Approximating Checkers for Simulation Acceleration

Biruk Mammo*, Debapriya Chatterjee*, Dmitry Pidan†, Amir Nahir†, Avi Ziv†, Ronny Morad†, Valeria Bertacco*
*University of Michigan †IBM Research Lab, Haifa
{birukw,dchatt,valeria}@umich.edu {pidan1,nahir,aziv,morad}@il.ibm.com

***Abstract*—Simulation-based functional verification is the key validation methodology the industry. The performance of logic simulators, however, is not sufficient to attain acceptable verification coverage on large industrial designs within the time-frame available. Acceleration platforms are a valuable addition to the verification effort in that they can provide much higher coverage in less time. Unfortunately, these platforms do not provide the rich checking capability of software-based simulation.**

We propose a novel solution to deploy those complex checkers, typical of simulation-based environments, onto acceleration platforms. To this end, checkers must be transformed into synthesizable, compact logic blocks with bug-detection capabilities similar to that of their software counterparts. Our "approximate checkers" trade off logic complexity with bug detection accuracy by leveraging novel techniques to approximate complex software checkers into small synthesizable hardware blocks, which can be simulated along with the design on an acceleration platform. We present a general checker taxonomy, propose a range of approximation techniques based on a checker's characteristic and provide metrics for evaluating its bug detection capabilities.

I. Introduction

Functional verification is a central component of digital design development, requiring a great deal of engineering resources and time. One of the mainstream verification methodologies entails using a logic simulator (a software application) that simulates the behavior of a design implemented in a hardware description language (Verilog or VHDL). A testbench provides stimuli to the design under verification (DUV), while a mix of embedded checkers, typically developed in a high-level language (Vera [1], *e* [10], C++, *etc.*), monitors the correctness of the DUV's activity. In simulation, the ability to observe and control any signal within the design allows close monitoring of the design's activity and provides comprehensive diagnosis capabilities.

Unfortunately, this methodology is severely limited by the vast complexity of modern digital designs. Longer and more complex regression suites are required to explore meaningful aspects of a design and yet, simulation speed degrades with increasing design size. As a result, significant portions of a design's state space are left unexplored during simulation. Alternative simulation/execution platforms, namely acceleration-based verification [17], [11], emulation [6], [16] and post-silicon validation [3], [7], are starting to attract more interest because of the performance boost they can provide. Among these, hardware simulation accelerators consist of large arrays of customized ASIC processors designed specifically to simulate basic logic components concurrently. To target such platforms, a DUV must first be synthesized into a structural netlist, which is then mapped to the platform. In current industry practices, the testbench remains on the host computer and controls the simulation running remotely on the acceleration platform (Figure 1). Selected signals are logged on the platform itself and periodically off-loaded to the host where they are checked by a number of software checkers to establish

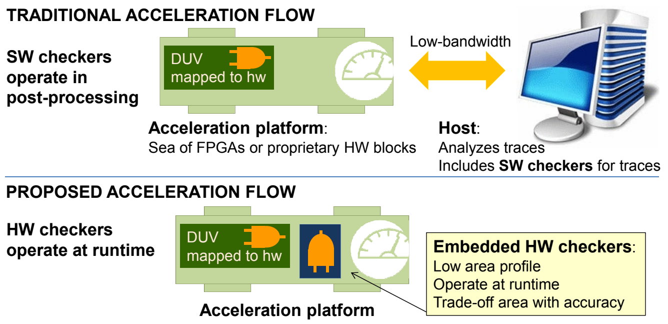

TRADITIONAL ACCELERATION FLOW
SW checkers operate in post-processing
DUV mapped to hw
Low-bandwidth
Acceleration platform: Sea of FPGAs or proprietary HW blocks
Host: Analyzes traces Includes SW checkers for traces
PROPOSED ACCELERATION FLOW
HW checkers operate at runtime
DUV mapped to hw
Embedded HW checkers: Low area profile Operate at runtime Trade-off area with accuracy
Acceleration platform

Fig. 1: **Proposed solution overview**. Traditional acceleration solutions rely on off-platform checkers running on remote host, requiring the transfer of logged data via a low-bandwidth link. Our solution proposes to embed checkers on platform for high performance checking. Our checkers must have a small logic profile while maintaining detection accuracy comparable to their host-based counterparts.

the functional correctness of the simulated design. Often the logging and off-loading activities become the performance bottleneck of the entire simulation.

To address this issue, recent research has shown promise in performing the stimulus generation part of the testbench directly on the acceleration platform [4]. Since test generation and checking are the yin and yang of verification, the software checkers should also be synthesized and embedded directly into the acceleration platform to fully unlock the advantages of acceleration by virtually eliminating the data transfer between host and platform. However, mapping checkers, particularly those utilizing behavioral golden models or complex software data structures, remains a challenge because (i) embedded checkers can only use synthesizable constructs, (ii) their logic complexity should not exceed the platform capacity and (iii) the performance impact incurred by the simulation of their logic components should not bring the acceleration performance down to that of a traditional off-line checking approach. An ideal embedded checker would be sufficiently small not to significantly impact the performance of simulation, while it would be functionally rich enough to provide meaningful value in correctness checking, and hence be a good substitute for a host-based software checker. To this end, we propose to develop embedded "approximate checkers": checkers embedded in the acceleration platform, compact enough to be transparent to acceleration performance, yet capable of detecting a significant fraction of a bug's manifestations.

Contributions: The primary objective of this work is to capture the design intent of a complex software checker into a compact hardware version that can be mapped along with the DUV onto an acceleration platform (see Figure 1). We propose a number of approximation techniques to reduce the logic overhead of the checker while preserving most of the checking capabilities of its software counterpart. We provide a taxonomy for common classes of checkers; we use these classes to reason about the effects of our approximation techniques in the absence of specific design knowledge.

The approximation process may lead to the occurrence of false positives, false negatives and/or delays in the detection of a bug. To properly analyze these effects we provide metrics to evaluate the quality of an approximation, and present two case studies to demonstrate our proposed solutions. Evaluation of our case studies indicate that we can achieve a reduction of approximately 60% in overall logic complexity with a negligible impact on checker quality.

II. Related Work

A rich choice of solutions is available for the validation of high-level behavioral models of digital designs, spanning both constrained test generation and formal property verification, enabling designers to specify complex assertions/checkers and expose bugs. Correspondingly, a wide range of languages exist to describe the structure and concepts needed: *e*, Vera, SystemVerilog, C++, *etc.*. Several research works in the past decade have focused on the efficient synthesis of formal assertions into realizable hardware descriptions [2], [14]. These techniques target specifically acceleration, emulation or in-silicon debug [8], [9]. Reconfigurable designs for debug architectures that enable verification engineers to create assertion checkers, transaction identifiers, triggers, and event counters in silicon have also been suggested [3]. However, assertion synthesis is an exact translation of individual properties and can generate extremely complex logic blocks, which can reduce or eliminate the acceleration advantage. In this work we focus on containing the logic overhead of the embedded checkers using approximation.

On the front of off-platform analysis, emulation platforms support tracing and off-loading of a selected group of signals [21], which can later be processed by a software checker. However, in this setup, the testbench still executes in software and communicates with the emulator via a fast bus connector, which is often a bottleneck [16]. In contrast, transaction-based acceleration (TBA) [19] is a paradigm in commercial simulation accelerators where the testbench itself has a software-hardware boundary and all interactions with the host are at the transaction-level, thus reducing the amount of data transferred between host and emulation platform. Best results are achieved when large components of the testbench are mapped to hardware [16].

Finally, approximation of logic functions has been proposed in other related domains, including binary decision diagrams (BDD) [18], timing speculation [15] and typical-case optimization [5], [12].

III. Checker Taxonomy

Our experience with several designs suggests that most checkers are intended to verify a similar set of design properties. Based on this observation, we present the following checker classification scheme:

Protocol Checkers: verify whether the DUV interfaces adhere to the protocol specification. An example is a checker that monitors the request-grant behavior for a bus arbiter checking that the arbiter sets the grant signal within a fixed number of cycles from receiving a request, or that it never issues a grant when some other requester owns the bus.

Control Path Checkers: verify whether the flow of data within the DUV progresses as intended. An example control path checker is one that monitors I/O ports of a router to check whether a packet accepted at an input port is eventually transmitted through the correct output port.

Datapath Checkers: verify whether data operations produce expected results. A datapath checker for an ALU, for example, verifies that the result of an addition operation is the sum of its operands.

Persistence Checkers: verify whether or not data items stored in the DUV remain uncorrupted. For example, a checker for a processor's register file may check that the contents of each register never change except upon a write command.

Priority Checkers: verify whether specified priority rules are held. A priority rule sets the order in which certain operations are to be performed, usually selected from some queue. Consider, for instance, a unified reservation station in an out-of-order processor that must prioritize addition operations over shift operations. A priority checker for this unit verifies that no shift operation is issued when additions are waiting.

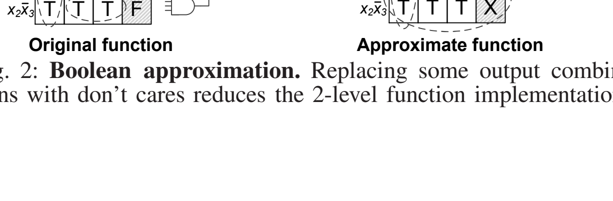
Occupancy Checkers: verify that buffers in the system do not experience over- or under-flow. For example, an occupancy checker may verify whether a processor dispatches instructions into its reservation station only when there is space available.

Existence Checkers: verify whether an item is present in a storage unit. In a processor cache, for example, an existence checker could verify whether the tag for cache access hit actually exists in the tag array of the cache.

IV. Approximation Techniques

To check for correctness, a checker may need to maintain information about the expected internal state of the design for extended periods. Thus, direct mapping of a software checker to hardware, when even possible, often leads to an extremely complex circuit block, possibly as large or larger than the design itself. Often the checker hardware could be simplified, targeting only functionality rather than also performance or power. For instance, a simple ripple-carry adder is sufficient for a datapath checker that verifies a Kogge-Stone adder. Below, we propose a number of approximation techniques to minimize the complexity of embedded checkers.

Boolean Approximation: can be used to reduce the complexity of any combinational logic block. The don't care set of the Boolean function implemented by the block can be augmented by simply changing some outputs from 1 or 0 to don't care (indicated by X). By appropriately selecting which combinations become don't cares, it is possible to greatly reduce the number of gates required for the function. An example is shown in Figure 2, where two minterms are set to don't care (highlighted by hashing), reducing the 2-level function's implementation from 6 to 4 gates. Boolean approximation often allows great reductions in circuit complexity with a minimal amount of don't care insertions. The transformation may lead to false positives or negatives for the checker: a 0 approximated to a 1 would lead to a false positive, and vice versa. Note that it is



$\bar{x}_0\bar{x}_1$ $\bar{x}_0x_1$ x_0x_1 $x_0\bar{x}_1$
$\bar{x}_2\bar{x}_3$ T F T T
$\bar{x}_2x_3$ T T F F
x_2x_3 T F T T
$x_2\bar{x}_3$ T T T F
Original function
$\bar{x}_0\bar{x}_1$ $\bar{x}_0x_1$ x_0x_1 $x_0\bar{x}_1$
$\bar{x}_2\bar{x}_3$ T F T T
$\bar{x}_2x_3$ T T F F
x_2x_3 T X T T
$x_2\bar{x}_3$ T T T X
Approximate function

Fig. 2: **Boolean approximation.** Replacing some output combinations with don't cares reduces the 2-level function implementation.

possible to apply the technique to a sequential logic checker by unrolling the combinational portion and then approximating the logic block obtained.

State Reduction: Embedded checkers may include storage elements for a wide variety of purposes. State reduction eliminates some of the non-critical storage to simplify both the sequential state and the corresponding combinational logic. Examples of non-critical storage are counter bits used for checking timing requirements of events at fine granularity and flip-flops for storing intermediate states in a checker's finite state machine (FSM). Figure 3 shows a portion of a protocol checker's FSM that verifies whether a signal is set for exactly one cycle. The "DELAY" state can be removed and the check is then performed in all the states following the "NEXT" state.

Even though the checker can no longer measure precisely the one cycle delay, it can still verify that the signal is only set for a finite number of cycles. This technique is particularly valuable for checkers containing a reference model of correct behavior. Indeed, for these checkers to operate correctly, the reference model's response must arrive before the design's response. A delay-removing an approximation would allow the checker to compute an estimate before the design's response. State reduction may also introduce false detections, either positive or negative.

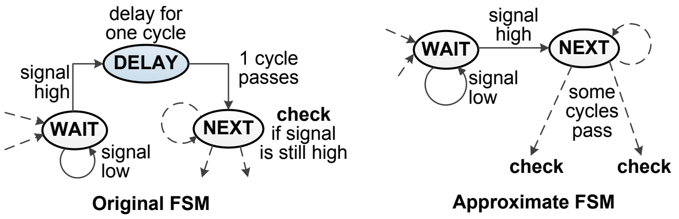


Fig. 3: **State reduction approximation** for a protocol checker. The delay state is removed leading to an approximate checking of events timing (duration of signal high).

Sampling and Signatures: The width of a datapath affects many aspects of a design, including the width of functional units' operands and of storage elements. To reduce the amount of combinational logic and storage required to handle wide data, an approximate checker can operate either with a subset of the data (sampling) or a smaller size representation (signature) derived from the data. Bit-fields, cryptographic hashes, checksums, and probabilistic data structures are valuable signature-based approximations, trading storage size for signature computation. A checker for an IPv4 router design, for instance, does not need to track all the data bytes of packets entering the system. In most cases, storing the control information and an XOR signature of the data is sufficient for checking purposes (see Figure 4). This approximation may result in both false positives and false negatives. As we show in Section VI-B, one can reasonably estimate the impact of these techniques, given the probability distribution of the data payloads and the nature of the design.

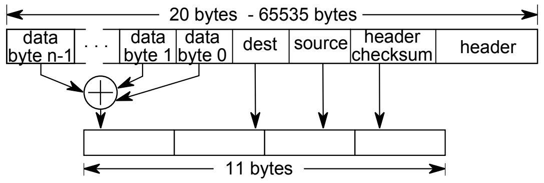


Fig. 4: **Signature approximation** for an IPv4 packet. A packet can be uniquely identified with high probability by using just a few bytes.

TABLE I: **Approximation to checker matrix**. Boolean approximation and state reduction are generic methods applicable to all.

	Boolean	State Reduction	Sampling	Signature
Protocol	✓	✓		
Control Path	✓	✓	✓	✓
Datapath	✓	✓	✓	✓
Persistence	✓	✓	✓	✓
Priority	✓	✓	✓	✓
Occupancy	✓	✓		
Existence	✓	✓		✓

The checker classes presented in Section III allow us to evaluate our proposed approximation techniques on a number of checkers without concern of the DUV's implementation details. Table I summarizes the results of this evaluation. Note that Boolean approximation and state reduction are general techniques applicable to all the classes of checkers. In contrast, sampling and signatures have limited scope as they are only appropriate for situations where monitoring a subset of possible events/combinations enables a detection.

V. Approximation Quality Metrics

An approximate embedded checker may be more relaxed or more restrictive than its original software counterpart. Thus, depending on the time and ways of a bug's manifestation, detection in the approximate checker may occur (or not) as in the original checker – *true positive (TP) or negative (TN)*; the bug may be missed by the approximate checker only – *false negative (FN)*; or the approximate checker may falsely flag the occurrence of a bug – *false positive (FP)*. In this context, it is important to evaluate the relative detection capability of an approximate checker with respect to the original. A good approximate checker should have a small rate of false positives and negatives. If the post-simulation diagnostic methodology is capable of ruling out false positives, then a high false positive rate would not be a critical issue for the approximate checker. We propose to evaluate the quality of an approximate checker with two common statistical metrics deployed in the evaluation of binary classification tests [13]: accuracy and sensitivity. For proper classification, each test has to be run either until a bug is detected (correctly, or wrongly) or until completion.

Accuracy ($\frac{TP+TN}{TP+FP+FN+TN}$) measures how accurate an approximate checker is in reproducing the results of the original checker. A high value exhibits only a few false positives and negatives.

Sensitivity ($\frac{TP}{TP+FN}$) evaluates the ability of a checker in detecting actual bugs (true positive rate). A high value of sensitivity indicates that most bugs that are detected by the original checker are also detected by the approximate checker.

When interpreting accuracy and sensitivity many additional aspects must be taken into account, including the input test vectors, the type of DUV, the class of the approximated checker, and the nature of the bugs. The next section presents two case studies to illustrate these aspects and their impact.

VI. Case Studies

We conducted cases studies on two experimental designs used for training verification engineers in industry and academia. One is a calculator design, similar in principle to a microprocessor with a restricted instruction set. The other is a 4x4 router design for a packet-switched network, capturing many elements of a typical network-on-a-chip (NoC) router.

Both designs include high-level software checkers which we manually translated into hardware descriptions to investigate the impact of our approximation techniques. These designs are smaller than industrial size designs, but they contain enough properties to be verified using checkers that span over all classes discussed earlier.

A. Calculator Design

Calculator 3, aka `calc3`, is used as an example in [20]. The design accepts add, subtract, shift, branch and register load and fetch commands from its four command ports, operates on its 16 32-bit wide internal registers, and responds with results through its four response ports. A successful branch makes `calc3` skip the command following the branch on the same input port. `calc3` supports up to 4 pending commands per port and out-of-order completion of commands, as long as there are no data hazards. Each command is associated with a unique 2-bit tag, reported when the command completes, along with 2 status bits indicating successful completion, a skipped command, or an overflow from addition/subtraction. Only the *fetch-register* command outputs data from the design.

The baseline black-box checkers for `calc3` were created by manually translating a high-level C++ software testbench into a Verilog description. Each port has a separate black-box checker ensemble working in the context of a common shadow register file, which maintains copies of the values that should be in `calc3`'s register file. The main components of the checkers for each port are shown in Figure 5 and they fall within four classes from Section III: a **protocol checker**, monitoring for correct tags and commands values; a **control path checker**, monitoring correct branch behavior and overflow/underflow result handling; a **datapath checker**, checking correct fetch and computation results; and a **priority checker** tracking data hazard constraints.

We used sampling as the main approximation technique for all checkers. The output checker and duplicate execution units were approximated by sampling a subset of the 32 bits for each operand. The approximated datapath for the output checker operates on the least significant 8 bits of data, except the comparator, which uses all 32 bits. This exception was necessary to ensure that branch decisions remained accurate. However, sampling leads to logic reduction in arithmetic-heavy execution units. Note that this scheme cannot detect overflows on its own; hence the approximate checker relies on the status bits of a response to learn about overflows and could potentially miss related bugs (false negatives).

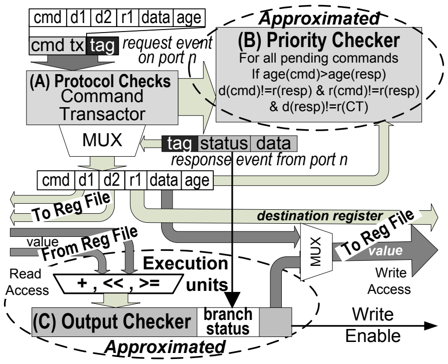


Fig. 5: **`calc3` checker ensemble for one port**. The checkers track tags (protocol), dependencies (priority) and computations (datapath and control). Since `calc3` does not output the results of arithmetic operations, these must be checked via a *fetch-register* command.

The priority checker is also amenable to sampling, where a completing command can be checked for priority violations only with respect to one out of four pending commands. This reduces the logic needed to implement the checker to approximately one fourth. Since incoming commands are mapped to different slots in the transactor based on current occupancy, there is a significant probability that a violating command is present in the slot being checked. Even though this is a weaker check than the original, a bug causing priority violations will still be detected with sufficient simulation runs. The command transactor (a protocol checker) was not approximated since the other checkers depend on its accuracy and even a slight approximation would introduce many false positives.

A significant amount of logic reduction can also be achieved by taking advantage of the fact that, unlike the software version, the checker resides in hardware, next to the DUV, where wire connections can be made directly to the DUV's components at virtually no cost. For instance, we can avoid maintaining a shadow register file by simply checking dynamically that the values to be written match with those computed by the checker. Thus, the shadow register file can be replaced by shared read ports with the design's internal register file and a register-write checker that checks the least significant 8 bits of register values to be written.

B. Router Design

`Router` is a VHDL implementation of a 4x4 router used internally at IBM for training new employees. It can accept variable-length packets from any of its four input ports and routes them to one of its four output ports based on entries in its statically configurable routing table. This is accomplished through a set of request, grant, ready, ack, nack, and command signals that connect to other routers or a configuration host. All `router` packets are composed of a source address byte, a destination address byte, up to 60 bytes of data, and a parity byte (bitwise XOR of all the bytes in the packet). Up to 16 incoming packets, not exceeding a total of 256 bytes, can be buffered at each of `router`'s input ports. `Router` rejects packets whose destinations do not exist in the routing table, whose parity is bad, or when there is not enough space in the input buffers. In addition, the interface signals must obey timing constraints as provided in the protocol specifications.

`Router` comes with a complete software checker environment designed in *e*. To investigate the effects of approximation, we manually developed an optimized but complete hardware version of the *e* environment. The architecture of the checker design is shown in Figure 6. The properties verified by the checkers are summarized below, based on the classes presented in Section III: a **protocol checker**, monitoring for correct timing and validity of a number of control signals (req, gnt, rdy, ack, nack); a **control path checker**, tracking validity of outgoing packets; a **datapath checker**, checking correct parity computation; a **persistence checker** to detect data corruption in input buffers and routing table; a **priority checker**, monitoring correct ordering of outgoing packets and output port mappings; an **occupancy checker** checking for input buffer overflows; and an **existence checker**, tracking the existence of entries in the input buffers and the routing table.

There could be situations in the `router` verification environment where a single checker implementation verifies

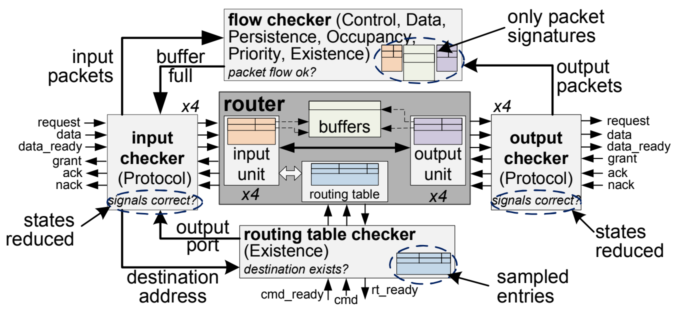


Fig. 6: **Hardware checkers for `router`**. The checkers track packets (several classes), I/O signals (protocol), routing table (existence).

multiple properties or vice versa. For example, a checker verifying that a packet transmitted through an output port is valid (control path) also checks that the routing table provides the correct mapping for the destination address (priority and existence). The input and output checkers in Figure 6 mainly consist of protocol checks. The flow checker is responsible for maintaining `router`'s buffers and most of the control path, persistence, priority, and occupancy checkers are within this module. It lets the input checker know if the buffer for an input port is full. The routing table checker verifies the existence of output port mappings in the routing table and predicts whether `router` would accept or reject a packet.

The state reduction, sampling, and signature techniques were investigated for this case study. Using the state reduction approach, a combined total of 40 states and 88 counter bits were eliminated from the input and output protocol checker units. This approximation is unlikely to cause false positives but might produce false negatives. The routing table is approximated with a sampling technique where only the 4 most significant bits (out of 8) of the destination and the mask are stored. This reduces storage requirements of the routing table by 44%. The existence check that utilizes the reference table should still be able to correctly detect when a destination actually exists in the routing table (true positive). However, due to the loss of half the information content, it may wrongly assume that a destination exists in the table when it does not (false positive). For a purely random input pattern, this existence checker will raise false positives half of the time. This fraction could be lower if the test inputs to the routing table were constrained to follow a specific pattern.

The signature approximation applied to the flow checker includes only the length of the packet (6 bits), the source and destination addresses and the parity, and it has a collision probability of 2^{-30}; sufficient for our design, which can have at most 64 packets in flight. This approximation alone corresponds approximately to 31% of the `router` logic. Reducing the packet signature further by removing source and destination addresses saves us about 32 bytes of storage overall while increasing the collision probability to only 2^{-13}.

VII. Experimental Results

We conducted multiple evaluations of our solution, by comparing results from our approximate checkers with results from their non-approximated, baseline hardware counterparts. We injected multiple bugs of varying complexity and location into our case study designs one at a time; for each bug, we run separate simulations with the approximate and baseline checkers. Each simulation could terminate either because a bug was detected or because the test ran to completion. Each bug detection (or lack thereof) by an approximate checker was compared to the corresponding detection by its baseline counterpart and then labeled as a true positive, true negative, a false positive or false negative.

Tables II and III describe the bugs injected into the designs. A total of 500 tests on `calc3` and 1,000 tests on `router` were executed for each design variant obtained by injecting a different bug. In most cases, test stimuli were randomly generated and uniform for all design variants. For the few hard-to-sensitize bugs, we inserted manually crafted input sequence snippets at random times in the simulation.

Figures 7 and 8 show the breakdown of the test outcomes: for each bug, we report how many tests resulted in each outcome type. The average effects of approximations on accuracy and sensitivity are shown in Table IV. For `router`, the different applied approximations were tested both separately and combined together. Note that for the *stuck* bug corresponding to a stuck bit in a `calc3` register, our approximation scheme was not able to detect any occurrence, since the stuck bit position (bit 20) is not monitored by the approximate checkers. Note in Figure 7 that the `calc3` checkers always give desirable outcomes, either true positive or true negative, for half of the bugs. For the rest, we get some false outcomes; however, there is still a significant rate of true positives (high sensitivity), which enables good bug diagnosis.

In the case of `router`, notice that the signature approximation technique produces only true positives and negatives, and thus has a perfect accuracy of 1. The state reduction technique has perfect accuracy for all bugs except *tim2*: this bug is never detected as it requires accurate timing checks, removed by the approximation. However, losing some cycle accuracy does

TABLE II: **Bugs injected in `calc3`**. The second column indicates which checker should detect the bug. In the approximate version, output checker bugs appear as register update mismatches.

id	checker	description
adds	cmd tx	only dispatch add commands, ignoring shift commands
ovr	output	add or subtract with overflow writes register
stuck	output	20th bit in register 13 is stuck
stall	cmd tx	11th add/shift/branch command stalled
blk1	output	second branch with same tag not blocked
dreg	output	a write-after-write hazard violation
blk2	priority	command with tag 11 is not blocked by command with tag 00
iraw	output	a read-after-write hazard violation (incoming command)
eraw	output	a read-after-write hazard violation (enqueued command)
skip	output	branch follower not skipped following branch

TABLE III: **Bugs injected in `router`** sorted by decreasing difficulty.

id	checker	description
under	flow	Input packet list underflow
2sent	flow	Packets sent twice
badin	input	Packets with bad parity accepted
ovr	output	Memory overwrite
ord	routing table	Wrong routing table search order
badout	output	Packets sent with bad parity
tim2	output	Wrong timing for data_ready signal
exist	routing table	Packet accepted when dest. not in routing table
tim1	output	Wrong timing for request signal
lock	input	Routing table not locking itself when searching

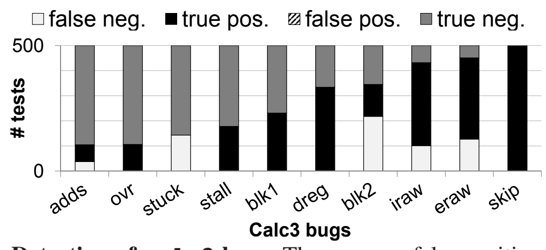


Fig. 7: **Detection of `calc3` bugs**. There are no false positives, since `calc3` approximations were designed to avoid them.

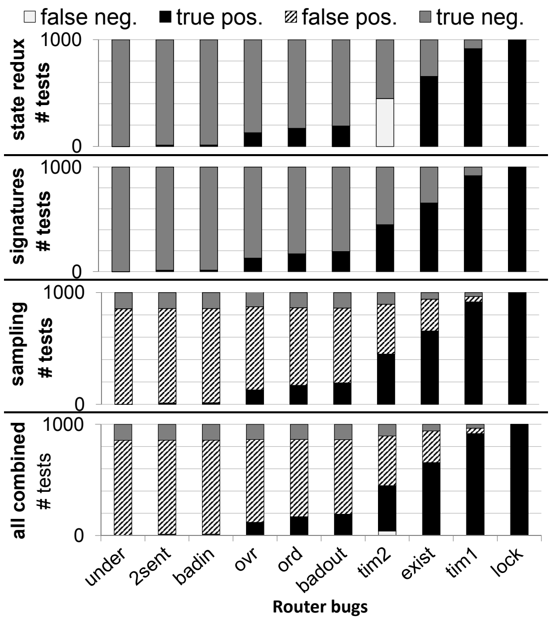


Fig. 8: **Detection of `router` bugs**. Different simulations were run for individual approximations and all combined.

TABLE IV: **Average approximation accuracy and sensitivity.**

metric	`calc3`	`router` st. redux	`router` signature	`router` sample	`router` all
accuracy	87.5%	95.5%	100%	45.8%	45.3%
sensitivity	74.9%	80%	100%	100%	89.1%

not affect all timing bugs, as *tim1* is still detected. Uniform degradation is observed for the sampling approximation on the routing table, by introducing false positives for all bugs. Whenever the routing table checker incorrectly assumes that an entry exists in the table, an execution mismatch occurs. To study the impact of input distribution on the approximate routing table checker, we conducted a separate test where we generated packet address masks with non-uniform probability to imitate real-world network traffic. We observed that when masks with a higher probability of zeroes in the most significant bits are supplied as inputs, the number of false positives is significantly reduced.

Based on our findings, we are able to make some general observations: i) sampling approximations give poor results if they do not take into consideration the nature of the data to be sampled. ii) When a checker combines multiple approximations, the worst performing one dominates. This is not a characteristic inherent to any approximation technique, but rather the result of the application of a technique to a checker that is not well suited for it. Moreover, as our results indicate, inaccuracies manifest in different ways for different types of designs and bugs.

Finally, we evaluated the logic complexity of the baseline embedded hardware checkers and compared against our approximate checkers. To this end, we synthesized both designs and the hardware descriptions of the baseline and approximate checkers using Synopsys' Design Complier, targeting the technology-independent GTECH library. Since the process of mapping a digital design onto an acceleration platform is very specific to the platform being used, we simply considered the total number of logic blocks generated as a reasonable indicator of logic size.

TABLE V: **Logic complexity of approximate checkers**. Overall checker overhead for `calc3` reduces from 87% to 36%. Overall checker overhead for `router` reduces from 57% to 23%.

unit	technique	original (#blocks)	approximate (#blocks)	reduction (%)
`calc3` output	sampling	4,810	1,332	68.1
`calc3` priority	sampling	2,928	782	73.3
`calc3` reg file	eliminate	7,945	1,031	87
`calc3` checker	**combined**	**20,473**	**8,565**	**58.2**
`router` input+output	state redux	3,764	2,664	29.2
`router` flow	signatures	146,910	56,983	61.2
`router` routing table	sampling	2,526	1,835	27.4
`router` checker	**combined**	**153,200**	**61,482**	**59.9**

VIII. Conclusion

Our case studies have demonstrated that checker approximation is a viable solution to reduce hardware overhead of complex checkers while still enabling a large fraction of bug manifestations to be detected. The application of the solutions presented in this work on actual industrial designs and acceleration platforms, with automated tools to generate approximate checkers, will be addressed in future work.

References

[1] Constrained-random test generation and functional coverage with Vera. Technical report, Synopsys, Inc, 2003.

[2] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In *Proc. CAV*, pages 538–542, 2000.

[3] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proc. DAC*, pages 7–12, 2006.

[4] A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, and A. Ziv. Threadmill: A post-silicon exerciser for multi-threaded processors. In *Proc. DAC*, pages 860–865, 2011.

[5] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge. Opportunities and challenges for better than worst-case design. In *Proc. ASP-DAC*, pages 2–7, 2005.

[6] 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.

[7] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Proc. DAC*, pages 244–248, 2001.

[8] M. Boulé, J.-S. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *Proc. ICCD*, pages 294 –299, 2006.

[9] M. Boulé and Z. Zilic. Automata-based assertion-checker synthesis of PSL properties. *ACM TODAES*, 13:4:1–4:21, 2008.

[10] Cadence. *Incisive Enterprise Specman Elite Testbench*, 2011. http://www.cadence.com/products/fv/enterprise_specman_elite.

[11] Cadence. *Palladium*, 2011. http://www.cadence.com/products/sd/palladium_series.

[12] K.-H. Chang, V. Bertacco, I. L. Markov, and A. Mishchenko. Logic synthesis and circuit customization using extensive external don't-cares. *ACM TODAES*, 15:26:1–26:24, 2010.

[13] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[14] S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti. Synthesis of System Verilog assertions. In *Proc. DATE*, pages 70–75, 2006.

[15] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. MICRO*, pages 7 – 18, 2003.

[16] I. Mavroidis and I. Papaefstathiou. Efficient testbench code synthesis for a hardware emulator system. In *Proc. DATE*, pages 888–893, 2007.

[17] M. D. Moffitt, M. A. Sustik, and P. G. Villarrubia. Robust partitioning for hardware-accelerated functional verification. In *Proc. DAC*, pages 854–859, 2011.

[18] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *Proc. DAC*, pages 445–450, 1998.

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

[20] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification*. Morgan Kaufmann Publishers Inc., 2005.

[21] Xilinx Verification Tool. *ChipScope Pro*, 2006. http://www.xilinx.com/ise/optional_prod/cspro.html.