Post-Silicon Bug Localization in Processors Using Instruction Footprint Recording and Analysis (IFRA)

Sung-Boem Park, Ted Hong, and Subhasish Mitra, *Senior Member, IEEE*

***Abstract*—Instruction Footprint Recording and Analysis (IFRA) overcomes challenges associated with an expensive step in post-silicon validation of processors—pinpointing the bug location and the instruction sequence that exposes the bug from a system failure. On-chip recorders collect instruction footprints (information about flows of instructions and what the instructions did as they passed through various design blocks) during the normal operation of the processor in a post-silicon system validation setup. Upon system failure, the recorded information is scanned out and analyzed offline for bug localization. Special self-consistency-based program analysis techniques, together with the test program binary of the application executed during post-silicon validation, are used for this purpose. Major benefits of using IFRA over traditional techniques for post-silicon bug localization are as follows: 1) it does not require full system-level reproduction of bugs, and 2) it does not require full system-level simulation. Simulation results on a complex superscalar processor demonstrate that IFRA is effective in accurately localizing electrical bugs with very little impact on overall chip area.**

***Index Terms*—Circuit marginality, design-for-debug, electrical bug, post-silicon validation, silicon debug.**

I. Introduction

POST-SILICON validation involves operating one or more manufactured chips in actual application environments to validate correct behaviors across specified operating conditions. According to recent industry reports, post-silicon validation is becoming significantly expensive. Intel reported a headcount ratio of 3:1 for design versus post-silicon validation [26]. According to [1], post-silicon validation may consume 35% of average chip-development time. Yerramilli [36] observes that the proportion of total design resources spent in post-silicon validation has been rising over the past years.

There are two types of bugs [26] that design and validation engineers worry about.

1) *Electrical bugs* caused by interactions between the design and physical effects, such as process variations, signal integrity, crosstalk and power-supply noise, temperature effects, etc. Such bugs generally manifest themselves only under certain operating conditions (temperature, voltage, and frequency). Examples include setup and hold-time problems, synchronization problems, noise, and circuit marginalities.
2) *Functional bugs*, also called *logic bugs*, caused by design errors. While most functional bugs are caught during pre-silicon verification, a small percentage of them are only exposed during post-silicon validation due to increasing design complexity and design-schedule constraints.

Post-silicon validation involves four major steps.

1) Detecting a problem by running a test program, such as end-user applications or functional tests, until a system failure occurs (e.g., system crash or unexpected exceptions).
2) Localizing the problem to a small region from the system failure, e.g., a bug in an adder inside an arithmetic logic unit (ALU) of a complex processor. The stimulus that exposes the bug, e.g., the particular ten lines of code from some application, is also important.
3) Identifying the root cause of the problem. For example, an electrical bug may be caused by a setup-time problem on a circuit path resulting in an error at the adder output for a certain input sequence.
4) Fixing or bypassing the problem by patching [8], [13], [29], [33], circuit editing [19], or, as a last resort, respinning.

As pointed out in [18], the second step dominates post-silicon validation effort and costs. Two major factors that contribute to the high cost of traditional post-silicon bug-localization approaches (details in Section V) are as follows.

1) *Failure reproduction* involves returning the hardware to an error-free state and reexecuting the failure-causing stimulus (e.g., instruction sequences, interrupts, and operating conditions for a processor) to reproduce the same failure. Unfortunately, many electrical bugs are very hard to reproduce (often referred to as Heisenbugs [14]). The difficulty of bug reproduction is exacerbated by the presence of asynchronous I/Os and multiple-clock domains. Techniques to make failures reproducible [15], [28], [32] are often intrusive to system operation and may not expose bugs.
2) Register transfer level (RTL) system simulation for obtaining golden responses is several orders of magnitude slower than silicon speed [10]. In addition, expensive external logic analyzers are required to record all signal values that enter and exit the chip through external pins [32].

Manuscript received December 13, 2008; revised March 18, 2009 and June 18, 2009. Current version published September 18, 2009. This paper was presented in part in [25]. This work was supported in part by the Semiconductor Research Corporation and in part by the National Science Foundation. The work of S.-B. Park was supported in part by a Samsung Scholarship, formerly the Samsung Lee Kun Hee Scholarship Foundation. The work of T. Hong was supported by a John and Kate Wakerly Stanford Graduate Fellowship. This paper was recommended by Associate Editor G. E. Martin.

S.-B. Park and T. Hong are with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA (e-mail: sbpark84@stanford.edu; tedhong@stanford.edu).

S. Mitra is with the Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305 USA (e-mail: subh@stanford.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2030595

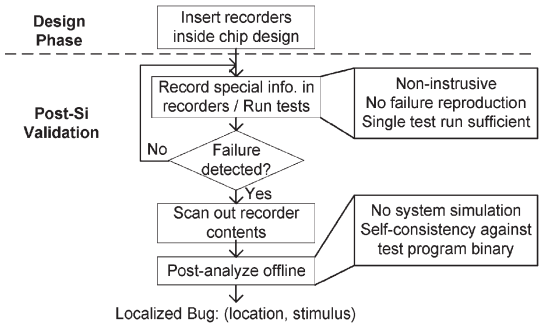


Fig. 1. Post-silicon validation flow using IFRA.

Due to the earlier factors, a functional bug typically takes hours to days to be localized, while electrical bugs require days to weeks and more expensive equipment [17].

This paper targets localization of electrical bugs in processors using a technique called Instruction Footprint Recording and Analysis (*IFRA*). Since there is little consensus about models of functional bugs [16], even though IFRA may be applicable, we did not investigate its applicability to functional bugs. Fig. 1 shows an IFRA-based post-silicon bug-localization flow. During chip design, a processor is augmented with low-cost hardware recorders (Section II-A) for recording instruction footprints. *Instruction footprints* are compact pieces of information describing the flows of instructions (i.e., where each instruction was at various points of time) and what the instructions did as they passed through various microarchitectural blocks of the processor. During post-silicon bug detection, instruction footprints are recorded in each recorder, concurrently with system operation, to capture the last few thousand cycles of history before a failure appears.

Upon detection of a system failure, the recorded footprints are scanned out through a boundary-scan joint test action group interface. IFRA uses special techniques to ensure that a single test run is sufficient in capturing all the necessary information. Hence, there is no need to reproduce the failure for localization.

Scanned-out footprints, together with the test program binaries executed during post-silicon bug detection, are postprocessed offline using special analysis techniques (Section III). These techniques identify the *bug location–time pair*: the location (e.g., the instruction queue control, scheduler, forwarding path, decoders) and the cycle in which an electrical bug had caused an error in a flip-flop in the design. The instruction sequence that exposes the bug (i.e., the *bug exposing stimulus*) is then derived from the bug location–time pair. The analysis techniques do not require any system-level simulation because they rely on checking for self-consistencies in the footprints with respect to the test program binaries.

Once a bug is localized using IFRA, existing circuit-level debug techniques [7], [18] can then quickly identify root causes of bugs. Hence, IFRA can enable significant gains in productivity, cost, and time-to-market. One method of circuit-level debugging is to derive thousands of structural test patterns from the bug exposing stimulus and apply them to the microarchitectural blocks in close vicinity to the location at which IFRA identified the bug (the localized block), while sweeping over voltage, frequency, and temperature ranges. Another method is to run the exposing stimulus while directing observation (e.g., trace buffers [1]) and control mechanisms (e.g., clock manipulation [18]) to the localized block or its neighboring microarchitectural block(s).

In this paper, we demonstrate the effectiveness of IFRA for a DEC Alpha 21264-like superscalar processor model because its architectural simulator [5] and RTL model [34] are publicly available. This processor contains aggressive performance-enhancing microarchitectural features (e.g., speculative, multiway, and out-of-order execution) present in many commercial high-performance processors [30]. Such features significantly complicate post-silicon validation, yet the structured architecture enables opportunities for efficient bug localization using IFRA. For simpler in-order processors (e.g., ARMv6, Intel Atom, SUN Niagara cores), the entire IFRA-based analysis can be significantly simplified.

Extensive IFRA simulations demonstrate the following.

1) For 75% of injected electrical bugs, IFRA pinpointed their exact bug location–time pair—composed of the location (1 out of 200 microarchitectural blocks) and the cycle (1 out of over 1000 cycles). For 21% of injected bugs, IFRA correctly identified their location–time pairs together with five other candidates (out of over 200 000 possible pairs) on average. IFRA completely missed correct location–time pairs for only 4% of injected bugs (i.e., either the location or the time or both were incorrect).
2) The earlier results were obtained without relying on system-level simulation and failure reproduction.
3) IFRA hardware introduces a very small area impact of 1% (dominated by 60 KB of distributed on-chip storage for storing instruction footprints). The area impact may be reduced by using part of an already-existing data cache [3] or by using trace buffers [1], if one exists.

Section II describes hardware support required for IFRA. Section III describes offline program analysis techniques performed on the scanned-out information. Section IV presents simulation results, followed by an overview of related work in Section V. The conclusions are presented in Section VI.

II. IFRA Hardware Support

We use an Alpha 21264-like four-way superscalar processor model [12] to explain the IFRA recording infrastructure. The shaded parts in Fig. 2 show three hardware components of IFRA.

1) A set of distributed instruction footprint recorders (denoted by "R" in Fig. 2) with dedicated storage. Each recorder is associated with a pipeline stage and collects instruction footprints corresponding to the associated stage.
2) An ID-assignment unit for assigning and appending an ID to each instruction that enters the processor.
3) A post-trigger generator for pausing or stopping recording.

When an instruction, with an ID appended, flows through a pipeline stage, it generates an instruction footprint corresponding to that pipeline stage which is stored in the recorder associated with that pipeline stage. An instruction footprint consists of the following features.

1) The ID corresponding to the instruction.

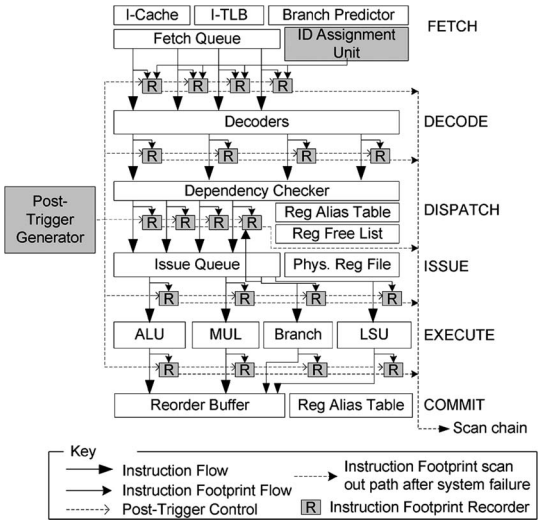


Fig. 2. Superscalar processor augmented with recording infrastructure. The figure is not to scale.

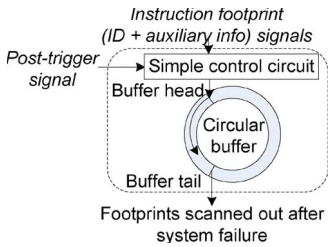


Fig. 3. Internal structure of a footprint recorder. Instruction footprints are recorded during test run and are scanned out after system crash.

2) *Auxiliary information* corresponding to the pipeline stage that tells us what the instruction did in the microarchitectural blocks contained in that pipeline stage.

Even if a processor were bug-free, there may be bugs inside the IFRA hardware. Electrical bugs affecting only IFRA hardware structures may result in the following conditions: 1) false post-trigger activation, or 2) errors in the recorded data but not in the pipeline.

Because the recording operation and the post-trigger generation are performed by independent hardware, errors affecting only one of the two do not cause false positives (i.e., a false indication of a bug in a bug-free processor). For the first case, the scanned-out data will indicate that the post-trigger was not supposed to activate. For the second case, since erroneous recorded data alone can never generate a post-trigger, the recorded data will be overwritten unnoticed.

A. Instruction Footprint Recorder

As shown in Fig. 3, each recorder consists of a circular buffer and simple control logic. Each pair of instruction ID and auxiliary information enters and fills up the circular buffer. The control circuitry is responsible for the following actions:

1) compacting idle cycles;
2) controlling circular buffer operations;
3) starting/resuming and stopping/pausing recording according to the post-trigger signal;
4) serializing buffer contents when they are scanned out.

TABLE I
AUXILIARY INFORMATION FOR EACH PIPELINE STAGE

Pipeline stage	Auxiliary information		Number of Recorders	Entries per Recorder
	Description	Bits per entry		
Fetch	Program counter	32	4	1,024
Decode	Decoded bits	4	4	1,024
Dispatch	2-bit residue of register name	6	4	1,024
Issue	3-bit residue of operands	6	4	1,024
ALU, MUL	3-bit residue of result	3	4	1,024
Branch	None	0	2	1,024
LSU	3-bit residue of result; memory address	35	2	1,024
Commit	Fatal exceptions	4	1	1
Total storage required for all recorders: Each entry contains an additional 8-bit instruction ID (explained later).			60 Kbytes	

Table I shows the auxiliary information collected in each pipeline stage. Decoded bits corresponding to an instruction, collected at the decode stage, tell us which functional unit the instruction is going to use (2 b), whether it uses a destination register (1 b) and/or a second operand register (1 b). The 2- and 3-b residues are obtained by performing mod-3 and mod-7 operations on the original values, respectively. The commit-stage recorder, rather than having a circular buffer, has one register that records the ID of the youngest committed instruction along with any fatal exception (Section II-C) it caused.

Synthesis results (using Synopsys Design Compiler with a TSMC 0.13μ library) show that the area impact of the IFRA hardware infrastructure is 1% on the Illinois Verilog Model [34], assuming a 2-MB on-chip cache, which is typical for current desktop/server processors. This overhead is largely dominated by the circular buffers present in the recorders. Wires connecting the recorders (shown in Fig. 2) operate at slow speed, and a large portion of this routing reuses existing on-chip scan chains that are present for manufacturing testing.

B. ID-Assignment Unit

A processor processes trillions of instructions during a system run, and there needs to be a way of distinguishing each of these instructions. It is difficult to use timestamps for instruction IDs for processors with multiple-clock domains with or without dynamic voltage and frequency scaling. Assigning consecutive numbers in a circular fashion (e.g., mod-64 for a processor with a maximum of 64 instructions in flight) does not work for complex superscalar processors supporting out-of-order execution and pipeline flushes. For processors with out-of-order execution, the program-counter (PC) value cannot be used as an instruction ID. Programs with loops may produce multiple instances of the same instruction with the same PC value. These multiple instances may execute out of program order.

Our special ID-assignment scheme, described as follows, works under all the earlier circumstances (proof given in Appendix I). For a processor with at most n instructions in flight, each instruction ID is $\log_2 4n$ bits wide. Instruction IDs are assigned to individual instructions as they exit the

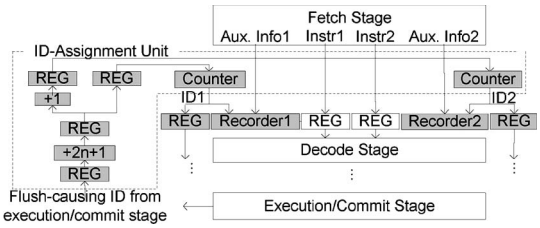


Fig. 4. ID-assignment unit for a two-way processor (enclosed by dashed box). Shaded parts indicate newly added hardware over existing processor hardware.

TABLE II
FAILURE SCENARIOS AND POST-TRIGGERS

Failure Scenario	Post-triggers	
	Soft	Hard
Array error	-	Parity check
Arith. error	-	Residue check
Fatal exceptions	-	In-built exceptions
Deadlock	Short (2 mem loads) instruction retirement gap	Long (2 secs) instruction retirement gap [22]
Segfault	TLB-miss + TLB-refill (for both data and intr. TLBs)	Segfault from OS; Address equals 0

fetch stage and enter the decode stage (Fig. 4). Since multiple instructions may exit the fetch stage in parallel at any given clock cycle, multiple IDs are assigned in parallel.

Instruction ID Assignment Scheme used by IFRA:

Rule 1: The first p instructions that exit the fetch stage in parallel are assigned IDs, $0, 1, 2, \ldots, p-1$.

Rule 2: Let ID X be the last ID that was assigned. If there are q instructions that exit the fetch stage in the current cycle in parallel, then q IDs, $X+1 \pmod{4n}$, $X+2 \pmod{4n}, \ldots, X+q \pmod{4n}$ are assigned to the q instructions.

Rule 3: If an instruction with ID Y causes a pipeline flush, then the ID X in Rule 2 is overwritten with the value of $Y+2n \pmod{4n}$. As a result, ID of $Y+2n+1 \pmod{4n}$ is assigned to the first instruction that is fetched after the flush. The flush is caused either by a mispredicted branch or an exception.

There exists sufficient time between instantiations of Rules 3 and 2 to overwrite the ID X with $Y+2n \pmod{4n}$ before newly fetched instructions arrive at the ID assignment unit. This is because it takes several cycles (e.g., Alpha 21264 has three pipeline stages within the fetch stage) for the instructions to propagate from the beginning to the end of the fetch stage.

C. Post-Trigger Generators

In order to ensure that the entire error-to-failure history is captured using reasonably sized recorders, we use early failure-detection mechanisms, *post-triggers,* for the failure scenarios listed in Table II.

We assume the presence of parity bits for arrays (e.g., register file, reorder buffer (ROB), register-free list, scheduler, and various queues) and residue codes for arithmetic units in ALUs and address calculators. Such parity bits and residue codes exist in several commercial processors [2], [27].

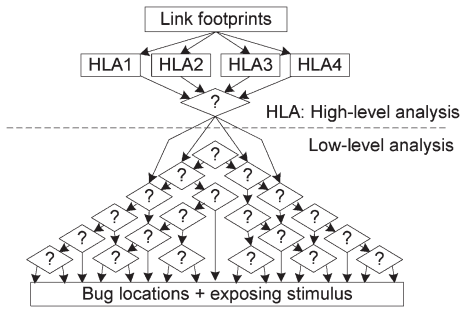


Fig. 5. Postanalysis summary. The low-level analysis decision diagram is for illustrative purposes only.

Fatal exceptions, such as unimplemented-instruction exceptions, arithmetic exceptions, and alignment exceptions, along with benign exceptions such as TLB misses, are already present in most processors.

Unlike the first three failure scenarios listed in Table II, the last two failure scenarios may be detected several millions of cycles after an error occurs. In order to prevent the history recorded in the recorders from being overwritten during this time, we introduce the notion of soft and hard post-triggers.

A *hard post-trigger* fires when there is an evident sign of failure and causes the recorders and processor operation to terminate. A *soft post-trigger* fires when there is an early symptom of a possible failure. It causes the recorders to pause but allows the processor to keep running. If a hard post-trigger for the failure corresponding to the symptom occurs within a prespecified amount of time, the processor terminates. If it does not fire within the specified time, recording resumes under the assumption that the symptom was false.

Segmentation faults (or segfaults) require OS handling and, hence, may take several millions of cycles to resolve. Null-pointer dereferences are detected by adding simple hardware to detect whether the memory address equals zero in the load/store unit. For other illegal memory accesses, a TLB-miss signal is used as a soft post-trigger. If a segfault is not declared by the OS while servicing the TLB miss, the recording is resumed on TLB refill. On the other hand, if a segfault is returned, then a hard post-trigger is activated. The pause in the recording may create a period of time that acts as a blind spot during post-silicon validation. In order to cover such blind spots, a separate set of functional tests specifically targeting TLB servicing must be designed to identify bugs that may appear during TLB misses.

Silent data corruption and live-locks are not covered by the current set of post-triggers. Use of a wider variety of post-triggers based on hardware assertions [1], [6], software assertions, and symptoms [34] is a topic of future research.

III. POSTANALYSIS TECHNIQUES

Once recorder contents are scanned out, footprints belonging to same instruction (but in multiple recorders) are identified and linked together using a technique called *footprint linking* (Section III-A). The linked footprints are also mapped to the corresponding instruction in the test program binary using the PC value stored in the fetch-stage recorder.

As shown in Fig. 5, four high-level postanalysis techniques (Section III-B) are run on the linked footprints, followed by a low-level analysis (Section III-C), represented as a decision

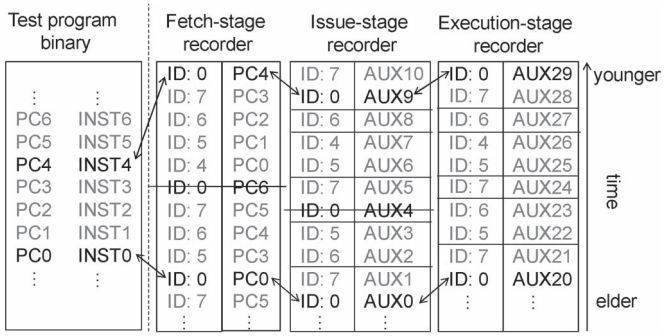


Fig. 6. Instruction footprint linking, with a maximum number of two instructions in flight. Rows are arranged vertically for issue and execute matrices.

diagram, which is used to obtain the final bug location–time pair(s) from which bug exposing stimuli are derived.

The postanalysis techniques rely on the concept of self-consistency, which checks for the existence of contradictory events in collected footprints with respect to the test program binary. While such checks are extensively used in fault-tolerant computing for error detection [4], [20], [24], [31], we use them for bug localization. Such application is possible because, unlike fault-tolerant computing, the checks are performed offline, enabling more complex analysis for localization purposes.

A. Footprint Linking

The footprint-linking algorithm undergoes the following steps (detailed description and reasoning behind each of the steps are presented in Appendix I).

Algorithm III.1: Top-Level Footprint Linking

Step 1: Scan out and format recorder contents for in-order stages (fetch, decode, dispatch, commit) into footprint vectors and out-of-order stages (issue, execute) into footprint matrices.

Step 2: For each vector/matrix, identify and label footprints corresponding to committed and uncommitted instructions.

Step 3: For each committed instruction, link its footprints.

Fig. 6 shows a part of a test program and the contents of three (out of many) recorders right after being formatted. Since we use short-instruction IDs (8-b for Alpha 21264-like processor), we end up having multiple footprints having the same ID in the same recorder and/or multiple recorders. For example, ID 0 appears multiple times in each recorder. After footprint linking, two of the footprints with ID 0 are identified as uncommitted and are crossed off. The rest are linked to indicate that they correspond to a single instruction (for clarity, only two instances of linking are explicitly shown in the figure).

B. High-Level Analysis

We use four high-level analysis techniques: 1) data-dependence analysis; 2) program-control-flow analysis; 3) load–store analysis; and 4) decoding analysis. Each technique is applied separately on the linked footprints and each discovers an inconsistency, if one exists, with an associated time of occurrence. A discovered inconsistency is associated with an entry point into the low-level analysis-decision diagram.

The time is given in terms of an entry in one of the footprint vectors/matrices, referred to as inconsistent footprint, representing a cycle within a particular pipeline stage.

If only one of them identifies an inconsistency, then the corresponding entry point into the decision diagram for low-level analysis is taken. If none of them discover an inconsistency, then there is a default entry point. If many of them identify inconsistencies, then the earliest occurring inconsistency, the one closest to the electrical bug manifestation, dictates the entry point into the decision diagram. Time comparison between inconsistencies is done by observing the fetch-stage footprints to which each inconsistent footprint is linked to. The entry linked to the eldest fetch-stage footprint corresponds to the earliest occurring inconsistency. Performing low-level analysis from each entry point corresponding to each inconsistency separately and then combining the results is a topic of future research.

1) Data-Dependence Analysis: This analysis technique verifies whether the instruction data-dependence order [30] is preserved, i.e., the analysis verifies that every committed instruction X is issued only after all instructions that would produce X's operands finish execution. First, using the serial execution trace obtained from the fetch-stage footprint vector and program binaries, the analysis obtains the footprint entries of the instructions that produce X's operands. Next, the footprint links of these entries (producers of operands of instruction X) are followed to the execute-stage footprint matrix. Next, for each of these links and for each committed footprint in the same or elder (earlier) execute-stage footprint matrix row, that footprint's link is followed to obtain the corresponding issue-stage footprint entry. The youngest (latest) among all these issue-stage footprint entries is the time after which X must issue (i.e., we verify that X's footprint occupies a younger (later) row as compared to all these issue-stage footprint entries). A failed check returns a data-dependence inconsistency, from which the low-level analysis inspects the scheduler; the analysis also returns X's issue-stage footprint entry as the inconsistent footprint (Appendix II, line 20.0).

2) Program-Control-Flow Analysis: This technique verifies program control flow. We perform four consistency checks on the PC sequence of the serial execution trace.

1) The PC increments by $+4$ except in the presence of a control-flow transition instruction (e.g., branch, jump).
2) A PC jump always occurs in the presence of unconditional transition instruction.
3) The PC jumps to the correct target in the presence of direct transition (with target address that does not depend on a register value).
4) The PC jumps to a legal target in the presence of register-indirect transition (with target address that depends on a register value). A *legal target* is an address that is part of the executable address space (determined from the program binary) and whose residue matches that of the recorded register residue.

With a violation in control flow from instruction X to instruction Y, the analysis returns a control-flow inconsistency, from which the low-level analysis inspects the PC register; the analysis also returns X's fetch-stage footprint entry as the inconsistent footprint(Appendix II, line 13.0).

3) Load/Store Analysis: This technique verifies that a stored value to a memory address matches the value that is later loaded from that same address. The check is performed on the load–store sequence obtained from execution-stage recorders. The checks as follows return six types of inconsistencies and are performed upon each value stored to an address a, denoted $w[a]$, and the subsequent (without an intervening store to a) values loaded from a denoted $r[a](1), \ldots, r[a](n)$.

a) Possible inconsistency types returned by checks:

type i)/type ii) The inconsistency is a store/load address error whose inconsistent footprint is the store/load instruction's footprint in the execution-stage footprint matrix.

type iii)/type iv) The inconsistency is a store/load data error whose inconsistent footprint is the store/load instruction's footprint in the execution-stage footprint matrix.

type v)/type vi) The inconsistency is a memory write/read error whose inconsistent footprint is the corresponding store/load instruction's footprint in the execution-stage footprint matrix. Since we only concentrate on pipeline bugs, these two inconsistencies are not passed onto low-level analysis but are only appended onto the list of candidate location–time pairs returned by low-level analysis.

b) Consistency checks:

1) If for all i, j $r[a](i) = r[a](j)$ (i.e., all loads were consistent) and for all i, $w[a] \neq r[a](i)$ and $n > 1$, then the multiple consistent loads suggest that the store address was incorrect. Return type i) and type v) inconsistencies (defined earlier) corresponding to the store $w[a]$.
2) If for all i, j $r[a](i) = r[a](j)$ and for all i, $w[a] \neq r[a](i)$, and $n = 1$, then since there is only one load, a bug in the load cannot be ruled out. Return type i) and type v) inconsistencies corresponding to the store $w[a]$ along with types ii), iv), and vi) inconsistencies corresponding to the load $r[a](1)$.
3) If there exists i, j such that $r[a](i) \neq r[a](j)$ (i.e., not all loads are consistent), then for each load $r[a](k) \neq w[a]$, return corresponding types ii), iv), and vi) inconsistencies.

4) Decoding Analysis: This analysis checks whether instructions are decoded correctly and whether they pass through the correct sequence of modules without disappearing or being modified erroneously in the middle.

The first consistency check is to verify, using the decode-stage bits, that only the recorder associated with the correct functional unit had recorded the instruction (e.g., an ADD instruction does not go into a multiplier unit). A failed check returns the footprint of the incorrectly executed instruction in the execute-stage footprint matrix as the inconsistent footprint.

The next three checks ensure that footprints contained in earlier pipeline stages should be a superset of footprints (both committed and uncommitted) contained in later pipeline stages. For example, a decode-stage footprint vector cannot contain a footprint that is not present in fetch-stage footprint vector. Due to finite-sized buffers, we do not perform this check for all footprints. These checks impose an additional requirement of post-trigger routing: Later pipeline stages must be stopped before or at the same time as earlier pipeline stages. Even if pipeline stages are not in the same clock domain, this can be done by stopping each stage from commit to fetch in sequence.

1) Consistency among footprint vectors: Create pointers $P1$, $P2$, and $P3$ to the youngest footprints of the fetch, decode, and dispatch footprint vectors, respectively. If $P1$ points to an uncommitted footprint with ID X, verify that if P2 does not point to X, then $P3$ does not either. A failure returns an erroneous flush inconsistency whose inconsistent footprint is the next elder flush-causing footprint in the dispatch footprint vector. If $P1$ points to a committed footprint X, verify that both $P2$ and $P3$ point to X. A failure returns an inconsistency whose inconsistent footprint is X's footprint in the footprint vector associated with the pipeline stage before the stage which was missing X's footprint. Continue this check by incrementing the pointers that point to X to the next elder entries of the respective footprint vectors. If a pointer cannot be incremented, stop.
2) Consistency among footprint matrices: Given the issue-stage matrix and execute-stage matrix, we use the fact that issue order and execute order for a given functional unit are the same. Each functional unit is associated with a column in the issue and fetch stage footprint matrices. The prior check is performed between the two columns associated with each functional unit.
3) Consistency along the vector–matrix boundary (dispatch-stage vector and issue-stage matrix). Algorithms similar to that of Algorithm III.1 are used to perform the following:
   a) For each committed instruction in the issue-stage matrix, the algorithm checks that it also appears in the dispatch-stage footprint vector. A failure returns a missing footprint inconsistency whose inconsistent footprint is X's footprint in the issue-stage footprint matrix.
   b) When performing step 6) of Algorithm A.III (identifying uncommitted instructions in the footprint matrix), check that the uncommitted instructions have a one-to-one mapping to uncommitted instructions in the dispatch-stage footprint vector. More than one uncommitted footprint indicates a duplicated uncommitted footprint in the matrix or a missing flush-causing footprint in the matrix. A failure returns an erroneous flush inconsistency whose inconsistent footprint is the footprint of the flush-causing instruction in the dispatch-stage footprint matrix.

C. Low-Level Analysis

The low-level analysis involves back-propagating discovered inconsistencies through hardware locations according to the decision diagram in Appendix II, while updating the bug time from the initial inconsistency. For each back propagation, the time is modified by comparing the new location to the prior location. For a change in pipeline stage given an instruction, the footprint links are followed to map the original time (an entry in a pipeline stage's footprint vector/matrix) to a time in the new pipeline stage (corresponding entry in the footprint vector/matrix). For a change in instruction within a pipeline stage,

TABLE III
ERROR INJECTION BITS

Description	Number of bits
PC, next PC	128
Memory Address used by Load/Store	128
Input/Output latch of Array Structures	82
Pointers to Array structures	23
Control states of Array Structures	4
Pipeline Registers	800
Valid Bits	26

since each row in footprint vector/matrix corresponds to a single cycle, the new time is determined by adding/subtracting the number of rows between the instructions.

D. Bug-Exposing Stimulus

The bug-exposing stimulus is derived from the bug location–time pair and the fetch-stage footprint vector. First, by following footprint links to the fetch-stage footprint vector, the time of the bug is mapped to an entry in the fetch-stage footprint vector. This entry partitions the fetch-stage footprint vector into two parts. Elder (earlier) footprints correspond to instructions before the error while younger (later) footprints correspond to instructions after the error leading up to the post-trigger event. The bug-exposing stimulus, the execution trace of instructions leading up to the cycle in which the bug first caused an error in a flip-flop, corresponds to the elder (earlier) set of footprints.

IV. RESULTS

We evaluated IFRA by injecting errors into a microarchitectural simulator [5] augmented with IFRA. We used an Alpha 21264 configuration (four-way pipeline, 64 maximum instructions in-flight, two ALUs, two multipliers, two load/store units), which gave 200 different microarchitectural blocks (excluding array structures and arithmetic units, since errors inside those structures are immediately detected and localized using codes, as discussed in Section II-C). Each block has an average size equivalent of 10-K two-input NAND gates. Seven benchmarks from SPECint2000 (bzip2, gcc, gap, gzip, mcf, parser, and vortex) were chosen as validation test programs as they represent different types of workloads. Each recorder was sized to have 1024 entries.

All bugs were modeled as single bit-flips at flip-flops to target hard-to-repeat electrical bugs. This is an effective model because most electrical bugs eventually manifest themselves as incorrect values arriving at flip-fops for certain input combinations and operating conditions [23].

Errors were injected in one of 1191 flip-flops (Table III). Note that no errors were injected in array structures, since they have built-in parities for error detection. Errors were injected in input/output registers and various control registers controlling the array structures. Pipeline registers in Table III include decoded opcode, register specifiers, immediate data, addresses to arrays, etc. Valid bits indicate whether a given instruction is valid or not in a pipeline register.

Upon error injection, the following scenarios are possible.

1) The error causes no system level effect [34].
2) The error does not cause any post-trigger mechanism to trigger, but the program output is incorrect.

TABLE IV
IFRA BUG-LOCALIZATION SUMMARY

Exactly Localized	75%
Correctly Localized with Candidates	21% min. 2, avg. 6, and max. 34 candidates
Completely Missed	4%

3) Failure manifestation with short error-detection latency, where recorders successfully capture the history from error-to-failure manifestation (including situations where recording is paused upon activation of soft post-triggers).
4) Failure manifestation with long error latency, where 1024-entry recorders fail to capture the history from error to failure (including soft triggers).

Cases 1) and 2) are related to the coverage of validation test programs and post-triggers and are not the focus of this paper. Any error injection run which does not result in the activation of any post-trigger within 100 000 cycles from the point of error injection are included in these categories.

Out of 100 000 error injection runs, 800 of them resulted in cases 3) and 4). Table IV presents results from these two cases. The "*exactly located*" category represents the cases in which IFRA returned a single and correct location–time pair (as defined in Section I). The "*candidate located*" category represents the cases in which IFRA returned multiple location–time pairs (called candidates), and at least one pair was fully correct in both location and in time. The "*completely missed*" category represents the cases where none of the returned pairs were correct. In addition, we pessimistically report all errors that resulted in case 4) as "completely missed."

All error injections were performed after a million cycles from the beginning of the program in order to demonstrate that there is no need to keep track of all the footprints before the appearance of an error. It is clear from Table IV that a large percentage of bugs were uniquely located to correct location–time pair, while very few bugs were completely missed, demonstrating the effectiveness of IFRA.

For "candidate located" cases, Table IV also reports statistics on the number of possible candidates out of a total of 200 000 possible candidate location–time pairs. When IFRA identified multiple candidates, on average, it correctly dismissed 99.8% of the possible bug locations.

V. RELATED WORK

Related work on post-silicon validation can be broadly classified into the following categories: formal methods [11], embedded trace buffers for hardware debugging [1], on-chip program and data tracing [21], clock manipulation [18], scan dump [7], check-pointing with deterministic replay [28], [32], and online assertion checking [1], [6], [9]. Table V provides a qualitative comparison of IFRA versus existing techniques.

Most of the techniques require failure reproduction and system-level simulation. If easy failure reproduction support is present, it will also help IFRA by allowing recorders to record unlimited lengths of history through repeated sampled recording and dumping.

On-chip storage of program and data traces [21], commonly used in embedded processors (e.g., ARM, Motorola's MPC, Infineon's Tricore), have some similarity with IFRA in that they also store program flow of the executed software. If one can

TABLE V
IFRA VERSUS EXISTING TECHNIQUES

Techniques	Formal methods	Trace buffer	Scan methods	Clock manipulation	Program & Data tracing	Checkpoint & replay	Assertion checking	IFRA
Intrusive?	(+)No	Depends	(-) Yes		Depends	(-) Yes	Depends	(+) No
Failure reproduction?	(-) Yes						N/A	(+) No
System-level simulation?	(+)No	(-) Yes					(+)No	(+) No
Area impact?	(-) Yes		(+) No	(-) Yes	(+) No		(-) Yes	(-) 1%
Applicability?	(+)General				(-) Processor		depends	(-) Processor

assume perfect hardware, capturing the signals at the asynchronous interfaces is sufficient to reconstruct all the internal signals using simulation [35]. However, such information is only valid before an error occurs, and no reconstruction is possible beyond the error. Since bug localization requires information from error to failure, the application of the technique to hardware debugging is limited.

Online assertion checking techniques are mostly used for detection and are complementary to IFRA in that such techniques can be efficiently used to generate post-triggers and also for fine-grained bug localization together with the postanalysis techniques supported by IFRA.

VI. CONCLUSION

IFRA targets the problem of post-silicon bug localization in a system setup, which is a major challenge in processor post-silicon design validation. Two major novelties of IFRA are as follows.

1) High-level abstraction for bug localization using low-cost hardware recorders that record semantic information about instruction data and control flows concurrently in a system setup, eliminating the need for failure reproduction.
2) Special techniques, based on self-consistency, to analyze the recorded data for localization after failure detection without full system-level simulation.

However, IFRA has its own limitations, opening up several interesting research directions.

1) The low-level postanalysis decision diagram is created manually based on a specific microarchitecture. Automated generation and construction of the decision diagram for any given processor is yet to be investigated.
2) The localization takes advantage of the structured architecture of processor designs and targets bugs directly related to the core and not the cache logic or interfaces. Application of IFRA still needs to be extended to homogeneous/heterogeneous multicore systems and system-on-chips consisting of nonprocessor designs.
3) IFRA does not currently support simultaneous multithreaded processors [30].
4) Bugs that may only cause performance slowdown but not critical for correct program runs are not targeted.
5) Sensitivity analysis and characterization of the interrelationships between postanalysis techniques, architectural features, error-detection mechanisms, recorder sizes, and bug types are yet to be performed.

APPENDIX I
FOOTPRINT LINKING

First, we state three premises and four assumptions of the underlying superscalar processor implementation. The premises demonstrate the generality of footprint linking, while the assumptions describe traits of our particular implementation that are shared with modern superscalar processors [30].

Premise 1—Pipeline With Out-of-Order Execution: There are four in-order (fetch, decode, dispatch, commit) and two out-of-order pipeline stages (issue, execute), as shown in Fig. 2. Instructions enter the centralized issue queue in-order (a process called dispatch), but exit the issue queue out-of-order (a process called issue). Instructions enter (during dispatch) and exit (during commit) the ROB in order. The maximum number of instructions-in-flight n equals the number of ROB entries in a superscalar processor.

Premise 2—Multiple-Clock Domains: The granularity of individual clock domains can be as fine as a single pipeline stage. Furthermore, for the execution stage, a domain can be made finer to only include a single type of functional unit. For example, if there are three ALUs and two load/store units, IFRA can support two separate domains with one containing all the ALUs and the other containing all the load/store units.

Premise 3—Dynamic Voltage and Frequency Scaling: Individual clock domains can undergo independent dynamic voltage and frequency scaling.

Assumption 1—Speculation Handling: Mispredicted branch instructions are detected by branch units in the execution stage and the corresponding pipeline flushes are initiated only after the branch instruction exits the execution stage.

Assumption 2—D-TLB Miss Handling: D-TLB misses behave similar to branch mispredictions in that they both cause pipeline flushes. However, an instruction with a D-TLB miss causes a pipeline flush only when the instruction reaches both the commit-stage and the head of the ROB (i.e., it is the instruction's turn to commit). For the purpose of footprint linking, we consider them to have committed. In terms of ID assignment, if the instruction causing the D-TLB-miss is assigned ID Y, the first instruction fetched after the resumption of IFRA recording will be assigned ID $Y + 2n + 1 \pmod{4n}$, similar to speculation handling. IFRA recording pauses between Y and $Y + 2n + 1$ because D-TLB misses are one of IFRA's soft-triggers

Assumption 3—External Interrupts and I-TLB Miss Handling: Both external interrupts (an asynchronous signal indicating a need for a program flow change—e.g., process context switch) and I-TLB misses are associated with the instruction at the tail of the ROB at the time of occurrence. After this, the processor stops fetching new instructions and allows the

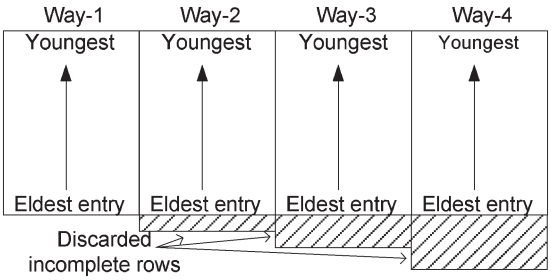


Fig. 7. Aligning four unwrapped circular buffers associated with a four-way pipeline stage. Incomplete rows are discarded.

instructions that are already in the pipeline to commit, before it pauses IFRA recording and invokes the handler for the interrupt or the I-TLB miss. Thus, neither interrupts nor I-TLB misses cause pipeline flushes. IFRA recording is resumed after the handler returns.

Assumption 4—Fatal-Exception Handling: Once an instruction with a fatal exception reaches the head of the ROB, the post-trigger generator halts the processor (Section II-C). Thus, fatal exceptions do not cause pipeline flushes.

Assumption 5—Binaries: All binaries are statically linked so their PCs can be inferred after system failure. Otherwise, we need to record complete instructions in fetch stage recorders, increasing the total storage from 60 to 76 KB.

Although we describe footprint linking based on the ID scheme presented in Section II-A, other shorter ID widths such as $\log_2(2n+2)$ or $\log_2 2n$ bits can be used instead. However, using $\log_2(2n+2)$ bits requires an expensive mod $(2n+2)$ operation, while using $\log_2 2n$ bits requires extra buffers to store flushed instructions.

A. Formatting the Set of Scanned-Out Footprints

After scanning-out footprints from individual recorders, the following five steps format the set of footprints for linking.

1) **Unwrap circular buffers:** The footprints scanned out from each circular buffer are unwrapped so that the youngest entry appears at the top and the eldest entry appears at the bottom (Fig. 7).
2) **Expand idle cycles:** While recording footprints, consecutive idle cycles are compacted to occupy a single entry. The compacted entries are expanded back so that each entry corresponds to a clock cycle.
3) **Align circular buffers:** The unwrapped circular buffers associated with each pipeline stage are collected and juxtaposed so that the youngest entries are aligned (Fig. 7). Due to the second premise earlier, each row in the juxtaposed buffers corresponds to the same clock cycle.
4) **Discard incomplete rows:** For a given pipeline stage, some recorders record longer history than others because they encounter more idle cycles. For simplicity of analysis, incomplete rows are discarded (Fig. 7). The resulting matrix of footprints is referred to as *footprint matrix*.
5) **Convert to footprint vector:** Footprint matrices associated with in-order pipeline stages are converted into *footprint vectors,* where (i,j)th entry of the matrix corresponds to $(4i+j)$th entry of the vector, for a four-way stage. Footprint matrices associated with out-of-order pipeline stages are retained.

B. Distinguishing Footprints With the Identical IDs

Due to out-of-order execution, different recorders may record instruction footprints in different orders. Furthermore, this ordering may differ from *program order*: the order in which instructions in the binary would be executed, if executed sequentially one at a time [30]. Ordering or determining the relative timing information of instructions/footprints across multiple pipeline stages is split into two steps. The first step, identifying whether a footprint corresponds to a committed or uncommitted instruction, is described in Appendix I-C. The second step, described in this section, uses this information to order and link committed instructions in program order.

Algorithm A.1: Linking—Step 3 of Algorithm III.1 Detailed
Given: Footprint vectors and matrices with footprints labeled according to whether they correspond to committed or uncommitted instructions.

Step 1: For each footprint vector and matrix, setup a pointer to the youngest entry. The pointer accesses the matrix in row-major order (left to right in a row and then from youngest to eldest row) so that matrices are treated as vectors in this algorithm.
Step 2: While the fetch-stage footprint vector pointer P does not point to a footprint marked as committed, increment P (i.e., move P downwards). If P cannot be incremented, stop. This is the next youngest instruction in program order (Theorem 1).
Step 3: Let $W =$ ID of the footprint pointed to by P.
Step 4: For each of the other footprint vectors and matrices, increment their pointers until a footprint with ID W is found. If ID W cannot be found in all the footprint vectors and matrices, stop.
Step 5: The order of instructions with the same ID cannot differ from program order (Theorems 1 and 2). Thus, these footprints correspond to a single instruction. Link these footprints.
Step 6: Increment P and go to step 2 (if not possible, stop).

Theorem 1: The relative order in which two committed instructions appear in any footprint vector never differs from their relative program order.

In other words, if instruction X appears before instruction Y in program order, then X will always occupy an elder entry than Y in any footprint vector. This is true because instructions enter the in-order pipeline stages in program order.

Theorem 2: The relative order in which two committed instructions with the same ID appear in any footprint vector or footprint matrix never differs from their relative program order.

In other words, if there are two instructions X and Y that have the same ID, and X appears before Y in program order, then X will always occupy an elder row than Y. The proof follows from Lemmas 2.1–2.5.

Lemma 2.1: If a ROB entry is occupied by an instruction with ID X, the instruction in the kth younger entry either has the ID $X+k \ (mod\ 4n)$ or $X+k+2n \ (mod\ 4n)$, $k \in \mathbb{Z}^+$, i.e., k is a positive integer, at any given time.

Proof: Base case of $k=1$. If X caused a flush, the newly fetched instruction Y after the flush would have an ID of $X+2n+1 \pmod{4n}$, as described in Section II-A; since

everything between X and Y is flushed, Y would be in the next entry. If X did not cause a pipeline flush, then the next entry would be assigned ID $X + 1 \pmod{4n}$ and would correspond to the next instruction in program order.

Inductive case: Suppose that Lemma 2.1 is true for the kth younger entry; thus, the kth younger entry contains an instruction with ID $X + k + 2nm \pmod{4n}$, $m \in \{0, 1\}$. Hence, the $(k+1)$th younger entry has ID $X + k + 2nm + 1 + 2nm' \pmod{4n} = X + k + 1 + 2nm'' \pmod{4n}$, $m', m'' \in \{0, 1\}$. It follows that Lemma 2.1 holds for all $k \in \mathbb{Z}^+$. ■

Lemma 2.2: If a ROB entry is occupied by an instruction with ID X, the instruction in the kth younger entry has an ID distinct from X, at any given time, where $1 <= k < n$.

Proof: From Lemma 2.1, the kth younger entry has an ID $X + k + 2nm \pmod{4n}$, $m \in \{0, 1\}$. To prove $X + k + 2nm \pmod{4n} \neq X \pmod{4n}$, it is sufficient to prove that $X + k + 2nm \neq X \pmod{2n}$ or $k \neq 0 \pmod{2n}$. ■

Lemma 2.3: All instructions in an n-entry ROB have distinct IDs at any given time.

Lemma 2.3 is a corollary of Lemma 2.2.

Lemma 2.4: The relative issue/execution order of two instructions with the same ID will never differ from the fetch order of the two instructions.

Proof: From Lemma 2.3, no two instructions with the same ID ever coexist in the ROB at any given time. Since only instructions that coexist in the ROB can switch their relative issue/execution orders from their fetched order [30], two instructions with the same ID cannot be issued/executed in an order different from their fetch order. ■

Lemma 2.5: The relative issue/execution order of two committed instructions with the same ID will never differ from the relative program order of the two instructions.

Lemma 2.5 is a corollary of Lemma 2.4.

C. Identification of Uncommitted Instructions

This section details the procedures to identify whether a footprint corresponds to a committed instruction or uncommitted one for in-order and out-of-order pipeline stages, respectively. For each of the cases, two categories of uncommitted instructions are addressed: 1) instructions that were fetched after the last committed instruction, that were not committed due to a hard post-trigger activation, and 2) instructions that were flushed from the pipeline.

1) Uncommitted Instructions in In-Order Pipeline Stages: Given a footprint vector, the following algorithm labels each vector entry with two fields: whether the footprint corresponds to a committed instruction and whether it corresponds to a flush-causing instruction. Note that, since a hard post-trigger does not allow instructions after the youngest committed instruction to commit, the youngest committed instruction is always labeled as flush-causing. The algorithm is shown by an example in Fig. 8. Proofs for Theorems 3–5 that are behind the algorithm are presented afterward.

Algorithm A.2: Labeling of a footprint vector

Step 1: Let $X =$ ID of the youngest committed instruction (Table 1) obtained from the commit-stage recorder.

Step 2: Setup a pointer P to the youngest (top most) entry of the footprint vector.

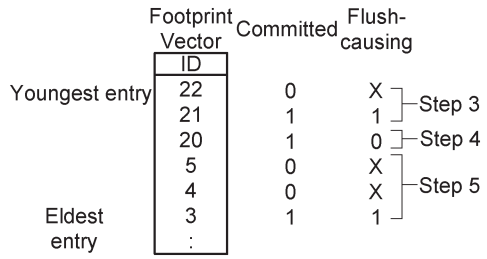


Fig. 8. Example footprint vector with labels. Assumes n to be eight and the youngest committed instruction to have an ID of 21.

Step 3: While X is not encountered, label the footprint pointed to by P as uncommitted (the flush-causing field is labeled a don't care) and increment P to the next elder entry. When X is encountered, label it as committed and flush-causing (Theorem 3). Before incrementing P, if P is pointing at the eldest entry of the vector, stop.

Step 4: While an ID-jump is not observed, label the footprint pointed to by P as committed and nonflush causing and increment P. An ID-jump is observed when the ID pointed to by P minus the ID of the entry pointed to by $P + 1$ (i.e., the next entry elder than P) is not $+1$ (mod 4n). The ID-jump indicates a pipeline flush (Theorem 4). Before incrementing P, if P is pointing at the eldest entry of the vector, stop.

Step 5: Let $W =$ ID of the flush-causing instruction, obtained by subtracting $2n + 1$ from the ID pointed to by P. While W is not encountered, label footprint pointed to by P as uncommitted (the flush-causing field is labeled a don't care), and increment P. When W is encountered, label it as committed and flush-causing (Theorem 5). Before incrementing P, if P is pointing at the eldest entry of the vector, stop

Step 6: Go to Step 4.

Theorem 3: Let X be the ID of the youngest committed instruction. The youngest entry in a footprint vector with ID X belongs to that youngest committed instruction. Furthermore, all younger entries correspond to uncommitted instructions.

Proof: Suppose there is a footprint vector entry with ID X that is younger than the entry corresponding to the youngest committed instruction denoted X'. Since footprint vectors correspond to in-order stage recorders, the only instructions that would have left a footprint after X' would be those fetched after X'. Under the ID-assignment scheme (Section II-A), the instruction fetched next after X' will have ID $S = X + 1 + 2m \pmod{4n}$, $m \in \{0, 1\}$. As X' is the last committed instruction, S does not commit and will occupy the head of the ROB. According to Lemma 2.1, the instructions fetched after S would be assigned IDs $X + 1 + k' + 2nm \pmod{4n}$, $m \in \{0, 1\}$, $1 <= k' < n$. Since all IDs in the ROB are of the form $X + k + 2nm \pmod{4n}$, $1 <= k <= n$ and none are equal to X, there cannot exist a footprint with ID X younger than the footprint corresponding to X', a contradiction. Thus, the youngest entry with ID X corresponds to the youngest committed instruction. Since the instructions fetched after X' are uncommitted, footprints younger than the footprint of X' correspond to uncommitted instructions. ■

Theorem 4: Let X be the ID for a footprint vector entry and W be the ID of the next elder entry. Then, $X - W \neq +1 \pmod{4n}$ iff X's entry corresponds to the newly fetched instruction after a flush.

Proof: (only if) Since footprint vectors correspond to in-order stage recorders, the order in which IDs appear in the vector is the same as their ID assignment order. Thus, if $X - W \neq +1 \pmod{4n}$, then rule 3) of the ID-assignment scheme (Section II-A) must have modified the ID register between W and X. Since modification occurs only after a flush has happened and since W and X are successive entries, X corresponds to the newly fetched instruction after the flush.

(*if*) Let W be the ID of a flush-causing instruction. The newly fetched instruction after W's flush would be assigned ID $X = W + 2n + 1 \pmod{4n}$, as described in Section II-A. Since footprint vectors correspond to in-order stage recorders, only instructions fetched after W and before the newly fetched instruction would have left a footprint. The first of these uncommitted instructions would be assigned ID $S = W + 1 \pmod{4n}$. According to Lemma 2.1, instructions fetched after S would be assigned IDs $W + 1 + k' + 2\text{nm} \pmod{4n}$, $m \in \{0, 1\}$, $1 <= k' < n$; including S, these IDs are of the form $W + k + 2\text{nm}$, $m \in \{0, 1\}$, $1 <= k <= n$. We prove that $W + 2n + 1 - W + k + 2nm \neq 1 \pmod{4n}$ for all k between 1 and n:

$2nm \neq k \pmod{4n}, m \in \{0, 1\}$ as $1 <= k <= n$

$\rightarrow 2n + 1 - k - 2n(1 - m) \neq 1 (\text{mod } 4n)$

$\rightarrow 2n + 1 - k - 2nm' \neq 1 (\text{mod } 4n), m' \in \{0, 1\}$

$\rightarrow W + 2n + 1 - (W + k + 2nm') \neq 1 (\text{mod } 4n).$

In addition, $X = W + 2n + 1 \text{ (mod } 4n)$ so $X - W \neq 1 \pmod{4n}$. Thus, the difference between the X (ID of the footprint entry corresponding to the newly fetched instruction after a flush) and the ID of the next elder entry would not be $+1$. ■

Theorem 5: If an instruction with ID X commits and is the newly fetched instruction after a flush, then the corresponding flush-causing instruction has ID $W = X - 2n - 1 \pmod{4n}$. The flush-causing instruction must have committed, and it appears in the footprint vectors as the youngest ID W elder than X; all entries between W and X correspond to flushed instructions.

Proof: Due to the ID-assignment scheme, a flush-causing instruction and the newly fetched instruction after a flush has a difference in IDs of $2n + 1$; since we assume that an instruction can only flush once [assumptions 1)–3)], $W = X - 2n - 1 \pmod{4n}$. Denote the instructions with ID W and X by W and X, respectively. Now, since X commits, W must also commit. This is true because if W did not commit, it must have been flushed. Thus, because X is younger than W and on the same execution path as W, X must also have been flushed, a contradiction. Finally, because W is the instruction that commits just before X commits, it is the youngest instruction with ID W that is elder than X. By Theorem 1, instructions are recorded in program order; thus, the first instruction with ID W elder than X is X's corresponding flush-causing instruction. ■

2) Uncommitted Instructions in Out-of-Order Pipeline Stages: Given a footprint matrix associated with an out-of-order pipeline stage, the following algorithms label each entry with whether the footprint belongs to a committed instruction. There is no need to label entries as flush-causing or not.

Algorithm A.3: Labeling of footprint matrices entries
Given: Fetch-stage footprint vector labeled with committed and flush-causing bits (Algorithm A.2).

Step 1: Setup a pointer F to the youngest entry in the fetch-stage footprint vector. Create an empty array U.
Step 2: Label all entries in the footprint matrix as uncommitted and unvisited.
Step 3: While F does not point to a committed entry, add the entry's ID to array U and increment F (i.e., select the next elder entry). If F cannot be incremented, stop.
Step 4: Let $W =$ ID of the entry pointed to by F.
Step 5: Pass ID W, and array U to Algorithm A.4 to label the footprint matrix entries corresponding to the committed instruction (W) as committed and uncommitted instructions (U) as uncommitted. If U is not empty, W can be considered the ID of a flush-causing instruction that flushes instructions with IDs in U.
Step 5: Clear array U and Go to Step 3.

Algorithm A.4: Labeling of footprint matrices entries
Given (Alg. A.3): W, the ID of the youngest unvisited committed instruction, and an array of uncommitted IDs U flushed by W.

Step 1: Setup a pointer P to the first entry of the youngest (top most) row of the footprint matrix.
Step 2: While P does not point to an unvisited entry with ID W, increment P, If an entry with ID W is not found, stop.
Step 3: P and F (Alg. A.3) point to entries corresponding to the same instruction (Theorem 2). Label P's entry as committed and visited.

For each $X \in U$

Step 4: Setup a pointer $Y = P$, decrement Y until an entry with ID $= W + 2n + 1 \pmod{4n}$ is found. The row of Y is the younger isolating row (Theorem 6). Increment Y to the first entry of the next row. If the ID is not found, $Y =$ first entry of the youngest row.
Step 5: Setup a pointer $E = P$, increment E until an entry with ID $X - n \pmod{4n}$ or $X - 3n \pmod{4n}$ is found. This is the elder isolating row (Theorem 7). Decrement E until the end of the prior row. If the ID is not found, $E =$ last entry of the eldest row.
Step 6: For all unvisited entries between Y and E (inclusive), if the ID equals X, then label the entry as uncommitted and visited.

Note: Pointers access the matrix in row-major order. Incrementing/decrementing pointers to footprint matrix entries means in row-major order, select the next entry elder/younger.

In order to determine whether the flushed instruction reached and left the stage, we only need to check for the presence of ID X in certain consecutive rows of the footprint matrix. The rows are bounded by a *younger isolating row* at the top and an *elder isolating row* at the bottom. The elder isolating row is always below the row that contains the flushed ID X, if there is any, and always above the row that contains another instance of

ID X that is the youngest among the ID X's dispatched before the flushed ID X. Similarly, the younger isolating row is always above the row that contains the flushed ID X, if there is any, and always below the row that contains another instance of ID X that is the eldest among the ID X's dispatched after the flushed ID X. If one has the ability to find the elder and the younger isolating row for a particular flushed ID, then it is trivial to find out whether the flushed instruction left the considered stage; if there is an ID X in the rows bounded by the isolating rows, then that is the flushed ID X we were looking for; if there is not any, we can conclude that the flushed instruction did not leave the issue stage.

For the rest of this section, let us denote W to be the ID of the identified flush-causing instruction and denote X to be the ID of an instruction that is flushed by ID W. The relationship between X and W is given by $X = W + k \pmod{4n}$, where $1 <= k <= n - 1$ (Lemma 2.1).

Identifying the younger isolating row:

Theorem 6: Suppose there is a flush-causing instruction W with ID W which flushes an instruction X with ID X. Then, the younger isolating row for X is the eldest row younger than W that contains an entry with ID $W + 2n + 1 \pmod{4n}$.

Proof: In the ID-assignment scheme, $W + 2n + 1 \pmod{4n}$ is assigned to the newly fetched instruction after the flush completes. Thus, by definition, ID $W + 2n + 1 \pmod{4n}$ must have entered the recorders strictly after all the previously flushed instructions. There will not be another instruction with the ID because flush-causing instructions can only cause a single flush.

If the ID cannot be found because the recording stopped before fetching a new instruction after the flush, then an imaginary row above the top row acts as the younger isolating row. ■

Identifying the elder isolating row:

Theorem 7: Suppose the algorithm has identified a flush-causing instruction W with ID W that flushes instruction X with ID X. Then, the elder isolating row for X is the youngest row elder than W that contains an entry with ID $X - n \pmod{4n}$ or $X - 3n \pmod{4n}$.

We use Lemmas 7.1–7.5 to prove the theorem. If the ID $X - n \pmod{4n}$ or $X - 3n \pmod{4n}$ cannot be found, an imaginary row below the bottom row acts as the elder isolating row. If the recorders were large enough, the ID would have been found in an elder row.

Lemma 7.1: If an instruction with ID X is occupying a ROB entry, then the instruction in the kth elder entry, $k \in \mathbb{Z}^+$, either has the ID $X - k \pmod{4n}$ or $X - k - 2n \pmod{4n}$.

Proof omitted due to similarity with Lemma 2.1.

Lemma 7.2: If an instruction with ID W commits, then the kth elder committed instruction, $k \in \mathbb{Z}^+$, has either ID $W - k \pmod{4n}$ or $W - k - 2n \pmod{4n}$.

Take the state of an infinite-sized ROB—one that does not remove committed instructions but behaves as a size n ROB in only allowing a maximum of n instructions in-flight—once the instruction with ID W commits. Then, all entries in the ROB elder than W are committed instructions. The kth elder entry represents the kth elder committed instruction and, by Lemma 7.1, either has ID $W - k \pmod{4n}$ or $W - k - 2n \pmod{4n}$.

Lemma 7.3: Before the instruction with ID X enters the ROB of size n, an instruction with ID $X - n \pmod{4n}$ or $X - 3n \pmod{4n}$ must have existed and committed.

Proof: Let W be the identified flush-causing instruction that flushes X; W commits because only flush-causing instructions that commit are identified and passed to the algorithm (see Algorithms A.3 and A.4). Consider the state just before X enters the ROB. W is present in the ROB so $X = W + j \pmod{4n}$ or $W + j + 2n \pmod{4n}$ for some j, $1 <= j <= n - 1$ (Lemma 2.1 and a size-n ROB). By Lemma 7.2, with $k = n - j$, an instruction with ID $W - k = W - n + j = X - n + 2nm \pmod{4n}$ or $W - k - 2n = W - n + j + 2n = X - n + 2nm \pmod{4n}$ exists and commits before W commits, $m \in \{0, 1\}$. Since X can enter the ROB at this time, there must be at most $n-1$ entries in the size-n ROB. By Lemma 7.1, the first through $(n-1)$th elder entries to X have IDs $X - j + 2nm' \pmod{4n}$, where $1 <= j <= n-1$ and $m' \in \{0, 1\}$. None of these IDs are equal to the ID $X - n + 2nm \pmod{4n}$, whose corresponding instruction must have committed before this time. Thus, there exists an instruction with ID $X - n \pmod{4n}$ or $X - 3n \pmod{4n}$ that commits before ID X enters the ROB. ■

Lemma 7.4: The flushed instruction with ID X, if it exists, cannot coexist with any of ID $X - n \pmod{4n}$ and $X - 3n \pmod{4n}$ in the ROB at any given time. The consequence is that ID X will always be in a row above the row that contains either $X - n \pmod{4n}$ or $X - 3n \pmod{4n}$.

Proof: Suppose that $X - n \pmod{4n}$, rather than $X - 3n \pmod{4n}$, is the committed one that is first encountered below the younger isolating row. Then, Lemma 2.2 tells us that the following IDs can be present in the ROB at the same time as $X - n \pmod{4n} : X - n + j \pmod{4n}$ or $X - n + 2n + j \pmod{4n}$, where $1 <= j <= n - 1$. However, none of them equal X. Now, suppose $X - 3n \pmod{4n}$ is the one of the two IDs that is first encountered below the younger isolating row. Lemma 2.2 tells us that the following IDs can be present in the ROB at the same time as $X - 3n \pmod{4n} : X - 3n + j \pmod{4n}$, $X - 3n + 2n + j \pmod{4n}$, where $1 <= j <= n - 1$. Again, none of them equal X. ■

Lemma 7.5: Another instance of ID X that is the youngest among the ID X's dispatched before the flushed ID X does not coexist with any of $X - n \pmod{4n}$ and $X - 3n \pmod{4n}$ in the ROB. Consequence is that the other instance of ID X will always occur in a row below the row containing either $X - n \pmod{4n}$ or $X - 3n \pmod{4n}$.

Proof: Suppose $X - 4n \pmod{4n}$ is in the ROB. Then, Lemma 2.2 tells us that the following IDs can be present in the ROB at the same time as $X - 4n \pmod{4n} : X - 4n + j \pmod{4n}$, $X - 4n + 2n + j \pmod{4n}$, where $1 <= j <= n - 1$. However, none them equal $X - n \pmod{4n}$ or $X - 3n \pmod{4n}$. ■

APPENDIX II
LOW-LEVEL ANALYSIS DECISION DIAGRAM

This section provides a brief description of the decision diagram used for the low-level analysis. The first 11 questions describe entry points into the decision diagram, and the rest describe the decision diagram proper. Note that the decision does not follow a single path (multiple candidate errors are possible); "OR" denotes when such a split should occur.

1) IF array error (→1.0)
2) IF arithmetic error (→2.0)
3) IF alignment exception (→10.0)
4) IF unimplemented instruction exception (→11.0)
5) IF integer overflow exception (→12.0)

6) IF deadlock (→21.0)
7) IF instruction access segfault (→13.0)
8) IF data access segfault (→14.0)
9) IF control flow analysis violation (→13.0)
10) IF data-dependence analysis violation (→20.0)
11) IF load/store analysis violation (→21.0)

1.0 Error in array element OR (→3.0)
2.0 Error in arithmetic unit OR (→3.0)
3.0 Error in exception generation unit
4.0 Error in register value at the output of execution stage
(→6.0) OR
IF using arithmetic unit (→2.0)
IF using load/store unit (→21.0)
IF using complex ALU, then error in complex ALU
IF using branch unit (→2.0)
5.0 Error in speculative register alias table
(→16.0) OR (→15.0) OR (→18.0)
Similar analysis to load-store analysis, but address is architectural register names and data is physical register names.
6.0 Error in register value at the input of execution stage
Displacement selection multiplexor (→7.0)
OR forwarding path (→8.0)
7.0 Displacement selection multiplexor
IF instruction is supposed to take immediate
IF operand residue does not match immediate residue
IF repeated inputs do not match output,
THEN error in multiplexor
ELSE Error in opcode or immediate (→17.0)
ELSE
IF immediate has been obtained from nonimmediate field of the kn instruction
THEN opcode or immediate (→17.0)
ELSE physical register file (→5.0)
8.0 Forwarding path
IF data-dependence analysis violation
THEN muxes + select signals
ELSE (→4.0) OR (→9.0)
9.0 Error in physical register file
(→4.0) OR wrong physical register name from RAT (→5.0) OR
Similar analysis to 5.0 except use register value instead of physical register name and use physical register name instead of architectural register name
10.0 Error in decoder part 1
(→6.0) OR (→3.0) OR
(Size bits flipped between output of decode to input of address generator) OR
IF instruction is unaligned access
(Wrong decode: unaligned decoded to aligned) OR
(Unaligned access bit flipped between output of decode and input of address generator)
11.0 Error in decoder part 2
(→3.0) OR
IF exception at decode stage
Wrong instruction written from icache (parity protection) OR
(→13.0) OR
IF fetched instruction does not match with instructions in binary
THEN error in fetch queue OR alignment&rotate unit
ELSE instruction word flip between fetch queue and input of decode stage OR wrong opcode decode OR wrong instruction written in fetch queue
ELSE IF exception at execution stage
Do the same check as above but the following is in addition:
Bitflip in decoded opcode field from output of decode stage to input of execute stage
12.0 Integer overflow
(→2.0) OR (→3.0) OR (→4.0)
13.0 Error in PC
IF control flow violation case 1
IF instruction went to branch unit
THEN opcode corruption (→17.0)
ELSE faulty nextPC select mux
IF control flow violation case 2
IF instruction went to nonbranch unit
THEN opcode corruption (→17.0)
ELSE faulty nextPC select mux
IF control flow violation case 3, 4
THEN (→4.0)
14.0 Error in address generator
(→6.0) OR (→2.0)
15.0 Error in architectural register name
Wrong decode or wrong decoded bits propagation (→16.0)
16.0 Wrong physical register name from register free list
17.0 Error in decoded bit propagation
IF decoded bits differ with resimulated result THEN error
18.0 Speculation recovery
IF after flush, results of flushed instruction are used rather than results prior to flush THEN incorrectly not initiated recovery
IF latest results are not seen but elder results are seen
THEN incorrectly initiated recovery
19.0 Error in architectural register alias table
Similar analysis to 5.0 but physical register name and architectural register names come from output of ROB
20.0 Error in scheduler
Scheduler array OR
IF ID duplication (from decode analysis)
Incorrectly cleared issued bits in the array OR
Queue pointer flip
IF ID disappearance (from decode analysis)
Incorrectly cleared valid bit in array
OR queue pointer flip
IF deadlocked
IF ID disappearance then valid bit flip from issue until execution
ELSE incorrectly setting valid bit in array
21.0 Error in load/store unit (load/store analysis)

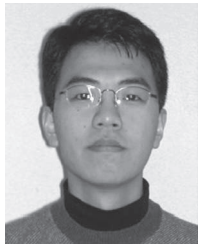
ACKNOWLEDGMENT

The authors would like to thank B. Gottlieb, N. Hakim, D. Josephson, P. Patra, and J. Stinson of Intel Corporation; O. Mutlu of Carnegie Mellon University; and E. Rentschler of AMD for helpful discussions and advice.

REFERENCES

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *Proc. Des. Autom. Conf.*, Jul. 2006, pp. 7–12.
[2] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, "A 1.3-GHz fifth-generation SPARC64 microprocessor," *IEEE J. Solid-State Circuits*, vol. 38, no. 11, pp. 1896–1905, Nov. 2003.

[3] A. Agarwal, R. L. Sites, and M. Horowitz, "ATUM: A new technique for capturing address traces using microcode," in *Proc. Int. Symp. Comput. Archit.*, Jun. 1986, pp. 119–127.
[4] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proc. Int. Symp. Microarchitecture*, Nov. 1999, pp. 196–207.
[5] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 56–67, Feb. 2002.
[6] A. A. Bayazit and S. Malik, "Complementary use of runtime validation and model checking," in *Proc. Int. Conf. Comput.-Aided Des.*, 2005, pp. 1052–1059.
[7] O. Caty, P. Dahlgren, and I. Bayraktaroglu, "Microprocessor silicon debug based on failure propagation tracing," in *Proc. Int. Test Conf.*, Nov. 2005, pp. 293–302.
[8] K. Chang, I. L. Markov, and V. Bertacco, "Automating post-silicon debugging and repair," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 2007, pp. 91–98.
[9] K. Chen, S. Malik, and P. Patra, "Runtime validation of memory ordering using constraint graph checking," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, Feb. 2008, pp. 415–426.
[10] D. W. Clark, "Large-scale hardware simulation: Modeling and verification strategies," in *CMU Computer Science: A 25th Anniversary Commemorative*, R. F. Rashid, Ed. New York: ACM, 1991, pp. 219–234.
[11] F. M. De Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang, "BackSpace: Formal analysis for post-silicon debug," in *Proc. Formal Methods Comput.-Aided Des.*, Nov. 2008, pp. 1–10.
[12] *Alpha 21264 Microprocessor Hardware Reference Manual*, Digital Equipment Corporation, Maynard, MA, Jul. 1999.
[13] M. D. Goddard and D. S. Christie, "Microcode patching apparatus and method," U.S. Patent 5 796 974, Aug. 18, 1998.
[14] J. Gray, "Why do computers stop and what can be done about it," Tandem Comput., Cupertino, CA, Tech. Rep. 85.7, PN 87614, Jun. 1985.
[15] M. W. Heath, W. P. Burleson, and I. G. Harris, "Synchro-tokens: Eliminating nondeterminism to enable chip-level test of globally-asynchronous locally-synchronous SoC's," in *Proc. Des. Autom. Test Eur.*, Feb. 2004, pp. 1532–1546.
[16] *International Technology Roadmap for Semiconductors*. 2007 ed.
[17] D. Josephson, S. Poehlman, and V. Govan, "Debug methodology for the McKinley processor," in *Proc. Int. Test Conf.*, Oct./Nov. 2001, pp. 451–460.
[18] D. Josephson, "The good, the bad, and the ugly of silicon debug," in *Proc. Des. Autom. Conf.*, Jul. 2006, pp. 3–6.
[19] R. H. Livengood and D. Medeiros, "Design for (physical) debug for silicon microsurgery and probing of flip-chip packaged integrated circuits," in *Proc. Int. Test Conf.*, Sep. 1999, pp. 877–882.
[20] D. J. Lu, "Watchdog processors and structural integrity checking," *IEEE Trans. Comput.*, vol. C-31, no. 7, pp. 681–685, Jul. 1982.
[21] C. MacNamee and D. Heffernan, "Emerging on-chip debugging techniques for real-time embedded systems," *Comput. Control Eng. J.*, vol. 11, no. 6, pp. 295–303, Dec. 2000.
[22] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors—A survey," *IEEE Trans. Comput.*, vol. 37, no. 2, pp. 160–174, Feb. 1988.
[23] R. McLaughlin, S. Venkataraman, and C. Lim, "Automated debug of speed path failures using functional tests," in *Proc. VLSI Test Symp.*, May 2009, pp. 91–96.
[24] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.
[25] S. Park and S. Mitra, "IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors," in *Proc. Des. Autom. Conf.*, Jun. 2008, pp. 373–378.
[26] P. Patra, "On the cusp of a validation wall," *IEEE Des. Test Comput.*, vol. 24, no. 2, pp. 193–196, Mar. 2007.
[27] P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones, "Soft-error resilience of the IBM POWER6 processor," *IBM J. Res. Develop.*, vol. 52, no. 3, pp. 275–284, May 2008.
[28] S. R. Sarangi, B. Greskamp, and J. Torrellas, "CADRE: Cycle-accurate deterministic replay for hardware debugging," in *Proc. Int. Conf. Dependable Syst. Netw.*, Jun. 2006, pp. 301–312.
[29] S. R. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas, "Patching processor design errors with programmable hardware," *IEEE Micro*, vol. 27, no. 1, pp. 12–25, Jan./Feb. 2007.
[30] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. New York: McGraw-Hill, 2005.
[31] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems—Design and Evaluation*, 3rd ed. Natick, MA: A.K. Peters, 1998.
[32] I. Silas, I. Frumkin, E. Hazan, E. Mor, and G. Zobin, "System-level validation of the Intel Pentium M processor," *Intel Technol. J.*, vol. 7, no. 2, pp. 37–43, May 2003.
[33] I. Wagner, V. Bertacco, and T. Austin, "Shielding against design flaws with field repairable control logic," in *Proc. Des. Autom. Conf.*, Jul. 2006, pp. 344–347.
[34] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Proc. Int. Conf. Dependable Syst. Netw.*, Jun./Jul. 2004, pp. 61–70.
[35] M. Xu, R. Bodik, and M. D. Hill, "Flight data recorder for enabling full-system multiprocessor deterministic replay," in *Proc. Int. Symp. Comput. Archit.*, May 2003, pp. 122–133.
[36] S. Yerramilli, "Addressing post-silicon validation challenge: Leverage validation & test synergy (invited address)," presented at the IEEE Int. Test Conf., Santa Clara, CA, Oct. 25, 2006.



Sung-Boem Park received the B.Eng. degree in electrical and electronic engineering from Imperial College, London, U.K., in 2005 and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, in 2007, where he is currently working toward the Ph.D. degree, under Prof. S. Mitra, with the Robust Systems Group, Department of Electrical Engineering.

He was a Graduate Intern Researcher with the Microarchitecture Research Laboratory, Intel Corporation, Santa Clara, CA. His research interests include postsilicon validation, computer architecture, and fault-tolerant computing.

Mr. Park was the recipient of the Design Automation Conference Best Paper Award in 2008.

Ted Hong received the B.S. degree in electrical and computer engineering from the University of California, Berkeley, in 2005 and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, in 2007, where he is currently working toward the Ph.D. degree under Prof. S. Mitra in the Robust Systems Group.

His research interests include test, post-silicon validation, robust system design, and computer architecture.

Subhasish Mitra (SM'06) received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA.

He was a Principal Engineer with Intel Corporation. He is currently an Assistant Professor with the Departments of Electrical Engineering and Computer Science, Stanford University, where he leads the Stanford Robust Systems Group. His research interests include robust system design, very large scale integration design, computer-aided design (CAD), validation and test, and design for emerging nanotechnologies. He has coauthored over 100 technical papers and has invented design and test techniques that have seen widespread proliferation in the semiconductor industry. His X-Compact technique for test compression is used by over 50 Intel products and is supported by major CAD tools. His work on imperfection-immune circuits using carbon nanotubes, jointly with his students and collaborators, has been highlighted by the Massachusetts Institute of Technology Technology Review, Semiconductor Research Corporation, EE Times, and several others.

Prof. Mitra was the recipient of the Presidential Early Career Award for Scientists and Engineers (the highest honor bestowed by the U.S. government on early career outstanding scientists and engineers), the National Science Foundation CAREER Award, a Terman Fellowship, the IEEE Circuits and Systems Society Donald O. Pederson Award for the best paper published in the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the Association for Computing Machinery (ACM) Special Interest Group on Design Automation Outstanding New Faculty Award, a Best Paper Award at the IEEE/ACM Design Automation Conference, a Divisional Recognition Award from Intel for a "Breakthrough Soft Error Protection Technology," and the Intel Achievement Award, Intel's highest corporate honor, "for the development and deployment of a breakthrough test compression technology."