IDLD: Instantaneous Detection of Leakage and Duplication of Identifiers used for Register Renaming

Yiannakis Sazeides[1], Alex Gerber[2], Ron Gabor[3], Arkady Bramnik[4], George Papadimitriou[5], Dimitris Gizopoulos[5], Chrysostomos Nicopoulos[1], Giorgos Dimitrakopoulos[6] and Karyofyllis Patsidis[6]

[1]University of Cyprus, [2]Google, [3]NVIDIA, [4]Intel, [5]University of Athens, [6]Democritus University of Thrace

Abstract—**In this paper, we propose a cost-effective microarchitectural technique capable of Instantaneously Detecting the Leakage and Duplication (IDLD) of the physical register identifiers used for register renaming in modern out-of-order processor cores. Leakage occurs when a physical register identifier disappears, whereas duplication occurs when the physical register identifier appears twice throughout the renaming logic. IDLD checks each cycle that a code calculated by xoring the physical register identifiers read from and written to arrays, used for managing physical registers allocation, renaming and reclamation, is zero. This invariance is intrinsic to the register renaming subsystem functionality and allows detecting an identifier leakage and duplication instantaneously. Detection of bugs in the complex register renaming subsystem is challenging, since: (a) its operation is not directly observable in program or architectural visible locations, (b) it lies in time-critical paths in the heart of every modern out-of-order core, and (c) it is often the target for optimizations in new core designs, and thus, more susceptible to bugs than legacy subsystems. We demonstrate that bugs in the renaming logic can be very difficult to root cause because, for numerous cases, it takes excessive time, e.g., millions of cycles, for a duplication or leakage to become an architecturally observable error. Even worse, activations of such bugs, depending on microarchitectural state, are often masked by subsequent hardware operations. Hence, an activation of a rarely occurring leakage or duplication bug during post-silicon validation can go undetected and escape in the field. The difficulty of root-causing register identifier duplication and leakage without IDLD is demonstrated using detailed bug modeling at the microarchitecture level, whereas the low overhead of IDLD is confirmed using RTL design analysis.**

Keywords*—*Post-silicon validation, register renaming, merged register file, microarchitecture, pipeline, design bugs

I. Introduction

As chip designs become more complicated, the gap between design and verification grows, and bugs in modern high-end processors are unavoidable, even with rigorous design processes and practices. In such a landscape, design correctness validation is a major challenge for the semiconductor industry [1] [2] that pushes manufacturers to spend extra effort, time, budget and chip area to ensure that the delivered products are operating correctly.

Design bugs are often detected during pre-silicon verification using simulation-time assertions that are useful for the detection of functional bugs before the design is synthesized. If such simulation time assertions are synthesized [3] they can extend the pre-silicon bug detection capability by taking into consideration also non-functional parameters, such as technology and electrical issues, during the simulation of synthesized designs. Unfortunately, such an approach faces several challenges [4] as functional checks are not straightforward or cost-efficient to synthesize. Additionally, they require extra design and validation effort and when synthesized they prolong the simulation time. Even if only subset of the assertions is synthesized, it is non-trivial how to determine which assertions hold the largest potential to detect bugs. What is more, design bugs can escape into silicon due to the limited coverage of pre-silicon methods (which are simulation-based and thus have a limited throughput) or inaccurate models of silicon behavior [1]. Several publicly available official erratum reports, from major processor vendors, present hundreds of bugs escaping to the market that affect both the performance and the functionality of the processor [5]-[11]. To this end, effective post-silicon validation methods become indispensable to detect design flaws and manufacturing defects in prototype chips.

Post-silicon validation allows detection of rare functional errors, but also electrical bugs that *manifest themselves only under certain conditions*, such as signal integrity, thermal effects, or process variations [12] [13]. Current industry (traditional) post-silicon validation methods mainly rely either on comparing the results of a program's execution to simulation-based reference/golden models, or on using multi-pass consistency end-of-test results [14]-[17]. However, both methods share a common drawback: *they are incapable of detecting a bug activation that does not affect the correct functionality of the validation program*. This occurs because the checking phase compares only the silicon's outcome with a pre-generated reference model. As a result, when a bug, which is activated during the execution of a validation program, is getting masked by other valid processor operations, or that bug only affects the performance of the validation test, the final output of the prototype chip will completely match the reference (bug-free) model. This kind of bugs that remain undetected during post-silicon validation, eventually slip in the market and, unfortunately, get activated in a user visible manner be degraded performance or incorrect output. It is crucial to stress that, while not detecting soft errors that are masked is desirable [18], not detecting during post-silicon a rarely activated bug because is masked,

depending on program and microarchitectural conditions, is not. Post-silicon validation phase is the last chance of verification teams to detect rare bugs and fixed them before the chip reaches the market, where a not-masked activation of the escaped bug will be user visible. To this end, such bugs are typically referred to as difficult-to-detect bugs.

Another challenging requirement of post-silicon validation is that after a bug is detected, it needs to be localized to identify what is its root-cause and eventually fix it. Bug localization during post-silicon validation can be quite expensive, as it may require several weeks and even months to complete, delaying the market entry of a product and resulting in grave economic consequences [19]. Unlike pre-silicon verification, post-silicon validation lacks observability and ease of control in the microarchitectural structures of the silicon prototype [20]. What makes bug localization so challenging is the potentially large time window between the bug activation cycle and its manifestation to an observable error (i.e., long bug detection latency). Such a large time-window needs to be analyzed with low-throughput simulators to root-cause the bug. Additionally, the fast detection of such bugs is of paramount importance since such bugs can be extremely difficult to be reproduced (i.e., the root-cause analysis can fail). For instance, a bug can be the result of a combination, at the same time, of a voltage droop and writing data over a critical control signal speed-path, which causes the data not to be written in an array. This combination of events is non-trivial to reproduce, especially, long after they occurred.

Bug localization and root-cause analysis can be accelerated by employing in silicon assertions/checkers [21] [22]. Anecdotal sources suggest that typically, simple checkers can detect only simple bugs, and thus, they usually offer limited coverage for difficult-to-detect bugs. On the other hand, expensive checkers can provide more comprehensive coverage but are costly, primarily in terms of area overhead [23]-[25] (we discuss these and other previous art in Section VII). Therefore, it is instrumental to design simple yet effective checkers that can quickly localize rarely occurring and difficult-to-detect bugs in complex and time-critical processor structures.

Recently, Intel reported in publicly available errata, escaped bugs related to register renaming logic (focus of our paper), which persist across multiple CPU generations. For example, due to an escaped bug in the 6th generation Intel CPUs (SKZ6 erratum [26]), short loops using registers AH/BH/CH/DH may cause unpredictable system behavior. Also, as a workaround to the ICL065 erratum [27], which states "under complex microarchitectural conditions, when move elimination is performed, unpredictable system behavior may occur", move elimination for general-purpose registers is disabled with a recent microcode patch on Ice Lake and Tiger Lake Intel processors [28]. Another very recent bug (February 2022), due to which the processor may incorrectly recover from a mispredicted branch due to a possible race condition in register checkpoint mechanism, has been also patched through a microcode update [29]. These recent examples clearly demonstrate that currently used post-silicon validation methods fail to detect severe bugs related to the register renaming logic of modern out-of-order processors.

To this end, we focus on the *quick and effective post-silicon validation of a critical microarchitectural structure*: the register renaming subsystem (RRS) [30] found in modern out-of-order (OoO) processor cores. RRS is in the heart of every modern OoO core, contains many complex hardware flows that implement various functions and optimizations, and is often the target of new core design improvements [31] [32], as well as of increases in the number of physical registers (Pdsts) that RRS contains and in the number of registers it renames per cycle. This interplay of existing RRS flows with new ones, along with increasing size and tight critical paths makes the RRS severely prone to design, timing, or electrical bugs [1] [2]. Moreover, post-silicon validation of the RRS is exceptionally challenging because RRS lies in time-sensitive paths [33] and any auxiliary hardware-based validation technique for detecting bugs should be simple with minimal (ideally zero) impact on cycle time, providing at the same time comprehensive bug detection. For all these reasons, post-silicon validation of the RRS requires a special treatment as compared to other structures (or even the core as a whole).

In this paper, we propose a simple yet quick and effective hardware technique, named **I**nstantaneous **D**etection of **L**eakage and **D**uplication (IDLD). IDLD is a hardware bug detection approach that checks that a code produced by xoring each cycle the physical register identifiers read from and written to arrays, used for managing physical registers allocation, renaming and reclamation, is zero. This basic invariance is derived from the register renaming subsystem functionality and allows detecting an identifier leakage and duplication instantaneously. IDLD focuses on rare bugs and manufacturing defects in the RRS control logic. As we demonstrate, such bugs primarily result in a physical register identifier (PdstID) being duplicated (i.e., a PdstID appears twice) or leaked (i.e., a PdstID disappears). For example, a PdstID is leaked when it is read from one RRS array but is not written in another, whereas a PdstID is duplicated when it is written in one RRS array without being released from another. Particularly, this work:

- Shows that PdstID leakage and duplication are manifestations of bugs in the control logic that manages the RRS physical register identifiers. PdstID leakage and duplication are new RRS bug models defined for the first time.
- Proposes IDLD: a novel RRS microarchitectural scheme, used during the post-silicon validation phase, which leverages fundamental RRS and design

properties to enable instantaneously the cost-effective detection of PdstID duplication and leakage.

- Justifies the significance of IDLD using a comprehensive microarchitecture-level bug modeling that reveals manifestation latency to an observable error often occurs millions of cycles after the bug activation and in more than a few cases the bug remains unnoticed.
- Designs a baseline RRS at the RTL level and enhance it with IDLD. Synthesis of the RTL designs reveals the minimal area overheads of IDLD.
- Presents another IDLD use case for the Store-Sets memory dependence predictor [34] and discusses IDLD's broader applicability.

II. Background & Motivation

Register renaming is a technique that enables OoO execution by eliminating false register dependences between instructions. Several implementations of register renaming have been proposed over the years [35] [36], however, in this work we focus on the register renaming with a merged register file, which is the typical implementation in most modern CPUs [37]-[43]. In such implementation, the results of operations are stored in a single physical register file that combines the architectural and speculative state. Register renaming with a merged register file uses a large pool of physical registers and translates a logical destination register (i.e., architectural register), of each data producing instruction, to a physical register. Figure 1 shows the RRS considered in this work, which consists of the following hardware arrays:

Free List (FL): FL is a first-in-first-out (FIFO) hardware structure, where PdstIDs are initialized each time the processor core is powered on, with each PdstID pointing to a different entry in the physical register file. A free Pdst is allocated to rename the logical destination register (Ldst) of an instruction. Its PdstID is sent to the reservation station (RS) where the renamed instruction waits to execute. When the instruction executes, it updates the physical register pointed by its PdstID.

Register Alias Table (RAT): RAT is a hardware array that keeps the most recent mapping of each logical register identifier to a PdstID. It is used to rename the input logical registers of an instruction. The renamed PdstIDs are forwarded to the RS of the instruction to determine when the instruction can be executed.

Reorder Buffer (ROB): ROB is a FIFO hardware structure with an entry allocated per instruction. Each ROB entry has a field to hold the PdstID that is evicted from the RAT by the instruction (if the instruction writes to a register). The Pdst is reclaimed (i.e., its PdstID returned in the FL) when the instruction retires.

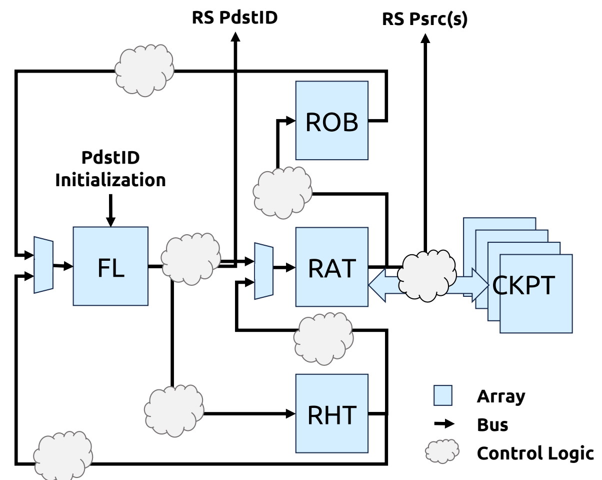

Figure 1. Register renaming with merged register file.

Checkpoint Table (CKPT): CKPT is used to take regularly snapshots of the RAT.

Register History Table (RHT): RHT is a FIFO hardware structure used to log the RAT changes per instruction, i.e., the logical destination register (if any) for an instruction and its allocated PdstID.

The CKPT and RHT are useful for expediting recovery of the RRS following pipeline flushes (e.g., due to a mispredict). When an instruction causes a pipeline flush, the RRS state is restored using the CKPT and RHT. First, the RAT is recovered from the closest previous checkpoint to the offending instruction. Then the RHT is used to perform a (positive) walk to update the RAT with information logged between the RHT entry associated with the restored checkpoint and the RHT entry of the offending instruction. The recovery process performs another (negative) walk of the RHT to return to the FL all the identifiers of the Pdsts allocated after the offending instruction. In addition to recovering the arrays state, the tail pointers of the RHT and ROB are restored to the position corresponding to the flush causing instruction. The FL head pointer is not restored since the wrong path Pdsts are returned in the FL during the negative walk using the FL tail.

RRS speed paths are very tight because in a superscalar OoO core it is required in a single cycle to rename multiple instructions together, so that in the next cycle another group of instructions can be renamed. As part of the renaming process, we determine which instructions can update the RAT with their allocated PdstID. This is non-trivial, as there may be more than one instruction that writes to the same logical register (Ldst), and this requires a multiplexing circuitry with numerous paths. The number of paths increase the wider a core gets. Additionally, register renaming related optimizations, such as move and idiom elimination [31], increase the possibilities for which PdstID is written in the RAT and increases the renaming logic complexity.

Moreover, adding a new RRS optimization to a future OoO core, such as register-equality [44] and memory-renaming [45], will require validation not only of the specific optimization but also its interplay with other existing flows and logic.

Consequently, RRS is non-trivial to validate. Moreover, adding extra circuitry to facilitate its validation should be done carefully to avoid increasing the RRS design complexity or increase the delay of its critical path with possible negative effects on cycle time or performance.

III. Bugs & Bug Models For RRS

This section discusses and justifies the types of bugs and bug models considered in this work for capturing the buggy behavior in the RRS control logic. The control logic is represented by the clouds in Figure 1 and is the one responsible for the allocation and reclamation of the Pdsts and the transfer of their PdstIDs between the various RRS arrays. We show that bugs within the RRS control logic can lead to serious malfunctions that compromise the entire processor core's functionality.

A. RRS Control Logic, Bugs and Bug Models

The control logic in the RRS, depending on the state of the core, generates several signals such as those shown Table I, (i) to control the updates of the RRS arrays, (ii) to control the updates of the read/write pointers for the RRS arrays that are maintained as FIFOs, and (iii) to determine the RRS arrays accessed locations. The Checkpoint signal is generated at regular intervals; in our design at every fixed number of ROB entry allocations, to checkpoint the RAT. The Recovery signal is generated after a squash/flush to recover the RAT from a checkpoint. The selected checkpoint is a fixed function of the ROB position that causes the squash/flush. Additionally, the RRS contains logic that determines which of the allocated Pdst identifiers in a clock cycle are used to update the RAT. This is needed because some of the instructions may update the same logical register.

For the other RRS arrays the selection (if any) of the PdstIDs that are used to update them is straightforward and we do not describe them further. Our discussion does not cover the complete set of signals in the RRS, but it reveals the numerous cases that may go wrong during register renaming in a processor core. A control logic bug can cause control signals not to be asserted or may cause a PdstID corruption. Such bugs can be the result of a design bug or timing error due to a weak electrical signal [1]. For instance, a weak signal can prevent a control signal to be asserted or it may corrupt a PdstID value to be written. Therefore, we use two bug models, which can describe the difficult-to-detect bug scenarios in the RRS: (i) *Control Signal Corruption*, a momentary control signal de-assertion when the signal should normally have been asserted (i.e., depending on the operation, this bug model can result in duplication or leakage or both) and (ii) *PdstID Corruption*, in which the PdstID gets corrupted when it is written in the RAT.

B. Criticality of RRS Control Logic

The severity of faults occurring within the RRS's control logic is demonstrated with a simple walkthrough example in Figure 2. Assume for example that a new instruction must be renamed and, due to a bug, the write-enable control signal of the RAT is momentarily stuck at the logical value low. This implies that the entry pointed by the instruction's Ldst in the RAT is not updated with the new PdstID (i.e., a leakage happens), since the write-enable signal is unasserted. Figure 2(a) shows the state of the RRS prior to the arrival of a new instruction. As soon as the new instruction arrives, the physical register R3 is allocated, causing the PdstID of R3 to be popped from the FL, as shown in Figure 2(b). At the same time, the current mapping of the new instruction's Ldst – that happens to be R1 in this example – must be copied from the RAT into the ROB, as also illustrated in Figure 2(b). Under normal circumstances, the renaming operation would then finish by overwriting the R1 identifier in the RAT with the R3 identifier that has been popped from the FL. However, since the RAT's write-enable signal is stuck at low because of the bug, the R3 PdstID is never written into the RAT, and the R1 PdstID still resides in the RAT, since it was not overwritten by R3. *This is a leakage scenario in RAT*. However, note that the R3 PdstID is (correctly) written in RHT (as shown in Figure 2(c)), because the bug does not affect the correct operation of RHT. Therefore, since the bug prevents the R3 PdstID to be written in the RAT, the newly renamed instruction will write its result into register R3 (as shown in Figure 2(b); the correct R3 PdstID is used by the current instruction), but any subsequent consumers will read data from register R1 (since, due to the bug, this is the current

TABLE I. REGISTER RENAMING SUBSYSTEM (RRS) CONTROL SIGNALS.

	Read Enable	Write Enable		Recovery	Checkpoint
FL	Advance read pointer	Update array	Update write pointer	–	–
ROB	Advance read pointer	Update array	Update write pointer	Move write pointer to offending entry+1	–
RHT	Advance read pointer *	Update array	Update write pointer	Move write pointer to offending entry+1	–
RAT	–	Update array	–	Checkpoint to RAT	–
CKPT	–	–	–	–	RAT to Checkpoint

** RHT uses two read pointers to perform a positive and negative walk during recovery*

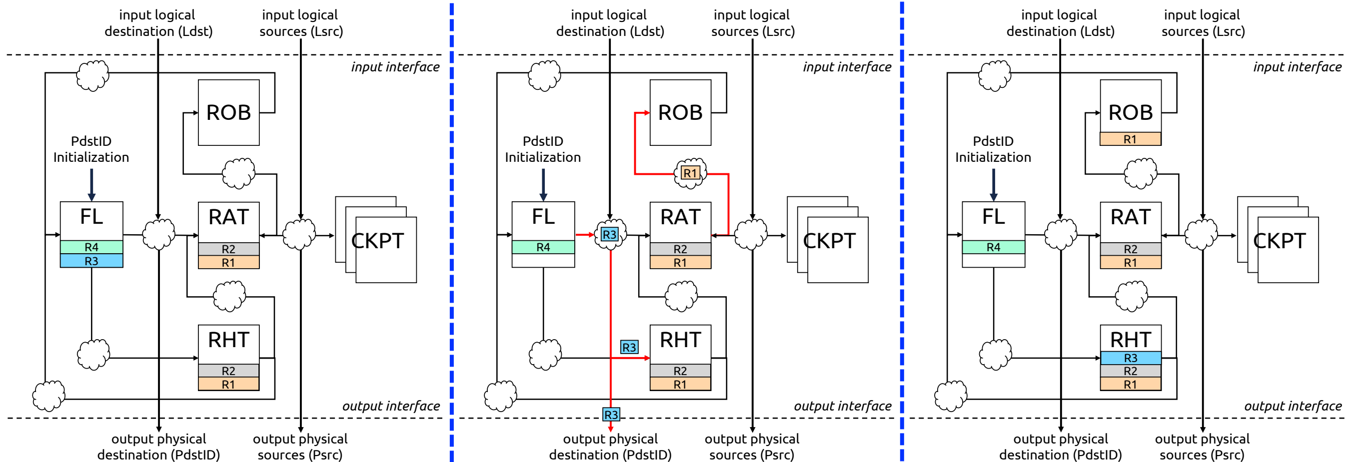

Figure 2. Walkthrough example demonstrating how a bug in write-enable signal of the RAT leads to leakage and duplication.

mapping in RAT). This scenario violates the correct dataflow and *will likely lead to* incorrect program execution. Further, as shown in Figure 2(c), the Pdst identifier of register R3 has now been dropped (i.e., it has leaked and cannot be found in the FL, RAT, or ROB), while forcing the R1 PdstID to be duplicated within the RRS (R1 PdstID now residing in both ROB and RAT). Note that all these problems would remain undetected by schemes that can only detect corruption in a PdstID, since no Pdst corruption has occurred [46] [47].

Even though this example focuses on a particular control signal of the RAT, equally severe issues may also arise due to bugs in other RRS control signals. For example, bugs affecting the read/write signals of the FL, ROB, and RHT could lead to similar behavior as that exhibited in the example above, which would adversely affect the functional correctness of the entire core.

It is important to note, that the bug activation, discussed in Figure 2, can have different effects depending on the microarchitectural state. Specifically, if the bug activation occurs in the correct path, as explained above, it is likely to become user visible (e.g., lead to a wrong output). On the other hand, if the bug activation occurs in the wrong path, it is possible to recover the corrupted PdstID from RHT after the pipeline flush (mispredict recovery) without any user visible effect. This underlines the criticality of detecting such bugs irrespective if their activation during testing happened to be masked since during field operation, under different microarchitectural conditions, the bug can remain unmasked and lead to a wrong output.

C. *Pdst Leakage and Duplication Bug Models*

In the previous subsection, we discussed the example shown in Figure 2, in which the primary effect of an RRS control logic bug is a leaked PdstID, and a secondary effect is a PdstID duplication. The examination of the bug manifestations for the various RRS control signals in Table I reveals that all bug manifestations in RRS logic can result in either or both a duplication and a leakage of Pdst identifiers.

Assume for example all the cases in which an array's write-enable signal incorrectly remains unasserted. This bug scenario will result in a PdstID leakage since the input PdstID is not written in the array. On the other hand, considering the case in which the read pointer of a FIFO hardware structure is erroneously not advanced (due to a bug in the RRS logic). This will result in a duplication the next time the array is read, since it will return the same PdstID. Now consider the case in which a wrong PdstID will be written in a RAT entry. This can correspond to the case where the FL has allocated one PdstID but, due to a bug, a different PdstID value is written in the RAT. This corresponds to a leakage, because the allocated register disappears, but also to a duplication since the wrong PdstID is getting written in the RAT while existing elsewhere in RRS. Another scenario could be that the RAT needs to be recovered due to a misprediction that leads to a pipeline flush, but, due to a bug, it is not recovered. This happens because the recovery signal remained unasserted, or it is recovered from a wrong checkpoint since the correct checkpoint was not taken (checkpoint signal remained unasserted due to a bug). This is similar to a PdstID corruption but instead of a single corruption, there can be multiple and, therefore, multiple PdstIDs are leaked and duplicated. To the best of our knowledge, there is no previous work which defines and presents in such a detail the bug models and their behavior, when they are activated in the control logic of the RRS. A bug model is an approximate behavior that abstracts the details of the actual physical causes of a bug and facilitates the development of methods to detect any physical bug that has the same behavior as defined by the bug model [48].

IV. Bug Modeling Analysis

A. *Microarchitecture Level Bug Modeling*

Before presenting our approach for the instantaneous detection of leakage and duplication of Pdst identifiers, we experimentally explore the behavior of the processor's

operation in a full-system setup for numerous activations of the three bug models in the RRS logic during the execution of actual workloads. In such a way, we gain a deeper understanding of the effects that escaped design bugs from pre-silicon verification, electrical bugs or manufacturing defects can produce during silicon execution, as well as the conditions that prevent these bugs from being detected during the post-silicon validation as it is currently employed in industry (see discussion in Section I). For this experimental analysis, we employed ten benchmarks (end-to-end execution for each benchmark) from the MiBench suite [49] with diverse behavior and the widely used gem5 simulator [50] based on the x86-64 ISA, which is a state-of-the-art cycle-accurate microarchitectural simulator.

The gem5 simulator is configured to model an OoO superscalar core using the renaming configuration presented in Section VI.A. Our experiments are based on bugs, which are activated in random clock cycles in the RRS gem5 implementation. We run 3,000 distinct simulations for each of the ten benchmarks (i.e., 30,000 bug occurrences in total), 2,000 for Read-enable and Write-enable Control Signal corruptions (broken into 1,000 runs for duplication bugs and 1,000 runs for leakage) and 1,000 runs for PdstID corruption. During each run, we consider one single bug activation in the RRS logic (i.e., 30,000 bug activations in total) and we record in detail all the output files, logs, and statistics that correspond to each execution. Through this experimental analysis, we can classify the effects of each of the three bug models into fine-grained bug effect classes.

For each run, we also keep track of the commit trace of the simulator. Therefore, we can monitor the bug activation cycle (i.e., in which cycle the bug is activated) and the bug manifestation cycle (i.e., at which time the bug affects the committed instructions; the commit trace becomes different from the bug-free commit trace). The results of this detailed analysis, help demonstrate the severity of the bug models and the reasons why the bugs defined in Section III are difficult to detect, and when they are detected, it is difficult to root-cause them due to their excessively long manifestation times.

As explained in Section I, bug activations that do not affect the functionality of the program are extremely difficult to detect during post-silicon validation. These are classified into three classes depending on their effect on execution:

Benign: When the execution terminates with no deviations from the bug-free execution and the output file is identical to the bug-free reference output.

Performance: This effect is functionally the same with Benign, but the difference is that there is a deviation in the cycles of the committed instructions. The program's output is correct, the instructions are committed correctly, but some instructions are not committed in the correct cycle (compared to the bug-free committed instructions).

Control Flow Deviation: This effect is again functionally the same with Benign, but the difference is that there is a deviation in the committed instructions (i.e., a deviation in the control flow). This means that in a correct clock cycle, a different instruction of the program is committed as compared to the bug-free committed instruction, but the output is identical to the bug-free reference output. This occurs when control flow diverges from program order and shortly re-converges back without an effect to the program output (e.g., this can happen for a conditional branch that irrespective of its direction, the control flow after the branch's execution re-converges at a control independent point in the program control flow).

B. Masking Effect and Bug Persistence

All the above classes of bug activation effects (Benign, Performance, Control Flow Deviation) do not affect the functionality of the program, and we collectively referred to all as a unified Masked class. Figure 3 shows the fraction of activations for each bug model in our experimentation that do not affect the program output. As we can see in Figure 3, there is a high probability for duplication and leakage bug models to get masked (i.e., the bug is classified to any of the three major classes: Benign, Performance, Control Flow Deviation) while using actual benchmarks, which of course stress and thoroughly exercise the RRS logic. More specifically, Figure 3 clearly shows how severe is the masking effect of the ten different programs. The leakage bug model has the highest masking probability (up to 71%). The duplication bug model has a lower, but undoubtedly, significant masking probability (up to 22%), while the PdstID corruption has the lowest masking probability (up to 3%). At the rightmost bars, we can see the average values for any bug model. Therefore, it is clearly shown that a high number of undetected bugs can occur due to the high probability of masking effects in the RRS, although the bug is activated.

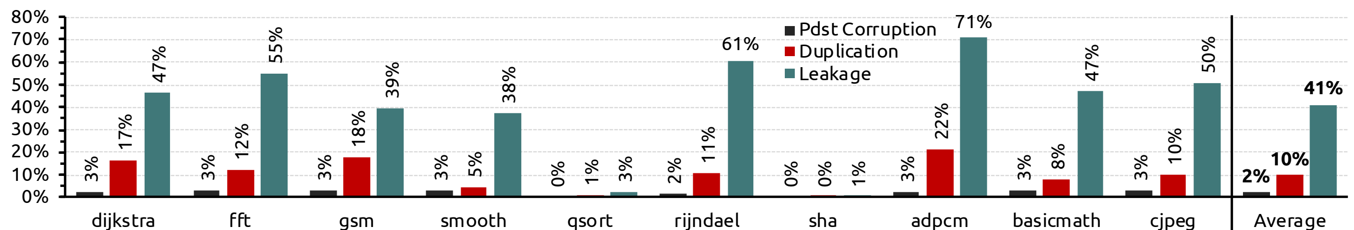

Figure 3. Masked bugs across all benchmarks and bug models used in this study (PdstID Corruption, Duplication, and Leakage).

Another take-away from Figure 3, is that different activation instances of the same bug model can either be masked or not masked, this suggests a dependence on microarchitectural conditions (e.g., whether a bug activation happens in the wrong path; see Section III.B). It is, therefore, desirable to develop a checker that is fast – detects bug as soon as it occurs – to avoid sensitivity to microarchitectural conditions.

It is important to note that the effects of masked faults (i.e., a PdstID disappears or is duplicated in the RRS) may persist after the end of the program's execution. For example, assume a Leakage scenario in which a PdstID is freed from the ROB and should return to the FL, but this PdstID is not written in the FL due to a bug (i.e., the FL write signal is unasserted), and thus, this PdstID will never be allocated again. In this scenario, the effect of the masked bug (i.e., the leaked PdstID) persists until the processor resets. However, there are cases in which the leaked PdstID can be recovered. For example, this happens when a leaked PdstID from the FL exists in the RHT, and after a squash, this PdstID is recovered and eventually returned to the FL. Figure 4 shows the percentage of masked bugs that their effect persists in the processor, even if the program finishes its execution. The results clearly demonstrate that even if the bug is activated without providing any indication of its occurrence (i.e., it gets masked), there is a probability (up to 81%) that the bug effect persists until the processor resets. When this occurs, it is very likely for this bug to affect another program's execution. Note that *sha* and *qsort* are not shown in Figure 4, since they both have zero probability of persisting bug effects. Overall, the analysis reveals that usually the largest fraction of masked bugs does not persist and will remain undetected by traditional post-silicon validation methods.

C. Bug Manifestation Times

In this section, we analyze the other tedious issue of post-silicon validation flow: bug localization and root-cause identification. If the time window between the bug activation and the bug manifestation (i.e., when the bug shows some evidence of its existence) is excessively long, it is extremely difficult for debug engineers to root cause the bug and thus fix it. Figure 5 shows the manifestation times in the x-axis. For a clear demonstration of our findings, we group the manifestation times in eight buckets on a logarithmic scale, as shown in the x-axis. The y-axis shows the number of bugs that belong to each bucket of the x-axis.

The results are shown for both non-masked bugs with the green-line, and masked bugs with some side effect ("Performance" or "Control Flow Deviation") with the red-line. The masked bugs can be detected if processors add a tracing mechanism that monitors for deviations from a reference trace in the program order or timing (a feature that is not available today). As shown in Figure 5, the 77% of the bugs which belong to the masked class but with some side-effect, and the 23% of the bugs that eventually affect the program's output (i.e., non-masked) manifest themselves between 10K and 100M cycles after their activation. It is evident that a large fraction of bugs in the RRS, either the non-masked ones or the masked with side-effects (assuming that they are detectable), have extremely long bug detection latency, making the root-cause analysis an exceedingly difficult task, because there are no means to determine the temporal or the spatial location of the bug during the root-cause analysis.

Additionally, 13.5% of the bugs are benign (not shown in a graph) with no evidence that a bug has occurred during a run. Note that, as we discuss in detail in Section VII, diverse-execution based approaches, such as QED [19] [51] or [52] [53] that aim at reducing the bug detection latency, cannot detect such kind of bugs that occur deep in the processor's pipeline, mainly because these approaches can detect bugs only if they affect the program's execution (i.e., bugs that eventually become visible to the ISA layer). This bug modeling analysis provides clear motivation for a fast, low-cost, and high coverage method for detecting RRS duplication and leakage.

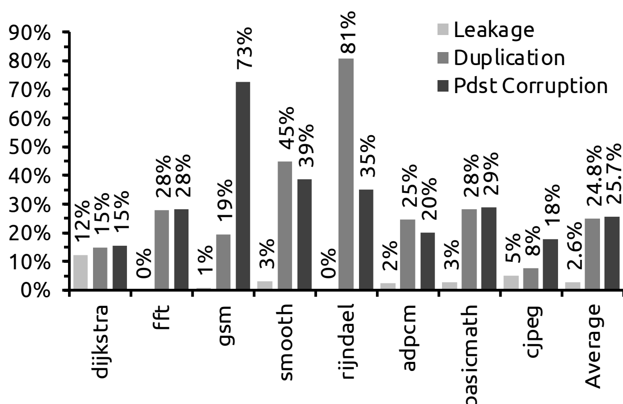

Figure 4. The percentage of the masked bugs that will persist in the processor until it is reset.

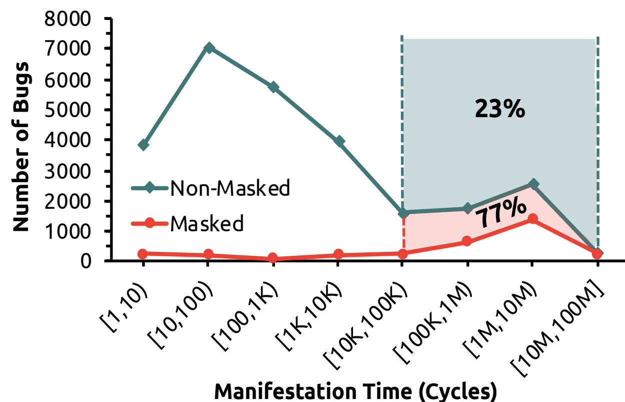

Figure 5. Number of bugs (y-axis) and bug manifestation times (x-axis) grouped into eight distinct buckets.

V. Detecting Errors In RRS Control Logic

In this work, we propose IDLD a bug error detection approach that can detect PdstID leakage and duplication instantaneously.

A. *Basic RRS Invariances*

The novelty of the proposed IDLD technique lies in recognizing and exploiting the following fundamental property and invariant operational characteristic of the RRS:

By design, PdstIDs during program execution move between three RRS arrays: FL, RAT and ROB. There is a known read that is the last read of a PdstID from each of these arrays at which point the PdstID is transferred to one and only one of the other arrays. Hence, there cannot be two writes of the same PdstID to any of these arrays without an intervening last read.

Specifically, a PdstID is allocated from the FL and used to update the RAT (unless it is written in the ROB when another younger instruction that gets renamed, together in the same cycle, has the same logical register destination). A PdstID remains in the RAT until it is evicted. A PdstID is evicted from the RAT when its associated logical register gets a mapping to a new physical register. The ROB holds the PdstIDs evicted from the RAT until the instruction that caused eviction commits at which time the PdstID is reclaimed by the FL. This means that each PdstID can be found at any given time in one and only one of the three arrays. Therefore, each PdstID that is read from RRS arrays in a cycle must be written in another array by the cycle end.

This invariance is present in systems with closed loop management of tokens. In such a system when a token is allocated it is always subsequently returned to the system and a token it cannot be returned to the system without first been allocated. The analogy with RRS is that a token corresponds to a PdstID that needs to be allocated from the FL and then reclaimed back in the FL. The main difference of RRS from a closed loop token management system is that between allocation and reclamation PdstIDs can reside in either the RAT or the ROB but not both. This analogy besides being instructive it also usefully identifies other use cases (see Section F).

The above invariance is generic and valid for the basic RRS implementation of an OoO core with merged register file without any physical register reuse or idiom elimination optimizations [31]. However, any such optimization in the RRS logic is compatible with IDLD, since depending on the optimization, minor modifications can be considered during the IDLD implementation. We demonstrate the effectiveness and the flexibility of IDLD in Section E, where we explain how IDLD can be made compatible with RRS optimizations.

B. *Proposed Post-Silicon Validation Method*

The main idea behind IDLD is to monitor that the RRS invariance presented in Section A is not violated. A violation of this invariance, henceforth referred to collectively as PdstID-invariance, can be the result of a bug, and can result in a PdstID duplication or a PdstID leakage or both (see Section III). The proposed scheme (shown in Figure 6) tracks the PdstID-invariance with low-cost hardware by computing each cycle the bitwise exclusive-or (XOR) of all the PdstIDs written/read to/from the FL, RAT and ROB and checking that it is always zero (shown as 0 in Figure 6). The XOR function trades-off detection accuracy with cost as there can be situations where the XOR remains the same when multiple PdstIDs are duplicated or leaked. Such situations are not expected to happen at once but rather be the result of incremental (one at a time) occurrence of duplication and leakage or both, which is detectable by the proposed scheme.

More specifically, the proposed scheme tracks separately the XOR of the PdstIDs read/written from/to the FL, RAT, and ROB, the three arrays that all PdstIDs must be found in. The three XORs are denoted as FL_{XOR}, RAT_{XOR} and ROB_{XOR}. Each of the three arrays uses a register to store its current XOR value and whenever a PdstID is inserted or removed from the array, it is XORed with the value in its register. The central invariance that the scheme checks is that when a PdstID is read from one array it should be written to another array – this holds true for the FL, RAT, and ROB. The PdstID-invariance is violated when $FL_{XOR} \oplus RAT_{XOR} \oplus ROB_{XOR} \neq 0$. In such a case, an error is detected indicating the bug activation in RRS. Figure 6 shows that the added IDLD logic does not lie in the access timing-critical path of any RRS array. Section VI evaluates the IDLD area and energy implications.

It is important to highlight that IDLD is intended for the post-silicon validation of prototype chips; it is not necessary during field operation. The feature can either be removed completely from the masks used for manufacturing market products or when the processor chips are released to the

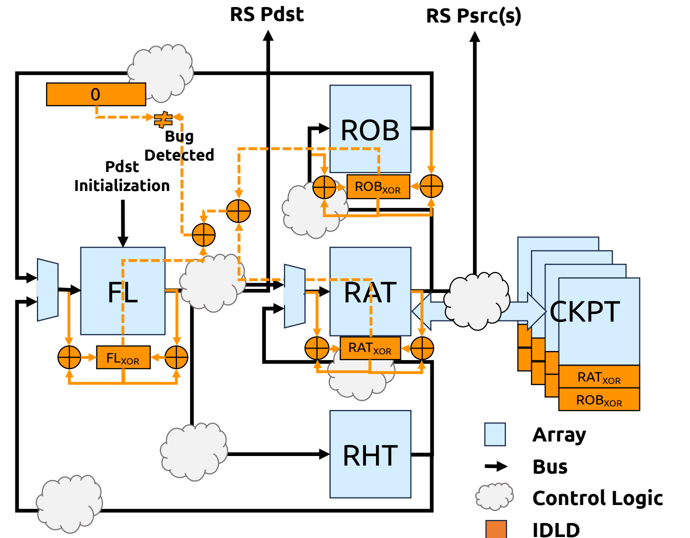

Figure 6. IDLD Protection for the RRS.

market. Thus, the IDLD hardware can be disabled using the "chicken bits" in the corresponding control registers [54]-[56]. Therefore, any power overhead of IDLD (see discussion in Section VI.A) is not a burden for in the field operation. However, IDLD can be re-activated if needed to focus on a customer's issue, since industry relies on enabling such features when a customer experiences an anomaly to help isolate/triage the bug as motivated in [57].

C. Pipeline Flushes

Logically speaking the PdstID-invariance should always hold, but due to practical implementation reasons processors realize pipeline flush actions over several cycles and possibly for some arrays independently and concurrently; this can lead to PdstID-invariance violations during the recovery phase without the presence of any bug. Consequently, PdstID-invariance must be checked at points of time when the processor is not in a recovery mode. Another implication of flushes is the extra bookkeeping required to correctly track the XOR of the PdstIDs they contain. After a flush, the content of the RAT is recovered from CKPT, without reading the previous PdstIDs and without inserting the new PdstIDs through regular RAT read and write ports. As a result, after recovery the RAT_{XOR} can be inconsistent with the RAT content. This problem can be easily resolved by including in a RAT checkpoint the RAT_{XOR}. This contributes to a small area increase (few bits per checkpoint). After recovering a RAT from a checkpoint, the RHT is walked to restore the RAT: updated from PdstIDs from RHT (positive reclamation of PdstIDs from the instructions between checkpoint and flush causing instruction; see Section II). Such updates are done through the regular RAT ports, so RAT_{XOR} is updated and eventually becomes coherent with the RAT content.

The ROB is naturally implemented as a FIFO queue in most cases. After a flush, the ROB tail pointer is moved back to the entry that caused the flush without reading out the PdstIDs in between. Therefore, the ROB_{XOR} becomes inconsistent with the ROB content. One way to overcome this inconsistency is to checkpoint the ROB_{XOR} on every RAT checkpoint. Subsequently, on a flush when a RAT and RAT_{XOR} is recovered and RAT is walked, the ROB_{XOR} is also recovered and walked with the PdstIDs evicted from the RAT during positive reclamation from the RHT. At the end, the ROB_{XOR} becomes coherent with the ROB content. The checkpoint cost of ROB_{XOR} is equal to that of checkpointing the RAT_{XOR} and is quite small. The extra checkpointed info is shown in Figure 6 as part of a RAT checkpoint but in an implementation, this information can be stored in a separate structure. The FL after a flush is updated from PdstIDs from RHT (negative Pdst reclamation from instructions that are flushed). Since these updates are done through the regular FL write port, the FL_{XOR} is always consistent with the FL content, so there is no need for special FL_{XOR} handling.

One of the IDLD strengths is the cost-effective debugging of multi-cycle RRS flows (e.g., flush recovery) by simply checking that IDLD's invariance is maintained after each execution of such flows.

D. IDLD Coverage

Overall, IDLD can detect instantaneously any bug activation that affects the correct RRS operation (not only the bug activations that affect the correct execution of a program, but also bug activations that do not affect the program's output) during post-silicon validation. Such scenarios describe any bug which occurs in the control logic (grey clouds in Figure 6) that causes PdstID duplication, PdstID leakage or combined duplication-leakage, and PdstID corruption during the write operation in an RRS structure. Note that the purpose of the proposed IDLD scheme is not to detect bugs that cause a Pdst corruption while a PdstID is already stored in FL, RAT, or ROB. Such simple bugs can be detected by other well-established schemes, like ECC [46] or circular parity [47]. Such schemes are orthogonal to IDLD and can be combined to provide a comprehensive RRS protection.

One subtle but important point about IDLD is that if the PdstID with value 0 gets duplicated or leaked, the proposed scheme will not detect it (XOR with zero does not cause a change). This can be fixed by logically extending all the PdstIDs by one bit with value 1. This bit should not be stored in the arrays but only used as an input constant in the XOR calculation for each array. In this work, we assume the various XORs are maintained using registers with size equal to the bits needed to encode a PdstID+1 bit, to account for the 0-value PdstID.

Another important coverage concern is the "infinite validation space": *we cannot know all the possible bugs that can exist in the silicon prototype, even with exhaustive validation* [56]. Apparently, as we demonstrated in Section IV, there are several difficult-to-detect bugs in RRS "pestering" the industry [29], not only due to their high probability to get masked, but also due to the long bug detection latency. IDLD aims at detecting any difficult-to-detect bug occurring in RRS, thus, significantly improving the post-silicon validation phase.

The IDLD method does not suffer false-positives – it needs to detect any bug activation - unless there is a problem with the IDLD logic itself. Although, possible, this is unlikely as IDLD is a simple design that lies off the critical path. In an extreme situation where IDLD feature is problematic, it can be disabled using control fuses provisioned in the design for this purpose.

E. Alternatives & Discussion

An alternative way [58] to track the PdstID-invariance is with a bit-vector that has as many bits as unique Pdsts (we refer to this as the bit-vector (BV) scheme). The bit position

corresponding to a Pdst is set when its PdstID is freed and unset when allocated. Duplication is detected when a PdstID becomes free, and its bit is already set. Leakage is detected by counting the number of free registers (bits set in the bitvector) in the free list when the pipeline is empty and checking that it is equal to the difference between the number of physical and logical registers. This approach is costly in terms of both state and logic. It requires as many bits as the number of unique Pdsts (100s of bits in modern cores). The bitvector access logic is complex since at any given time several different Pdsts are allocated or freed. Moreover, the vector needs recovery in case of flushes, which adds overhead.

In comparison, IDLD requires significantly less state, in the order of the bits needed to encode a Pdst, and its logic is simple: the bitwise XOR of the circulated PdstIDs. Both IDLD and bit-vector can detect a Pdst duplication and leakage, but IDLD can detect a bug instantaneously when it occurs whereas the BV scheme only when a duplicated PdstID is reclaimed or when the pipeline is empty. The latency for such events is not bounded, for example, a duplicated PdstID stored in a RAT table entry will not be reclaimed until an instruction renames the logical register corresponding to that entry and evicts it from the RAT. However, even this is not sufficient since it is possible that the other copy of a duplicated PdstID is allocated by the time the other duplicated PdstID is reclaimed. What is more, the bit-vector scheme is unable to detect bugs that are masked, for example, a bug activation that occurs while in the wrong path and the bug activation is masked (see Section III for an example where leakage is recovered from the RHT). In the evaluation section (Section VI) we compare the coverage of IDLD against that of the BV scheme.

Another way to track the PdstID-invariance is by counting the number of free and allocated registers and checking that their sum is equal to the number of unique Pdsts. This scheme is inexpensive as it requires $\log_2$(#Pdsts) bits and can detect PdstID duplication and leakage. However, unlike IDLD, this scheme cannot detect a combined duplication and leakage, since the total number of PdstIDs remains invariant (x+1–1=x). Further, it cannot capture corruption in a PdstID and counting is more complex than the bitwise XOR.

As discussed in Section A, IDLD represents an effective post-silicon validation approach. However, there is a possibility that, due to implementation specific RRS optimizations, the invariants in which IDLD is based on, to be violated without any bug occurrence. E.g., when a physical register is reused during the move elimination optimization [31], its PdstID will appear more than once in the RRS. Similarly, the identifiers for hardwired registers of zero and one used to realize the 0/1-idiom elimination optimization [31] can also appear multiple times in the RRS. When such optimizations are employed, as in many modern high-end cores, IDLD can be easily configured to provide bug coverage for them, by leveraging the control signals that enable them. For instance, a control signal will be communicated to the RAT when a second instance of a PdstID is created (without been allocated from the FL which means the FL_{XOR} is not updated) for a move elimination. The signal is needed so that the PdstID is marked in the RAT as duplicated (need this to determine when the specific PdstID can be returned to the FL). This signal can be used to inform IDLD to not take into consideration at that time the corresponding duplicated PdstID for a RAT_{XOR} or ROB_{XOR} calculation. If this signal, due to a bug, is not activated it will cause IDLD assertion because the RAT_{XOR} or ROB_{XOR} will be updated without the FL_{XOR} being updated.

It is important to note that IDLD is easy to port and scale across different core generations if the basic RRS microarchitecture remains the same (e.g., as in Figure 6). This will be the case for when a core gets wider, uses complex design optimizations to hit the target frequency, the number of registers increase, and additional register renaming optimizations are employed. This is to say that the main IDLD design and validation effort will be incurred once and thereafter IDLD can easily be reused and ported across microarchitectural generations.

F. *Other Use Cases*

The IDLD approach is applicable for debugging other circuits with closed-loop functionality that manage fixed resources, or information. An example of such a circuit is the Store-Sets Memory-Dependence-Predictor (MDP) [34]. A MDP is used in modern cores to minimize the penalty from memory order violations. For the Store-Sets MDP predictor (shown in Figure 7 where acronyms are explained), when an ID, unique identifier for each store currently in the pipeline, is entered in the LFST table; the entry needs to be removed from LFST subsequently. Otherwise, if the ID is not removed, a load may cause execution to hang because it can have a dependency on a store that has left the pipeline. Note that if this bug occurs in the correct path the bug affects correct functionality and availability. LFST insertions are removed when the store's address is computed or read before it is overwritten by another store instance that happens to map

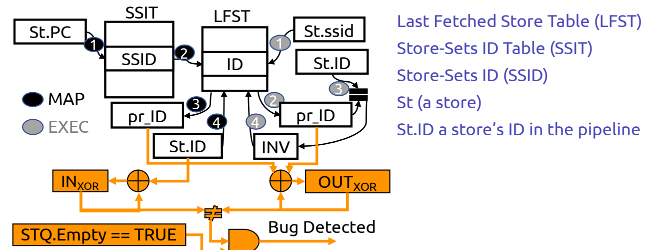

Figure 7. IDLD use case for MDP. Black circles represent the MDP flow for a store instruction at the map stage and the grey circles the flow when the store address is known at execute.

to the same LFST entry. The interested reader can refer to [34] for more details on this MDP operation. For the purposes and scope of this paper, what is important is the requirement that each LFST insertion is eventually removed. Such invariance clearly suggests opportunity for IDLD use as shown in Figure 7. IDLD uses two registers to track the XOR of the ID's that are inserted and removed from the LFST table. The other important part is to identify when to check for invariance violation: the two XORs should be equal but they are not. One option for this is every time a counter, that is incremented on insertions and decremented on removals, becomes zero. A possibly simpler alternative is to do this whenever the Store Queue (SQ) of the core is empty. To facilitate frequent checking, in case the SQ removals are lagging the insertions, i.e., rarely SQ becomes empty, one can take a checkpoint of the insertion XOR when a specific SQ entry is allocated and compare the checkpoint with the removal XOR when that SQ entry commits. This is insufficient, however, because the removal XOR is updated out-of-order (e.g., whenever a store address becomes known). One way to overcome this is to compare with a second version of the removal XOR that is updated only from SQids that are between the current SQ tail and the SQ position where checkpoint is taken and the IDLD invariance gets checked. This second removal XOR gets a copy of the original removal XOR after each invariance check. Due to space limitations, we do not provide details about the MDP use case (e.g., handling of pipeline flushes).

The IDLD approach is applicable to any system where there is incoming and outgoing information flow from read and write ports, and it is a system invariance that the overall outgoing and incoming info should match. This has applicability in many situations (bus communication, exchanges between NoC links, FIFOs etc.). The requirements for adopting IDLD are: i) to identify that such invariance exists (in some situations is non-obvious, e.g., the MDP use case), ii) determine the conditions for when the invariance holds and can be checked (e.g., for the MDP use case one possibility is when STQ is empty), iii) introducing additional state to enable invariance checking that is otherwise not possible,(e.g., checkpointing the RAT_{XOR}, FL_{XOR}, ROB_{XOR} and recovering it on flushes for the RRS use case), and iv) introducing checkpoint state to facilitate more frequent checks (e.g., checkpointing the out XOR in the MDP use case when a specific SQ entry is allocated and inserted).

VI. Experimental Evaluation

A. IDLD Hardware Implementation Analysis

The IDLD technique is evaluated in terms of hardware area, and energy (note that any energy consumption overhead is only presented for the completeness of the IDLD presentation and does not affect the field operation of the processor which implements IDLD). Timing results are not present as IDLD is completely off the critical path. A complete RRS is implemented as a fully functional, cycle-accurate module in SystemVerilog. Specifically, all hardware structures and logic described in Section II are implemented and integrated into a fully functional RRS. We investigate 1-wide, 2-wide, 4-wide, 6-wide, and 8-wide register renaming. In this way, we cover both scalar (single-issue) and high-performance superscalar (multiple issue) OoO pipelines. The baseline RRS and the baseline extended with the proposed IDLD technique are fully implemented in SystemVerilog. Both designs are (1) thoroughly validated at the RTL level for functional correctness using many tests to capture both normal and corner-case behaviors, with and without bug occurrences; (2) synthesized to a commercial 45 nm standard-cell library under worst-case conditions (1.1 V, 125 °C); and (3) placed-and-routed using the Cadence digital implementation flow. The RRS arrays are implemented as standard-cell-based memories, using flip-flops in the place of SRAM cells, following an internal clock-gated organization like [59]. The implemented RRS supports 128 physical registers, which determine the size of the RHT and FL (i.e., 128 entries each), and it includes a 96-entry ROB, a 32-entry RAT, and 4 RAT checkpoints.

B. Experimental Results for Area Overhead and Energy

The post-place-and-route results pertaining to the hardware cost of the baseline and IDLD designs are summarized in Table II. The table reports the area and energy for each design. As shown in Table II, the proposed IDLD design has a small area increase (up to 12% for the 8-wide compared to the baseline). For example, IDLD can achieve 84,377μm^2 area overhead at 1.1 V (for 8-wide register renaming), as opposed to 75,998μm^2 area overhead for the baseline. The key here is not the absolute values of the baseline, but the relative difference between the baseline and IDLD. The provided numbers in Table II clearly indicate that IDLD scales well all the way up to 8-wide renaming. Note also that the numbers shown in Table II refer to the RRS only, and not to the full OoO core and that while we increase the width of the core, we do not scale the number of Pdsts and the size of the RRS structures (i.e., the additional area overhead of IDLD should be negligible). An estimate of the overall area contribution of IDLD to a state-of-the-art OoO

Table II. Area and Power for Baseline and IDLD for Different Write Port Counts (% is Overhead Relative to the Baseline).

	Baseline		IDLD			
Ports	**Area(μm^2)**	**Energy(pJ)**	**Area(μm^2)**		**Energy (pJ)**	
1	36,891	6.04	37,891	(3%)	6.28	(4%)
2	53,441	7.64	54,903	(3%)	8.38	(10%)
4	65,480	11.14	73,701	(12%)	12.29	(10%)
6	73,001	13.12	80,258	(10%)	14.29	(9%)
8	75,998	13.71	84,377	(11%)	15.38	(12%)

core is about 0.12%. This is based on the area breakdown for a 2-way OoO core with a merged register file at 45nm, which shows renaming taking ~4% of the real estate. Given our design increases by 3% the area of a 2-way RRS at 45nm, and RRS corresponds to 4% of the core area, then 4% x 3% = 0.12%.

Although the power overhead of IDLD does not burden the field operation, since IDLD is enabled only for the validation purposes (see Section V.B), we present it for the completeness. The IDLD mechanism incurs an energy overhead over the baseline implementation that ranges from 4% to 12% for 1-wide and 8-wide register renaming, respectively. The energy consumption numbers refer to the total energy of the RRS (not the whole OoO core).

Overall, the results in Table II show that IDLD is cost-efficient mechanism in terms of all hardware metrics. Both baseline and IDLD behave similarly when scaling to wider renaming (e.g., 8-wide), as the observed trends are dominated by the increase in the complexity of the renaming logic.

C. *Bug Modeling Evaluation & Bug Detection*

Exploiting further our microarchitecture-level bug modeling, in this section we also present some results about the distribution of any bug effect in RRS and the bug detection capability of IDLD compared to the traditional end-of-test checking method. Apart from the three bug effect classes discussed in subsection IV.B, there are also four more classes which belong to any observable bug effect.

SDC (Silent Data Corruption): The execution finishes normally (and the commit trace is comparable to the bug-free trace), but the program output is different as compared to the bug-free reference output, without observable indications.

Timeout: The execution is not finished within a certain amount of time, equal to 2.5 times the bug-free execution time. These executions are externally stopped to resolve potential deadlock or livelock situations.

Assert: The execution is unexpectedly terminated due to a high-level condition that the simulator is unable to handle. This means that the simulator cannot decide how a real system would behave and raises an assertion.

Crash: The execution does not reach the end, because it is interrupted by a catastrophic event. As a result, no program output is produced. A crash may refer to a process crash (killed process) or a system crash (kernel panic).

Figure 8 presents the detailed results for each benchmark for the Control Signals bug model. Different workloads provide different behavior for each bug effect class. The ramifications of control logic bugs vary arbitrarily depending on workload characteristics and execution patterns. This makes the validation process and the generation of validation tests more difficult and lowers the probability of the bug in the RRS logic to be detected. Figure 9 summarizes the results of our experiments for bug detection capability. IDLD can

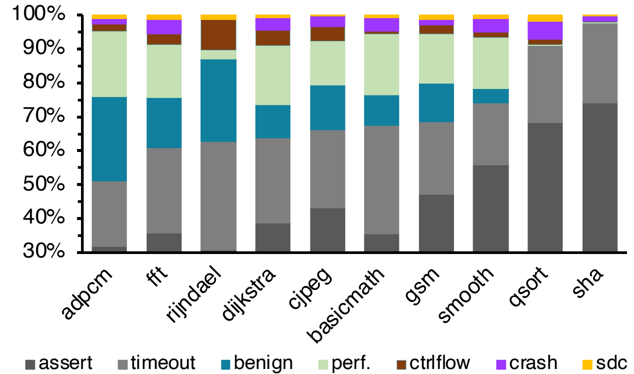

Figure 8. Breakdown of Outcomes for Bug injections in Control Signals per Benchmark.

detect all 30,000 bug occurrences (20,000 bugs for control signals corruption (duplication and leakage) and 10,000 bugs for Pdst corruption); i.e., bug coverage 100%.

Conversely, the traditional end-of-test checking techniques detects only 24,632 bugs (bug coverage 82.1%). This difference is due to the limitations of current validation techniques since they miss bugs that do not affect the program's output. As argued earlier, although a bug manifests itself during the post-silicon validation, there is a great chance to not affect the output of the validation test. Thankfully, IDLD guarantees the instantaneous detection, not only of bugs that provide visible evidence of their existence (e.g., SDC, crash, assert, timeout), but also for masked bug occurrences, therefore, such difficult-to-detect bugs become detectable.

In Figure 10, we also analyze the coverage of the BV method when combined with the traditional end-of-test checking post-silicon validation. As the results reveal, BV offers only 1% higher additional coverage when combined with the traditional end-of-testing method. Thus, a significant fraction of the bugs remains undetected even with the BV (about 17%). As discussed earlier, a bug in the RRS has a high probability of being masked without any persist effect, for example, when it occurs in the wrong path, something that the BV method is unable to detect as it only checks for bugs

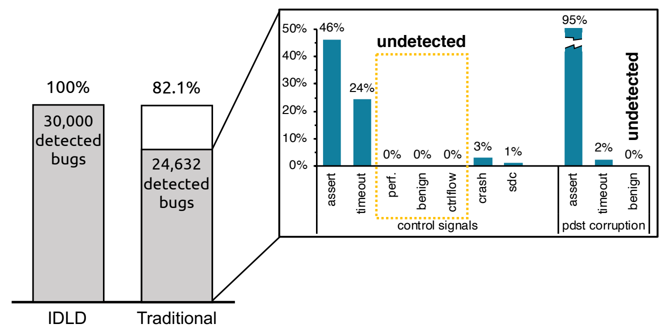

Figure 9. Bug detection capability of IDLD and traditional end-of-test checking post-silicon validation.

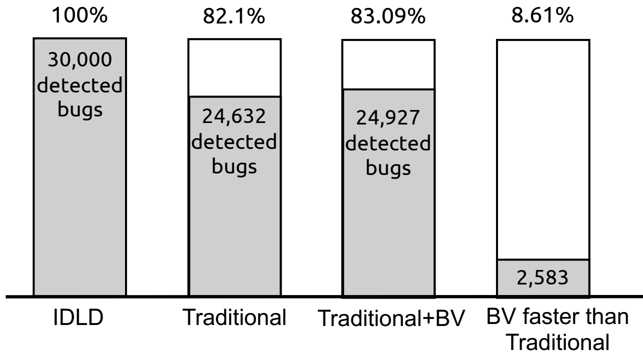

Figure 10. Bug detection capability of IDLD, traditional end-of-test checking post-silicon validation, traditional+BV.

on reclamation and when the pipeline is empty. The graph also shows the fraction of bugs the BV detects before the traditional end-of-testing approach. This is about 8.6% of the bugs, which indicates that a significant fraction of bugs (74.5%) is detected first by the traditional end-of-test checking. An analysis of the latency of the faster BV bug detections (not shown in a graph), reveals that BV detections may occur even up to millions of cycles after their activation. This analysis highlights the advantages offered by IDLD with its 100% coverage, instantaneous detection latency and simplicity.

VII. RELATED WORK

DIVA [23] is a technique that uses a simple checker core to validate what a more complex core is doing. It delivers very high bug coverage capable of detecting various types of errors in the core. The main drawbacks of DIVA are redundant execution units and new paths to the register file and caches. Regarding register renaming, DIVA leverages the availability of a split speculative and architectural register file for dynamic verification (validate values read using renamed registers by comparing prior to commit the architectural register values read using the logical register names). With a merged register file, as the one evaluated in this work and which is more widely used in recent products [37]-[42], there is no distinction between architectural and speculative state and unclear whether a cost-efficient DIVA variation exists for such scenario. Numerous techniques are employed in products to detect and, in some cases, correct errors due to faults [60]-[68]. However, few techniques can detect bugs that do not cause a corruption in a stored or generated value. For instance, an ECC protected array cannot detect a bug that prevents to write into an array. Even redundant lock-step systems cannot detect faults due to systematic bugs [69], since the same fault is activated in the redundant systems. Another previously proposed RRS protection method relies on the use of a regiment of simple checkers that monitor whether many basic microarchitectural invariances are upheld [70]. The approach of [70] detects corruption in the RRS arrays by checking that a physical register is reclaimed after being allocated. Invariance checking in [70] requires extending the functionality of time critical structures, for example adding read ports to a structure tracking operand readiness. It also suffers from delayed detection since an error is detected not when it occurs, but when it causes an invariance violation. This means that errors that get masked between these points of time are not detected at all. While not detecting soft errors that get masked is desirable, not detecting a bug that happens to get masked is not. IDLD overcomes such limitation.

Other approaches for bug and fault detection rely on a comparison of signatures that encode what should happen during execution vs. what actually happens [24] [25] [71] [72]. The schemes in [71] [72] require instruction set and compiler support, however, in our work we consider hardware only approaches for instantaneous detection of bugs. A key advantage of the proposed IDLD scheme is that it needs simple microarchitecture support and does not require ISA extensions and compiler support. Along the same lines, QED [51] and all its extensions cannot directly localize bugs at the level of hardware granularity, since they mainly aim at detecting faults in pre-silicon verification, and in post-silicon validation they can only detect bugs that primarily affect the program's execution. Moreover, hardware-based QED approaches provide expensive checkers in terms of area overhead [19]. Since timing/electrical bugs cannot be easily reproduced in simulation, software-based techniques such as QED or others which are based on diverse execution [52]: (1) cannot guarantee the bug detection, and (2) even when they detect the bug, the root-cause analysis will be impossible, since the bug is virtually non-reproducible. In [73] a column parity-based approach is proposed to protect FIFOs by tracking the information that enters and leaves a FIFO. IDLD is applicable to complex sub-systems with non-FIFOs, and it can check for invariance violations more frequently.

Other hardware approaches entail significant area and power overheads (reported to be 5-10% overall) and pervasive changes in time sensitive pipeline paths, e.g., extending arrays (physical register file [25] and renaming table [24]), multiplexers, buses and buffers in the OoO core engine to propagate and store multi-bit signatures. While the schemes in [24] [25] provide comprehensive bug coverage, including the RRS, unlike IDLD, they cannot capture bugs that cause only Pdst leakage because they do not affect correctness. [24] [25] can detect duplication, i.e., when two Pdsts point to the same physical register, when the invariances checked by [24] [25] are violated, but unlike IDLD (which detects instantaneously), there is no time guarantee for when this will occur (it depends on whether bug activation is not masked and on few other conditions). In any case, there is a value for cost-effective schemes that can target the bug protection of a specific core sub-system as we propose for the RRS in this paper. The specific choice for a

bug protection technique is clearly driven by return-on-investment (ROI) vs. overhead. For processors that already employ numerous RAS features the ROI from a new core-wide coverage scheme may be lower than just adding protection to a critical core sub-system that enables to detect critical bugs.

VIII. Conclusion

The paper shows that control logic bugs in the RRS of a modern OoO core result in duplication and leakage of Pdsts used in renaming. The work uses microarchitectural bug modeling to report that a significant percentage of bugs in RRS will not affect the program's functionality (i.e., masked), although they manifest themselves during post-silicon validation, and thus, they are extremely difficult to be detected. Even when they are not masked and manifest into deviations from the expected output, the control flow or the performance, these manifestations occur much later (often millions of cycles) after the bug activation time. These findings underline the significant challenge faced by a root causing effort of bugs in the RRS control logic. These findings motivate the proposed IDLD, a simple cost-effective hardware technique that can detect leakage and duplication in the RRS instantaneously after the bug occurrence. Synthesis analysis of an RTL design of an RRS with IDLD reveals minimal area overhead as compared to the baseline RRS design.

Acknowledgment

This research has been supported by the European Union Horizon 2020 programme through a H2020 Tetramax project TTX (Grant 761349), the H2020 UniServer project (Grant 688540), the FP7 Clereco project (Grant 611404), and by a Cisco and an Intel Research grant. The first three authors performed part of this work while working with Intel at the Israel Design Center in Haifa.

References

[1] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances," Design Automation Conference (DAC), 2010, pp. 12-17. doi: https://doi.org/10.1145/1837274.1837280

[2] Amir Nahir, "Post-Silicon Validation – Tackling 4 Billions Risks per Second," MEDIAN Workshop, 2012.

[3] H. Sohofi and Z. Navabi, "Assertion-based verification for system-level designs," Fifteenth International Symposium on Quality Electronic Design, 2014, pp. 582-588, doi: https://doi.org/10.1109/ISQED.2014.6783379.

[4] H. Foster, "Synthesizing assertions into hardware for faster silicon debug," Tech Design Forum, July 2012 https://www.techdesignforums.com/practice/technique/synthesizing-assertions-into-hardware-for-faster-silicon-debug/

[5] "Intel Core 2 Extreme Processor X6800 and Intel Core 2 Duo Desktop Processor E6000 and E4000 Sequence Specification Update," 2008.

[6] "Revision Guide for AMD Athlon 64 and AMD Opteron Processors," 2005.

[7] "5th Generation Intel® CoreTM Processor Family, Intel® CoreTM M Processor Family, Mobile Intel® Pentium® Processor Family, and Mobile Intel® Celeron® Processor Family, Specification Update," October 2015, Revision 015, Reference Number 330836-015.

[8] "Mobile 4th Generation Intel® CoreTM Processor Family, Mobile Intel® Pentium® Processor Family, and Mobile Intel® Celeron® Processor Family, Specification Update," November 2015, Revision 028, Reference Number 328903-028.

[9] "Intel® Xeon® Processor E3-1200 v3 Product Family Specification Update," January 2016, Reference Number 328908-014US.

[10] "Intel® Xeon® Processor E7-8800/4800 v3 Product Family, Specification Update," March 2016, Reference Number 332317-007US.

[11] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs," International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015, pp. 517-529. doi: https://doi.org/10.1145/2694344.2694366

[12] P. Patra, "On the cusp of a validation wall," IEEE Design & Test of Computers, vol. 24, no. 2, March-April 2007, pp. 193-196. doi: https://doi.org/10.1109/MDT.2007.54

[13] G. Papadimitriou, D. Gizopoulos, A. Chatzidimitriou, T. Kolan, A. Koyfman, R. Morad, and V. Sokhin, "Unveiling difficult bugs in address translation caching arrays for effective post-silicon validation," International Conference on Computer Design (ICCD), 2016, pp. 544-551, doi: https://doi.org/10.1109/ICCD.2016.7753339

[14] A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, and A. Ziv, "Threadmill: A post-silicon exerciser for multi-threaded processors," Design Automation Conference (DAC), 2011, pp. 860-865. doi: https://doi.org/10.1145/2024724.2024916

[15] J. Goodenough and R. Aitken, "Post-silicon is too late avoiding the $50 million paperweight starts with validated designs," Design Automation Conference (DAC), 2010, pp. 8-11, doi: https://doi.org/10.1145/1837274.1837279

[16] B. Bentley, "Validating the intel pentium 4 microprocessor," Design Automation Conference (DAC), 2001, pp. 244–248. doi: https://doi.org/10.1145/378239.378473

[17] D. Josephson, "The good, the bad, and the ugly of silicon debug," Design Automation Conference (DAC), 2006, pp. 3-6, doi: https://doi.org/10.1145/1146909.1146915

[18] S. S. Mukherjee, C.r T. Weaver, J. S. Emer, S. K. Reinhardt, T. M. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor." International Symposium on Microarchitecture (MICRO), 2003, 29-42. doi: https://doi.org/10.1109/MICRO.2003.1253181

[19] E. Singh, D. Lin, C. Barrett and S. Mitra, "Logic Bug Detection and Localization Using Symbolic Quick Error Detection ," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018. doi: https://doi.org/10.1109/tcad.2018.2834401

[20] A. Nahir, M. Dusanapudi, S. Kapoor, K. Reick, W. Roesner, K.-D. Schubert, K. Sharp, and G. Wetli, "Post-silicon validation of the IBM POWER8 processor," Design Automation Conference (DAC), 2014, pp. 1–6. doi: https://doi.org/10.1145/2593069.2593183

[21] D. Lee and V. Bertacco, ""MTraceCheck: Validating non-deterministic behavior of memory consistency models in post-silicon validation," International Symposium on Computer Architecture (ISCA), 2017, pp. 201-213, doi: https://doi.org/10.1145/3079856.3080235

[22] C.-H. Hsu, D. Chatterjee, R. Morad, R. Ga and V. Bertacco, “ArChiVED: Architectural checking via event digests for high performance validation,” Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014, pp. 1-6. doi: https://doi.org/10.7873/DATE.2014.330

[23] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," International Symposium on Microarchitecture (MICRO), 1999, pp. 196-207. doi: https://doi.org/10.1109/MICRO.1999.809458

[24] J. Carretero, P. Chaparro, X. Vera, J. Abella, and A. González, “End-to-end register data-flow continuous self-test,” International Symposium on Computer Architecture (ISCA), 2009, pp. 105-115. doi: https://doi.org/10.1145/1555754.1555770

[25] R. Nathan and D. J. Sorin, "Nostradamus: Low-cost hardware-only error detection for processor cores," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014, pp. 1-6, doi: https://doi.org/10.7873/DATE.2014.173

[26] “Intel Core X-Series Processor Family Specification Update,” Rev. 009, Document Number: 335901-009, February 2020, https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/6th-gen-x-series-spec-update.pdf

[27] “10th Generation Intel® Core™ Processor Specification Update,” Rev. 013, Document Number: 341079-013, October 2021, https://cdrdv2.intel.com/v1/dl/getContent/615213

[28] A. Abel and J. Reineke, "Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures”, 2021, https://arxiv.org/pdf/2107.14210.pdf

[29] “Intel February 2022 Microcode Update,” https://access.redhat.com/articles/6716541#register-checkpoint-race

[30] Robert M. Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units," IBM Journal of Research and Development, vol. 11, no. 1, Jan. 1967, pp. 25-33. doi: https://doi.org/10.1147/rd.111.0025

[31] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar and A. Yoaz, “A novel renaming scheme to exploit value temporal locality through physical register reuse and unification,” International Symposium on Microarchitecture (MICRO), 1998, pp. 216-225. doi: https://doi.org/10.1109/MICRO.1998.742783

[32] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, 1. Miller, and M. Upton, “Hyper-threading technology architecture and microarchitecture,” Intel Technology Journal, vol. 6, no. 1, 2002. pp. 4-15, 2002.

[33] S. Palacharla, N. P. Jouppi, and J. E. Smith, “Complexity-effective superscalar processors,” International Symposium on Computer Architecture (ISCA), 1997, pp. 206-218. doi: https://doi.org/10.1145/264107.264201

[34] G. Z. Chrysos, and J. S. Emer. "Memory dependence prediction using store sets," International Symposium on Computer Architecture (ISCA), 1998, pp. 142-153. doi: https://doi.org/10.1109/ISCA.1998.694770

[35] A. Gonzalez, F. Latorre and G. Magklis, “Processor Microarchitecture. An Implementation Approach,” Morgan & Claypool Publishers, 2011. https://doi.org/10.2200/S00309ED1V01Y201011CAC012

[36] J. E. Smith and A. R. Pleszkun, “Implementation of precise interrupts in pipelined processors,” International Symposium on Computer Architecture (ISCA), 1985, pp. 36–44. doi: https://doi.org/10.1145/327070.327125

[37] I. E. Papazian, “Next Generation Intel Xeon(R) Scalable Server Processor: Icelake-SP,” HotChips31 Symposium (HCS), 2020. doi: https://doi.org/10.1109/HCS49909.2020.9220434

[38] R. E. Kessler, “The Alpha 21264 microprocessor,” IEEE Micro, vol. 19, no. 2, pp. 24–36, March-April 1999. doi: https://doi.org/10.1109/40.755465

[39] K. C. Yeager, “The Mips R10000 superscalar microprocessor,” IEEE Micro, vol. 16, no. 2, pp. 28-41, April 1996, doi: https://doi.org/10.1109/40.491460

[40] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, Alan Kyker and Patrice Roussel. “The microarchitecture of the Pentium® 4 processor,” Intel Technology Journal, 2001.

[41] I. Anati, D. Blythe, J. Doweck, H. Jiang, W. f. Kao, J. Mandelblat, L. Rappoport, E. Rotem and A. Yasin, “Inside 6th gen Intel® Core™: New microarchitecture code named Skylake,” HotChips28 Symposium (HCS), 2016. doi: https://doi.org/10.1109/HOTCHIPS.2016.7936222

[42] M. Clark, “A new x86 core architecture for the next generation of computing,” HotChips28 Symposium (HCS), 2016. doi: https://doi.org/10.1109/HOTCHIPS.2016.7936224

[43] J. Mandelblat, “Intel’s Next Generation Microarchitecture Code Name Skylake,” Intel Developer Forum, 2015.

[44] A. Perais, F. A. Endo and A. Seznec, “Register sharing for equality prediction,” International Symposium on Microarchitecture (MICRO), 2016, pp. 1-12. doi: https://doi.org/10.1109/MICRO.2016.7783707

[45] A. Moshovos and G. Sohi, “Streamlining Inter-Operation Memory Communication via Data Dependence Prediction.” International Symposium on Microarchitecture (MICRO), 1997, pp. 235-245. doi: https://doi.org/10.1109/MICRO.1997.645814

[46] N. J. Wang, J. Quek, T. M. Rafacz and S. J. Patel, “Characterizing the effects of transient faults on a high-performance processor pipeline,” International Conference on Dependable Systems and Networks (DSN), 2004, pp. 61-70, doi: https://doi.org/10.1109/DSN.2004.1311877

[47] R. Gabor, Y. Sazeides, A. Bramnik, A. Andreou, C. Nicopoulos, K. Patsidis, D. Konstantinou, and G. Dimitrakopoulos, “Error-Shielded Register Renaming Sub-system for a Dynamically Scheduled Out-of-Order Core,” Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019, pp. 812-817, doi: https://doi.org/10.23919/date.2019.8715194

[48] M. Bushnell and V. Agrawal, “Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits,” Springer Publishing Company, Incorporated, 2013.

[49] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, “MiBench: A free, commercially representative embedded benchmark suite," International Workshop on Workload Characterization (WWC), 2001, pp. 3-14. doi: https://doi.org/10.1109/WWC.2001.990739

[50] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T.r Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” SIGARCH Computer Architecture News, vol. 39, no. 2, pp. 1–7, May 2011, doi: https://doi.org/10.1145/2024716.2024718

[51] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. Abdel Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra, "QED: Quick Error Detection tests for effective post-silicon validation," International Test Conference (ITC), 2010, pp. 1-10, doi: https://doi.org/10.1109/test.2010.5699215

[52] N. Foutris, D. Gizopoulos, M. Psarakis, X. Vera, and A. Gonzalez, “Accelerating microprocessor silicon validation by exposing ISA diversity,” International Symposium on Microarchitecture (MICRO), 2011, pp. 386-397. doi: https://doi.org/10.1145/2155620.2155666

[53] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," International Conference on Computer Design (ICCD), 2008, pp. 307-314, doi: https://doi.org/10.1109/ICCD.2008.4751878

[54] C. Turner, “Safety and security for automotive SoC design,” ARM, 2016. https://docplayer.net/42298283-Safety-and-security-for-automotive-soc-design.html

[55] J. Fruehe, “AMD Epyc brings new RAS capability increasing Reliability, Availability and Serviceability in the latest AMD Design.” Moor Insights and Strategy, June 2017.

[56] I. Wagner and V. Bertacco, “The Verification Universe” in “Post-Silicon and Runtime Verification for Modern Processors”, Springer, 2011.

[57] N. Foutris, D. Gizopoulos, X. Vera, and A. Gonzalez, “Deconfigurable microprocessor architectures for silicon debug acceleration,” International Symposium on Computer Architecture (ISCA), 2013, pp. 631–642. doi: https://doi.org/10.1145/2485922.2485976

[58] RISC-V Boom RTL Simulation Assertions for Leakage and Duplication https://github.com/riscv-boom/riscv-boom/blob/ad64c5419151e5e886daee7084d8399713b46b4b/src/main/scala/exu/rename/rename-freelist.scala#L95

[59] P. Meinerzhagen, S. M. Y. Sherazi, A. Burg and J. N. Rodrigues, “Benchmarking of Standard-Cell Based Memories in the Sub-VT Domain in 65-nm CMOS Technology,” in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 1, no. 2, pp. 173-182, June 2011. doi: https://doi.org/10.1109/JETCAS.2011.2162159

[60] R. R. Hornish, “777 Autopilot Flight Director System,” Digital Avionics Systems Conference (DASC), 1994, pp. 151-156, doi: https://doi.org/10.1109/dasc.1994.369488

[61] “MP2128 3X MicroPilot's Triple Redundant UAV Autopilot,” White paper, 2015.

[62] S. Siceloff, “Shuttle Computers Navigate Record of Reliability,” NASA, Tech. Rep. June, 2010.

[63] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb, “IBM's S/390 G5 microprocessor design,” IEEE Micro, vol. 19, no. 2, pp. 12-23, March-April 1999, doi: https://doi.org/10.1109/40.755464

[64] S. S. Mukherjee, M. Kontz and S. K. Reinhardt, “Detailed design and evaluation of redundant multi-threading alternatives,” International Symposium on Computer Architecture, 2002, pp. 99-110, doi: https://doi.org/10.1145/545214.545227

[65] “Intel® GO™ Autonomous Driving Solutions.” https://www.intel.com/content/dam/www/public/us/en/documents/platform-briefs/go-automated-accelerated-product-brief.pdf, 2017

[66] “TMS570LS3137 Product Manual,” Texas Instrument, 2015

[67] D. Henderson, “POWER8 Processor Based Systems RAS Introduction to Power Systems Reliability, Availability, and Serviceability,” Whitepaper, 2016

[68] K. T. Nguyen, “New Reliability, Availability and Serviceability (RAS) Features in the Intel Xeon Processor Family.”, Whitepaper, 2017

[69] https://www.kvausa.com/random-failure-vs-systematic-failure

[70] V. Reddy and E. Rotenberg, "Coverage of a microarchitecture-level fault check regimen in a superscalar processor," International Conference on Dependable Systems and Networks (DSN), 2008, pp. 1-10. doi: https://doi.org/10.1109/dsn.2008.4630065

[71] A. Meixner, M. E. Bauer and D. J. Sorin, “Argus: Low-Cost, Comprehensive Error Detection in Simple Cores,” in IEEE Micro, vol. 28, no. 1, pp. 52-59, Jan.-Feb. 2008, doi: https://doi.org/10.1109/micro.2007.18

[72] A. Meixner and D. J. Sorin, “Error Detection Using Dynamic Dataflow Verification,” International Conference on Parallel Architecture and Compilation Techniques (PACT), 2007, pp. 104-118, doi: https://doi.org/10.1109/PACT.2007.4336204

[73] I. Sideris and K. Pekmestzi, “A column parity based fault detection mechanism for FIFO buffers,” Integration, vol. 46, no. 3, 2013, pp. 265-279. doi: https://doi.org/10.1016/j.vlsi.2012.03.004