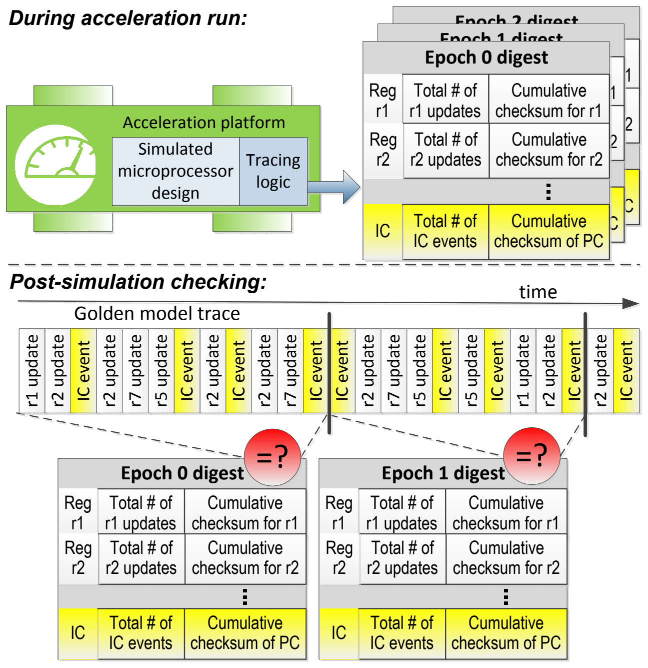

Figure 1: Overview of sequence-by-sequence checking. Digests containing cumulative checksum of updates to each architectural register, along with a count of the total number of updates are computed during each epoch. This digest is compared against that generated from the golden model during the post-simulation checking phase.

used to compute and record such event digests. During the offline checking phase, these event digests are compared by epoch-by-epoch against the golden model. The digests consist of cumulative checksums of updates to each architectural register, along with a count of the total number of updates. A checksum is also built from the series of completed instruction addresses during the epoch. Counting update events is important for tackling possible issues relating to event correlations. Indeed, from the count, we can infer whether an update event to a certain architectural register has either moved ahead enough to be in the subsequent epoch or backward to be in the preceding epoch.

One may think of two possible downsides of our approach. The first one is that the cumulative nature of the checksum may reduce the sensitivity to discrepancies between corresponding architectural state update events. We approach this issue by using sufficiently long checksums for each architectural resource. While this approach may entail more recorded data, the impact is insignificant since the long checksums are amortized over the length of the epoch. The other possible downside is that, after we identify a discrepancy in the cumulative record of a large number of events, it is no longer possible to localize which update was the cause of such discrepancy. However, if the regression's length is in the order of billion cycles, then narrowing down a discrepancy to a window of a few thousands cycles is already very valuable. Other fine-granularity methods, such as [5], can then be applied to analyze the region of interest in detail.

3.2 Complete checking flow

The proposed sequence-by-sequence validation mechanism is a process that iteratively checks the consistency between the simulation trace of the DUV and that of the golden model. This process takes two inputs: the trace's digest from the DUV and the unmodified trace generated by the golden model. The checking task is

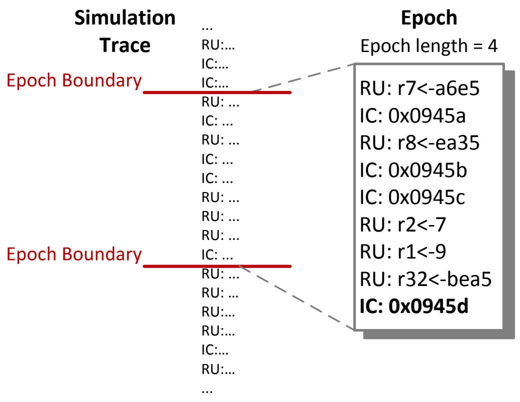

Figure 2: Epoch of epoch length 4. An epoch is a sequence of instruction events consisting of a fixed number of IC-events and an arbitrary number of RU events. The last entry of an epoch is always an IC-event.

about manipulating the golden model trace to fit it on the DUV digest. We work epoch by epoch. When we succeed in matching an epoch, we move to the next one. If we fail, we flag a bug.

An *epoch* E is a contiguous portion of a simulation trace, consisting of a sequence of IC and RU events, interleaved in any way. The number of IC events in an epoch must be fixed, and it is called the *epoch's length*. The number of RU events within an epoch may vary. The last entry of an epoch must always be an IC-event. Finally, for each architectural register we define a metric, called *RU length*, which corresponds to the number of RU events updating that register within the epoch. Thus, each epoch has an RU-length vector associated with it, with one entry for each architectural register.

In our approach, we compare epochs obtained in acceleration against those from the golden model execution, by checking the checksums derived from the epochs. The epoch's comparison flow, which is illustrated in Figure 3, consists of three main parts:

- ***Epoch segmentation***: Segmentation is a process that aligns the *epochs* obtained from the DUV with those from a golden model. After a segment is identified via the epoch length of the golden trace, this step examines whether the last IC-event of the segment matches the last IC-event of the digest obtained from the DUV. A mismatch here reveals a bug for incorrect program flow.
- ***RU events adjusting***: The goal of this step is to match the RU-length vectors between the golden model's epoch and the DUV epoch. As mentioned before, it is possible that the separation between an RU event and its corresponding IC event is such that the two are logged in separate epochs in the DUV execution. Therefore, before our final check, we must guarantee that the epoch's digests from our two traces include the same amount of RU events for each architectural register. To this end, we may move some RU events between adjacent epochs in the golden model's trace, in order to match the RU length for all the registers. We operate in the golden model's trace, since we only have a digest for the DUV trace. If we cannot find a set of moves of RU events that matches the DUV trace's digest, we flag a bug for missing RU event(s).
- ***Checksum computation***: This is the final step of the checking process. We construct the digest (*i.e.*, checksums for all architectural registers and checksum of PC-values for all IC-events) from the segmented and adjusted epoch in the golden trace, and then compare it against the digest from the DUV trace. If everything matches, we move on to the next epoch, otherwise we flag an error.

The checking process iterates through these three steps for the entire regression, one epoch at a time. Whenever any of the steps

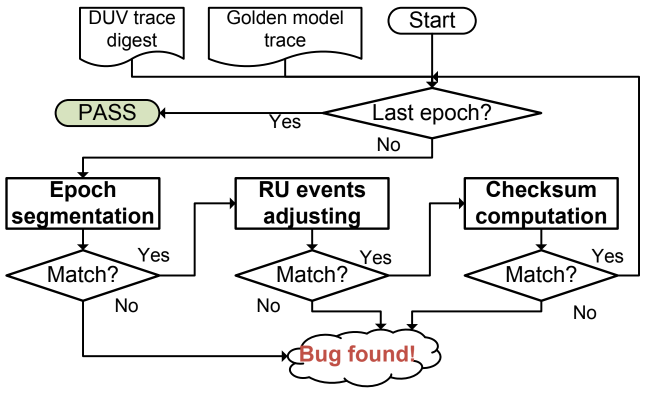

Figure 3: ArChIVED's checking flow proceeds epoch-by-epoch through three main steps: epoch segmentation, RU events adjusting and checksum computation.

reports a failure, a bug is reported and the post-simulation checker terminates. Our solution can report the time of the bug occurrence at the granularity of an epoch length. Other solutions can then be deployed to further refine the localization (for instance [5]).

3.3 Checking steps

In this section we elaborate on the tasks entailed by each of the checking steps.

3.3.1 Epoch segmentation

In the "segment epoch" step, we process corresponding epochs of the same length – that is, epochs including the same number of IC events. If the two epochs were to match each other they should end at the same IC event, since we have already matched all the previous epochs. When the last IC event does not match, we flag an error, indicating that there is mismatch between the execution flow in the acceleration platform and that in the golden model.

3.3.2 RU events adjusting

As explained in Section 2.3, the lack of event-correlation is one of the main challenges for SBS validation. The RU events corresponding to an IC-event may appear up to k IC-events earlier or later. However, all the IC-events and RU events for a particular register follow program order, which is a key tenet for the mechanism of our solution. This step computes the RU-length for each register in the golden trace's epoch, and compares it against the value ob-

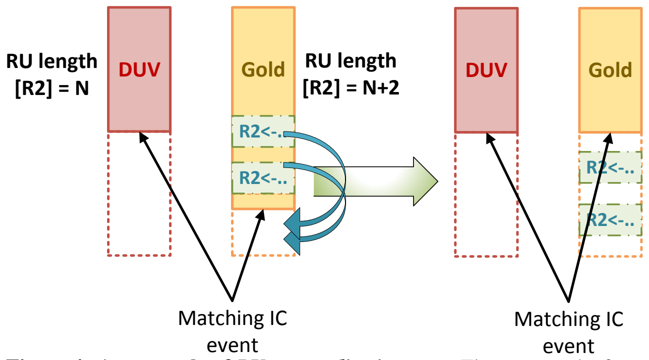

Figure 4: An example of *RU event adjusting* step. The two epochs from the DUV and the golden model have the same IC length and the boundary IC event is matched. However, the epoch from the golden model is two RU events longer than E_{DUV} with respect to register R_2. Thus, the *RU event adjusting* step searches for 2 RU events updating register 2 in the golden model epoch, starting from its end. Once found, it moves them forward to the next epoch.

tained from the DUV. For each register with a different length, our checker attempts to move RU events in the golden model across the epoch's boundaries, until it can attain a match.

The process considers one register at a time. It first adds or subtracts, to the number of RU events computed for the epoch, those RU events that have been propagated forward or borrowed backward from the previous epoch. If, at this point, the sum matches that of the DUV digest, the work for this register is completed. Otherwise, it will push forward to the next epoch, or borrow from it, the number of RU events required to make the two sums match.

Note that this stage only needs to work within the neighboring epochs. Indeed, the epoch length has been selected to be sufficiently long so that RU events can never land more than one epoch before or after the IC-event to which they relate. The epoch length depends on the specific microarchitecture under verification. Thus, if we cannot borrow a sufficient number of RU events for a given register, that indicates a bug of missing RU event(s) or of excessive RU event displacement. Figure 4 illustrates this process with an example. In the figure we show an epoch in which the golden model trace includes two additional RU events with respect to register 2. Thus the adjustment step moves the last two RU events relating to register 2 to the following epoch.

3.3.3 *Checksum computation*

While the previous steps have already ruled out many bug manifestation possibilities, some other manifestation types still remain. For example, RU events may occur with incorrect register values. Moreover, the wrong ordering among RU events updating the same register also reveals a bug, often leading to erroneous behavior. To address these types of issues in our SBS checking scheme, we calculate a set of digests from the golden model's trace for the epoch, and compare it against the set of digests we have for the corresponding DUV's epoch. The digest consists of running checksums on each architectural register and PC-values of IC-events.

To compute the digest of the DUV, every architectural register should be equipped with checksum computation and storing logic. Given the possibility of corruption or wrong ordering in IC-events, we also need to compute the checksum from the PC-value of IC-events. The necessary hardware support is detailed in Section 3.4. To compare the digests, we compare the vector extracted from the DUV against that extracted from the golden model. If any pair of checksums are not equal, our checker reports an error. There are a few desirable characteristics that the checksum scheme of choice should have:

1. small logic footprint in hardware;
2. on-the-fly checksum computation (instead of block-based), so that less intermediate computation storage is required;
3. a checksum that is sensitive to event ordering, that is, without the presence of aliasing, the checksum scheme should be able to distinguish between update orders $a \longrightarrow b$ and $b \longrightarrow a$ to capture bugs manifesting as a wrong event order;
4. low aliasing.

We study below two simple checksum schemes, XOR and rotate-and-XOR, and analyze their qualities with respects to the characteristics above.

XOR checksum scheme.

The XOR checksum scheme simply updates the checksum by applying an exclusive-or operation between the current epoch's temporary checksum value with the next architectural event. For each epoch, our checker initializes the checksum to 0 and repeatedly applies the above operation through the entire epoch. The advantage of this checksum scheme is that the logic footprint is extremely small. Moreover, because of its simplicity, one can easily apply incremental updates to this checksum.

However, XOR cannot preserve the ordering information between events. For example, if we have three binary messages, 1001, 1010, and 0011, regardless of the order of calculation, the XOR checksum will generate the same result, that is, 0000. We will note in Section 4.4 that this checksum is very vulnerable to certain kinds of errors. In practice, this drawback renders XOR an untenable candidate for our SBS checking framework.

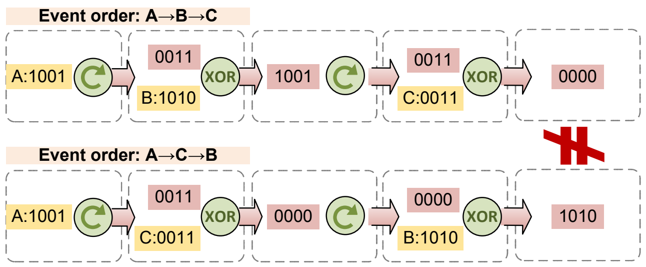

Figure 5: An example of rotate-and-XOR checksum scheme. The rotate-and-XOR checksum scheme is capable of distinguishing among different event orderings.

Rotate-and-XOR checksum scheme.

One direct improvement to XOR checksum is to apply a rotation operation before updating the checksum, to take the ordering information into account. The rotate-and-XOR scheme left-rotates the accumulated checksum by one bit before updating it with a new message. While this mechanism successfully preserves the ordering information to some extent, it imposes additional overhead, usually entailing only several short wires, which is negligible.

Figure 5 illustrates the benefits of this approach with an example. We use the same sequences considered in the previous subsection and show that, when using the rotate-and-XOR checksum we obtain two distinct checksums, namely 0000 and 1010.

3.4 Hardware requirements

To implement sequence-by-sequence checking, the microprocessor design needs to be instrumented with two kinds of tracing logic components: RU-events counters and checksum computation logic. We equip each architectural register with these two components to record its related RU length and keep track of the cumulative checksum corresponding to that register. We also need additional checksum computation logic for IC-events as well.

- **RU-length counter:** We use this to record the number of RU events that update a same register. As the execution of DUV proceeds, whenever a register update event is reported in the acceleration platform, the event is analyzed so that the index of the destination register is obtained, and the corresponding counter is incremented. All counters are reset to 0 at the beginning of every epoch.
- **Checksum computation logic:** The checksum computation logic must hold the current checksum value in a checksum-register and update it according to the selected checksum algorithm, when a regression is executed. The value in the checksum-register at the end of each epoch is the cumulative effect of this epoch, and is reported as the checksum of the corresponding architectural register for that epoch in the digest. The checksum-register will also need to be reset after each epoch in the DUV.

The logic necessary to implement this support is very simple and straightforward. Because of their extremely limited complexity, we will also need additional logic and buffers to stream out the digest data of an epoch over the duration of the following epoch. We estimate that the logic overhead to implement these structures would have no substantial effect on acceleration performance.

4. EXPERIMENTAL RESULTS

This section first describes our experimental setup and then presents the recording rate trade-offs and the detection accuracy for different variants of our checking scheme.

4.1 Experimental setup

Our experimental environment is built on the gem5 simulator [2]. We chose the ARMv7 ISA as the underlying architecture, which has a total of 64 integer registers (int regs) and 168 special purpose registers (misc regs). We collected architectural traces, consisting of register-update events (RU events) and instruction-complete events (IC events), by executing test programs using the Alpha-21264-based cycle-accurate O3CPU model in gem5. To test the efficacy of our solution, we analyzed a total of 5 benchmarks including a full execution of an eight-queen problem solver, and full or partial executions of four SPECCPU2006 integer benchmarks [9].

To provide the hardware support, as described in Section 3.4, we equip each register with an RU-counter and a set of checksum logic, including a 32-bit checksum-register and the checksum-computing logic. The bitwidth of RU-counters, however, varies throughout the experimental evaluation, since it depends on the epoch length selected.

4.2 Error injection

We obtain our test traces by mimicking the execution traces generated by a buggy processor model being simulated on an acceleration platform in the following three steps: i) we first execute the test program on the simulator to obtain a golden trace. ii) We reorder the RU events in these traces in a constrained-random fashion to generate legal variants of the golden trace. In this step, every RU event will be within a range of a small number of IC-events from its associated IC-event. iii) Random errors are then injected into these variant traces. In our current setup, we inject one random error into each of the variants. The distribution of injected errors (as shown in Table 1) basically follows the distribution described in [5]. Considering the epoch-based nature of our solution, our SBS checking scheme must tackle several types of manifestations of buggy behaviors, including some that are not present in a traditional IBI scheme [5].

We now present the categories of errors we considered in our experiment. For RU events, the bug manifestations (symptoms) studied include:

- **VanishedRU:** An architectural register update event takes place in the golden model, but does not appear in the DUV trace.
- **ExtraRU:** An architectural register update event appears in the DUV traces, but not in the golden model.
- **CorruptedRU:** The updated value of a register in the DUV does not match the expected value as recorded in the golden model.
- **MigratedRU:** The order between at least two register update events which update the same destination register is incorrect with respect to the golden model.

For IC-events, the bug manifestations include:

- **VanishedIC:** An architectural instruction completion event takes place in the golden model, but does not appear in the DUV trace.
- **CorruptedIC:** The PC value of an instruction completion event in the DUV trace is inconsistent with that in the golden trace.
- **ReorderedIC:** The order between at least two instruction completion events is incorrect with respect to the golden model.

Note that MigratedRU bugs can be categorized as *register value mismatches* because of their immediate detection nature; *i.e.*, such bugs are no different from a register value mismatch when the checker can observe this event as soon as it is produced. From a SBS perspective, however, since the checker can only see the accumulated results, MigratedRU errors could be easily masked due to checksum aliasing. A similar situation also occurs with ReorderedIC bugs. Hence, we treat them as separate categories.

Table 1: The distribution of types of bug manifestation in the pool of traces. Note that "0.00%" denotes the situation where such kind of bug doesn't manifest in that benchmark.

	bzip2	libquantum	sjeng	mcf	8queens
VanishRu	57.93%	66.21%	57.93%	64.83%	62.41%
ExtraRu	26.90%	21.38%	22.07%	12.41%	17.72%
CorruptRu	7.59%	2.07%	9.66%	11.03%	7.66%
MigrateRu	5.52%	8.28%	8.28%	11.72%	7.45%
VanishIc	0.69%	0.69%	0.69%	0.00%	1.52%
CorruptIc	0.69%	0.00%	0.00%	0.00%	1.45%
ReorderIc	0.69%	1.38%	1.38%	0.00%	1.79%
Total	100.00%	100.00%	100.00%	100.00%	100.00%

Table 2: Comparison of recording rates of several epoch-length variants of our scheme (SBS) against prior work [5].

Scheme	Epoch length	Checksum width	Average recording rate
IBI	-	32	108.80
IBI	-	5	27.90
SBS	10	32	1506.24
SBS	100	32	163.15
SBS	1,000	32	17.57
SBS	10,000	32	1.92
SBS	100,000	32	0.20

4.3 Recording bit rate

We evaluated the proposed framework by analyzing its effectiveness in detecting bugs and its performance from a recording rate perspective. We first demonstrated the efficacy of our SBS approach by comparing the recording bit rate of our approach with that of a classic IBI solution.

Assuming that in our SBS checking scheme, we transmit one set of event digests off platform every N cycles, the recording rate R_{SBS} can be calculated as follows:

$$\begin{aligned} R_{SBS} = \frac{IPC}{N}((B_{RUdigest} + B_{RUlength}) \times R \times EPI_{RU} \\ + (B_{ICdigest} \times 1)) \end{aligned}$$

where $B_{RUdigest}$ is the number of bits for recording the message digest of RU events each time and $B_{RUlength}$ is the number of bits for recording the RU length; $B_{ICdigest}$ is the number of bits for recording the digest of IC events; R denotes the number of registers; EPI_{RU} represents the average number of RU events reported per instruction. Note that, in this equation, we also assume that only 1 IC event is reported for each instruction.

We also notice that an IBI checking scheme is a special case of a SBS scheme with N equals to 1. Moreover, since IBI checking schemes record data on a per-cycle basis, they do not require register counters to store the RU event counts related to individual registers. They also do not need to record each register value every cycle. However, they should still record a few extra bits to specify the index of the destination register of the current RU event. Thus, the recording rate R_{IBI} for an IBI scheme can be calculated as follows:

$$\begin{aligned} R_{IBI} = IPC((B_{RUdigest} + B_{reg-idx})) \times EPI_{RU} \\ + (B_{ICdigest} \times 1)) \end{aligned}$$

Table 3: Percentage of bugs detected by XOR and rotate-and-XOR (RnX) checksum schemes. Similar to Table 1, N/A denotes the situation where the certain kind of bug doesn't manifest in that benchmark.

Bug category	**VanishedRU**		**ExtraRU**		**CorruptedRU**		**MigratedRU**		**VanishedIC**		**CorruptedIC**		**ReorderedIC**	
Chksum scheme	XOR	RnX	XOR	RnX	XOR	RnX	XOR	RnX	XOR	RnX	XOR	RnX	XOR	RnX
bzip2	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	0.0%	**25.0%**	100.0%	100.0%	100.0%	100.0%	0.0%	**100.0%**
libquantum	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	0.0%	**50.0%**	100.0%	100.0%	N/A	N/A	0.0%	**100.0%**
sjeng	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	0.0%	**16.7%**	100.0%	100.0%	N/A	N/A	0.0%	**100.0%**
mcf	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	0.0%	**52.9%**	N/A	N/A	N/A	N/A	N/A	N/A
8queens	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	0.0%	**86.1%**	100.0%	100.0%	100.0%	100.0%	0.0%	**65.4%**

where $B_{reg-idx}$ is the number of bits for specifying the index of the destination register of the curreny RU event. In the following discussion, we assume a test regression execution with $EPI_{RU} = 2$ and $IPC = 0.9$.

With these parameters, an IBI checking scheme that records complete event messages (32 bits) (IBI(32), for short), which theoretically can obtain the highest detection accuracy, has a bit rate of $0.9 \times ((32+8) \times 2 + 32) = 100.8$ bits/cycle, in which 8 bits are used to indicate the index of the destination register of the current RU event. Under the same execution, an IBI scheme with event message compressed into 5-bit checksums, similar to [5], (IBI(5), for short) can achieve a bit rate of $0.9 \times ((5+8) \times 2 + 5) = 27.9$ bits/cycle.

However, since our SBS approach equips every architectural register with a RU-counter and checksum logic, if the epoch length is small, we suffer from heavy recording rate overhead. The recording rate of our SBS scheme can be calculated as follows: assume that we always use 32-bit checksums in our SBS checker. For a SBS checking scheme with an epoch length of 10 (SBS(10,32), for short), since we need 4-bit RU-counters to conservatively store the number of RU events for each register, the theoretical recording bit rate is $\frac{0.9}{10}[(32+4)\times(64+168)\times 2+32] = 1506.24$ bits/cycle. It is not surprising that this number is much larger than the bit rate of IBI(32), since we augment our RU events with a large amount of information. Fortunately, this bit rate can be drastically amortized if we increase the length of the epoch, even though it also widens the RU-counters. For instance, using the above equation, we can obtain a bit rate of 17.57 bits/cycle for the SBS(1000,32) scheme; similarly, the bit rates of SBS(10000,32) and SBS(100000,32) would be 1.92 bits/cycle and 0.20 bits/cycle, respectively. Table 2 reports the complete set of calculations.

We can conclude that, if we increase the size of the epoch, we can possibly achieve more than 99% reduction of recording bit rate compared to the IBI(5) configuration.

4.4 Detection accuracy

A large data-compression ratio may also cause a substantial loss of information and thus degrade checking accuracy. To ascertain whether this is the case, we first conducted a study on detection accuracy. Since an epoch length of 100,000 gives us the largest improvement with respect to our objective, *i.e.*, the minimization of the recording bit rate, we select this epoch length for this study.

We implemented the two checksum schemes described in Section 3.3.3 (*i.e.*, XOR and rotate-and-XOR) to study how this factor affects the detection ratio. According to our results, reported in Table 3, we achieved a 100% detection ratio for all the bugs from categories VanishedRU, ExtraRU, CorruptedRU, VanishedIC, and CorruptedIC. This result, not only demonstrates the effectiveness of our 3-stage checking approach, but it also suggests that it may be a good choice to trade temporal precision (inversely proportional to epoch length) for spatial precision (proportional to the bitwidth of the underlying checksums).

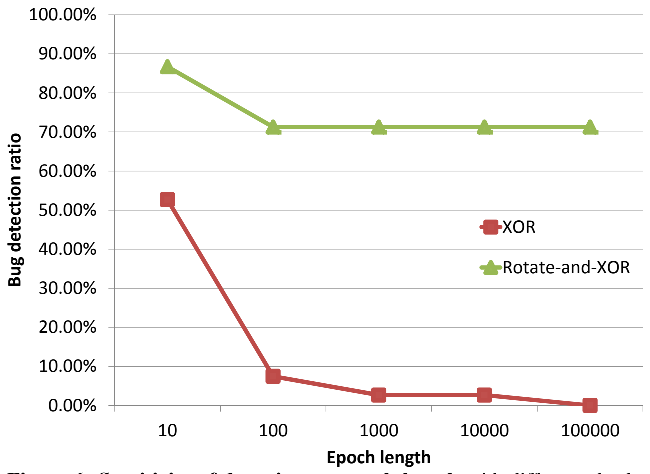

Figure 6: Sensitivity of detection vs. epoch-length with different checksum schemes for MigratedRU and ReorderedIC bugs.

However, our SBS checking does suffer from aliasing if the bugs are from the MigratedRU or ReorderedIC categories, especially for the configurations using the XOR scheme. With an XOR checksum scheme, we can observe from Table 3 that none of the MigratedRU bugs or the ReorderedIC bus can be detected (except for the configuration with a 10 IC-events epoch length, as discussed below). This result is not surprising and it is already implied by the discussion in Section 3.3.3. In short, bugs related only to message reordering cannot be detected by a simple XOR checksum scheme, unless data bits are also corrupted.

On the other hand, we found that the rotate-and-XOR scheme is still able of capturing a large portion of bugs in these two categories. Moreover, when we studied the buggy traces labeled *false negative*, we noticed that, for some of the MigratedRU bugs, the two reordered RU events actually write the same value to the register. While bugs with this type of manifestation might exist in the DUV, we cannot identify this kind of error if the simulation trace is our only clue. Thus, we believe this result demonstrates that the rotate-and-XOR scheme is both a simple and effective solution for preserving temporal information within each epoch.

Finally, we conducted a sensitivity analysis on epoch length variation. In this study, we focus only on the detection of all the bugs from the MigratedRU and the ReorderedIC categories, because bugs from every other category are 100% detected under every epoch-length configuration. The results are presented in Figure 6. We found that while the detection ratio of XOR scheme degrades from around 50% to almost 0% as the epoch length grows from 10 to 1,000, the degradation of detection ratio of the rotate-and-XOR scheme is less significant.

It is worth noting that, the fact that some of the MigratedRU and ReorderedIC manifestations can be captured by the XOR scheme under small epoch length is not surprising. Whenever an event is reordered across the epoch boundary, the manifestations can be detected by the first two stages of our checking solution.

5. CONCLUSION AND FUTURE WORK

In this work, we proposed a novel scheme for architectural validation of microprocessor designs for acceleration platforms. This method, called SBS-checking, enables highly accurate architectural validation at a very low recording rate on platform. The solution provides the same, or even better quality of checking, compared to previous solutions, but at much higher performance. As part of our future work, we intend to develop more accurate checksum schemes to detect certain types of bug manifestations.

6. ACKNOWLEDGMENTS

This work was supported by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

7. REFERENCES

[1] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Trans. on CAD*, 16(6):609–626, 1997.

[2] N. Binkert et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.

[3] M. Boule, J.-S. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *Proc. ICCD*, 2006.

[4] Cadence. *Palladium*. `http://www.cadence.com/products/sd/palladium_series`.

[5] D. Chatterjee, A. Koyfman, R. Morad, A. Ziv, and V. Bertacco. Checking architectural outputs instruction-by-instruction on acceleration platforms. In *Proc. DAC*, 2012.

[6] J. Darringer et al. EDA in IBM: past, present, and future. *IEEE Trans. on CAD*, 19(12), 2000.

[7] M. Denneau. The Yorktown simulation engine. In *Proc. DAC*, 1982.

[8] G. Ganapathy, R. Narayan, C. Jorden, M. Wang, and J. Nishimura. Hardware emulation for functional verification of K5. In *Proc. DAC*, 1996.

[9] J. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4), 2006.

[10] Y.-I. Kim, W. Yang, Y.-S. Kwon, and C.-M. Kyung. Communication-efficient hardware acceleration for fast functional simulation. In *Proc. DAC*, 2004.

[11] J.-G. Lee, W. Yang, Y.-S. Kwon, Y.-I. Kim, and C.-M. Kyung. Simulation acceleration of transaction-level models for soc with rtl sub-blocks. In *Proc. ASPDAC*, 2005.

[12] M. Shabtay, D. Leonard, B. Maya, and S. Michael. Building transaction-based acceleration regression environment using plan-driven verification approach. In *DvCon*, 2007.

[13] D. Victor et al. Functional verification of the POWER5 microprocessor and POWER5 multiprocessor systems. *IBM Journal of Research and Development*, 49(4.5), Jul. 2005.