ArChiVED: Architectural Checking via Event Digests for High Performance Validation

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

*University of Michigan, Ann Arbor, MI, USA
{hsuch,valeria}@umich.edu

†IBM, Austin, TX, USA
dchatte@us.ibm.com

‡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 validating 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 requires 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 for the acceleration platform, because of the sizable amount of data that must be transferred off-platform for comparison against the golden model. In this work we propose a sequence-by-sequence (SBS) checking approach that is efficient and practical for acceleration platforms. Our 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 that SBS checking delivers the same bug-detection accuracy as traditional IBI checking, while reducing the amount of traced data by more than 90%.**

I. 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 led to a significant increase in associated verification effort. In this context, simulation-based validation continues 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.

To achieve sufficient simulation coverage for these long regressions, 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). This crucial requirement for simulator performance has led 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.

The boosted simulation performance, however, comes at the cost of reduced checking and debugging capabilities. For instance, these platforms are designed to only simulate synthesizable logic descriptions, leading to challenges in integrating software checkers onto them. To overcome this issue, one often needs to rely on a remote host and transfer the data to be checked off the platform. This approach comes with limitations: specifically, the limited transfer bandwidth of the platform greatly constrains the observability of internal signals. Another drawback is related to the fact that most checkers designed for microprocessor validation execute in lock-step, frequently suspending the simulation and eroding the performance advantage of acceleration. These issues, unfortunately, render this checking approach infeasible.

978-3-9815370-2-4/DATE14/©2014 EDAA

Another verification technique considers a "log and then check" approach, utilizing recording mechanisms on platform. In this approach, only a relevant subset of a design's signals / events are recorded during simulation. The recorded data is then checked off-line for consistent behavior. However, as the number of recorded signals increases, simulation performance degrades quickly, thus the need of minimizing the recording bit-rate. But 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 popular family of checking schemes, called instruction-by-instruction (IBI) checking, which are able to identify any architectural state deviation from the golden reference and thus providing bug detection and diagnosis capabilities. 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. 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 methods to attain the objective of IBI checking, namely, validation of the architectural state updates with respect to a golden model.

Contributions. In this work, we introduce a novel "sequence-by-sequence" checking scheme that checks the validity of the updates on the architectural state by sequences of instructions. This approach drastically reduces the volume of recorded data, while still discerning most discrepancies 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: i) minimal average recording bit-rate; ii) error-detection ratio comparable to IBI checking; iii) digests require only a small logic footprint on platform; iv) fine-grained diagnosing capability. We observe that, even for digests derived from long (>10,000) instruction sequences, sensitivity to architectural state discrepancies is not diminished when using appropriate compression schemes.

II. Background and related work

Acceleration and emulation platforms have become increasingly popular over the past decade and today are vital in the validation of complex designs [1,7,9,12]. Modern acceleration platforms can typically reach simulation performances between 10kHz to 1MHz by mapping a design's structural logic description to large arrays of customized processing elements and executing the simulation in a concurrent fashion [4,7,8,17].

While acceleration and emulation platforms can provide high-performance simulation, they also add substantial challenges to the validation process: complex software checkers that cannot be synthesized (*e.g.*, golden models) cannot go on the platform. A work-around to this limitation has been to keep these checkers off-platform; however, the overhead of transferring the monitored

signal data off the platform severely limits the viability of this work-around. As a result, current industry methodologies have focused on limiting the number of synchronization events between host and platform by: i) accumulating interactions between the design and the testbench into longer and infrequent transactions [13,16], ii) synthesizing the simpler checkers into hardware for simulation alongside the design [3], or iii) recording the values of critical design signals during simulation on-platform and off-loading the data at the end to check for consistency with a software checker [6]. Among these solutions, variants of the latter approach are dominant for microprocessor validation on acceleration and emulation platforms. For instance, [6] proposes to decouple event-tracing from checking efforts. An abstraction of updates to the architectural state, non-speculative updates of register values (RU) and non-speculative instruction commits (IC), is recorded in a buffer on-platform, and then transferred off-platform to compare against the golden model events, while the test regression is run in lockstep between the two models. The upside of this approach is the capability of detecting any behavioral deviation between them instantaneously. Unfortunately, even with this abstraction, the latency overhead imposed on the accelerator is still much higher than what is typically tolerable (∼50% slowdown in the worst case) in high-performance validation flows.

To mitigate this problem, a run-time verification scheme, as the one proposed in [14,15], could be adopted to reduce the recording rate by examining the correctness of the execution at a basic block granularity. However, the average size of the basic blocks is usually only tens of instructions. This advantage would be easily erased by the extra data that must be transferred to the checker, and thus this solution cannot solve the problem.

A. Challenges

To achieve our goals, two major challenges must be overcome.

Handling the lack of event correlation: Modern acceleration platforms in the industry often employ a set of distributed register files to achieve higher access bandwidth, instead of a single monolithic one. The different delays between register files and the corresponding logging devices lead to different amounts of delay in the event-reporting, and thus eliminates the natural correlation between an instruction commit (IC) event and the register update (RU) events that the instruction has generated. Fortunately, 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 before or after the IC event. In our solution, we leverage this observation to synchronize RU events from an acceleration with those from the golden model. In the evaluation, we use the parameter k to express the bound described, where k is expressed in terms of IC events, instead of cycles. We also leverage two other observations: i) all IC events still occur in program order on platform, and ii) all updates to a given architectural register appear in program order in the trace.

Reducing the amount of traced data: The recording rate necessary to trace relevant events for a thorough checking is prohibitive for acceleration platforms. Hence, a straightforward tracing approach would erode away the performance advantage. Thus, it is compulsory to perform at least some on-platform compression using additional logic. Previous attempts to tackle this problem [6] record all events but compress data values associated with them (*e.g.*, updated register values) using checksum schemes. The solution we propose here achieves much higher compression density by summarizing sets of events.

III. Sequence-by-sequence checking

Traditional IBI-checking approaches require recording each architectural event on the DUV and compare it against a golden model. When operating on an acceleration platform, this solution entails the ability to record and transfer event data off-platform at high bandwidth [6]. In contrast, our sequence-by-sequence (SBS) solution accrues and compresses several architectural events over a period of time before it transfers the information off-platform for comparison with a golden model. Thus, the storage and transfer bandwidth demands are amortized over a period of execution and can be controlled by a user by extending or shortening the window length. For our solution to be effective, however, we need to organize the event data to avoid the loss of critical information and bug detection potential in this accrual and compression process.

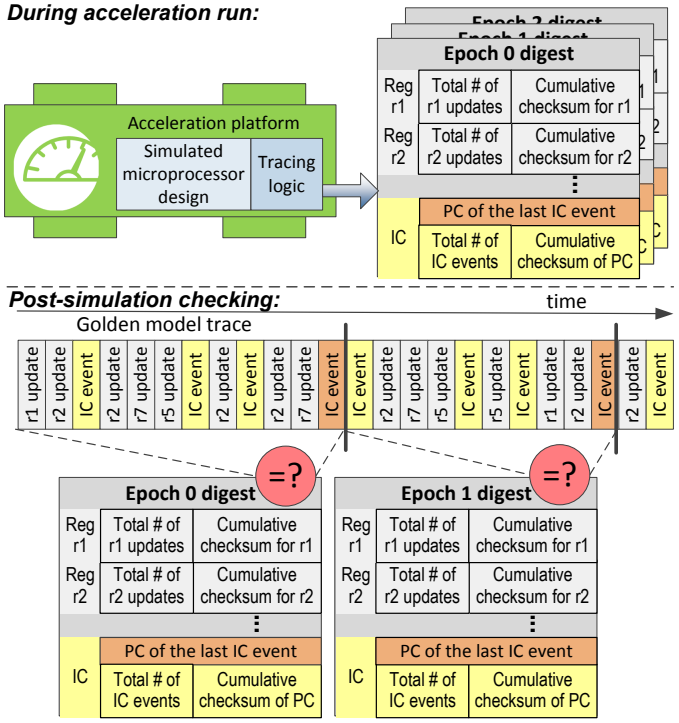

Fig. 1: Overview of our sequence-by-sequence (SBS) checking. Epoch digests are generated during execution on the acceleration platform and then transferred off-platform. In post-simulation, these digests are compared against the architectural events reported by the golden model execution to detect the possible occurrence of a bug.

Figure 1 presents a high-level overview of our solution. During the execution on the acceleration platform, architectural event digests are computed for each window of simulation, called an *epoch*. Low-overhead additional logic can be used to compute and record such event digests. Digests are then transferred off platform. During the off-line checking phase, these event digests are compared, 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 (the latter is used to align the acceleration digest with the golden model one as discussed in Section II-A). They also include the address of the last committed instruction and a checksum, which is built from the addresses of instructions committed during the epoch.

One may think of the possible downsides of our approach. The first one lies in the use of a checksum, which may limit the sensitivity to discrepancies between corresponding architectural update events. We address this issue by using sufficiently long checksums for each architectural resource. While this solution may entail more data recording, the impact is insignificant since the long checksums are amortized over the entire length of the epoch. Another possible downside is that, after we identify a discrepancy in the cumulative record of a large number of events,

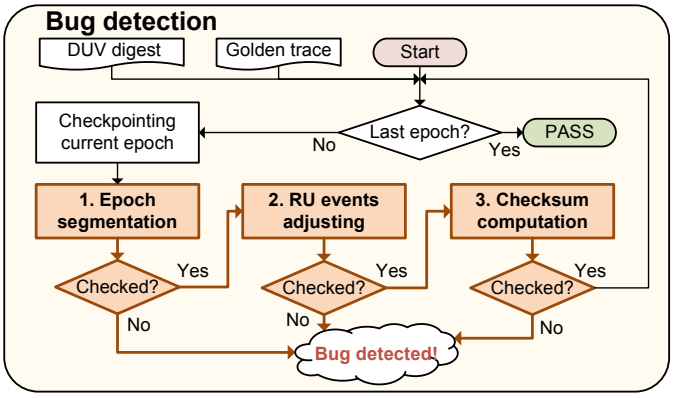

Fig. 2: ArChiVED's checking flow proceeds epoch-by-epoch through three main steps: epoch segmentation, RU events adjusting and checksum computation phase.

it is no longer possible to localize which update was its cause. However, if the regression's length is in the order of billion cycles, narrowing down a discrepancy to a window of a few thousands cycles is already a very valuable accomplishment. Once a bug has been detected, we can use our same framework for an in-depth analysis of the region of execution surrounding the bug as discussed in Section IV.

A. Complete checking flow

Our SBS solution checks iteratively the consistency between a simulation trace generated on an acceleration platform and a golden model. This process takes two inputs: the trace's digest from the acceleration and the unmodified trace generated by the golden model. The checking task manipulates the golden model trace to fit on the digest. We process the trace and digest 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* is a contiguous portion of a simulation trace, consisting of a sequence of instruction commit (IC) and register update (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 comparing the checksums derived from the traces. The epoch's comparison flow, which is illustrated in Figure 2, consists of three main parts:

1. ***Epoch segmentation***: This step aligns *epochs* obtained from the accelerator simulation 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 accelerator. A mismatch reveals a bug due to incorrect program flow.
2. ***RU events adjusting***: The goal of this step is to match the RU length vectors between the golden model's epoch and the digest's epoch. As discussed in Section II-A, 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 obtained from the accelerator. For each register with a different length, our checker attempts to move RU events in the golden model across the epoch's boundary, until it can attain a match. The process considers one register at a time. It operates in the golden model trace, since we only have a digest of the accelerator trace. 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 digest, the work for this register is completed. Otherwise, it pushes 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, epoch length must be 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. If we cannot find a set of RU events that matches the digest, we flag a bug for missing register updates. Figure 3 illustrates this process with an example. In this example, the golden model trace has two additional RU events associated with register 2. Thus the adjustment step pushes the last two RU events relating to register 2 to the following epoch.
3. ***Checksum computation***: This is the final step of the checking process. While the previous steps have already ruled out many bug manifestation possibilities, other manifestation types still remain. For example, RU events may occur with incorrect register values. Moreover, the wrong ordering among RU events updating a same register is also an indicator of a bug, often leading to erroneous behavior. To address these types of issues in our SBS-checking scheme, we calculate a set of checksums for all architectural registers and for the PC values of all IC events from the golden model's trace for the epoch, and compare it against the checksum from the accelerator's digest. If everything matches, we move on to the next epoch; otherwise we flag a bug for incorrect event orderings or corrupted event values.

The checking process iterates through these three steps for the entire regression, one epoch at a time. Whenever any of the steps report a failure, then the coarse-grained analysis terminates. Our solution then launches the fine-grained bug diagnosis phase to find the bug's first occurrence down to the granularity of tens of instructions.

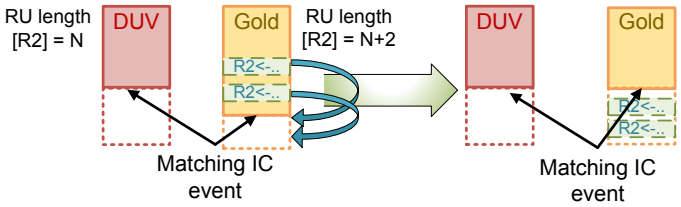

Fig. 3: An example of *RU event adjusting* step. The two epochs from the accelerator and the golden model have a matching IC event boundary. However the golden model has two additional updates for register 2, which we push forward to the next epoch.

B. Checksum computation

To compare the digests, we compare the checksum vector from the digest against the one generated from the golden model's trace. 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:

i) small logic footprint in hardware;
ii) on-the-fly checksum computation (instead of block-based), so that less storage is required;
iii) a checksum that is sensitive to event ordering, so to capture bugs manifesting as a wrong event order;
iv) low aliasing.

We study below three simple checksum schemes, architectural-state, XOR and rotate-and-XOR, and analyze their qualities with respects to the characteristics above. In addition, we also consid-

ered and evaluated a cyclic-redundancy check scheme (32-bits) and we summarize our evaluation in Section VI-D.

Architectural state checking. Architectural state checking is the most popular checking scheme in industry. In this scheme, the checker regularly compares the register file and the PC register values of the accelerator's digest and the golden model. This scheme is an ideal approach in terms of the extra logic footprint and on-platform storage needed. Unfortunately, according to Section VI-D, this scheme is insensitive intra-epoch event reorderings, and has a high probability of aliasing.

XOR checksum. The XOR checksum scheme simply updates the checksum by applying an exclusive-or operation between the current epoch's temporary checksum value and the next architectural event through the entire epoch. This scheme features an extremely small logic footprint. However, XOR cannot preserve information ordering between events and, in practice, this drawback renders XOR an untenable candidate for our SBS checking framework. We will note in Section VI-D that this checksum is very vulnerable to certain kinds of errors.

Rotate-and-XOR checksum. 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. This successfully preserves ordering information to some extent, at the cost of a small extra overhead, usually entailing only several short wires.

IV. Bug diagnosis

Once the SBS checking process completes, we have identified most bugs down to an epoch-long interval, usually 10K to 100K instructions long. At this point we re-tune our solution to support the diagnosis of the bug, but analyzing only the identified interval to narrow down the bug to a window of tens of instructions. Most, if not all, acceleration platforms include checkpoint/restore capabilities so that one portion of a same test can be re-executed many times – this feature is typically used for diagnostic purposes. We leverage precisely this feature to carry out our SBS diagnostic solution and we set up the checkpoint/restore mechanism to take snapshots of the system's architectural state at the beginning of each epoch. It is straightforward to setup the golden model to also take checkpoints at the beginning of each epoch.

As soon as a bug is detected by SBS, the remote host signals the acceleration platform to suspend execution and restore the previously checkpointed architectural state, while the same rollback is applied to the golden model's execution. We then run our SBS solution starting from this initial state and with a much smaller epoch length (a tenth or a hundredth of the original length), and perform exactly the same checking activity at a finer granularity. This process can be iterated until the epoch length is sufficiently small to diagnose the root cause of the issue. Note that any epoch length used in this process must still be greater than the parameter k (Section II-A) and representing how much RU events can be displaced from their corresponding IC event (for our experimental evaluation, the parameter $k = 10$).

V. Hardware requirements

To compute the digest of the DUV, every architectural register should be equipped with logic to compute and store the checksums. In addition we need counters to count the number of RU events on a per-register basis and the number of IC events. Finally, we need logic and storage to compute the checksum of PC values corresponding to instructions that were committed.

RU events are counted by extracting the register index of each register file write-operation and incrementing the corresponding counter. IC events can be counted by monitoring the number of instruction completions. Checksum values are accrued on an epoch-by-epoch basis in dedicated checksum registers.

Because of our high compression rate, in our setup, the time it takes to generate an event digest is much longer than the time to transfer that digest off-platform. Thus we only need storage for 2 trace buffers: at any point one is being generated and the other is being transfered. The actual overhead entailed by the two trace buffers depends on the checking granularity selected for the analysis under consideration, but fluctuates within 2-2.4kB for storage. As for logic footprint overhead, we need one XOR gate for each bit of the checksum computation (32 bits wide) for each of the registers in the sytem (˜230 in our case), for an estimated total of 7500 XOR gate or, equivalently, 30,000 NAND gates. Since modern processor cores map to at least a few millions logic gates, our overhead for checksum computation is no more than 2-3%. Note that special-purpose registers have usually fewer bits, leading to significantly smaller overhead than estimated here.

Finally, we assume that the platform is already equipped with checkpointing/rollback capabilities. Usually, this entail maintaining a shadow copy of each architectural register in the system. In addition, we require storage to store a digest being transferred off platform while the next digest is being generated. Digests are particularly small in our design, so this little storage allows the simulation to continue uninterrupted until a bug is detected. Overall, these hardware additions are minimal and we estimate them to have little impact on the platform's performance.

VI. Experimental results

In this section, we evaluate the bug detection accuracy and the recording rate required by our solution. We also compare SBS checking against an architectural-state checker solution, which constitutes the state-of-the-art approach in the industry. This architectural checker operates by taking a snapshot of the system's state at the end of each epoch and comparing it against the corresponding snapshot from the golden model.

A. 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-commit events (IC events), by executing test programs using the cycle-accurate out-of-order O3CPU model in gem5. The design was augmented with the components described in Section V: counter, 32-bit checksum-register and logic to compute the checksums for each architectural register. The bitwidth we used for the counters varies with each experiment, depending on the epoch length selected. We used 8 distinct testbenches from the in SPEC CPU2006 integer benchmarks [10], with full or partial execution of each testbench (depending on its length). To exercise the system broadly we strove to include a broad spectrum of applications in our pool.

B. Bug model and symptoms

The main type of bugs targeted in acceleration are deep and complex functional bugs manifesting at the microarchitectural and architectural level. Indeed, electrical bugs, those due to transistor sizing, to logic implementation, or layout cannot be detected with this methodology, since the design is mapped to a non-native infrastructure, either FPGA or specialized logic-computing units. To recreate this type of bugs in our evaluation, we model functional bugs in the design as random bit flips and occasional instruction losses. These manifestations represent the effect of a complex bug manifesting only when certain rare conditions occur together during the test execution. We strive to inject bugs broadly

in each of the key design modules: fetch stage, instruction buffer, execution stage and register file.

Only one bug is injected during each testbench execution. We select the injection time randomly among an early, middle or late phase of the execution. The bug is activated for a varying amount of time and then de-activated: this is to emulate the temporary occurrence of a rare execution scenario. Each testbench is executed many times (approximately 350); we then eliminate redundant situations. Most of them are due to many runs where the bug does not manifest – that is, the execution is unperturbed by the bug injection, a byproduct of the fact that we inject very narrow bugs, which only manifest when a rare combination of events occurs in the design. After this pruning, we are left with approximately 30 buggy variants for each testbench execution. We keep a few non-buggy traces to evaluate the sensitivity to false positives of our solution. Inspired by [6], we then classify the traces based on the first symptom that manifests in them:

- **VanishedRU:** an architectural register update event takes place in the golden model, but not in the accelerator trace.
- **ExtraRU:** an architectural register update event appears in the accelerator trace, but not in the golden model.
- **CorruptedRU:** the value of a register update in the accelerator trace does not match the one from the golden model.
- **ReorderedRU:** two or more register update events are out or order with respect to the golden model.
- **VanishedIC:** an instruction commit is missing from the acceleration trace.
- **CorruptedIC:** a mismatch in the program counter values between acceleration and golden model for instructions commits.
- **ReorderedIC:** one or more IC events are generated out of order in the acceleration with respect to the golden model.

C. Recording bit rate

We evaluate the proposed framework by analyzing its effectiveness in detecting bugs and its performance from a recording rate perspective. We first demonstrate 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 transfer one digest off-platform every N cycles, the recording rate R_{SBS} can be calculated as follows:

$$RR_{SBS} = [Reg(RU_{cnt} + RU_{cks}) + IC_{cks} + PC] \times \frac{IPC}{N}$$

where Reg is the number of registers; RU_{cnt} is the size of the RU counters, and RU_{cks} and IC_{cks} are the sizes of the RU and IC checksums; PC refers to the number of bits for recording the PC value of the last IC event of the epoch and IPC indicates the number of instruction committed per cycle.

We also notice that IBI checking is a special case of SBS with $N = 1$. IBI checking does not require counters because it records register values for each instruction. It only requires one register checksum and a few bits to indicate which register did the update. So, for IBI, the recording rate can be computed as:

$$RR_{IBI} = [(RU_{cks} + RU_{idx}) \times EPI + IC_{cks}] \times IPC$$

where EPI represents the average number of RU events reported per instruction. For our analysis we set $EPI = 2$ and $IPC = 0.9$. Note also that our target ARM architecture is 32-bit wide and has 64 integer registers and 168 special-purpose registers.

Table I reports a number of solutions points for both IBI and SBS checking. An IBI checking scheme that does not compress RU and IC events ($RU_{cks} = IC_{cks} = 32bits$), IBI(32) for short, presents a recording rate of 100.8 bits/cycle. If events are compressed down to 5-bits checksums as in [6], the recording rate is 27.9 bits/cycle. In contrast, the recording rate of our solution is highly dependent on epoch length, and can be unacceptable for small lengths: for a SBS scheme using 32-bit checksums and an epoch length of 10 – SBS(10,32) – the recording rate adds up to a whopping 757.44 bits/cycle (mostly due to the additional counters and PC value). Fortunately at more reasonable epoch lengths, such as SBS(32,1000) or SBS(32,10000), the recording rate is much smaller, even much smaller than for IBI, and very practical. Indeed, at practical epoch lengths, we can attain a 99% reduction in the recording bit-rate compared to an IBI solution. Finally, note that the architectural-state checking approach requires similar recording rates as SBS. We show in the next section that , while these latter two solutions are comparable in required recording rate, their bug detection accuracy is very different.

TABLE I: Comparison of recording rates of several epoch-length for our scheme (SBS) against two IBI checking approaches and the epoch-based architectural-state checking.

Checking scheme	Epoch length					
	1	10	100	1,000	10,000	100,000
IBI(32)	**100.8**	-	-	-	-	-
IBI(5)	**27.9**	-	-	-	-	-
SBS(32)	6948	757.44	82.01	8.83	0.97	**0.10**
ArchState	2248	224.8	22.48	2.25	0.22	**0.02**

D. Detection accuracy

We evaluated bug detection accuracy (fraction of bugs that is detected by comparing the digests with the golden model) of SBS using both checksum schemes described in Section III-B, and compared them against that of the architectural-state checker. Figure 4 indicates that the XOR and rotate-and-XOR checksum scheme achieved a detection ratio of >85% and 93%, respectively, on average. Moreover, both schemes are practically insensitive to aliasing, as suggested by the fact that the quality of results is unaffected by the epoch length. We further analyzed the false negative situations with a small epoch length (*i.e.*, 100) and the rotate-and-XOR scheme: we found a few to be caused by checksum aliasing, and most to be due to a change in instruction opcode but with no impact on the architectural state at any time during the execution (*i.e.*, no impact on any of the RU or IC events.) Overall, these findings demonstrate that SBS is effective in detecting bugs in acceleration and provides a valuable trade-off between temporal vs. spatial accuracy in bug detection. The results for the architectural-based checker in Figure 4 suggest significantly lower accuracy in bug detection across all testbenches and epoch lengths.

Finally, we analyze the relation between epoch length and detection accuracy. Figure 4 indicates that SBS' bug detection accuracy is fairly insensitive to epoch length. Indeed the accuracy degradation between an epoch length of 100 and one of 100,000 is only 7%. In contrast the architectural-state checker degrades quickly with longer epochs, up to 30%. With the next study, we also found that the rotate-and-XOR checksum scheme shines particular by detecting rarely-occurring bugs, where its detection accuracy is the only one to be consistently high over a broad range of epoch lengths (Figure 5).

This final study evaluates the detection abilities of SBS for rarely-occurring bugs, i.e., ReorderedRU and ReorderedIC. Since these bugs are due events occurring out-of-order, they can be easily masked by checksum schemes that are insensitive to order (XOR). Moreover, these bugs are extremely challenging to generate; thus, we inject them by manipulating the traces in post-execution.

We also tested an incremental CRC-32 scheme on these rarely-occurring bugs over a range of benchmarks. While this scheme can reach 87% in detection accuracy for the smallest benchmark (*i.e.*, mcf) when using an epoch length of 100, its detection accuracy rapidly degrades to less than 60% as the epoch length

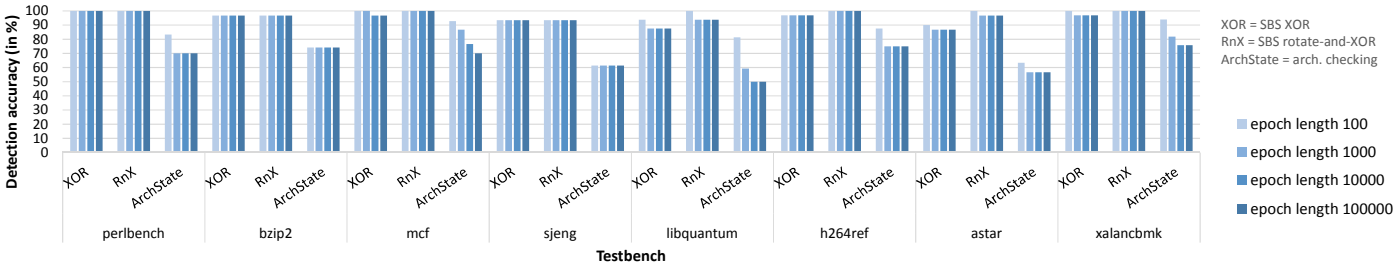

Fig. 4: Detection accuracy of the checking schemes under study over a range of testbenches.

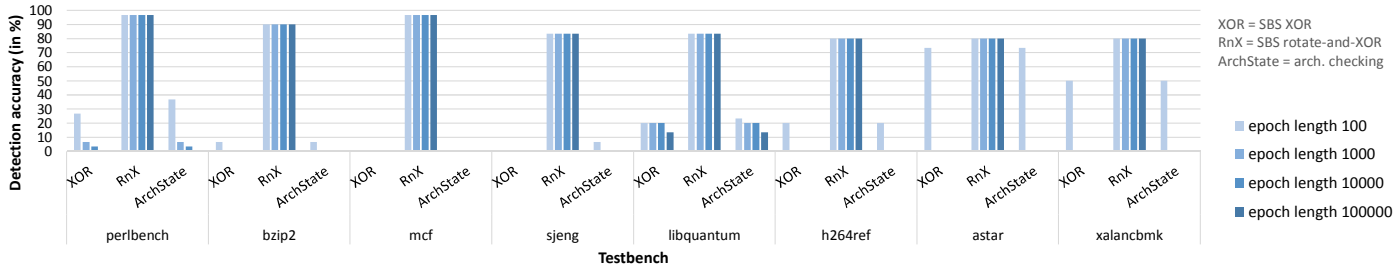

Fig. 5: The detection accuracy of different checking schemes for bugs that produces ReorderedRU or ReorderedIC symptoms.

grows. With all other tested benchmarks, CRC-32's accuracy never exceeds 70%. Considering the complexity of a parallel CRC-32 implementation [5,11] and its unremarkable detection performance, we believe it is not the ideal choice for our goals.

Figure 5 plots the findings of this experiments. The chart indicates that, while both SBS checking with XOR checksums and the epoch-based architectural-state checking fail to effectively capture bugs that manifest as ReorderedRU or ReorderedIC symptoms, the rotate-and-XOR checksum scheme brings in vast improvements in detecting bugs manifesting through these challenging symptoms, achieving at least 80% of accuracy across all testbenches. Moreover, we can observe from the Figure that this scheme's accuracy remains almost unaffected over a broad range of epoch lengths, making it an excellent candidate for the SBS checking flow. In contrast, the other schemes' detection degrades quickly at longer epoch lengths because of the higher chance of mutual cancellation of the reordering effects.

E. Performance overhead

Since our evaluation has been conducted on an architectural simulator rather than an acceleration platform, it is impossible to precisely estimate the performance overhead brought by our checking infrastructure. Therefore, we estimate it by comparing against the evaluation in [6], where the authors evaluate the AWAN acceleration platform simulating an IBM's POWER processor. In [6], the authors find that the biggest contributors to performance overhead are due to the draining of the trace buffers and, with smaller incidence, the additional logic footprint. Based on our Section V analysis, our logic overhead is much smaller than in [6], while storage requirements are comparable, thus, overall, we estimate a smaller contribution to performance overhead due to extra logic. Our buffer-draining overhead, estimated to be approximately 0.5%, is bound to be much smaller than theirs, since we have a much lower recording rate.

VII. Conclusion and Future Work

In this work, we proposed a novel scheme for architectural validation of microprocessor designs with acceleration platforms. This method, called SBS-checking, enables not only a highly accurate architectural validation at a very low recording rate on platform, but also provides fine-grained diagnosis capability. Overall, our solution enables much higher performance due to a >90% reduction in recording bit-rate, for the same quality of checking, compared to previous solutions.

Acknowledgements. This work was supported in part by NSF grant #0746425 and by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We would also like to thank Andrea Pellegrini for his contribution in providing insightful suggestions and the setup of the experiments.

References

[1] J. Babb et al. 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] G. Campobello, G. Patane, and M. Russo. Parallel crc realization. *IEEE Trans. on Computers*, 52(10), 2003.

[6] D. Chatterjee et al. Checking architectural outputs instruction-by-instruction on acceleration platforms. In *Proc. DAC*, 2012.

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

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

[9] G. Ganapathy et al. Hardware emulation for functional verification of K5. In *Proc. DAC*, 1996.

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

[11] C. Kennedy and A. Reyhani-Masoleh. High-speed parallel CRC circuits. In *Proc. ACSSC*, 2008.

[12] Y.-I. Kim et al. Communication-efficient hardware acceleration for fast functional simulation. In *Proc. DAC*, 2004.

[13] J.-G. Lee et al. Simulation acceleration of transaction-level models for SoC with RTL sub-blocks. In *Proc. ASPDAC*, 2005.

[14] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proc. MICRO*, 2007.

[15] A. Meixner and D. Sorin. Error detection using dynamic dataflow verification. In *Proc. PACT*, 2007.

[16] M. Shabtay et al. Building transaction-based acceleration regression environment using plan-driven verification approach. In *DvCon*, 2007.

[17] P.-S. Tseng et al. Simulation/emulation system and method, 1999. U.S. Patent No. 6009256A.