Hybrid Checking for Microarchitectural Validation of Microprocessor Designs on Acceleration Platforms

Debapriya Chatterjee*, Biruk Mammo*, Doowon Lee*, Raviv Gal†, Ronny Morad†, Amir Nahir†, Avi Ziv†, Valeria Bertacco*

*University of Michigan
{dchatt,birukw,doowon,valeria}@umich.edu

†IBM Research Lab, Haifa
{ravivg,morad,nahir,aziv}@il.ibm.com

***Abstract*—Software-based simulation provides a convenient environment for microprocessor design validation, where a number of complex software checkers are integrated with the simulated design to identify discrepancies between design and specification. Unfortunately, the performance of software-based simulation is vastly inadequate to achieve sufficient coverage for large microprocessor designs with complex microarchitectures. Hence, acceleration and emulation platforms are heavily deployed in the industry for high-performance validation. However, software checkers cannot be directly incorporated into such platforms, forcing designers to craft ad-hoc solutions. Adapting checking solutions for software simulation to acceleration platforms presents the following constraints: i) only a limited number of signals can be monitored per cycle for checking purposes so as to retain acceptable simulation performance, and ii) the overhead of the added checking logic must be minimal.**

In this work, we explore a novel solution to adapt software-based checkers for individual microarchitectural blocks to acceleration platforms, by leveraging a hybrid approach. Our solution exploits embedded logic and data tracing for post-simulation checking in a synergistic fashion to limit the associated overhead. Embedded logic can be used for synthesized local checkers as well as to compress the traced data and thus limit recording overhead. We analyze several trade-offs associated with checking accuracy and logic / recording overhead for different microarchitectural blocks of an out-of-order superscalar processor design. We strive to provide valuable insights on how to adapt such software checkers to the acceleration environment using our hybrid approach. We find that, by leveraging simple embedded checkers and data compressors (15-25% logic overhead), we can achieve excellent checking accuracy even when aggressively compressing the data for transfer (only 15-25 bits/cycle), and localize bugs up to 5,900 cycles sooner than an architectural-level checker.

I. Introduction

Verification remains the most effort-demanding phase in the modern microprocessor design process. Due to continued shrinking of transistor sizes over many technology generations, the complexity of modern microprocessor designs has increased tremendously over the last few decades. Complex superscalar out-of-order processor microarchitectures have expanded well beyond the high-performance server and workstation space, as even the next-generation mobile application processors have adopted such complexity [3]. This, in turn, has led to an increase in the effort dedicated to verification. Simulation-based validation is the primary methodology for verification in the industry: a large number of regression suites are simulated on different abstractions (architectural, RTL-level, structural) of the design, to check for adherence to specification. Therefore, simulation performance is key to the success of simulation-based validation. Unfortunately, the performance of software-based RTL simulation on large designs ($>$ 1B gates), such as modern microprocessors, is vastly inadequate (only 1-10 cycles/sec) to achieve satisfactory coverage. Hence, many verification teams in the industry have shifted some of their effort to acceleration and emulation. Indeed, these platforms provide orders-of-magnitude better performance compared to software-based simulation.

To its credit, software-based simulation provides a feature-rich environment for verification, which is critical in validating and debugging a design. A number of checkers are connected and simulated with the processor design at various phases. Often, these checkers include end-to-end correctness checks for correct architectural execution and memory access protocols, as well as localized checkers for individual microarchitectural blocks. Checker-centric validation, although very successful for software-based simulation, does not extend to acceleration or emulation environments in a straightforward manner. Acceleration and emulation platforms can only simulate synthesizable logic; hence, even though the design can be synthesized and simulated at high performance, the testbench and checking environments do not extend into the realm of acceleration [11]. Moreover, lockstep execution of software checkers on a host paired with the design simulated on an accelerator is not tenable, since it degrades overall performance to an unacceptable level. It is, therefore, critical to adapt checkers to acceleration platforms to fully leverage high-performance simulation for verification and debugging. Current industry methodologies on this front have focused on limiting the number of synchronization events between the host running the checkers and the accelerator by: i) accumulating short and frequent interactions between the design and the testbench into longer and infrequent transactions [11], [15], ii) recording the values of critical design signals during simulation on-platform, and then off-loading the log at the end to check for consistency with a software checker [7], iii) synthesizing some of the checkers into hardware for simulation alongside the design [5], [12]. These approaches suffer from simulation slowdown due to large log transfers or large logic overhead. Our proposed solution strives to overcome these shortcomings.

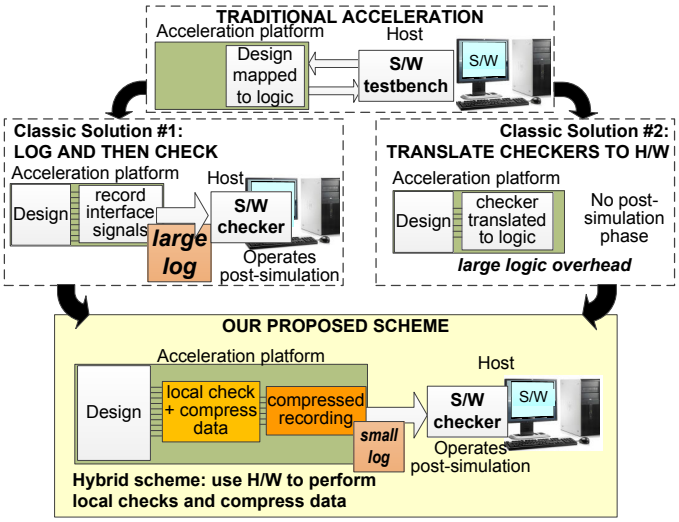

Fig. 1: **Hybrid checker-mapping approach**. We use a mix of embedded logic and data-compression techniques for data that must be logged. For the latter, a software checker analyzes the logged data after simulation.

Any checking solution adapted for acceleration platforms must consider several tradeoffs regarding checking capability and performance. Recording a large number of signals during simulation can incur a performance penalty due to inherent constraints of the acceleration platforms (since simulation must be suspended when the internal buffers of the acceleration platform fill up with recorded signal values). Thus, in such an approach, the average number of recorded bits per simulation cycle must be small enough to introduce only an acceptable degree of slowdown. Adding extra logic to be simulated on the platform alongside the design, if there is room, can also incur slowdowns. Some of the processor-based acceleration platforms do not have a strict logic capacity limit but once the amount of logic that can be simulated in true parallel fashion is reached, any additional logic is simulated in time-multiplexed fashion inducing slowdown. Hence, if embedded logic (such as synthesized checkers) is to be simulated with the design for checking purposes, it must be as small as possible. In an effort to reduce recorded bits and synthesized checker logic, some of the capabilities of the original software-

based checkers may also be lost. In view of these constraints, a desirable solution towards checking on acceleration platforms should have minimal logic footprint and record only a small number of bits per cycle, while providing the same quality of results as the original checker in software-based simulation.

Contributions. In this work, we propose a novel checker adaptation methodology (see Figure 1) for microarchitectural blocks, which synergistically uses embedded logic and post-simulation software checkers to provide high quality checking capability with very small performance overheads. Checkers that have a small logic footprint when synthesized can be embedded and simulated with the design – we call these "local assertion checkers". On the other hand, checkers that must adopt the "log and then check" approach because of their complexity, compress activity logs relevant to the check using on-platform compression logic and then perform the check off-platform – we call these "functionality checkers". We demonstrate our methodology on the software verification environment for a modern out-of-order superscalar processor design. This environment contains a number of microarchitectural-block checkers and an architectural checker, which together provide a setup representative of real world experiences. Adapting these checkers to an acceleration environment provides a challenge that is representative of those faced by verification engineers working in the field. We provide insights on how the checking activity for a microarchitectural block can be partitioned into local assertion checkers and functionality checkers. We classify essential blocks of an out-of-order processor from a checking perspective. Finally, we demonstrate novel techniques to reduce the amount of recorded data with the aid of lightweight supporting logic units, leading to only a marginal accuracy loss. This study is performed at a conceptual level, which involves manual partitioning of checkers for the case-study design.

II. Related Work

A plethora of solutions are available for simulation-based validation of digital designs using software-based simulation [16]. A simulation-based validation environment commonly involves checkers that are connected to the design. These checkers are written in high-level languages, such as C/C++, SystemVerilog, and interface with the design via a testbench. Unfortunately such validation schemes cannot leverage the performance offered by hardware-accelerated platforms for validation, namely simulation acceleration, emulation, or silicon debug. Prior research has investigated synthesis of formal temporal logic assertions into synthesizable logic [1], [8], targeting those platforms [4], [6]. Techniques for using reconfigurable structures for assertion checkers, transaction identifiers, triggers and event counters in silicon have also been explored [2]. However, synthesizing all checkers to logic is often not viable for multiple reasons. Software checkers are often developed at a higher level of abstraction for a design block, thus a direct manual translation to logic will run into the challenge of addressing logic implementation details and can be error prone. Though these checkers can be translated into temporal logic assertions and subsequently synthesized with tools such as those described in [1], [6], the size of the resultant logic is often prohibitive for our context. Indeed, the logic implementation of a checker implementing a golden model for a microarchitectural block is often as large as the block itself, and such vast overhead is not tolerable for large blocks. Schemes to share checking logic by multiple checkers via time-multiplexing has been proposed for post-silicon domain [9], however the range of multiplexing is too limited to offer the same degree of flexibility as software checkers. Recent research has focused on reducing logic overhead by sacrificing checking accuracy [12], but did not consider the benefits of complementing that approach with signal tracing.

The possibility of adopting conventional software testbenches for acceleration and emulation platforms has been considered in prior work as well. The testbench still executes in software and communicates with the platform over a bus: in this setup the communication often becomes the bottleneck [13], [10]. Transaction-based acceleration (TBA) [15] attempts to overcome this bottleneck by bundling several interactions between the testbench and the platform into larger, yet less frequent transactions.

Some recent silicon-debug solutions, such as IFRA [14], introduce additional logic into the design to trace the flow of an instruction through various microarchitectural blocks and a post-simulation analysis tool analyzes the trace to locate the manifestation of a possible design bug. However, IFRA relies on post-triggers for detecting processor execution divergence, and it is not nearly as flexible as software checkers used during simulation. This solution is orthogonal to IFRA as it attempts to adapt traditional pre-silicon verification infrastructure for acceleration platforms, rather than developing a dedicated post-silicon solution.

On the data logging front, acceleration and emulation platforms permit recording the values of a pre-specified group of signals [17], which can be later verified for consistency by a software checker. Recently, a solution was proposed for adapting an architectural checker for a complex microprocessor design to an acceleration platform [7] using this approach: low overhead embedded logic produces a compressed log of architectural events, which is later checked by an off-platform software checker. However, an architectural checker cannot provide the level of insight on design correctness, which a number of local checkers for microarchitectural blocks can. At the architectural level, the information gathered is limited to events modifying the architectural state of the processor; in contrast, microarchitectural checkers track events occurring in individual microarchitectural blocks, generally entailing many more signals. Hence, adapting several such checkers provides a much greater challenge, but it is much more rewarding from design debugging perspective.

III. Synergestic Checking Approach

The most common method of checking microarchitectural blocks involves implementing a software reference model for the block. The design block updates a scoreboard during simulation, which, in turn, is checked by the software reference model [16]. This approach is viable in software simulation, but not directly applicable to acceleration platforms. Since acceleration platforms only allow simulation of synthesizable logic, one option is to implement the reference model in hardware; however, as explained in Section II, this option is often impractical. Another option is to record all signal activity at the microarchitectural blocks' I/O and cross-validate it against a reference model maintained in software after the simulation completes. However, that solution requires recording of a large number of bits in each cycle, leading to an unacceptable slowdown during simulation. Thus, neither solution outlined scales well to complex microarchitectural blocks. We propose a two-phase approach that solves this problem by making synergistic use of these two methods while avoiding the unacceptable overheads of both. It may be argued that we do not always need fine-grain checking capabilities on acceleration platforms: engineers may turn off signal tracing during an overnight run and only record the final outcome of the test, while fine granularity checking requiring extensive tracing is enabled only on sighting of a bug. Unfortunately, this approach prevents accelerated simulation to obtain the same-level of coverage as software-based simulation, since a buggy behavior can often be masked in the final test outcome. Moreover, in this approach debugging is still performed at a compromised simulation performance, while the proposed approach achieves both higher coverage as well as debugging support without sacrificing performance.

The first phase performs cycle-by-cycle checking using embedded local assertion checkers on-platform. It focuses on monitoring the correctness of the target block's interface activity and local invariants,

Block	Local assertion (cycle-accurate, simple, low level)	Functionality (event-accurate, complex, high level)
Reorder Buffer (ROB)	Do ROB head and tail pointers ever cross each other?	Do instructions retire with correct destination register values?
Reservation Station (RS)	Are the contents of the RS flushed 1 cycle after an exception?	Are instructions released with correct operand values?
Load-Store Queue (LSQ)	Are loads whose address has been computed sent to the data cache within a bounded number of cycles (unless flushed)?	Do store operations send correct data to the correct address?
Map Table (MT)	Are all register values marked as in-flight, are in the ROB, or are in the register file?	Are correct tags provided to each dispatched operand field whose value is not ready?

TABLE I: **Examples of local assertion checkers vs. functionality checkers** for core blocks of an out-of-order processor with an Intel P6-like microarchitecture. Note that functionality checks are relatively more complex and high-level.

which can be expressed as local assertions. During this phase, we also log and compress (with embedded logic) relevant microarchitectural events to enable off-platform overall functionality checking. In the second phase, the logged data is transferred off-platform and compared against a software model to validate the functional activity of the block. This approach is illustrated in Figure 2. The main idea behind this two-phase approach is the separation of local assertion checking from functionality checking for a block.

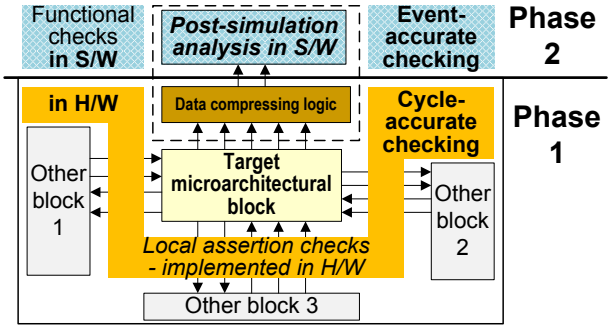

Fig. 2: **Two-phase checking**. Local assertion checks are performed by embedded logic in a cycle-accurate fashion, while microarchitectural events are logged and compressed on platform with additional logic, and then evaluated for correctness by an off-platform functionality checker after simulation.

Local assertion checking often requires simple but frequent monitoring. Hence it must be performed in a cycle-accurate fashion and can often be achieved via ***low overhead embedded logic***, with minimal platform performance loss. Indeed most local assertions are specified over a handful of local signals and can be validated in an analysis' windows of a few cycles (*e.g.*, a FIFO queue must flush all of its content upon receiving a flush signal, FIFO head and tail pointers should never cross over, *etc.*). These checkers do not require large storage of intermediate events; rather they must maintain just a few internal states to track the sequential behavior of relevant signals.

In contrast, **functionality checking** can be carried out in an event-accurate fashion. From a functionality perspective, most microarchitectural blocks can be abstracted as data structures accessed and modified through events of read and update operations. The main goal of functionality checking is then to verify the legality and consistence of operations on this data structure. In addition to monitoring the data associated with events, an event-accurate checker also needs to perform bookkeeping of the internal contents of the microarchitectural block, and thus an embedded logic implementation would be grossly inefficient. Therefore, for functionality checking, the data associated with events should be recorded and transferred off-platform for ***post-simulation analysis in software***, where the validity of the recorded sequence of events is checked. Since events need to be recorded only as they occur, there is no need to record signal values on every simulation cycle. Moreover, we notice that we can further reduce the amount of data recorded by leveraging ***on-platform compression***, while still achieving high-quality functionality checking. Table I provides examples of local assertion and functionality checks for a few blocks.

A. *Checker partitioning guidelines*

It is technically possible, though inefficient, to express any collection of checks entirely as an embedded hardware or entirely as a post-simulation software checker (preserving cycle-accurateness via tracing cycle numbers, if needed). The partitioning of software-based checkers into local assertions and functionality checkers amounts to one of the most critical design decisions for the verification infrastructure and requires the involvement of a verification engineer who can extract the aspects that can be mapped into local assertions. However, there are high-level guidelines that we gained from experience and that can be used to guide and simplify this task. As discussed above, verifying the high-level functionality of a block is naturally a perfect fit for event-accurate functionality checking, whereas verifying simple interface behavior and component-specific invariants with cycle bounds is a better fit for local assertion checking. The primary criterion when making this distinction should be whether event-accuracy is sufficient or cycle-accuracy is needed to implement a check. Another governing principle is that the logic footprint of a synthesized local assertion should be small. Hence, a sufficiently complex interface check that will result in a large logic overhead upon synthesis should be implemented as a post-simulation software checker instead. Once a checker is selected for local assertion checking, it can be coded as a temporal logic assertion and synthesized with tools such as those in [1], [6]. Note, however, that for our evaluation, we simply coded the assertions directly in synthesizable RTL.

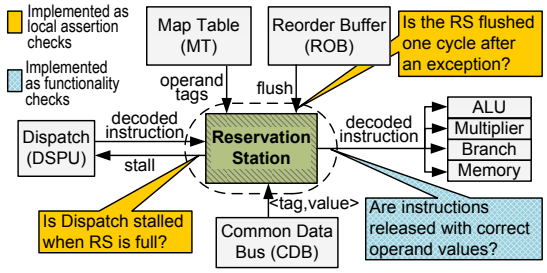

Fig. 3: **Reservation station block:** interfacing units and example checks.

Let us illustrate our approach using the reservation station (RS) block shown in Figure 3. This block stores instructions whose source operands are not yet ready and/or are waiting for a functional unit (FU) to become available. It receives input data from: i) a dispatch unit (DSPU) to allocate new instructions, ii) a map table (MT) to get source registers' tags (for not-yet-ready source values), iii) a reorder-buffer (ROB) to receive unique tags for each newly allocated instruction, and iv) a common data bus (CDB) to gather source operand values as they become available on the CDB. The RS unit releases instructions, whose source operands are ready, to multiple functional units. Thus we can abstract the RS to a data structure that *allocates* an entry upon receiving a new instruction from the DSPU, receives *updates* from the CDB, and *releases* ready instructions to the FUs.

One can identify several properties that must be upheld for the correct operation of the RS, three of which are shown in Figure 3: i) the DSPU must stall when the RS is full, and resume when a

slot is available in the RS, ii) the RS must clear all of its contents on the cycle following an exception, and iii) each instruction must be released by the RS with the correct source operand values. The first two properties are localized to a few interface signals and require cycle-accuracy. Their checkers are also simple to implement, resulting in small logic footprints when synthesized; hence, suitable for implementation as local assertions. The third property pertains to correctness of source operand updates for instructions waiting in the RS block. Its checker must monitor source operand updates from the CDB, identify the instructions updated, and preserve the operand values for later verification upon release. Note that the checker must take action only during an update event from the CDB and a release event from the RS. Since this checker verifies a complex high-level property spanning the lifetime of an instruction in the RS and can perform its checks in an event-accurate manner, it is best implemented as a functionality checker.

B. Microarchitectural block classification

We have observed that most microarchitectural blocks can be evaluated as black-boxes that receive control and data inputs from other microarchitectural blocks or shared buses, and either output information after some amount of processing, or perform internal bookkeeping. We label the events associated with input, output, and internal bookkeeping as *allocate*, *release* and *update*, respectively. We present a general classification of microarchitectural blocks' behavioral characteristics based on the types of events they receive and generate. This classification can help us decide the appropriate checker type, as well as the type of compression scheme, that fits best a given unit. We identify three main types of microarchitectural structures: the first two types are state-heavy structures (excluding caches), and are suitable for our event-based functional checking, while the third is best fit for local assertions. The three structures are discussed below and illustrated in Figure 4.

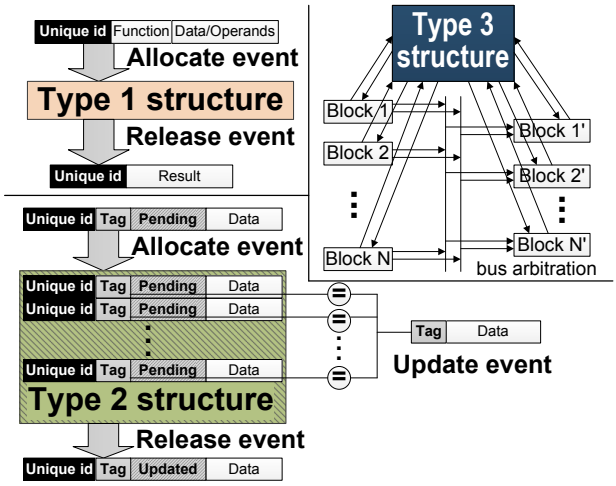

Fig. 4: **Classification of microarchitectural structures**. Based on their behavioral characteristics, microarchitectural blocks can be classified into three types of structures.

Type 1: Structures with allocate and release

This type of structure accepts bundles of information indexed by a unique tag, stores them temporarily, performs some operations on the data in the bundles, and finally releases the transformed bundles. Common examples are processor functional units: they receive operand values indexed by unique ROB-tags, operate on them and release the result to the CDB. For this type of structure, our functionality checker must simply log the allocate and release events during simulation and must find the matching reference events during post-simulation analysis. The unique ROB-tags associated with each bundle are used to identify and match events.

Type 2: Structures with allocate, update and release

This type of structure also targets bundles of information indexed by a unique tag; however, in this case the bundles include placeholders for values to be updated with a final value (by a third party) while the bundle is being processed by the structure. Update events generally relate only to specific entries in the structure, which are identified via a unique tag provided in the update event. A bundle is released (becomes "ready") when all its expected updates have been received. The "ready"-ness criteria is defined by the microarchitectural context. The control information associated with each bundle are the tags for the value updates and the "ready"-ness bits.

Block	**allocate event**	**update event**	**release event**
ROB	dispatch	execution complete	inst. retire
RS	dispatch	register value available on CDB	inst. issue
MT	dispatch	execution complete (tag match)	inst. retire (tag match)
LSQ	dispatch	i) address update on computation ii) value update on cache response	inst. retire

TABLE II: **Allocate, update and release events for essential blocks of an out-of-order processor with Intel P6-like microarchitecture**.

This type of structure comprises some of the core microarchitectural blocks of an out-of-order processor. For instance, in an ROB, allocation occurs when a new instruction is dispatched. The update event (in this case just one) occurs when the instruction execution is completed, as indicated by the common data bus producing the corresponding tag. The release event is the instruction's retirement, indicated by the head entry in the ROB buffer. Table II presents a few of the structures of this type included in an Intel P6-family out-of-order processor: for each structure, we indicate the allocate, update and release events. Several other families of microarchitectures for out-of-order processors (*e.g.*, MIPS R10K) exhibit similar behavior.

Type 3: Combinational/Arbitration structures

The last type of structure consists mainly of combinational blocks, or blocks with a small amount of state information. They are often used for issue, dispatch, or bus-arbitration logic. They must make arbitration decisions based on pending requests in the current cycle: here local assertion checkers generally suffice.

C. Compression of recorded data

In our solution, our functionality checkers gather all the relevant events for each microarchitectural block, as described above, by logging the associated control and data signals on-platform for later transfer and post-simulation analysis. During the logging process, however, we also compress the collected information so as to reduce transfer time. Our goal is to achieve compression in the recorded information without sacrificing accuracy. From a verification perspective, control signals are more informative than data signals; hence, the guiding principle is to preferentially compress data content over control information. Indeed, the control information is critical in keeping the post-simulation software checker in sync with the design block. Since compression is performed using an embedded logic implementation, we want to leverage low-overhead compression schemes, such as parity checksums, which can be computed with just a few XOR gates. In [7] it was shown that blocked parity checksums are generally sufficient to detect value corruptions due to functional bugs in modern complex processor designs. In light of this, we devised three compression techniques, presented below.

1) Data checksum with lock-step control: Often, the post-simulation software must be able to follow the same sequence of control states as the design block to validate its behavior. Based on this observation, we compress the data portion of all events using a checksum (see Figure 5.1), while keeping control information intact. In cases where design and test constraints place limits on data range (*e.g.*, when the address field in a data packet is restricted to use only a specific address range), considering only portions of the data may

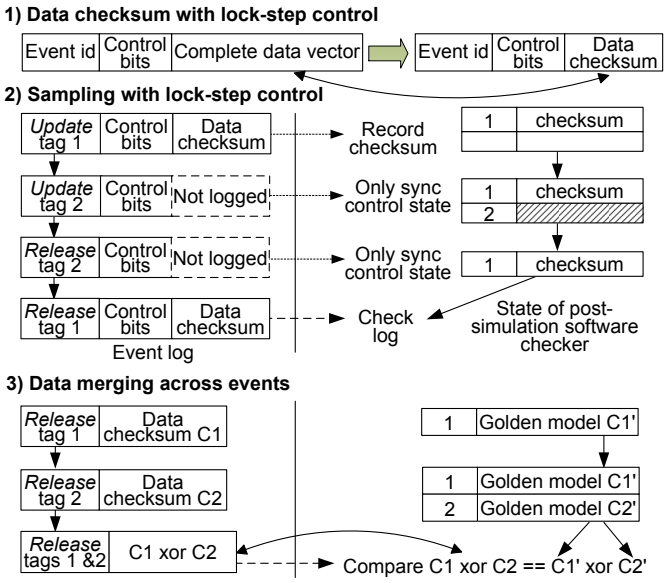

Fig. 5: **Compression techniques for functional events**.

allow for further compression of the recorded information. Moreover, some types of events may undergo additional compression steps as discussed in the next two subsections.

2) Sampling with lock-step control: Taking advantage of the relative importance of control and data signals further, it is sometimes sufficient to record allocate, update and release events with their corresponding control signals, and simply drop the data components of the event (see Figure 5.2). In addition, a number of release events contain control signals that do not affect the state of a microarchitectural block. For such events, either sampling intermittent events, or considering only parts of the control signals is a viable approach.

3) Data merging: This technique computes checksums across multiple events. Instead of checking individual release events, we can construct their "checksum digest" (see Figure 5.3). This is beneficial when only a small fraction of release events may contain errors. Also, note that this approach is complementary to individual event checksums, since the data associated with each event is already compressed.

Choice of technique: The first two techniques are applicable to both type 1 and type 2 structures for compressing data associated with all 3 types of events, while the data merging technique is only applicable to release events for these structures. Some examples of checksum decision choices on a case-study design are presented in IV-A and IV-B.

IV. Experimental evaluation

We applied our checker-mapping solution to a number of representative microarchitectural checkers that are part of the verification environment of a 64-bit, 2-way superscalar, out-of-order processor design, resembling the Intel P6 micro-architecture and implementing a subset of the Alpha ISA. Due to the absence of any open-source out-of-order microprocessor design we used a student project as our design and developed a verification environment around it. Our verification environment consisted of multiple C/C++ microarchitectural block-level checkers, one for each of the blocks reported in Figure 6, and an architectural golden model checker (*arch-check*) connected to the design via a SystemVerilog testbench. We also equipped our verification environment with a time-out condition on instruction retirement, indicating whether the processor had hung (μP *hang*).

We synthesized the design (excluding caches) using Synopsys Design Compiler targeting the GTECH library. Table III reports the contribution of each block to the total logic size: as it can be noted,

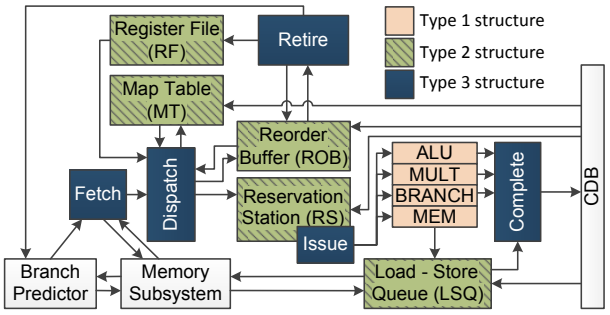

Fig. 6: **Microarchitectural blocks** in our experimental testbed.

the lion's share is contributed by structures that are state-heavy, such as ROB, RS and LSQ. Developing acceleration-based checkers for such blocks has been traditionally a challenge as: i) if the checker is entirely implemented in hardware, the logic overhead becomes unacceptable – comparable in size to their design counterpart, and ii) these blocks generate many events, thus logging entails lots of storage, data transfer and analysis. Hence, we believe that the validation of these blocks will benefit the most from our solution.

	ROB	RS	LSQ	MULT	RF	Others
# GTECH blocks	158,163	53,086	46,765	40,894	24,702	30,546
% of total	44.7%	15.0%	13.2%	11.5%	7.0%	8.6%

TABLE III: **Block sizes** of our testbed design. The design has a combined 16-entry RS, 64-entry ROB, 32-entry LSQ and an 8-stage multiplier.

We evaluated the feasibility of our hybrid checking methodology by analyzing several schemes on the checkers in our testbed. The validation stimulus was generated using a constrained-random generator that created a test suite of 1000 assembly regressions. In evaluating the quality of our solution, we considered the three most relevant metrics: **average number of bits recorded per cycle**, **logic overhead** and **checking accuracy**. The first metric reflects the amount of data to be recorded on platform and later transferred; we estimated the second one by using tracing logic similar to [7]; the third one is obtained by comparing our hybrid checkers against the bug detection quality of a software-only checker in a simulation solution. Note that, industry experience suggests that the average bits/cycle metric is the most critical for acceleration performance. A recording rate of only 162 bits/cycle is reported to induce a 50% slowdown for the acceleration platform used in [7]. We use this metric as a proxy for relative performance overhead independent of any specific platform architecture.

We injected a number of functional bugs in each microarchitectural block to evaluate the bug-detection qualities of our solution. To measure the checking accuracy of any compression scheme, the full set of regressions were run with only one bug activated at a time, and this process was repeated for each bug to create an aggregate checking accuracy measure. Each microarchitectural checker was only evaluated over the bugs inserted into its corresponding design block. The quality of each checker is based on its accuracy of detection. We quantified this aspect by computing what percentage of the bug manifestations it detected.

While we investigated our solution on most type 1 and type 2 blocks, and some of the type 3 blocks of Figure 6, for reasons of space we report below our findings for only one representative block for each category: a multiplier checker (type 1), a RS checker (type 2) and a dispatch checker (type 3). Note that, as discussed in Section III-B, type 3 checkers can typically be fully mapped to local assertions and do not require a functionality checking component.

A. Multiplier Checker

Our design contains an 8-stage pipelined multiplier, which *allocates mult* instructions with operand values obtained from the issue buses, and *releases* the result on the CDB after computation. Associated with each instruction are data signals: two 64-bit operand

values on the input side, one 64-bit result on the output side and a 6 bit-wide control signal (the tag), on both directions. For this block, the only local assertion checked whether an instruction completes within 8 cycles excluding stall cycles, while functionality checking was used to verify computation performed by the block. A *data checksum with lock-step control* compression, possibly with *data merging*, is a natural choice for compressing events for the functionality checker; using a checksum scheme on the data output while preserving the whole 6 bits of control. However, to verify the result of the computation, we still need all operands' bits for each passing instruction. Finally, *sampling with lock-step control* can also be used as long as we keep track of all instructions going through the block but record operands and results in a sampled fashion. Table IV details the set of compression schemes for evaluation.

Name	Compression scheme
csX	compress 64-bit output into X checksum bits
merge	merge results of 5 consecutive release events
samp	record data for only 1 out of 5 instructions going through the block

TABLE IV: **Multiplier checker - Compression schemes**.

We modeled five distinct functional bugs (see Table V) on the multiplier block to evaluate the quality of our checking schemes.

Multiplier's functional bugs	
- Incorrect multiplier pipeline control signal	- Partial result over-written
- Wrong product bit assignment at pipeline stage	- Corruption of result's MSB
- Release of result on the wrong lines of the CDB	

TABLE V: **Multiplier checker - injected functional bugs**.

Figure 7 explores the trade-off between the checking accuracy of different data compression schemes and their average recording rate: the first bar on the left is for the the full software checker tracing all active signals and leading to an average rate of 25 bits/cycle. In contrast, the hardware-only checker does not entail any logging. Note that the average recording rate is much smaller than the total number of interface signals for the block, since only a fraction of instructions require a multiplier. The other bars represent sampling (low logging rate and low accuracy), merging, and various checksum widths. Note that our checksum compressions provide very high accuracy at minimal logging cost. We believe this is due to i) the ability of checksums to detect most data errors, and ii) the fact that some control flow bugs impact data correctness as well.

Figure 8 plots the logic overhead for all the compression schemes evaluated, relative to the multiplier block size. Note that the merging scheme has a slightly higher tracing logic overhead since it needs additional logic to maintain the running checksum.

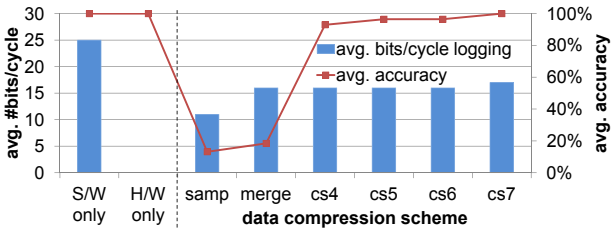

Fig. 7: **Multiplier-checker - Accuracy vs. compression**. The checksum schemes achieve almost perfect accuracy, logging only about 16 bits/cycle.

B. Reservation Station (RS) Checker

The reservation station (RS) block is especially interesting since it is central to the out-of-order core architecture as it interfaces with almost all other microarchitectural blocks and generates the most events. The task of a RS is to hold instructions whose operand values have not yet been produced. The local assertions of the synergistic RS checker verify the simpler aspects: stall, flush and instruction residency time bounds. From the functionality checking perspective,

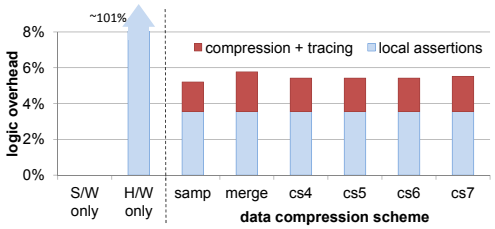

Fig. 8: **Multiplier-checker - Logic overhead** relative to the multiplier hardware unit for a range of compression schemes. Note that the logic overhead for the local assertion checkers does not vary between schemes since the same local assertion is used for every scheme.

the dispatch unit allocates instructions to the RS, the common data bus (CDB) updates the RS with ready operands, and the RS releases ready instructions to the functional units. Associated with each instruction are control and data signals: a unique ID tag, operand "ready"-bits, operand values if operands are ready, or corresponding tags otherwise, and decoded instruction data. The control signals for an allocate event (instruction tag, operand tags and readiness bits) require only 20 bits. The data signals for the same event, on the other hand, require a total of 326 bits. We observed this skewed distribution of data and control signals in the *update* and *release* events of most other blocks as well. Thus, we expect our *data checksum with lock-step control* method to be very effective in compressing these events. Table VI lists the compression schemes that we studied for this block.

Name	Compression scheme
csX	both 64 bit operand fields are compressed to X bit parity checksum, control and tag stay intact, all other data bits are ignored
twX	Only last X bits of the 6 bit tag are recorded, control bits stay intact, operand fields and all other data bits are ignored

TABLE VI: **RS checker - Compression schemes**.

We modeled seven distinct effects of functional bugs (see Table VII) for the RS block to evaluate the accuracy of our solution. Note that these bug manifestations capture a large class of design bugs, since a number of functional bugs produce similar effects.

Reservation station's functional bugs	
- Wrong operand value on issue	- Wrong ROB tag released on issue
- CDB update overwrites ready operand	- Missing a CDB update
- Corruption of decoded instruction within RS	- Ready instruction in RS stalled
- Erroneous ROB tag recorded on allocate	

TABLE VII: **RS checker - Modeled functional bugs**.

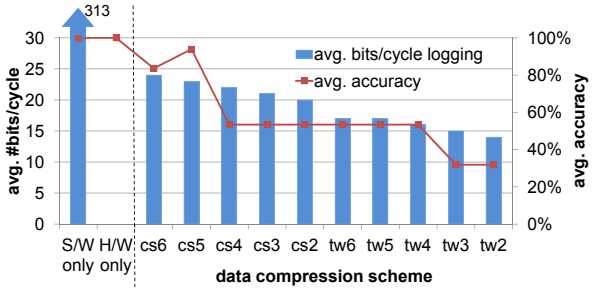

Fig. 9: **RS-checker - Accuracy vs. compression**. A cs5 (64-bits operands compressed to 5 bits) compression scheme delivers an almost perfect bug detection accuracy at a recording rate of only 23 bits/cycle.

Similarly to what we did for the multiplier checker, Figures 9 and 10 report the trade-off between accuracy and logging sizes, and logic footprints for several compression variants applied to the RS checker. In this case, the software checker must log a very high 313 bits/cycle. Note from Figure 9, that the "**cs5**" scheme provides almost perfect accuracy at a logging rate of only 23 bits/cycle, a 92.7% reduction. In Figure 10 we note that the same scheme can be implemented with

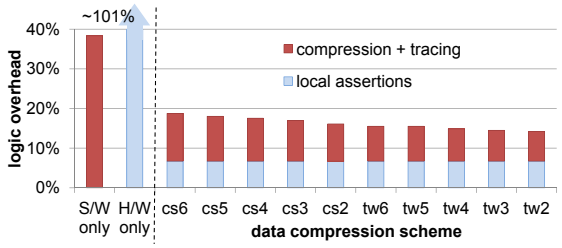

Fig. 10: **RS-checker - Logic overhead** relative to the RS hardware unit for a range of compression schemes.

a fairly small logic overhead (18%), compared to the extreme H/W only checker (101%). Finally, we observe that the logic footprint of the various tracing schemes decreases with the number of checksum bits calculated, as one would expect.

Finally, in Figure 11, we show a distribution of which of the checkers in our verification infrastructure detect each bug first. The portions labeled **RS-local** and **RS-func.** represent the fraction detected by the local assertion and functionality checker of our synergistic RS-checker as described in this paper. **other-check** refers to another microarchitectural block checker, and **arch-check** is the fraction detected by the architectural checker (see the beginning of Section IV). Finally, **μP hang** represents the fraction detected by our timeout monitor. For the S/W-only version, both RS-func. and RS-local checks are implemented as post-simulation software checkers. While some bug manifestations are only detected by the architectural checker, overall, all bugs are detected by at least one of the checkers in place. Among the variants of our solution, the synergistic checker with cs5 compression alone, localizes more than 60% of the bug manifestations, at a logic overhead of only 18%, indicating that our solution is effective for a wide range of bugs.

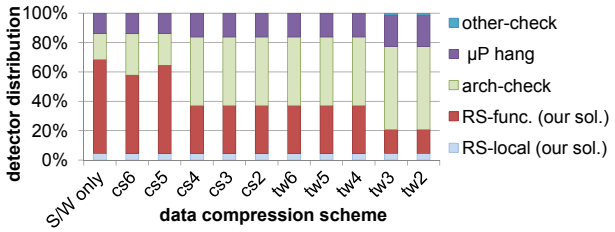

Fig. 11: **RS-checker - First detecting checker over various compression schemes**. In most cases either the embedded checker or the functionality checker is able to detect a bug before it propagates to the architectural level.

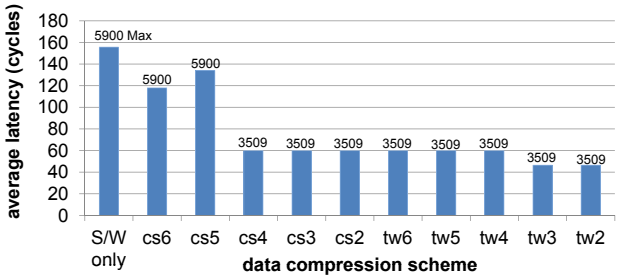

Fig. 12: **RS-checker - average latency between detections at micro-architectural (earlier) and architectural (later) level**. The maximum bug detection latency is reported above the bars. Our RS checker with cs5 scheme can detect a bug up to 5,900 cycles before it propagates to the architectural level.

Note that, almost all bugs eventually manifest at the architectural level; hence, the primary reason to validate at the microarchitectural level is to localize the bug in time and space more accurately, easing debugging. To evaluate the success of our solution on this front, we also conducted a study on the latency of bug manifestation in Figure 12 , where we plot the difference in cycles between when a bug is detected by our solution and when it is detected at the architectural level. It can be noted that we can flag a bug up to 5,900 cycles earlier, a great benefit for debugging purposes.

C. Dispatch Checker

The main purpose of the dispatch logic is to dispatch each decoded instruction with correct value or tag (depending on whether the value is available). The block is implemented with combinational logic to select the correct source of tag/value provider based on associated flags. All checks for this block are implemented via local assertions whose footprint was less than 80% of the size of the block. Even though this is a relatively large footprint, the block itself is very small w.r.t. the whole design, hence the approach is affordable.

V. Conclusions

We presented a hybrid solution for acceleration-based checkers. Our solution leverages a combination of local assertions, data compression hardware and off-platform post-simulation analysis for checking the complex functionalities. We found that our solution is effective in delivering high quality bug detection capabilities at low recording rates (15-25 bits/cycle) and logic overhead (<25%) over a broad range of micro-architectural blocks, while enabling much earlier detection than an architectural-level check (up to 5,900 cycles).

References

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

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

[3] ARM. *A15*. http://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf.

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

[5] M. Boulé and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *Proc. ICCD*, 2005.

[6] M. Boulé and Z. Zilic. Automata-based assertion-checker synthesis of psl properties. *ACM Trans. Des. Autom. Electron. Syst.*, 13:4:1–4:21, February 2008.

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

[8] S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti. Synthesis of system verilog assertions. In *Proc. DATE*, 2006.

[9] M. Gao and K.-T. Cheng. A case study of Time-Multiplexed Assertion Checking for post-silicon debugging. In *Proc. HLDVT*, 2010.

[10] Y.-I. Kim and C.-M. Kyung. Tpartition: Testbench partitioning for hardware -accelerated functional verification. *IEEE Design & Test*, 21:484–493, 2004.

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

[12] B. Mammo, D. Chatterjee, D. Pidan, A. Nahir, A. Ziv, R. Morad, and V. Bertacco. Approximating checkers for simulation acceleration. In *Proc. DATE*, 2012.

[13] I. Mavroidis and I. Papaefstathiou. Efficient testbench code synthesis for a hardware emulator system. In *Proc. DATE*, 2007.

[14] 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.

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

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

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