Checking Architectural Outputs Instruction-By-Instruction on Acceleration Platforms

Debapriya Chatterjee†, Anatoly Koyfman‡, Ronny Morad‡, Avi Ziv‡ and Valeria Bertacco†

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

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

ABSTRACT

Simulation-based verification is an integral part of a modern microprocessor's design effort. Commonly, several checking techniques are deployed alongside the simulator to detect and localize each functional bug manifestation. Among these, a widespread technique entails comparing a microprocessor design's outputs with a golden model at the architectural granularity, instruction-by-instruction. However, due to exponential growth in design complexity, the performance of software-based simulation falls far short of achieving an acceptable level of coverage, which typically requires billions of simulation cycles. Hence, verification engineers rely on simulation acceleration platforms. Unfortunately, the intrinsic characteristics of these platforms make the adoption of the checking solutions mentioned above a challenging goal: for instance, the lockstep execution of a software checker together with the design's simulation is no longer feasible.

To address this challenge we propose an innovative solution for instruction-by-instruction (IBI) checking tailored to acceleration platforms. We provide novel design techniques to decouple event tracing from checking by including specialized tracing logic and by adding a post-simulation checking phase. Note that simulation performance in acceleration platforms degrades when increasing the number of signals that are traced; hence, it is imperative to generate a compact summary of the information required for checking, collecting and tracing only a few bits of information per cycle.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*

General Terms

Design,Verification

Keywords

Simulation Acceleration, Checking, Checking on Acceleration

1. INTRODUCTION

Verification remains one of the most challenging and time consuming activities in the modern microprocessor design process. Shrinking transistor sizes have enabled a massive increase of microarchitectural complexity in microprocessors over the past decades. As a result, the verification effort needed for these designs has also increased tremendously. Simulation-based validation is still the workhorse of verification in the industry: a large collection of test regression suites are simulated on different models (architectural, RTL-level, structural) of the processor under verification, to check whether the design adheres to the original specification. To attain an acceptable degree of functional verification coverage for a modern microprocessor, billions of simulation cycles are executed on each new revision of the processor in development. Clearly, the success of simulation-based validation is closely tied to simulation performance. Unfortunately, the performance of software simulation tools on complex designs, such as microprocessors, falls far short (1-10 cycles per second) of what is required to complete validation in a reasonable amount of time. Hence many functional verification teams in the industry rely on acceleration and prototyping platforms to meet their verification performance needs. However, even if simulation performance is much higher on these platforms, checking the functional correctness of a complex system, such as a processor design, presents many new challenges:

Mapping checkers to acceleration platforms: Many software-based checkers designed for microprocessor validation are expected to execute in lock-step with the RTL or gate-level simulation of the processor model. However, acceleration platforms can only simulate synthesized logic descriptions. Hence, if a software checker is sufficiently complex that it cannot be easily mapped into hardware, the checking solution cannot be brought onto the acceleration platform. Lock-step execution of software checkers in the host is infeasible since it would require frequent transfers of values from the platform and thus hinder performance unacceptably.

Acceleration performance: The performance of acceleration degrades when increasing the number of recorded/monitored signals or events. This effect is present even in software simulation-based validation; however, it is much more prominent in acceleration. Indeed, tracking a large number of signals in acceleration can degrade its performance to the point of cancelling its benefits over software simulation (as discussed in Section 4). Given an accelerator architecture and a design mapped to it, it is possible to estimate the slowdown incurred from the number of bits being traced. Thus, checkers that require to monitor a large number of signals cannot be adopted in acceleration in a straightforward fashion.

In the wake of these challenges there is a growing need of innovation to transform traditional microprocessor checking methodologies so that they fit in the constraints of accelerator/prototyping platforms while still delivering comparable verification quality and accuracy as their software simulation-based counterpart.

In this work, we present an architectural checking solution for microprocessor cores on acceleration platforms. Our solution performs what we call "instruction-by-instruction" (IBI) checking, that is, it verifies the outcome of each instruction completed in the accelerated simulation by comparing it with an architectural golden model. We achieve our goal by applying a number of major transformations to a baseline software simulation-based validation methodology. We solve the challenge of mapping complex logic to the platform by decoupling the recording of events from their checking. We also address the challenge of poor performance due to large data tracking by computing a summary of the information required by the checker before transferring the data off-platform. The proposed solution retains almost all the capabilities of its software counterpart but does not compromise the performance of acceleration. We successfully deployed and evaluated this solution in the validation of an upcoming IBM POWER processor.

2. RELATED WORK

Simulation accelerators and emulation platforms have been traditionally used to boost the productivity of the microprocessor validation effort [8, 12], and they play an even more critical role today, in light of the increased complexity of these designs. However, acceleration-based flows usually have a coarse checking granularity, that is, they can label a test as passed or failed after its completion but, in case of failure, no additional information is available related to the time/location of the bug manifestation. Comparing architectural state between a purely software-simulated design model and a golden architectural software model at instruction boundaries, or at other synchronizing boundaries, has also been a commonly deployed method for microprocessor validation[14, 5]. The key reason why this methodology has not yet been considered for acceleration, with the golden model running in software on a host platform, is that connecting these two components (golden model and accelerated design) is both difficult (due to lack of debugging support) and detrimental to performance [5]. Obtaining scan values from a silicon prototype and comparing them to a RTL golden model to detect divergence analysis during post-silicon debug has been proposed in [4]; however, this solution is only used to diagnose electrical faults.

More recent silicon-debug solutions, such as IFRA [11], introduce additional logic into the design to trace the flow of an instruction through various microarchitectural blocks and use this information with a post-simulation analysis tool to locate the manifestation of a possible design bug. Though our solution has a similar organization, *i.e.*, decoupled tracing and checking components, we are interested in the manifestation of a failure in the architectural state. Moreover, IFRA cannot detect divergence of the processor execution from the ideal model on its own, in fact it relies on post-triggers for this information. Our solution is focused on detecting the first point of divergence in the architectural state, hence it solves an orthogonal problem. Certain runtime verification techniques such as DIVA [2], introduce a lightweight companion processor to check the architectural state of the main processor, but these solutions operate at runtime, past design debug. Standards for hardware support for a variety of trace and control instrumentation for system debug has been proposed for the embedded domain [10], though they are not meant to be used for debugging the processor core design.

3. IBI BACKGROUND

Instruction by instruction (IBI) checking, or golden model based validation, is a well known checking technique that has been used in processor verification for many years [9, 13]. IBI compares the architectural events produced by each executed instruction with those required by the processor specification. This technique provides a simple way to distinguish deviations from the desired behavior. It does not depend on the internal implementation of the processor, and can be used with any microarchitecture implementing the same instruction set. An additional benefit of this approach is the relative ease of debugging: the corresponding checker recognizes the exact spot of the deviation in time and thus it enables the time localization of the problem.

A typical IBI checking methodology works as follows. A test generator (*e.g.* [7, 1]) produces a test program containing the results expected by the processor specification after each instruction (the expected results). These results are usually obtained using a software that can calculate the expected results after each instruction, known as a golden model. Then the checker environment compares these results to the ones produced by the processor simulator for the same test program [9, 13]. The checker environment needs to identify when an instruction execution completes and what resources were modified because of the instruction execution. It also needs to account for the behavior that cannot be predicted by the golden model (*e.g.* external interrupts), or are not fully defined by the specification (*e.g.* values of some registers become “undefined" when exceptions occur).

4. ACCELERATION BACKGROUND

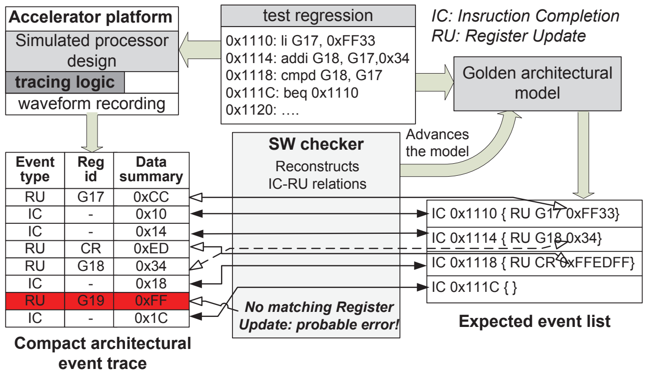
To boost the performance of simulation, a number of platforms have recently attracted interest as alternatives to software-based simulation: acceleration [6], emulation / proto-typing platforms [3] and post-silicon validation [11]. Hardware-accelerated simulation platforms are composed of large arrays of customized ASIC processors, specifically designed to simulate logic gates concurrently. To target these platforms, the design under verification (DUV) must be synthesized into a structural netlist, and then the corresponding logic gates are mapped to the execution substrate. Acceleration platforms have limited logic capacity, and even within their capacity limit, they may experience a performance penalty for large designs and depth of simulation. This logic capacity limit prohibits the mapping of any arbitary checking solution into equivalent hardware and simulating it alongside the design. Thus, only checkers that result in low logic overhead can be tolerated. In our case, we evaluated our solution on a single core of an upcoming POWER processor, which in itself fit within the capacity limit of the accelerator used in our evaluation. The limit was also maintained after the addition of the logic required by our technique.

Acceleration platforms usually allow the collection and transfer of waveforms for debugging purposes [6], but the transfer slows down the simulation, eroding the key benefit of acceleration. In general, the more signals are observed and transferred, the lower the acceleration performance. However, the precise relation between acceleration performance impact and signals traced depends on the architecture of the accelerator. Reducing the number of recorded signals per cycle (thus the trace data generation rate) is extremely important for a successful checking solution for acceleration platforms. Emulation platforms have very similar tradeoffs, except that the hardware acceleration fabric consists of programmable look-up tables (FPGAs). Hence, our solution could be adapted to emulation.

5. IBI FOR ACCELERATION PLATFORMS

In this section we present our instruction-by-instruction checker solution for acceleration platforms. Our technique enables this validation methodology on fast accelerated simulations, thus boosting the amount of simulation cycles that can be checked within a given amount of time. In our solution, we run the same test on the processor model simulated in the acceleration platform and on the golden model running on the off-platform host, and then compare results. To make the comparison possible, we need to collect relevant information about the retired instructions and architectural resources modified from the acceleration platform, and transfer it off-platform. The actual comparison is then performed by a dedicated software checker, capable of running the golden model on the same test and compare the two sets of results. As mentioned earlier, the acceleration advantage decreases when increasing the amount of recorded information and the size of simulated logic. Hence, one of our design goals is to record as little information as possible and incur as little hardware overhead as possible, all while delivering accurate bug detection capabilities.

Based on the observations above, our solution comprises the following two components: i) a dedicated, on-platform, logic block to record a compact summary of architectural events and ii) an off-platform software checker module that considers the recorded data and analyzes it in light of a golden model output. This decoupled approach enables us to get around one of the fundamental challenges discussed previously, minimizing on-platform logic


Accelerator platform
Simulated processor design
tracing logic
waveform recording
test regression
IC: Insruction Completion
RU: Register Update
Golden architectural model
SW checker
Reconstructs IC-RU relations
Advances the model
No matching Register Update: probable error!
Expected event list
Compact architectural event trace

```
0x1110: li G17, 0xFF33
0x1114: addi G18, G17,0x34
0x1118: cmpd G18, G17
0x111C: beq 0x1110
0x1120: ....
```

Event type	Reg id	Data summary
RU	G17	0xCC
IC	-	0x10
IC	-	0x14
RU	CR	0xED
RU	G18	0x34
IC	-	0x18
RU	G19	0xFF
IC	-	0x1C

```
IC 0x1110 { RU G17 0xFF33}
IC 0x1114 { RU G18 0x34}
IC 0x1118 { RU CR 0xFFEDFF}
IC 0x111C { }
```

Figure 1: **Overview of our solution** to provide IBI checking on acceleration platforms. The Figure illustrates the test running on the platform (left) and on the off-platform software (right). The bottom left table shows an example of data transferred off-platform.

overhead. However, it also imposes a substantial redesign of the checking approach. We will check instruction completions and registers only (similar to many other IBI solutions) because memory behavior is very difficult to trace and predict in modern architectures. To achieve this we will record two types of events on the acceleration platform - instruction retirements and register updates. We do not focus on memory behavior, as it is common for other golden model solutions, since that requires specialized solutions beyond the scope of this work. We then compress the collected information on-platform to minimize the amount of data transferred. As a result, we must only record and transfer a few bits per cycle, thus maintaining the acceleration performance advantage. The on-platform tracing logic is simulated along with the processor in the acceleration platform. To minimize data recording, we do not track information that ties registers to a specific instruction; instead, we rely on the off-platform software, to reconstruct these connections based on the information recorded. Figure 1 presents an overview of our solution showing the components on the accelerator and on the off-platform software. It also outlines the type of data that is traced and transferred.

5.1 Hardware-based data tracing

From a high level standpoint the collection of information for our purposes appears to be straightforward; however, when applied to an industry processor, many aspects become challenging. The processor in question is a modern, server class, superscalar out-of-order processor with simultaneous multi-threading allowing 8 simultaneous threads per core. Hence, each architectural event is a complex combination of several microarchitectural events. To correctly identify and log individual architectural events, we need a number of microarchitectural monitor points, mapped together with the design onto the accelerator. The main architectural events to be collected for our purposes can be grouped into the following 3 major classes:

Instruction completion: Since the underlying processor is out-of-order, we can only obtain a finalized instruction retirement event when an instruction is committed. This information is gathered from the group completion table of the processor design, where instruction completion events are built from micro-operation completion information.

General purpose register activity: This group of registers includes integer general purpose registers (GPR), floating point registers and vector registers (VR). Accessing update events and values incurs an additional layer of indirection due to register renaming deployed in out-of-order microarchitectures.

Special purpose register activity: Special purpose registers (SPR), such as several status registers, are easier to handle, since they are directly mapped and have explicit signals that identify a write to a special purpose register. We chose to collect information on a subset of special purpose registers that are either part of or closely related to the architectural state.

Note that we record all instruction completion events and all update events on the monitored registers. However, we perform lossy compression on the data associated with each event, *i.e.* completed instruction addresses or values written to a register, to reduce the number of bits recorded on the acceleration platform.

5.2 Off-platform software checker

As discussed in previous sections, our instruction-by-instruction checker strives to identify all discrepancies between the simulated processor behavior and its golden model. A processor's architectural state is defined by the values of the architectural registers (including general purpose registers, certain special purpose registers that affect execution flow and program counter) and the contents of memory. We assume that events that are not captured by the golden model (such as memory updates due to shared memory) do not appear in the test case. Thus, our single core processor model can be considered to be executing correctly, as long as program flow and architectural state are identical to that of the golden model. Hence, tracking the completion of instructions (program flow) and any modification to architectural registers is sufficient to check the correctness of execution. We encountered two key challenges in developing the off-platform checker, discussed below:

Reconstruction of instruction flow: A significant problem we had to address was the lack of close time correlation between an instruction retirement and its register events. This information cannot be reconstructed simply from the acceleration trace. Thus, in our solution we maintain a list of all registers that should have been modified by a completed instruction. We expect that for each such register, the first modification report that appears after the completed instruction will contain the correct value, and this report will appear within a bounded number of cycles. This solution is based on the assumption that registers are modified only after the corresponding instruction completes, and all associated register modifications are reported within a bounded number of cycles. However, we also had to consider the case where a register update is received before its corresponding instruction completion: in this case we must search for a matching event from the golden model over a few instructions downstream. If we do not find the matching event within a few instructions, we flag an error. We have run experiments to compare the results reported by a state-of-the-art software-based IBI checker to the results reported by our solution. We learned that the only difference lies in identifying which instruction is the root of the execution path deviation from the golden model execution (when such deviation exists). Our checker may report an instruction that is close to the actual deviating instruction (usually the next instruction), which we found satisfactory for effective debugging.

Handling interrupts for checking purposes: External interrupts and other non-deterministic events are not predictable by the golden architectural model; however, they are still included in the acceleration traces. External interrupts can still be identified from the address of the corresponding interrupt handler and specific values of the related control registers. Our solution mimics the effect of the interrupt routine by modifying the associated status registers and other architectural resources in the golden model and then it resynchronizes the model with the trace.

6. DATA COMPRESSION

As discussed in Section 5.2, a central goal of our work is to keep the amount of data recorded per cycle at a bare minimum, to maintain the performance advantage of acceleration, while still providing acceptable detection accuracy. To this end, we compress the data associated with each event, such as register update values and addresses of completed instructions. A lossy compression scheme, such as a checksum is ideal for this purpose, since we are only interested in identifying value deviations. So, as long as a different value produces a different checksum with high likelihood, it serves the purpose. Moreover, another important aspect in the development of our solution, is that the additional hardware required to implement the compression scheme should have minimal logic overhead and minimal logic depth. Hence, a compression scheme that involves little additional logic and does not add substantial delay to the critical path is favored over a more complex scheme.

6.1 Register update values

Value discrepancies in register updates can often be discerned using a checksum over a small subset of the bits, without requiring a complete value comparison. We strive to use only a few (say, less than 8) bits of encoded information for each register value field (32 bit / 64 bit). The basic idea is to compute a checksum from the value generated by the simulated hardware and perform the same operation on the value generated by the reference model for each register update in the software checker. For the sake of our checker solution, a checksum match is considered a valid register update. Since all checksum schemes are a hash function from a set of size 2^{64} (for 64 bit registers) to a set of size 2^c, where c is a small value, some amount of aliasing is unavoidable. However, we found that blocked parity schemes, presented below, provide sufficient accuracy in practice for the typical error scenarios that we encountered.

Blocked parity schemes partition the data vector into several distinct blocks and then compute single bit checksums for each block. The concatenation of these bits provides the final checksum. This approach is guaranteed to detect any bit value difference, as long as the number of single bit errors within each block is odd. A benefit of this approach is that its computation is extremely low cost in hardware, simply requiring a few XOR gates. However, this approach is ineffective for scenarios where errors manifest with an even number of localized bit-flips, which may occur all within one, or a few, blocks. To address this situation we build blocks on non-contiguous bits, scattering the bits over the checksum blocks. With this technique, an error affecting a few contiguous bits has a much higher chance of detection. The experimental evidence presented in Section 8 supports this intuition.

6.2 Retired instruction addresses

The data associated with each retired instruction is the address of the committed instruction. To compress this values we use a very simple scheme, recording only the last few bits of the address. Even though this scheme is prone to aliasing, it works very well in practice. Indeed, it allows us to identify an execution divergence from the golden model fairly precisely, since the probability of execution starting at an aliased address leading to the same sequence of register updates as the correct execution is extremely low.

7. ON-PLATFORM TRACING UNIT

As discussed earlier, there are several types of data collected on the acceleration platform originating in different regions of the design at a variable rate. To manage this flow of data, we developed a novel unified scheme to collect and organize it for on-platform storage, before it can be transferred off-platform. To this end, we first need a mechanism to identify which registers are updated on a particular cycle or which instruction groups have completed, so that we only record new values for the relevant registers/addresses. Second, we need a mechanism to present this data in a structured fashion, so that it can be recorded efficiently by the acceleration platform's data logging mechanism. We note that, although the maximum number of simultaneous events in a clock cycle can be quite high, the average number of events per cycle is fairly small. Hence, a recording mechanism that can handle transient peaks in the number of events and can present data at a constant rate to the platform's debug support unit would be ideal. A possible solution to this second requirement is a first-in first-out buffer that allows the storing of up to a few entries at a time and it is drained at a constant rate. This section discusses how we achieved these requirements.

7.1 Select and encode logic

The first task of the tracing unit focuses on selecting and encoding different types of events as they are flagged during a clock cycle. In the platfom there are a number of data lines and corresponding valid lines coming from different parts of the processor and corresponding to different special purpose registers or instruction completion events that we want to track. Our goal is to be able to store the relevant data at each cycle (as signaled by the corresponding valid lines) while also tracking the correct source for the data. By doing so the off-platform software is able to reconstruct the sequence of events to be checked against the golden model.

The goal of the select and encode logic unit can be formally expressed as follows: given a collection of N signal lines, presented as an ordered list, up to any M lines among those can request data logging on any given clock cycle. The task of this unit is to identify and encode the position in the list of the M lines in preparation for storing them along with the data itself. Ultimately, these positions will be used to identify the source of the corresponding data value. This problem is also known as the "detect and encode all ones" problem: one straightforward solution would be to use a chain of priority encoders: the first encoder is responsible for the highest order position, which is then masked and the entire vector of N lines is passed down to the next encoder. While simple, this solution creates a deep combinational logic block, which could hamper the performance of acceleration.

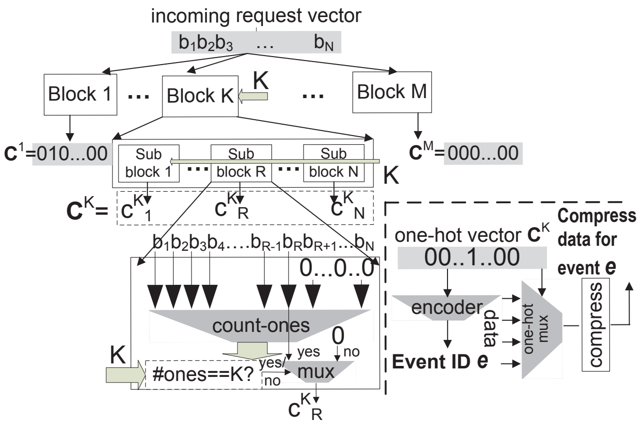


Figure 2: **Detector block to identify the source of data** to be logged in a given clock cycle of simulation acceleration.

Our goal in developing this unit is to develop a design that is most suited for acceleration platforms, even if it may entail a non-minimal area footprint in silicon. To this end, we devised an alternative solution, that has a much smaller logic depth. Our solution uses a parallel detection scheme, where each detection block is responsible for generating a one-hot encoded vector corresponding to the line position for which the block is responsible, if that line has data available. If no logging data is generated from that line during a cycle, the block should simply output a vector of zeros. Figure 2 illustrates our solution: we use M detection blocks, since we have at most M lines generating data within one cycle. Each block receives in input a value K, and generates a one-hot encoded vector where the 1-bit is in the position of the K-th line producing data in that cycle. For instance, if during a cycle lines 4, 7 and 11 produce data to be logged, then block 1 should have a one in position 4, block 2 should have a one in position 7 and block 3 should have a one in position 11.

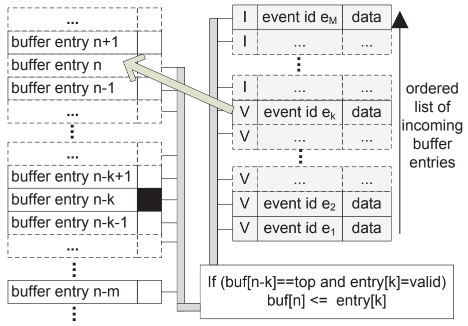


Figure 3: **Trace buffer writing unit**. Each buffer entry is associated with a writing unit. Each unit determines which data logged in the cycle should be stored in the position for which it is responsible.

7.2 Trace buffer

Once the relevant data has been selected and encoded for logging, we need a hardware block to record the architectural events. To this end we use a trace buffer that must be capable of handling up to M entries in each clock cycle, while allowing a constant R entries to be read. Such a buffer is typically realized via a circular buffer with read and write pointers. However, multiplexors are needed to realize these pointers. Unfortunately, they also increase the logic depth of the design, particularly when the number of buffer entries is large. Hence, we adopted an alternative design, where the buffer is implemented as a shift-buffer, so that the constant number of read operations in each simulation cycle corresponds to a constant number of shifts. A bit is associated with each entry to indicate the first free entry, and independent write units are associated with each buffer entry. Each write unit has access to its corresponding entry and the M preceding ones, and it determines what to write in its entry based on the number of write operations to be completed in the cycle. This design is shown in Figure 3: the implementation is parallel and logic depth is kept minimal.

8. OPTIMIZING DESIGN PARAMETERS

In this section we discuss a number of analyses that we conducted to optimize our checker design. The most influential parameter for the performance of our checker is the number of bits traced per cycle. This parameter is determined by the product of two other parameters: (i) the number of trace buffer entries being drained per clock cycle and (ii) the number of bits per entry in the trace buffer. The first one has to be equal or greater than the average rate of traced events generated by the processor, since we are using a finite size buffer. The other is determined by the number of bits required to describe the event type (which is a fixed value), along with the number of data-bits associated with each entry (a variable value). Below we discuss how we computed the near ideal value for each of these components.

Trace buffer size: Since the number of write operations to the buffer varies from cycle to cycle, while draining rate remains constant, we need to ensure that the average draining rate is higher than the average generation rate. Even then, bursts of generation may create backlogs in the buffer. In Figure 4 we show how different buffer draining rates affect instantaneous buffer occupancy. We derived a queuing theory-based estimate for our buffer size, which

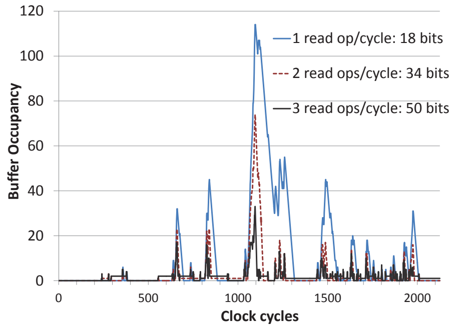


Figure 4: **Buffer occupancy peaks at varying rates of draining**.

ensures a very low probability of overflow, while using the lowest possible draining rate. Assuming a pessimistic generation rate, a draining rate of 3 entries/cycle was found to be sufficient and a corresponding buffer size of 512 is adequate.

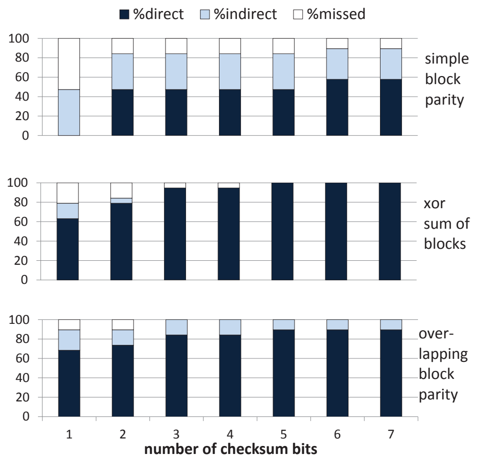


Figure 5: **Detection accuracy of a range of checksum schemes**. Register value discrepancies can be either detected at register update (direct), or in downstream computation (indirect), or missed.

Checksum width: The number of bits recorded per buffer entry is dictated by the number of data bits. We want to store a minimal number of bits in the checksum, while still detecting value discrepancies caused by a functional bug. Hence, we investigated the detection accuracies of several blocked parity schemes, as described in Section 6.1, over buggy traces diverging on a register value update.

To this end, we varied the number of checksum bits from 1 to 7, while the original register values are 64-bits wide. We studied three different checksum schemes as reported in Figure 5, and estimated the minimum checksum bit width required to detect typical value discrepancies. The schemes we evaluated are: (i) Simple blocked parity, where a single parity bit is computed from each portion of register data and appended to the final checksum. (ii) XOR sum of blocks, where the checksum is obtained by applying bitwise XOR to all same size sub-blocks of the register value. (iii) Overlapping block parity, similar to (i), but with overlapping partitions. The sample size for this study was 500 traces with register value corruptions similar to those of actual buggy traces. From Figure 5, it can be gathered that typical discrepancies can be detected with as little as 5 bits of XOR sum of blocks. This allows storing only 8 (event-type ID, one out of 256) +3 (thread ID) +5 (checksum)=16 bits in each trace buffer entry. In addition to this we have 1-bit indicating that the values at the buffer output are valid and another 1-bit to indicate buffer overflow. These 2 bits are constant overhead irrespective of number of entries read.

9. EXPERIMENTAL RESULTS

Our solution was implemented for an upcoming POWER processor core design on the AWAN accelerator [6] platform. We evaluated the capability of our solution to detect bugs as well as its performance. The IBM SixthSense tool-chain was used to design and synthesize the hardware blocks for our solution. The processor core netlist consisted of a few million logic gates, and the resulting logic overhead was within 20%.

9.1 Bug detection capability

Any discrepancy of the processor's behavior from the golden architectural model due to a probable functional bug is detected as one of the following situations (symptoms) by our IBI checker:

1. Register value mismatch: Updated value of a register does not match with predicted value from golden model;
2. Unexpected register update: An architectural register update event takes place in the design but not in the golden model;
3. Unaccounted register update: A register update event takes place in the golden model but does not occur in the design;
4. Wrong instruction: The instruction address of an executed instruction is in disagreement with the golden model;

We obtained a set of 145 architectural event traces that exposed actual functional bugs. These 145 constituted the entire set of buggy traces that we had access to. To evaluate the bug detection capability of our checker, we ran the same traces on our off-platform software checker to determine if our accelerator-based checker could also detect the occurrence of the bugs. All 145 testcases exposed a bug in our setup; in addition the symptoms reported matched those of the software-based golden model solution. We report in Table 1 the distribution of the bugs detected according to the type of symptom flagged by our checker. As it can be noted, a large portion of the issues are due to unaccounted/unexpected register updates. All these problems were detected within 5 instructions from the first point of golden model/accelerator divergence.

Symptom	#occurences
Register value mismatch	21
Unexpected register update	30
Unaccounted register update	89
Wrong instruction	5

Table 1: **Distribution of bugs detected by our solution**.

Since we do not compress the information regarding which architectural register (among the monitored subset) is updated, we detect all discrepancies that are not affected by checksum aliasing. However, even in this latter case, often the program flow diverges substantially due to the bug, and we can still flag the issue a few instructions downstream.

9.2 Tracing overhead

The amount of logic added for on-platform tracing purposes may impact the performance of the simulation. However, this is only the case if the overall logic size mapped to the platform (design + checkers) exceeds a certain threshold, dependent on the accelerator's characteristics. When abiding this threshold, the performance degradation due to the tracing logic comes from two sources (i) additional logic to simulate (ii) signal recording time.

To evaluate these effects, we measured the simulation acceleration performance of the POWER core design in several situations. The stimuli used for this study were regression tests lasting few million cycles. First, we run a baseline design with no tracing logic. Then we added the tracing logic, but without observing the trace buffer output. Then we also enabled tracing for the typical case, that is, 3 buffer entries are read per cycle, amounting to 50 bits of recorded information per cycle. Finally, we considered an extreme situation where 10 buffer entries are read per cycle, for a total of 162 bits. Figure 6 summarizes our findings, normalized to the simulation performance (between 10-100 kHz) of the baseline design with no tracing logic.

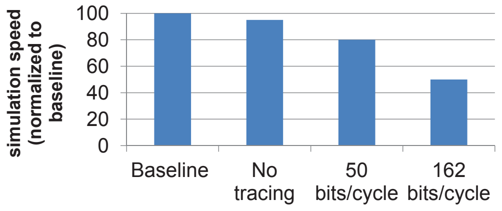


Figure 6: **Impact of tracing logic** on acceleration performance.

From Figure 6 we gather that our solution introduces only a 5% slowdown due to the tracing logic alone, and another 15% due to data logging. Even the extreme situation causes no more than a 50% slowdown in acceleration performance, a value still order of magnitudes better than software-based simulation.

10. CONCLUSIONS

In this work we presented a novel microprocessor design checking solution that provides architectural checking against a golden model for simulation acceleration. Our solution provides the same bug detection quality as its software-based counterpart. It enables architectural validation on acceleration platforms with negligible accuracy loss and moderate performance loss of approximately 20%.

Acknowledgments

This work was developed with partial support from the Air Force Office of Scientific Research, Grant #FA9550-09-1-0164. The authors would also like to thank Ariel Birnbaum, Brian Mestan, Chris Abernathy and Kenneth Ward from IBM.

11. REFERENCES

[1] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-Pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test*, 21(2), 2004.
[2] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. MICRO*, 1999.
[3] M. Boulé, J.-S. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *Proc. ICCD*, 2006.
[4] O. Caty, P. Dahlgren, and I. Bayraktaroglu. Microprocessor silicon debug based on failure propagation tracing. In *IEEE Transactions on Computers*, 2005.
[5] Y.-S. Chang, S. Lee, I.-C. Park, and C.-M. Kyung. Verification of a microprocessor using real world applications. In *Proc. DAC*, 1999.
[6] J. Darringer, E. Davidson, D. Hathaway, B. Koenemann, M. Lavin, J. Morrell, K. Rahmat, W. Roesner, E. Schanzenbach, G. Tellez, and L. Trevillyan. EDA in IBM: past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), 2000.
[7] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology for microprocessors using the Genesys test-program generator. In *Proc. DATE*, pages 434–441, March 1999.
[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. M. Ludden et al. Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems. *IBM Journal of Research and Development*, 46(1):53–76, 2002.
[10] A. Mayer, H. Siebert, and K. McDonald-Maier. Boosting debugging support for complex systems on chip. *Computer*, 40(4), 2007.
[11] S.-B. Park, T. Hong, and S. Mitra. Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *IEEE Trans. on CAD*, 28(10), 2009.
[12] V. Popescu and B. McNamara. Innovative verification strategy reduces design cycle time for high-end SPARC processor. In *Proc. DAC*, 1996.
[13] D. W. Victor et al. Functional verification of the POWER5 microprocessor and POWER5 multiprocessor systems. *IBM Journal of Research and Development*, 49(4), 2005.
[14] J.-S. Yim, Y.-H. Hwang, C.-J. Park, H. Choi, W.-S. Yang, H.-S. Oh, I.-C. Park, and C.-M. Kyung. A C-based RTL design verification methodology for complex microprocessor. In *Proc. DAC*, 1997.

Supplementary material

S1: Insights on micro-architectural to architectural event translation

Instruction completion: To properly sample the information required for each instruction at its correct completion time, we leverage the fact that instruction retirement is performed in the program order. This remains true even when the processor has an out-of-order microarchitecture. This observation is utilized to create instruction completion events from the retirement of an instruction at the reorder buffer. However, there is another layer of complexity associated with identifying instruction completion events. Even though the processor has a RISC architecture, for performance reasons each instruction is sub-divided into micro-operations. At the retirement stage, the actual visible events are micro-operation retirements. Hence, to generate retirement events at the architectural level, micro-ops completions are collapsed together based on the directives of the instruction dispatch table. The associated instruction address is also gathered from the instruction dispatch table.

General purpose registers: All general purpose architectural registers are dynamically mapped to registers in the physical register file; hence, a map entry is associated to each register update event, and it provides the architectural index along with the tag that indexes the written value in the physical register file. From this information, we collect the most recent written value to an architectural register on the notification of a write event. This is achieved using a significant amount of decoder and multiplexing logic.

S2: Select and encode logic - insights

As mentioned in Section 7.1, a straightforward solution to this problem entails using a cascade of priority encoders. However, combinational implementations of this approach would lead to an unacceptable logic depth, while sequential solutions would require more than one clock cycle to compute the output value.

Though the cascade solution and some slight variations of it (optimizing the detection part but retaining the cascaded logic structure) describes a correct logic implementation, the problem that remains is the enormous logic depth required for implementing even a small cascade depth. This would lead to forming a very long critical path in the logic and would severely limit simulation performance in the accelerator. Solutions that employ sequential logic to get around the cascading problem exist but we can not use this solution since we are not assured to have a fixed number of idle cycles where absolutely no request arrives, we need to finish detect and encode in one cycle only.

In contrast, our solution, shown in Figure 2, is combinational with a small logic depth. In the Figure, we use M concurrent detection blocks, each receiving in input a distinct value K. Each block outputs a one-hot vector with a 1 in position of the K-th input line producing data in that cycle. Blocks for which the input value K is larger then number of data lines producing data will simply output a zero vector.

To achieve this functionality, each block is organized into N sub-blocks in order, one for each of the incoming data lines. The input value K is passed along to all sub-blocks. The one sub-block connected to the line with the K-th data value must output a 1 is there is a new value on the line, and a 0 otherwise. All other sub-blocks simply output 0. In order for a sub-block to determine if it is connected to the line with the K-th data value, it counts the number of lines providing data values in line indices lower than its own order position. The only logic required to implement this solution includes a tree of few-bits adders and a comparator of only $\log_2 M$ bits for each sub-block.

S3: Trace buffer size selection

To determine an adequate size for the trace buffer, we applied the following reasoning. Let's call I the average number of instructions completed per clock cycle (*i.e.* IPC) and W the average number of register updates per completed instruction (including SPR updates). Then, the average number of entries generated per cycle is: $G = I \times (1 + W)$. The draining rate of the tracebuffer (D) has to be greater or equal than the generation rate to avoid overflow, *i.e.*, $D \geq G$. In addition, we need to take burst into account: if the average of G over a window of T clock cycles is $G_T > D$, then a backlog of $(G_T - D) \times T$ entries has been created, which the buffer must accommodate. As it can be noted, the buffer always drains at steady state; however, its instantaneous behavior maybe adversely affected by small draining rates. Our ideal situation is to have a minimal draining rate and experience overflow rarely.

To determine the probability of overflow we use a theoretical model from queuing theory: our problem can be modeled as a variant of a M/M/1 queuing model, where the average number of writes per cycle is the arrival rate λ and the service rate μ is the number of entries drained per clock cycle. This model holds since the number of events reported per clock cycle can be considered as a Poisson arrival process with mean rate $\lambda = G$, whereas the average rate of departure is $\mu = D$. Now, the probability of queue occupancy q exceeding a given value Q, $P(q > Q)$ is given by $(\frac{\lambda}{\mu})^{Q+1}$. M/D/1 is a more accurate model since our draining rate is constant; however, since there is no closed form expression for this case, we conservatively approximated it with M/M/1. We can plot this probability as a function of buffer size, for the worst case generation rate. To obtain the worst case generation rate we used an artificial regression that dispatches a large number of independent ADD instructions for each of the 8 threads simultaneously. This forces the processor to operate at high throughput and approach the theoretical maximum of the average rate of event generation.

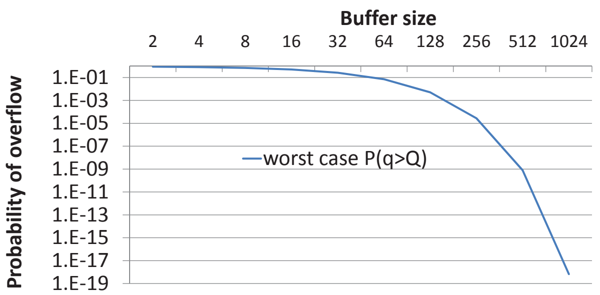


Figure 7: **Overflow probability for finite buffer size assuming M/M/1 queuing model**.

We observed an average instruction completion rate I of 1.2 per clock cycle, along with an average rate of register updates per instruction W of 1.0. We pessimistically assumed 0.4 SPR writes per instruction completion, leading to a total generation rate of $1.2 \times (1 + (1.0 + 0.4)) = 2.88$, which is then rounded to the closest larger integer, leading to a buffer draining rate of 3. With these parameters, we plotted the overflow probability in Figure 7 for different buffer sizes and concluded that a buffer size of 512 corresponds to a minimal probability of overflow.