Hardware/Software-Based Diagnosis of Load-Store Queues Using Expandable Activity Logs

Javier Carretero, Xavier Vera, Jaume Abella, Tanausú Ramírez, Matteo Monchiero, Antonio González
Intel Barcelona Research Center, Intel Labs – UPC
Barcelona, Spain
javier.carretero.casado@intel.com, xavier.vera@intel.com, jabella@ac.upc.edu,
tanausux.ramirez@intel.com, matteo.monchiero@intel.com, antonio.gonzalez@intel.com

Abstract

The increasing device count and design complexity are posing significant challenges to post-silicon validation. Bug diagnosis is the most difficult step during post-silicon validation. Limited reproducibility and low testing speeds are common limitations in current testing techniques. Moreover, low observability defies full-speed testing approaches. Modern solutions like on-chip trace buffers alleviate these issues, but are unable to store long activity traces. As a consequence, the cost of post-Si validation now represents a large fraction of the total design cost. This work describes a hybrid post-Si approach to validate a modern load-store queue. We use an effective error detection mechanism and an expandable logging mechanism to observe the microarchitectural activity for long periods of time, at processor full-speed. Validation is performed by analyzing the log activity by means of a diagnosis algorithm. Correct memory ordering is checked to root the cause of errors.

1 Introduction

Nowadays, the major bottleneck of the design cycle is the verification of complex microprocessors. Intel reported a headcount ratio of 3:1 for design vs. post-silicon validation [20] during 2005. In fact, silicon debug has become the most costly activity in the development of a new design (35% on average for 90nm designs) [1]. Moreover, it is forecasted [22] that the increasing costs of equipments for post-silicon validation will make products prohibitively expensive in the future. This is known as the Design and Verification Gap.

The first step during post-silicon validation iteration consists in the detection of a failure, by means of error detection mechanisms, through specific testing applications or through system crash or deadlock. Then, the engineers' objective is to pinpoint the location and cause of the failure [1, 13, 16]. Finally, the root cause of the error [21, 24] must be fixed, by means of error circumvention or by RTL recoding.

The most difficult part during post-silicon validation is the step related to bug localization [19]. This includes determining the location and cause of failure, as well as the sequence of processor activity during a period of time that resulted in the activation of a bug or fault.

Techniques like full scan [10] are adopted widespread. In this approach, the circuit under debug is stimulated and its output is captured after the manifestation of a specific trigger event [2]. The captured response (*scan dump*) is then sent and interpreted by the debug software. Also, this software uses the JTAG interface [12] to control the testing. However, the system state is not preserved after a *scan dump* is obtained and hence, this causes difficulties when debugging failures for whom engineers need to observe consecutive internal states. On the other hand, on-chip trace buffers [2, 11] allow real-time observation of a continuous flow of internal signals (*traces*), but the main drawback lies in the fact that the limited size of the buffer restricts the analysis window.

Several challenges make post-silicon validation a time-consuming and complex task. First, failures need to be reproducible. Reproducibility allows engineers to replay the error and examining the state before and after the failure.

Second, post-Si validation often requires obtaining a-priori golden outputs by means of tedious functional and RTL simulations. However, since simulation is orders of magnitude slower than full-speed system testing, it translates into more expensive products, or products with a higher number of potential bugs.

Third, post-silicon validation poses the problem of long error detection latency. Often, specific test programs are run at full-speed to either detect crashing conditions or a mismatch in the expected golden output [23]. However,

978-1-4244-9435-4/11/$26.00 ©2011 IEEE

bugs often are activated because of complex interactions among processor structures and manifest at the application level output long after they have been triggered [15]. The long error manifestation latency is aggravated by the fact that during post-silicon validation there is limited processor internal observability.

To reduce the impact of these issues, we present a novel hybrid hardware/software solution to diagnose defects/bugs during post-silicon validation. Our solution offers the benefits of high validation coverage and debugging support with negligible area and performance overhead. Our post-Si validation approach is based on a simple and scalable in-hardware activity logging mechanism that observes selected internal processor activity during program execution, at full-speed. This allows alleviating the problem of internal observability, reproducibility and testing speed. On a second axis, we propose to integrate a continuous timely error detection technique with the logging mechanism in order to enable automated post-silicon validation. This allows minimizing the error detection latency problem and the golden output problem, while at the same time reducing the required space for on-chip logging buffers. The last component of our system is a diagnosis algorithm that analyzes the activity logs and can automatically pinpoint the root cause of a failure.

This validation approach targets the diagnosis of errors in the Load Store Queue (LSQ), one of the most stressed and complex component in a modern core.

The rest of the paper is organized as follows. Section 2 details our processor configuration. Section 3 introduces our post-Si validation technique. Next, Sections 4 to 7 delve into the specific details of the implementation. Results are presented in Section 8. Previous work is reviewed in Section 9. Finally, Section 10 summarizes this paper.

2 Baseline Processor

Our proposed system has been implemented and evaluated on a top of a complex out-of-order processor that resembles the Intel® Core™microarchitecture [25].

Specifically, its data cache holds up to 32KB of data storage and it is configured as a 8-way cache. Also, it features two shared read/write ports, able to retrieve 32 bytes each. These ports allow a full cache line access per cycle (64 bytes). However, it is only possible to use either the ports as two read ports or one write port, not a combination of both. The second level cache supports a 4MB storage space, organized in 16 ways and offers a single shared read/write port of 64 bytes. The memory system implements a MSHR able to support up to 8 outstanding misses. The processor can allocate up to 4 micro-operations per cycle (potentially 4 loads), and retire out from the ROB up to 4 micro-operations per cycle. Up to 1 non-bogus store can update

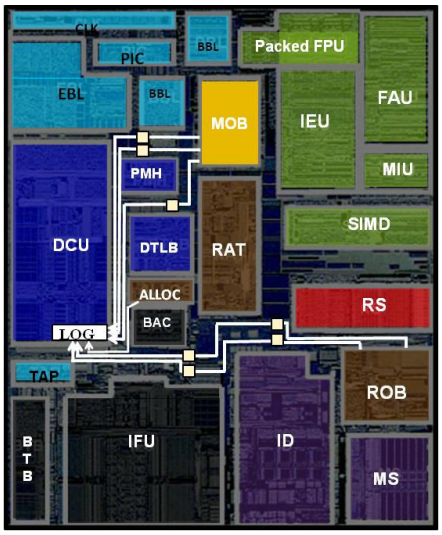


Figure 1. Event generation location places and event driving latches

the architectural state every cycle, although up to 4 bogus stores can be retired too. The *Load Queue* has 30 entries, whereas the *Store Queue* holds up to 20 stores. Two loads can potentially access every cycle the data cache and scan the store queue for forwarding situations. Similarly, one store can scan the load queue per cycle to detect ordering violations.

3 Post-Si Validation System Overview

Our post-Si validation approach uses a hybrid hardware/software core-level mechanism that logs the microarchitectural activity for later analysis. This mechanism allows storing traces that reflect the system activity during program execution for long periods of time, at processor full-speed. We define an *event* as a microarchitectural activity or a change of state related to the circuit under debug. A selected set of *event types* are tracked by the logging mechanism and an aggregate of them constitute the activity log used for debugging.

With small changes in a couple of subroutines of the OS, the log can be stored in one or more pages of the memory space of each application being run. This way, we can store long logs without adding big memory structures in the processor or impacting the performance of applications when stealing large part of the cache. Events generated by the processor are temporally stored in a small hardware buffer while waiting for an idle write port in the cache. The data cache acts as a proxy to the rest of the memory hierarchy.

Connected to the logging mechanism, we propose adding a lightweight and timely error detection mechanism.

Its purpose is to detect the failure as soon as it can propagate a data corruption. It is also necessary in order to maintain the log unpolluted from microarchitectural events past the error manifestation point.

The last component of our validation method is a software-based diagnosis algorithm. Once an error is detected by the error detection mechanism, this algorithm will examine the log and will root the cause for such error.

Next, we detail how the different components are integrated and how they interact with each other.

4 Event Generation

The first necessary modification consists in collecting the microarchitectural LSQ activity. Every memory operation has associated several types of activity events that may be generated out of program order. The activity log is built incrementally by aggregating the events that occur within the same cycle. It reflects the activity introduced in the pipeline in a *timely manner*.

Figure 1 shows the layout locations where events are generated. We have identified 4 event types used to perform LSQ failure diagnosis. Each event carries some important piece of information that is used later by the software diagnosis:

1. **`ALLOC` event**: we generate this event when a load is allocated. The information associated to this event consists of the memory size it reads from memory (3 bits), its position within the *Load Queue* (5 bits), and the *Store Queue* head and tail pointer values upon its allocation (both 5 bits).

2. **`COMMIT` event**: we generate it when a store commits. This event contains the store position within the *Store Queue* (5 bits), and a bogusness bit indicating whether the store belongs to a wrong control path or not (1 bit).

3. **`AGEN` event**: we generate an `AGEN` event for every executed store. This event contains: the store position within the *Store Queue* (5 bits), its effective size (3 bits), its linear address (32 bits) and two extra fields indicating whether the store detected a load introducing a *memory order violation* (1 bit) and the corresponding load position within the *Load Queue* (5 bits).

4. **`LDEXEC` event**: for every executed load, we generate a `LDEXEC` event that indicates the load's *Load Queue* position (5 bits), its linear address (32 bits), the read port used to move the load out of the *Load Queue* (1 bit), and two extra fields to tell whether there was a *store-forwarding situation* (1 bit) and the forwarded *Store Queue* position (5 bits).

Every event type and its associated information are generated in one pipeline stage and one microarchitectural structure. This means that there is no need to gather information from other parts of the core. As a consequence, events are generated locally but stored on a centralized structure, called the '*LOG buffer*'. Due to layout constraints it may happen that the delay required to move events to the hardware log may vary depending on the pipeline location. To solve this issue we add latches so that every event type generated during the same clock cycle arrives to the log at the same time. The number of latches to be inserted per event type is determined by the worst delay. Nevertheless, inserting latches does not pose any problem to the operating frequency because events do not need to be logged on the very same cycle they are generated.

Entries that constitute the log are not meant for specific event types. Hence, any event can be written to any position within the log. Moreover, log entries are not pre-reserved but used as they are needed. In order to distinguish among event types, we add decode information to every generated event. This adds 3 bits per event: 2 for the event type and 1 to indicate if it is valid or not. Hence, an `ALLOC` event requires 21 bits, a `COMMIT` event 9, an `AGEN` event 49 and a `LDEXEC` event 47.

4.1 Event Fusing Optimization

The quantity of information required per event type is different. `ALLOC` and `COMMIT` events require fewer bits than `AGEN` and `LDEXEC` events, because the size of the addresses dominates over the rest. This means that small events will have spare bits in the log entries.

We make use of this situation and propose to fuse `ALLOC` and `COMMIT` events. This allows us reducing the number of events to be written to the log per cycle, at the expense of higher software diagnosis complexity.

`ALLOC` events are fused by storing the number of loads allocated in the same cycle (this needs 2 bits, because a maximum of 4 loads can be allocated per cycle), their corresponding sizes (12 bits), the first load *Load Queue* position (5 bits), the *Store Queue* head pointer and the tail *Store Queue* pointer values observed during their allocation (20 and 20 bits, respectively). This optimization makes an `ALLOC` event 62 bits long. Our architecture allows only one store commit per cycle. However, `COMMIT` events for bogus stores are compressed into a single event. To do so, we indicate the number of bogus stores retired (2 bits, because at most 4 instructions can be retired per cycle), the initial store's *Store Queue* position (5 bits) along with the bogusness bit set to true (1 bit). As a consequence several `COMMIT` events can be fused to 11 bits.

After applying this optimization, every event stored in the log will require 64 bits of space. Notice that a maximum

Failure Scenario	Description	Analysis Window
FWD_FROM_YOUNGER	Load was forwarded from a younger store	From ALLOC
FWD_FROM_IDLE	Load was forwarded from an idle STQ position	From ALLOC
FWD_FROM_BOGUS	Load was forwarded from a bogus store	From ALLOC
FWD_FROM_OLDER_NOT_YET_EXECUTED	Load was forwarded from a previous store that did not compute its address	From farthest AGEN event
FWD_FROM_OLDER_YET_NON_OVERLAPPING	Load was forwarded from an older non matching store	From farthest AGEN event
FWD_AND_KILL_FROM_YOUNGER	Load was forwarded but there was a younger store that wrongly killed the load	From ALLOC
FWD_AND_KILL_FROM_BOGUS	Load was forwarded but then was killed by a bogus store	From ALLOC
FWD_AND_KILL_FROM_OLDER_BUT_OLDER_THAN_FWD	Load was forwarded from an older matching store but there was a matching store older than the forwarding one which wrongly killed the load	From ALLOC
FWD_AND_KILL_FROM_OLDER_YET_NON_OVERLAPPING	Load was forwarded but a previous non matching store later killed the load	From ALLOC
FWD_BUT_KILL_NOT_PERFORMED	Load was forwarded but there was an older store, younger than the forwarding one which should have invalidated the load	From ALLOC
FWD_BUT_OTHER_POSSIBLE_FWD	Load was forwarded from an older store but should have been forwarded from a store older than the load but younger than the wrong forwarding store	From farthest AGEN event
KILL_FROM_YOUNGER	Load was killed by a younger store	From ALLOC
KILL_FROM_BOGUS	Load was killed by a bogus store	From ALLOC
KILL_FROM_OLDER_YET_NON_OVERLAPPING	Load was killed by an older store but its address did not match	From ALLOC
KILL_BUT_INVALID_ST_ACTING_AFTER	Load was killed by the correct store, but afterwards another store performed an invalid kill or fwd action	From ALLOC
KILL_BUT_VALID_YOUNGER_FWDABLE_STORE	Load was killed by an older matching store, but there was an older store younger than the killing one that should have performed a forwarding	From farthest AGEN event
KILL_BUT_VALID_YOUNGER_SHOULD_KILL	Load was killed by an older matching store, but an older store younger than the killer one should have killed it	From ALLOC
FWD_NOT_PERFORMED	Load should have been forwarded	From farthest AGEN event
KILL_NOT_PERFORMED	Load should have been killed by an older matching store	From ALLOC

Table 1. Summary of failure scenarios with their description and the required analysis window

of 6 events can be generated per cycle: 1 ALLOC, 1 AGEN, 2 LDEXEC and 2 COMMIT events (whereas if no optimization is applied, 11 events can be generated per cycle in the worst case).

4.2 Address Hashing Optimization

A second optimization, consists in reducing the size of the larger events so that more event entries can fit in a given area budget. To do so, we compress addresses of AGEN and LDEXEC events. In this case, a full 32 bit address would be reduced to a smaller number of bits by means of *address hashing*. Depending on the hash size, we may have AGEN events ranging from 18 to 48 bits and LDEXEC events ranging from 16 to 46 bits (1-bit and 31-bit hashes, respectively).

Clearly, this design alternative allows minimizing the required size of an event in the log, at the expense of some loss in the diagnosis coverage. Note that this optimization would not allow applying the '*event fusion*' optimization, and viceversa.

When using a reasonable hash size (like 8-bit), every event in the log will require 32 bits of space.

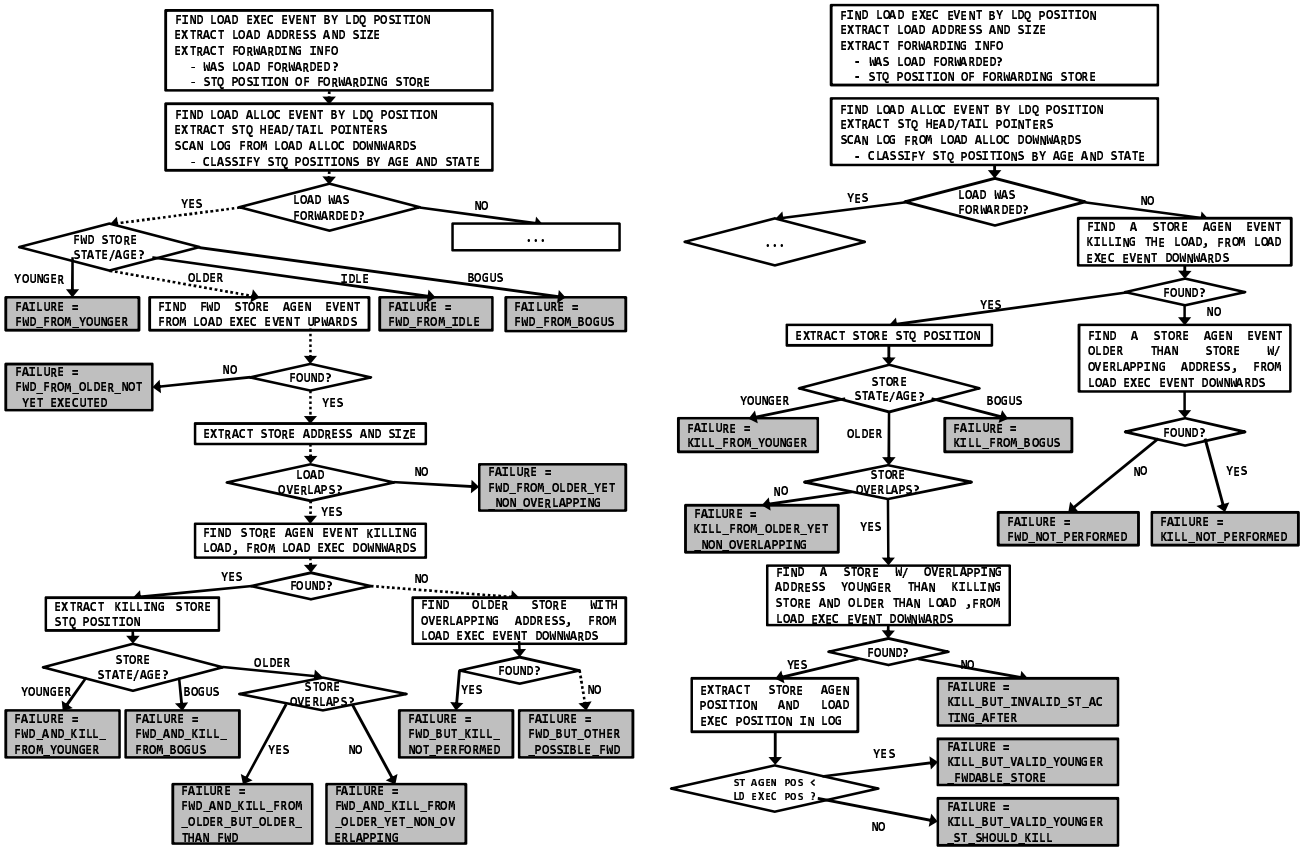
5 Error Detection Mechanism

As part of the validation system, we add a concurrent, timely error detection mechanism. This feature allows detecting any failure (including a design bug) as soon as it can propagate a data corruption, and allows having a precise, unpolluted state in the processor microarchitecture and in the activity log upon error detection (no events past the error manifestation point are logged). Being concurrent allows detecting failures arising from multiple sources of failures (including design bugs and transient faults).

Recently, some works have proposed *ad-hoc* solutions for checking the memory ordering in a targeted and effective way [5, 17, 18]. Specifically, we have chosen the *MOVT* mechanism [5] because of its area and performance overhead. The *MOVT* mechanism relies on a tiny cache-like structure that is updated at commit time and keeps track of the last store to each cached address. The detection of a failure in the LSQ logic is done at the commit time of a load.

6 Diagnosis Algorithm

The logging of the processor activity is done in parallel to processor operation. When an error is detected a fail-



(a) Failing load was forwarded at execution time

b) Failing load was not forwarded at execution time

Figure 2. High-level code snippet of the diagnosis algorithm showing failure type determination

ure is flagged and we insert into the log the information of the committing load that observed a failure (its *Load Queue* position).

Different levels of precision may be implemented, depending on the amount of information that designers want to obtain as feedback. We have identified two possible diagnosis levels. For example, the diagnosis algorithm may signal a failure case where "*a load at LDQ position 2 with address* `0x82ba1700` *and size* `4` *has been nullified by an older store with address* `0x92ba1700`" or it may even extend it with the information that "*the load should actually have been forwarded from the store at STQ position* `8` *with address* `0x82ba1700` *and size* `8`". Clearly, the second output provides more valuable information for debuggers because besides determining the failure that actually happened during the processor operation, it also allows to determine the expected behavior. However, it is clear that as we increase the diagnosis precision, the bigger will be the number of events to be analyzed (the analysis window).

Based on previous pre- and post-Si product bugs, we consider 19 different common failure scenarios in the LSQ logic. They are described in Table 1. The first column corresponds to the failure name, the second column describes the failure scenario and the third column indicates the size of the analysis window required to identify the actual failure scenario. Specifically, two different window sizes are required to diagnose the considered failure scenarios. The first group of failures can be diagnosed by considering an analysis window starting at the failing load `ALLOC` event (and ending in the last logged event). The second group of failures can only be diagnosed when increasing the analysis window up to the farthest `AGEN` event of that store that at the failing load `ALLOC` time was older than the load and whom the processor did not `COMMIT` before the failing load `LDEXEC` time. This analyis window also allows determining the expected failure-free case.

A diagnosis algorithm must minimize the number of events to be analyzed. Due to efficiency reasons, we implement an algorithm based on classifying failures depending on a *decision tree*. As nodes are visited (groups of failure scenarios), the failure scenarios are refined depending on the outcomes of different tests. Figure 2(a) and Figure 2(b) depict the implemented diagnosis algorithm at a high level.

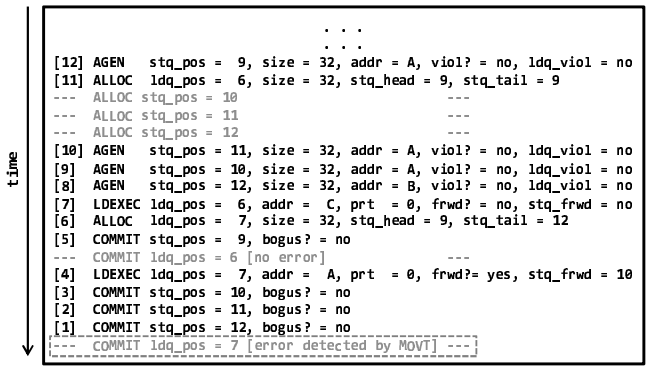


Figure 3. Log capturing a LSQ failure

Note that this code snippet does not provide the expected failure-free LSQ behavior.

The algorithm is constructed in such a way that the first failure types to be considered are those who require the smallest analysis window. Later, if these failures types do not correspond to actual failure case, the rest of failure types are considered (increasing the analysis window). Hence, the number of events to examine in order to identify what went wrong is not fixed a priori. It depends on the actual failure case, and the degree of diagnosis precision desired by testers.

Figure 3 shows an example of a short log capturing a failure in the LSQ operation. To clarify things, events colored in light grey are not captured in the log but have been added to clarify the temporal evolution of the microarchitectural activity. The diagnosis algorithm will determine that the failing load is the one in slot 7 in the *Load Queue* (load 7). This information is provided by the error detection mechanism. The failing load `LDEXEC` event occupies position <4> in the log, and it indicates that the load has address A and was forwarded by the store in slot 10 of the *Store Queue* (store 10). The failing load `ALLOC` event occupies position <6> and it indicates that upon its allocation, stores 9/10/11/12 were already in the *Store Queue* and were older. Scanning the log from this event down to event in position <4>, we can refine all store ages and states. Since store 9 was `COMMIT`ed in position <5>, before the failing load executed, it is now considered as idle (it disappeared from the *Store Queue*). Also, older stores 10/11/12 are determined to be not bogus. Hence, the algorithm follows the edge called `OLDER`, as shown with dashed lines in Figure 2(a). Then, the forwarding store `AGEN` event is found (analysis window is extended to position <9>) and its address and size are obtained. For this example, store 10 overlaps with the failing load (it has the same address and size) and was executed before. Next, the algorithm finds no `AGEN` event killing the failing load (from event <4> downwards). Finally, the analysis window is extended to event <10> to find that

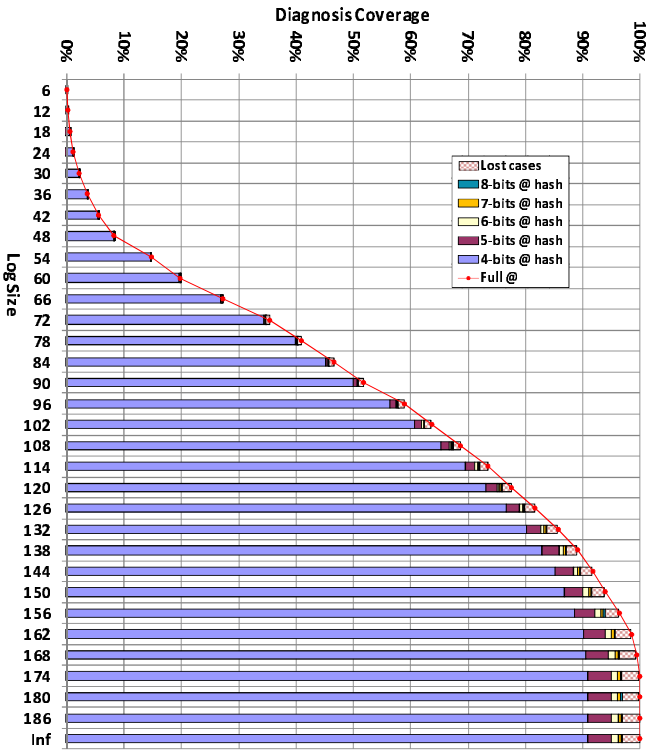


Figure 4. Required log size vs coverage

there is an `AGEN` event from store 11, younger than store 10, and it is overlapping. Hence, the diagnosis algorithm concludes that load should have been forwarded by store 11 (`FWD_BUT_OTHER_POSSIBLE_FWD` failure scenario).

6.1 Diagnosis Coverage versus Log Size

The diagnosis algorithm described in the previous section is able to identify faults for an ideal scenario where the log is unbounded and there is no limit on the number of events that can be logged per cycle. Figure 4 shows the average required number of logged events to root a fault for the `SPEC2K` benchmark suite (see Section 8). In this case, we have used the highest level of diagnosis precision, and have also applied the '*event fusion*' optimization. As one can see, if our log keeps the last 180 events, we are able to root almost all possible faults (99.96%). In case the log is bounded to a fixed size, a failure will not be diagnosable if the algorithm runs out of events in the log and has not taken any decision.

Finally, it is important to note that the *address hashing* optimization has implications on the achievable diagnosis coverage. A failure observed for a load will not be able to be diagnosed in case there is more than one store whose address hashing matches the load address hash. Similarly, in case there is more than one store whose full address

matches the load address, then the load will not be diagnosable for any hash size smaller than the length of the address. Whereas in the first case this can be alleviated by increasing the hash size, in the latter case this can only be solved by avoiding *address hashing*.

Figure 4 also depicts the overall diagnosis coverage loss for different hash sizes for our benchmark suite. It is interesting to note that 8-bit hashes addresses shows the best trade-off since they represent the 99.84% diagnosability potential of the *address hashing* optimization. However, the percentage of faults that cannot be diagnosed when using this technique ('*Lost cases*') is 2.91%. Moreover, some specific failure scenarios would never be detected when using the *address hashing* optimization (such as `FWD_BUT_OTHER_POSSIBLE_FWD`, `KILL_BUT_VALID_YOUNGER_FWDABLE_STORE`, etc.). Since the '*event fusion*' and the '*address hashing*' optimizations are exclusive, we opt to use the former one.

7 Logging System Implementation

From the previous upper-bound coverage results, it seems impractical to implement purely in hardware 180 hardware log entries. To solve these issues, we propose a more adaptable hybrid hardware/software solution. We modify the OS to sequester one or more physical pages from the application being run (4KB) to work as a circular buffer for the events. Connected to the data cache, we introduce a small hardware buffer to temporally store the events generated by the processor. This buffer sends the events to the main log (in memory) through the data cache, whenever its write port is idle. Events will be stored in event cache lines and will be stored on any way, controlled by the cache replacement policy. Moreover, these cache line can be replaced as needed by the application in an adaptive manner and can move through the memory hierarchy.

7.1 Required Hardware Changes

The required changes introduced in the processor are depicted in the left hand-side of Figure 5(a) and have been tagged as '*LOGGING system*'. The inputs to the logging component are two: (*i*) the monitored events that have been generated in their corresponding pipeline stages and (*ii*) a signal from the retire logic indicating whether the data cache ('*DL0*') write port is going to be available (idle) during the current cycle. Next, we will explain all different components in detail.

Merging line: We use a special buffer called '*merging line*' which is as big as a cache line (64 bytes). The main purpose of the '*merging line*' is to: (*i*) offload as many events whenever the cache write port is idle, (*ii*) cluster the LSQ events in the minimum number of cache lines and (*iii*) reduce power by using less idle cache write cycles. This also allows tolerating many consecutive cycles for which the cache write port is busy.

The '*merging line*' is dumped to the data cache whenever the '*Dump logic*' determines so. This happens whenever the port is idle and there is at least one event in the '*merging line*', although different policies can be implemented. Note that when using the '*event fusion*' optimization, the '*merging line*' is able to store 8 events.

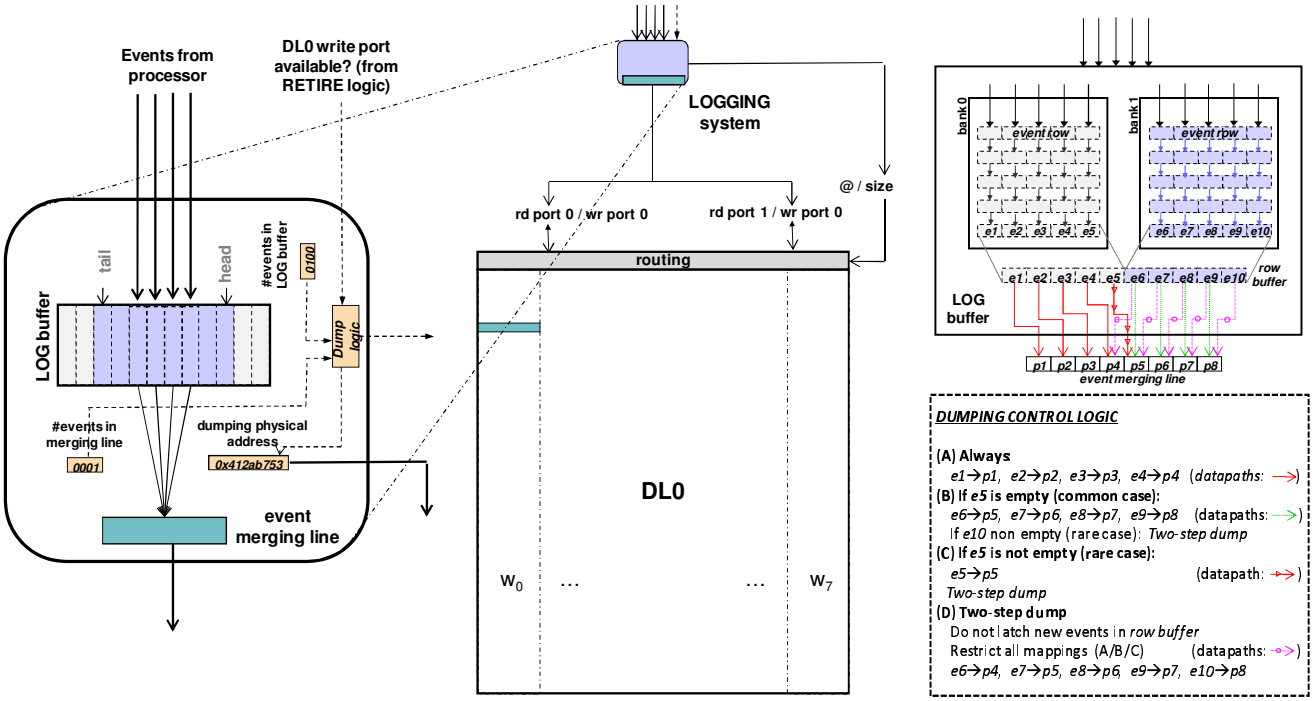
LOG buffer: The '*LOG buffer*' stores the events generated by the processor and is designed so that every cycle it can store a fixed number of generated events, unless it is full. Events are moved out to the '*merging line*' each cycle if enough space is available.

As Figure 5(b) shows, the '*LOG buffer*' is organized as a multi-banked structure (instead of a multi-ported structure). Events generated by the processor in the same cycle are stored together in an '*event row*' (a group of as many latches as the number of writable events per cycle). Every '*bank*' is organized as multiple chains of '*event rows*', and every cycle the events inside an '*event row*' advance and are latched into the next '*event row*', if the destination is not being used and the source has some events. Events from the same cycle are written into banks in a rotative manner. As a consequence, '*event rows*' in the same positions in different banks must advance simultaneously, so that time ordering across banks is maintained.

Two '*event rows*' are read every cycle out from the two banks into a buffer called '*row buffer*'. This allows reading the events generated from two different cycles, Note that a multi-ported configuration would allow reading events from as many different cycles as the number of read ports. Some empty events may be present in the '*row buffer*' and in the '*event merging line*', so we mark them at write time so that the diagnosis algorithm may identify them.

The events in the '*row buffer*' are then moved into the '*event merging line*', once it has been dumped to the memory hierarchy (it is empty). A clean '*event merging line*' allows associating fixed positions among the events in the buffer and the positions in the '*event merging line*', avoiding full shuffling trees. On the other hand, this restricts using the full capacity of the '*event merging line*' and can introduce cycles where the dumping cannot be performed. Given that the number of non-empty events in the buffer may surpass the capacity of the '*event merging line*', it may be necessary to dump the events in two different cycles. However, the '*LOG buffer*' can be designed in such a way that a single-step dump can be performed for the common case where 4 or less events are generated per cycle (a total of 8 events, the size of the '*event merging line*'). The '*dumping control logic*' manages this, as Figure 5(b) shows.

In case the cache port is busy for several cycles, the '*LOG buffer*' may also end up being full. As a consequence,



(a) LOGGING System (b) LOG buffer organization

Figure 5. Hardware scheme of the activity logging mechanism

some events may not be logged. Also, it may happen that during a cycle more events are generated than the number of events that potentially can be written. To address these issues we propose that whenever an event cannot be added to the log, the very first next successfully written event will be extended with a '*barrier*' bit. From the diagnosis algorithm perspective this means that in case a '*barrier*' bit is found the failure is not diagnosable.

Physical memory addressing: The logging system also contains a pointer to the physical memory position (aligned to a cache line boundary) where the '*merging line*' will be offloaded. The '*dumping physical address*' is incremented after the '*merging line*' has been moved to the data cache, and the increment is triggered by the '*Dump logic*'. Hence, events are stored in consecutive positions in memory.

Also, there is no need to perform a TLB translation. This eludes the design complexity of dealing with TLB misses that are not caused by the application itself.

7.2 Sequestering a page through the OS

From a software perspective, we opt to have the OS responsible of sequestering the physical page and providing the '*dump physical address*' to the hardware.

Upon process creation, the OS gets as many consecutive physical pages as needed by the diagnosis, and then assigns it to the logging system. For LSQ diagnosis, one physical page suffices. This physical page is considered by the OS as pinned (cannot be swapped). Then, the OS communicates the physical address to the hardware component. To do so, we consider the '*dumping physical address register*' as a memory mapped register and accessing it by regular I/O instructions.

Application switching: Upon a process switch-out, the OS will read the corresponding physical address pointer and store it in the process OS structure for next use. The logging physical pages will be invisible to any process and their addresses will not be stored in any translation table (hence, not accessible). Only the OS will know about their existence.

7.3 Putting It All Together

Once a failure is detected by the error detection technique, the following steps are taken:

1. Pending events residing in the '*LOG buffer*' and the '*event merging line*' are drained off through the data cache.

2. The information gathered by the error detection mechanism (*Load Queue* position of the load raising the error), the head and tail physical address pointers are then dumped to the logging physical page. This information is stored in the first 64 bytes of the logging

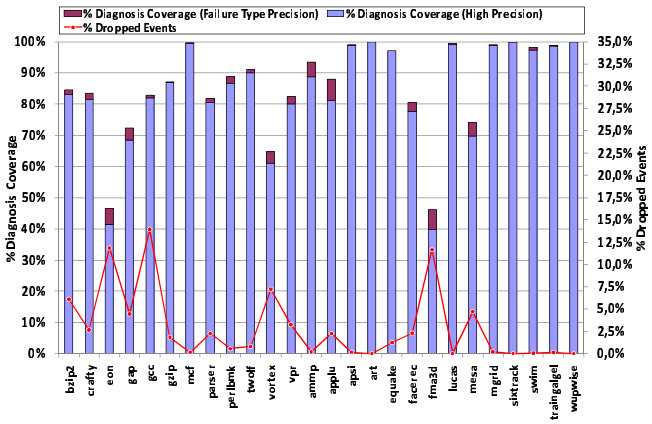


Figure 6. Diagnosis coverage results

physical page (hence, the logging of events would start on the second cache line boundary of that page).

3. A `MACHINE CHECK` exception is thrown. We rely on existing features to report hardware errors [8]. The processor modifies the respective control and status registers from the corresponding error-reporting register banks in order to indicate that a LSQ error has been reported.

Once the exception has been thrown, the OS takes the final steps.

4. The exception routine will identify an error in the LSQ and will dump the logging physical pages to a file dump for later analysis in another CPU.

8 Evaluation

This section evaluates our technique in terms of diagnosis coverage, area, power and performance overheads. We have evaluated it for a processor that resembles the Intel® Core™microarchitecture. The full set of `SPEC2K` benchmarks have been used. Every benchmarks has been run for 100M instructions, with a warm-up period of 100M instructions.

8.1 Diagnosis Coverage

For every failure scenario and program, we have simulated the injection of 1000 effective faults. We have allowed them to propagate and being detected. For each of them, we have run the diagnosis algorithm to compute the diagnosis coverage. The highest diagnosis precision level has been considered.

The '*LOG buffer*' structure has been sized and configured so that a sweet spot is achieved with respect diagnosis coverage and area overhead. Our system supports a total of 12 '*events rows*' (split in 2 banks), each one able to sustain a maximum of 5 event writes per cycle. This means that potentially the '*LOG buffer*' can keep up to 60 valid events. Despite the maximum number of events generated per cycle is 6, this situation happens seldom. Our analysis shows that allowing 5 writable events accounts for 99.96% of the cycles.

The diagnosis capability shown by our technique varies from one application to another. Figure 6 shows the achieved diagnosis coverage when running the whole `SPEC2K` benchmark suite on a logging system configured as our best choice. On average, our system is able to achieve a diagnosis coverage of 87.79%. Figure 6 also shows the diagnosis coverage that can be achieved when using a lower precision in the diagnosis algorithm. Specifically, this precision level allows the algorithm to pinpoint the actual failure type, not the expected failure-free scenario. Results show that diagnosis can be increased to 89.84%, on average.

It is interesting to highlight that even when the diagnosis algorithm is not able to pinpoint the root cause for a *given* fault injection, if we permanently allow the same fault to manifest again and again, then the system is able to diagnose all the failures and failure types (at least on one application).

There is not a linear relationship between the percentage of dropped events and the diagnosis coverage. Even when dropping a similar percentage of events, the percentage of loads that can be completely diagnosed can vary (`gcc-gap`, `crafty-gap`, and `mcf-perlbmk`). We also notice that applications such as `eon` and `fma3d` obtain poor diagnosis coverage because they exhibit phases where a huge amount of consecutive cycles (15 or more) the data cache write port is used by the application. During the length of this period, more events need to be allocated than the number of events that can be dumped from the '*event merging line*' to the memory hierarchy (a maximum of 8). When these phases dominate, a large percentage of events are lost and diagnosis coverage decreases. For these glass-jaw cases even a '*LOG buffer*' of 32 '*event rows*' would not be able to provide diagnosis coverage above 60%, which is already too expensive for a post-Si technique.

8.2 Performance overhead

We have compared the performance impact with respect to approaches that sequester a cache way or a group of adjacent sets with an equivalent storage capacity (4KB). Results in Figure 7 show that in the worst case, a slowdown of 2.71% is introduced when using our hybrid hardware/software logging approach. When using an approach that reduces a way (1 out of 8), the worst performance slowdown is 4.23%. Reducing the cache an equivalent number

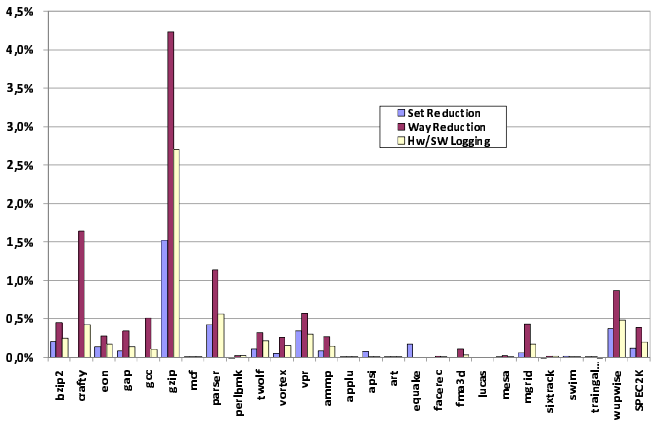


Figure 7. Induced performance overhead

of sets (8 out of 64) introduces less performance overhead (worst case is 1.52%), assuming set re-mapping is enabled. However, way-reduction and set-reduction pose a problem: as the required log size increases, the performance overhead introduced would rapidly surpass the performance overhead introduced by our hybrid approach. This is due to the fact that cache lines devoted to store logging information cannot be evicted from the data cache, and hence less effective cache lines can be used by the application. On the other hand, our approach allows any log size and cache lines can be evicted from the data cache to upper level caches. On average, the way-reduction, set-reduction and our approach suffer a slowdown of 0.12%, 0.39% and 0.20%, respectively.

8.3 Area, Power and Delay overheads

We have quantified the area, dynamic ready energy per access and cycle time overhead for the '*LOGGING system*' under the 65nm technology. The area, dynamic read energy and cycle time ratio with respect to the data cache are 1.65%, 5.47% and 15.36%. When comparing the area against the whole core, our technique requires an area overhead of 0.24%. It is important to remark that power and performance penalties are paid only during post-Si validation time. Once a processor has been verified, the logging feature will be deactivated.

9 Related Work

To our knowledge, few works have attempted to increase the efficiency of diagnosis in microprocessors. Bower et. al [4] proposed a pure hardware mechanism to diagnose and repair hard faults for ALUs and buffer structures. To do so, it relies on a global error detection mechanism (DIVA [3]) and small saturating error counters for every deconfigurable unit present in the processor.

On the other hand, Li [14] uses a software-based diagnosis mechanism. The scheme relies on error detection mechanisms based on software-symptoms to flag errors (hence, not timely). It is also based on a state checkpointing mechanism to roll back the faulty core to a clean state. Upon rollback, a detailed log is generated to record the execution trace that activated the fault. Then, a golden trace is generated on a fault-free core. Both traces are compared to diagnose errors. However, the scheme does not allow reproducibility and little information is given on how the traces can be logged.

IFRA [19] is a similar scheme to ours. It records the microarchitectural information in distributed small on-core hardware circular buffers. The small size restricts the achievable diagnosis coverage, because the number of loggable events is fixed a priori. Also, since IFRA relies on a set of error detection mechanisms with big detection latencies, the achievable diagnosis coverage for a given log size is lower than if it used timely error detection techniques.

Related to multiprocessor memory validation, DACOTA [9] reconfigures a portion of the cache to log memory accesses. It partitions statically the cache and it does not rely on a timely error detection mechanism, but on periodic execution-diagnosis phases (enabled by a checkpointing mechanism) that introduce big performance overheads. DACOTA is able to detect errors by finding cycles among memory accesses, a similar idea used in previous works [6, 7]. These validation techniques construct and send logs to a checker core. The special hardware checker runs concurrently and analyzes the traces. It is important to note that these works target multiprocessor correctness, not uniprocessor correctness.

10 Conclusions

We have presented a novel solution to diagnose failures in the LSQ during post-Si validation. Our post-Si validation technique incorporates a low-cost logging mechanism that observes selected system activity during normal program execution. With small changes in the OS, the log can be stored in one page of the memory space of the application being run. This way, we can increase the coverage by storing long logs without adding big area overheads. Upon error detection, the log is dumped from the memory for automated diagnosis. Our results show that, on average, we can achieve a high-precision diagnosis coverage of 87.79% with just a 0.24% area overhead with respect to the core. Moreover, the performance slowdown introduced during post-silicon validation (due to logging purposes) is around 0.20%, on average.

11 Acknowledgements

This work has been partially supported by the Spanish Ministry of Education and Innovation under grants TIN2007-61763 and TIN2010-18368. Additionally, this work has also been supported by the Generalitat de Catalunya under grant 2009GR1250.

References

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *In the Proceedings of the 43rd annual Design Automation Conference (DAC '06)*, pages 7–12, New York, NY, USA, 2006. ACM.

[2] E. Anis and N. Nicolici. On using lossless compression of debug data in embedded logic analysis. In *In Proceedings of the IEEE International Test Conference (ITC'07)*, 2007.

[3] T. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of International Symposium on Microarchitecture (MICRO-32)*, 1999.

[4] F. Bower, D. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the International Symposium on Microarchitecture (MICRO-38)*, 2005.

[5] J. Carretero, X. Vera, P. Chaparro, and J. Abella. On-line failure detection in memory order buffers. In *Proceedings of the IEEE International Test Conference (ITC'08)*, 2008.

[6] K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'08)*, 2008.

[7] Y. Chen, Y. Lu, W. Hu, T. Chen, H. Shen, P. Wang, and H. Pan. Fast complete memory consistency verification. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'09)*, 2009.

[8] I. Corporation. Intel® 64 and IA-32 architectures software developer's manual (volume 3a). pages 623–674, 2010.

[9] A. DeOrio, I. Wagner, and V. Bertacco. DACOTA: Post-silicon validation of the memory subsystem in multi-core designs. In *In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA-15)*, 2009.

[10] E. Eichelberger and T. Williams. A logic design structure for lsi testability. In *In the Proceedings of the 14th Design Automation Conference (DAC '77)*, pages 462–468, Piscataway, NJ, USA, 1977. IEEE Press.

[11] A. Hopkins and K. McDonald-Maier. Debug support strategy for systems-on-chips with multiple processor cores. *IEEE Transactions on Computers*, 55(2):174–184, 2006.

[12] IEEE JTAG 1149.1-2001 Std. Ieee standard test access port and boundary-scan architecture. *IEEE Computer Society*, 2001.

[13] D. Josephson. The good, the bad, and the ugly of silicon debug. In *In the Proceedings of the 43rd Annual Design Automation Conference (DAC '06)*, 2006.

[14] M. Li, P. Ramach, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *In Proceedings of the 38th International Conference on Dependable Systems and Networks (DSN)*, 2008.

[15] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *In ASPLOS XIII*. ACM, 2008.

[16] R. Livengood and D. Medeiros. Design for (physical) debug for silicon microsurgery and probing of flip-chip packaged integrated circuits. In *In the Proceedings of the 1999 IEEE International Test Conference (ITC '99)*, 1999.

[17] A. Meixner and D. Sorin. Dynamic verification of sequential consistency. In *In the Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pages 482–493, Washington, DC, USA, 2005. IEEE Computer Society.

[18] A. Meixner and D. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *In the Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*. IEEE Computer Society, 2006.

[19] S.-B. Park and S. Mitra. IFRA: Instruction footprint recording and analysis for post-silicon bug localization of processors. In *Proceedings of the Design Automation Conference (DAC'08)*, June 2008.

[20] P. Patra. On the cusp of a validation wall. *IEEE Design & Test of Computers*, 24(2):193–196, March-April 2007.

[21] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas. Patching processor design errors with programmable hardware. *IEEE Micro*, 27(1):12–25, 2007.

[22] Y. Siva. Addressing post silicon validation challenge: Leverage validation & test synergy. In *In Proceedings of the IEEE International Test Conference*, 2006.

[23] I. Wagner and V. Bertacco. REVERSI: Post-silicon validation system for modern microprocessors. In *In Proceedings of the IEEE International Conference on Computer Design (ICCD'08)*, 2008.

[24] I. Wagner, V. Bertacco, and T. Austin. Shielding against design flaws with field repairable control logic. In *In the Proceedings of the 43rd annual Design Automation Conference (DAC '06)*, 2006.

[25] O. Wechsler. Inside Intel®Core™microarchitecture: Setting new standards for energy-efficient performance. *Intel White Paper*, 2006.