# IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors

Sung-Boem Park Subhasish Mitra
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, CA

## ABSTRACT

*The objective of IFRA, Instruction Footprint Recording and Analysis, is to overcome the challenges associated with a very expensive step in post-silicon validation of processors – bug localization in a system setup. IFRA consists of special design and analysis techniques required to bridge a major gap between system-level and circuit-level debug. Special hardware recorders, called Footprint Recording Structures (FRS's), record semantic information about data and control flows of instructions passing through various design blocks of a processor. This information is recorded concurrently during normal operation of a processor in a post-silicon system validation setup. Upon detection of a problem, the recorded information is scanned out and analyzed for bug localization. Special program analysis techniques, together with the binary of the application executed during post-silicon validation, are used for the analysis. IFRA does not require full system-level reproduction of bugs or system-level simulation. Simulation results on a complex super-scalar processor demonstrate that IFRA is effective in accurately localizing bugs with very little impact on overall chip area.*

**Categories and Subject Descriptors**
B 7.2 Design Aids – Verification, B8.1 Reliability, Testing and Fault-Tolerance, B 8.2 Performance Analysis and Design Aids

**General Terms** Reliability, Verification

**Keywords** Validation, verification, debug, design for debug

## 1. INTRODUCTION

*Post-silicon validation* involves operating one or more manufactured chips in actual application environment to validate correct behaviors across specified operating conditions. According to recent industry reports, post-silicon validation is becoming significantly expensive. Intel reported headcount ratio of 3:1 for design vs. post-silicon validation [Patra 07]. According to [Abramovici 06], post-silicon validation may consume 35% of average chip development time. [Yeramilli 06] observes that the increasing use of design resources and equipment costs in post-silicon validation makes it prohibitively expensive in the future.

Post-silicon validation involves three major activities [Josephson 06, Livengood 99, Wagner 06, Sarangi 07]: detecting a problem (e.g., through system crash, segmentation fault or error detection) by applying proper stimulus; localizing and identifying the root cause of the problem; and, fixing or bypassing the problem. Post-silicon bug localization involves identifying the *location-time pair* of a bug; i.e., the hardware design block where the bug is located, and the clock cycle when the bug produces an error. The bug localization step often dominates post-silicon validation efforts [Josephson 06] and is the focus of this paper.

Major factors that contribute to the high cost of current post-silicon bug localization approaches (details in Sec. 6) are:

1. Most of them require failures to be reproducible. *Failure reproduction* involves returning the hardware to an error-free state, and re-executing the failure-causing stimulus (including instruction sequences, interrupts, and operating conditions) to reproduce the same failure. In a system environment, it may be very costly to reproduce a failure, especially for *electrical bugs* [Josephson 06], which manifest themselves only under certain operating conditions. Examples of electrical bugs include setup and hold time problems, synchronization problems, noise, circuit marginalities, etc. This reproducibility problem is exacerbated by the presence of asynchronous I/Os, multiple clock domains, etc. Techniques to make failures reproducible, e.g., [Heath 04, Sarangi 06, Silas 03], may be intrusive to system operation and may not expose important bugs.

2. RTL system simulation for obtaining golden responses is several orders of magnitude slower than silicon speed, and also requires expensive external logic analyzers to record all primary I/O signals in cycle accurate fashion [Silas 03].

The objective of *IFRA,* Instruction Footprint Recording and Analysis, is to overcome these post-silicon bug localization challenges. IFRA consists of:

**1. Low-cost hardware recorders** for recording *instruction footprints* — semantic information describing data and control flows of dynamic instructions as they pass through various parts of a processor. These recorders are different from traditional trace buffers [Anis 07, MacNamee 00]. This recording is done in a non-intrusive manner so that the original execution behavior of a system is not perturbed.

**2. Post-failure analysis techniques** for using recorded footprints and the binary (or the assembly code) of the program executed during post-silicon validation, in order to identify the location-time pair of a bug. The location is provided in terms of microarchitectural blocks, such as control FSMs for various arrays of storage elements, pipeline registers, adders, decoders, etc. These analysis techniques do not require failure reproducibility or RTL simulation.

Once a bug is localized using IFRA, existing circuit-level debug techniques [Caty 05, Josephson 06] can then quickly identify the root cause of bugs, resulting in significant gains in productivity, cost, and Time-to-Market / Time-to-Volume. As a side benefit, the IFRA approach also provides insights into stimuli that expose post-silicon bugs.

We demonstrate the effectiveness of IFRA for an Alpha 21264-like superscalar processor model. This design is sufficiently complex, yet its structured architecture provides opportunities for efficient bug localization. Our primary target is to localize electrical bugs, since they require considerable post-silicon validation efforts [Patra 07]. Extensive IFRA simulations demonstrate:


*DAC 2008*, June 8–13, 2008, Anaheim, California, USA

1. For 75% of injected bugs, IFRA exactly pinpointed their correct location-time pairs. For 21% of injected bugs, IFRA correctly identified their location-time pairs together with 2 to 6 other candidates on an average. IFRA completely missed correct location–time pairs for only 4% of injected bugs.

2. IFRA does not require any system-level simulation or failure reproducibility.

3. IFRA hardware introduces very small area impact of 2% (including 50KBytes of distributed on-chip storage).

Major contributions of this paper are:

1. Introduction of IFRA to bridge a major gap between system-level and circuit-level debug, by allowing very quick localization of bugs to a few design blocks from anomalous system-level behaviors.

2. Low cost methodology for recording control and data flows of dynamic instructions in a compact and non-intrusive manner.

3. Off-line program analysis techniques to analyze the recorded information for bug localization without requiring system-level simulation and failure reproduction.

4. Demonstration of the effectiveness of IFRA for a complex super-scalar processor.

Section 2 presents an overview of IFRA. Section 3 describes the IFRA recording infrastructure. Section 4 describes off-line program analysis techniques performed on the recorded information. Section 5 presents simulation results, followed by an overview of related work in Sec. 6, and conclusions in Sec. 7.

## 2. IFRA OVERVIEW

Figure 2.1 shows a post-silicon debug flow using IFRA. During chip design, a processor is augmented with low-cost hardware recorders for recording instruction footprints (Sec. 3). During post-silicon validation, instruction footprints are recorded concurrently during system operation in a circular fashion to capture the last few thousand cycles of history before a failure manifests. After a failure manifests, the recorded footprints are scanned out through a Boundary-scan JTAG interface [Parker 03]. The footprints, together with the binary of the program executed during post-silicon validation, are then post-processed using special analysis techniques (Sec. 4) for bug localization.

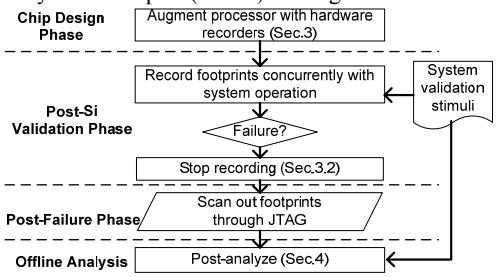


Figure 2.1. Post-silicon debug flow using IFRA.

## 3. IFRA RECORDING INFRASTRUCTURE

We use an Alpha 21264-like superscalar processor model [Alpha 99], to explain the IFRA recording infrastructure. The shaded parts in Fig. 3.1 indicate additional hardware required: *footprint recording structures* (FRS's) – a set of distributed recorders with dedicated storage, and a post-trigger circuit.

### 3.1 Footprint Recording Structure (FRS)

Each FRS is associated with each way of a pipeline stage (e.g. four FRS's are associated with a 4-way fetch stage), and records footprints of instructions when they leave the pipeline stage. An instruction's footprint corresponding to a pipeline stage consists of:

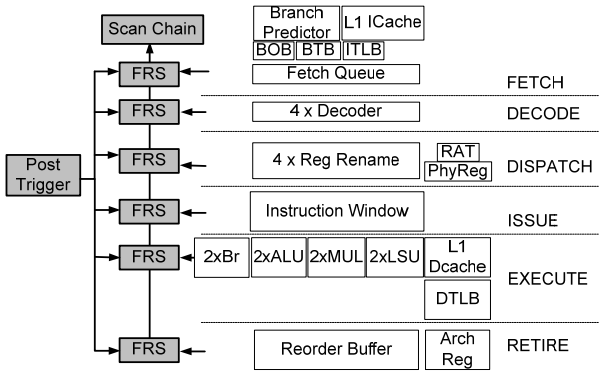


Figure 3.1. Superscalar processor augmented with recording infrastructure.

1. The instruction's identification number, which is used to uniquely identify it.

2. *Auxiliary data*, which tells us what the instruction did in the microarchitectural blocks contained in that pipeline stage.

Figure 3.2 shows the internal structure of an FRS. The main circular buffer acts as storage for instruction footprints and the idle cycle FSM is responsible for maintaining the number of consecutive idle cycle counts.

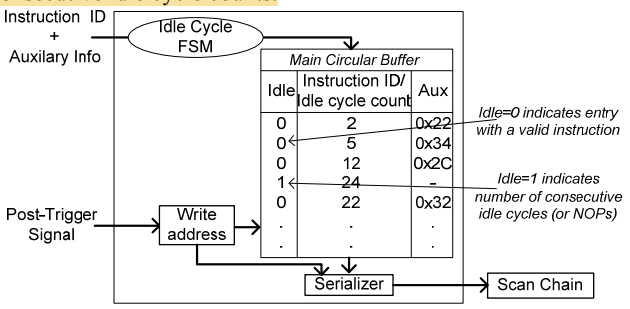


Figure 3.2. FRS internal structure

The auxiliary data records information specific to the pipeline stage where the FRS is inserted. Table 3.1 shows auxiliary data collected in each pipeline stage of an Alpha 21264-like 4-way superscalar processor (detailed configuration in Sec 5). The third column indicates the number of bits of auxiliary information for each FRS, and the last column indicates the total number of FRS's required for each pipeline stage.

Instruction IDs are assigned to individual instructions before they enter the FRS's associated with the fetch stage (Fig 3.3). Since the FRS's for the fetch stage also store PC (Table 3.1), the FRS's entries serve as mapping between instructions in binary (or assembly code) and all instructions in-flight having instruction IDs. For a processor with at most $n$ instructions in-flight, it is sufficient to represent instruction IDs using $\log_2 2n$ bits to uniquely identify every instruction. The assignment is done in a circular fashion. The formal proof is given in [Park 08].

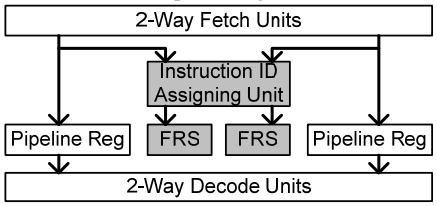


Figure 3.3. Instruction ID assignment for a 2-way processor: Shaded parts are newly added hardware.

Table 3.1. Auxiliary information for each pipeline stage.

| Pipeline stage | Auxiliary information | # bits per FRS | #FRS's |
|---|---|---|---|
| Fetch | PC | 32 | 4 |
| Decode | Part of decoded instruction (detail in Sec. 4.5) | 4 | 4 |
| Dispatch | Parity of renamed registers | 3 | 4 |
| Issue | Parity of operands | 2 | 4 |
| ALU | 3-bit residue of result + ALU error signal | 4 | 2 |
| Branch | 3-bit residue of operand | 3 | 2 |
| MUL | 3-bit residue of result + MUL error signal | 4 | 2 |
| LSU | Address + residue of value | 35 | 2 |
| Commit | Events (misprediction, cache/tlb miss, interrupts) + committed bit | 5 | 4 |

### 3.2 Failure Detection using Post-triggers

In order to ensure that the entire error-to-failure history is captured using reasonably sized FRS's, we assume the presence of early failure detection mechanisms, *post-triggers,* for the failure scenarios listed in Table 3.2. Detection of any one of the failure scenarios terminates the recording.

Table 3.2. Failure scenarios and post-triggers.

| Failure Scenario | Post-triggers | |
|---|---|---|
| | Soft | Hard |
| Array error | - | Parity check |
| Arith. error | - | Residue check |
| Exceptions | - | In-built exceptions |
| Deadlock | Short (2 mem loads) instruction retirement gap | Long (2 secs) instruction retirement gap |
| Segfault | Tlb-miss + Tlb-refill | Segfault from OS; Address equals 0 |

We assume the presence of parity bits for arrays (e.g. register file, reorder buffer, register rename table, register free list, scheduler, and various queues). We also assume the presence of residue codes for arithmetic units in ALUs and address calculators. Such parity bits and residue codes exist in several commercial processors [Ando 03, Leon 06, Sanda 08]. Unimplemented instruction exceptions and arithmetic exceptions are already present in most processors.

Unlike the first three failure scenarios listed in Table 3.2, 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 FRS's to be 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, while a *soft post-trigger* fires when there is an early symptom of possible failure. A hard post-trigger causes the recording and the processor operation to terminate. A soft post-trigger causes the recording in all FRS's to pause, but allows the processor to keep running. If a hard post-trigger for the failure corresponding to the symptom occurs within a pre-specified amount of time, the processor stops. If the hard post-trigger does not fire within the specified time, the recording resumes assuming that the symptom was false.

For deadlocks, a soft post trigger event fires when no instruction retires within the time required to perform two memory loads. The corresponding hard post trigger event is two additional seconds of no retirement.

For the last failure scenario, segmentation fault (or segfault), there is a single hard post-trigger to detect null-pointer dereference, and a pair of soft and hard post-triggers to detect illegal reading/writing into unallocated memory or writing into read-only memory. Null-pointer dereference is detected by adding simple hardware to detect whether the memory address equals zero in the Load/Store unit. For other illegal memory accesses, TLB-miss is used as the soft post-trigger. If segfault is not declared by the OS while servicing the TLB-miss, the recording is resumed on TLB-refill. Since the recording is paused in the event of a soft post-trigger, there may be a period of time that may act as a blind spot during post-silicon validation.

## 4. POST-ANALYSIS TECHNIQUES

After a hard post-trigger fires, all FRS entries, together with the write addresses of the circular buffers, are scanned out through the JTAG interface. The localization analysis begins by combining FRS contents with the binary of the executed program to build a detailed *global control-data flow* (Sec. 4.1). This step enables us to tell where each dynamic instruction was present at each time instance. Next, four high-level post-analysis techniques (Sec. 4.2 - Sec. 4.5) targeting different parts of the processor are run on the global control-data flow. If any one of the techniques identifies inconsistencies in the flow, backtracing is performed to find all the dynamic instructions that the inconsistency depends on. The residue and parity bits, collected by FRS's (shown in Table 3.1), associated with these dynamic instructions are checked for consistency: parities/residues of operands used by consumer instructions must match parities/residues of results produced by producer instructions. This check refines the localization to give the final location-time pair. Note that, the parity/residue checks done for refinement are different from the checks done to detect array errors and arithmetic errors described in Sec 3.2. As a side benefit, the post-analysis also provides several thousand cycles worth of instruction sequences causing the error and leading to failures.

### 4.1 Combining Distributed FRS Contents

The FRS's collect instruction footprints in a distributed manner without any explicit timestamps or synchronization. The following method of creating a global picture from the distributed data works for a processor with single and multiple clock domains, where each domain could undergo dynamic voltage and frequency scaling. The proof can be found in [Park 08].

1. The PCs from the fetch-stage FRS's are mapped to the instructions in the binary.
2. Select the last committed instruction in the commit-stage FRS's and find its instruction ID. For each of the FRS's in other pipeline stages, find the last instruction footprint with the same instruction ID as this last committed instruction. All these footprints correspond to footprints of a single dynamic instruction.
3. Repeat step 2 for all the committed instructions starting from the latest to the earliest committed instructions present in the commit-stage FRS's.
4. Repeat step 2 on the non-committed instructions present after the last committed instruction in the commit-stage FRS's.
5. Discard all other non-committed instructions.

### 4.2 Data Dependency Analysis

Our first post-analysis approach is to verify whether data dependency order is preserved, i.e., if there is a producer-consumer relationship in the serial execution trace, whether the consumer instruction executes after the producer instruction has

produced its result. The analysis is performed on instruction issue sequence (obtained from issue-stage FRS's) and the serial execution trace (derived from the fetch-stage FRS's and commit-stage FRS's). Consider the example in Fig. 4.1. Architectural register names (rather than physical register names) are obtained from the assembly code instruction mapping done in step 1 of combining.

Figure 4.1 illustrates an example of data dependency analysis. Since instruction with ID 0 shown in the serial trace produces a value on R0, while the instruction with ID 3 consumes a value from R0, data dependency exists between those two instructions. Instruction with ID 0 enters the ALU while the instruction with ID 3 enters the multiplier (shown in the execution-stage FRS's). Assume that the two functional units are in different clock domains, and also assume that the ALU has a latency of 3 cycles. Since the two dependent instructions are in different clock domains with a possibility of dynamic frequency scaling, it is not possible to directly check their relative timing. However, we know that the issue-stage FRS's must be in a single clock domain, and thus know that the instructions with ID 3 and ID 5 must be issued at the same time (shown in the issue-stage FRS's). In this case, the distance between 0x03 and 0x00 is only two cycles, which is shorter than the 3-cycle latency of the ALU. This implies the consumer instruction with ID 3 was issued prematurely, before the producer instruction with ID 0 has completed.

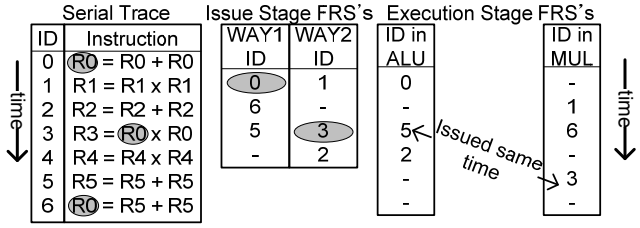


Figure 4.1. Data dependency analysis example.

Any inconsistency identified using this analysis is attributed to an error in one of following microarchitectural blocks of the processor: registers at dispatch, issue, execute stages; issue buffer entries and issue buffer's control; forwarding paths among dispatch, issue, execute stages; register files; and, register renaming of dispatch stage. A detailed analysis method is used for a more refined localization to one of these blocks and also for finding the time of error occurrence. This analysis method can be found in [Park 08].

### 4.3 Program Control Flow Analysis

The next post-analysis technique verifies program control flow. There are four illegal cases that are checked for:

1. Transition in the absence of control flow transition instruction (instruction that changes the PC, e.g., branch, jump).
2. No transition in the presence of unconditional transition instruction (instruction that always changes PC value)
3. Illegal target in the presence of direct transition (with target address that does not depend on a register value).
4. Illegal target in the presence of indirect transition (with target address that depends on a register value).

Figure 4.2 shows an example serial execution trace (derived from FRS data from fetch and commit stage) illustrating all four illegal cases. The first two cases can be checked by checking PC. The third case can be easily checked since the instruction itself contains all the necessary information to compute the target address. The fourth case is checked by checking whether the address addition has been done correctly using residue arithmetic.

A violation in the program control flow can be attributed to an error in one of the following microarchitectural blocks: address calculator in execution stage; all pipeline registers between fetch and execution stages; forwarding path between execution and fetch stage; speculation recovery; and, register renaming. A detailed analysis method for refined localization to one of these blocks can be found in [Park 08].

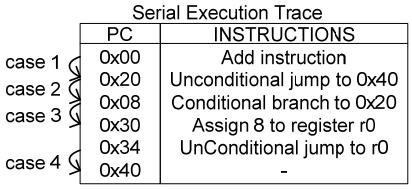


Figure 4.2. Four illegal cases of control flow transitions.

### 4.4 Load/Store Analysis

The third post-analysis technique involves verifying that a stored value to a memory address matches the value that is later loaded from that same address. In the absence of DMA activity, which may modify the memory content, any mismatch indicates a bug in the load/store unit or memory system (cache, memory controller, memory, etc) external to the processor core. In order to check for such mismatches, for each load/store instruction, we record memory addresses and residue of memory contents in Load/Store-unit FRS's. A detailed localization approach can be found in [Park08]. Memory addresses affected by DMA activities may be factored out during post-analysis by recording the instructions sent to DMA engines using external logic analyzer.

### 4.5 Decoding Analysis

This technique checks whether all committed instructions are decoded correctly and whether they pass through the correct sequence of modules without disappearing or being distorted in the middle. Recording part of the decoded instruction bits (which functional unit should the instruction go to, how many operands does it use and whether it requires a destination register) verifies the operation of the decoder. Checking that instructions went to the correct functional units ensures that there was no corruption in the decoded opcode field of pipeline registers. Corruption of pointers or states associated with regular array structures is checked by observing whether instructions appear or disappear in the middle of pipeline. For example, corruption of the empty flag bit of an issue buffer results in sudden disappearance of instructions. This analysis is the most complicated one and the details can be found in [Park08].

## 5. RESULTS

We evaluated IFRA by injecting errors into a micro-architectural simulator augmented with IFRA, as described in Sec. 3. Post-analysis techniques described in Sec. 4 are used for bug localization. We used Simplescalar 3.0 architectural simulator [Simplescalar] with Alpha 21264 configuration (4-way pipeline, 72 maximum instructions in-flight, 2 ALUs, 2 multipliers, 2 load/store units). For this particular configuration, there are 50 different microarchitectural blocks. Each block has an average size equivalent of 80K 2-input NAND gates. The detailed list of microarchitectural blocks can be found in [Park 08]. Seven benchmarks from SPECint2000 (bzip2, gcc, gap, gzip, mcf, parser, vortex) were chosen as validation test programs. The FRS's were designed to collect information according to the setup described in Table 3.1. Each FRS was sized to have 1,024 entries.

All bugs were modeled as single bit-flips to target hard-to-repeat electrical bugs that pose major post-silicon validation

challenges. Many electrical bugs affect speed paths [Silas03], and speed paths manifest themselves as incorrect values arriving at flip-flops for certain input combinations and operating conditions.

Errors were injected in one of 1,191 flip-flops (Table 5.1). 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 5.1 include decoded opcode, register specifiers, immediate data, address, offset, etc. Valid bits indicate whether a given instruction is valid or not in a pipeline register.

Table 5.1. Error injection bits.

| Description | # 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 |

Upon error injection, the following scenarios are possible:

1. The error that has no effect at the system level.
2. The error that does not cause any post-trigger mechanism to trigger, but produces incorrect program output.
3. Failure manifestation with short error latency, where FRS's successfully capture the history from error injection to failure manifestation (including situations where recording is stopped upon activation of soft post-triggers).
4. Failure manifestation with long error latency, where 1024-entry FRS's fail to capture the history from error injection to failure (including soft triggers).

Cases 1 and 2 are related to coverage of validation test programs and post-triggers, and are not the focus of this paper. Hence, error injection runs resulting in these cases are ignored and not reported. Any error injection run which does not result in the activation of any post-trigger within 100K cycles from error injection are included in this category. For errors resulting in cases 3 and 4, we report results in Tables 5.2 and 5.3. For case 4, we pessimistically report that our IFRA approach completely misses correct bug location-time pair (included under "completely missed" category in Tables 5.2). All error injections were performed after a million cycles from the beginning of the program in order to demonstrate that the history between error injection and failure manifestation is sufficient for effective post-silicon bug localization as is the case with IFRA.

Tables 5.2 and 5.3 present results from 800 error injections that resulted in cases 3 and 4. The "*exactly located*" category represents the cases in which the injected errors were uniquely located to the correct location-time pair. The percentage of bugs belonging to this category must be very high for an effective bug localization technique. The "*candidate located*" category represents the cases in which the correct location-time pairs of error injections were identified, however with additional incorrect pairs identified as candidates. The "*completely missed*" category represents the cases where the correct location-time pairs do not appear in the list of candidates. An effective bug localization technique must have very few "completely missed" cases. Note that all debugging techniques were used for all injected errors, but credit was given to the technique that finally localized it. **It is clear from Table 5.2 that a large percentage bugs were uniquely located to correct location-time pair, while a very few bugs were completely missed, demonstrating the effectiveness of IFRA.** For "candidate located" cases, Table 5.3 reports the statistics on the number of possible candidates. It is clear from Table 5.3 that the number of such candidates is very small.

Table 5.2a IFRA bug localization summary.

| | |
|---|---|
| Exactly Localized | 75% |
| Correctly Localized with Candidates | 21% |
| Completely Missed | 4% |

Table 5.2b. IFRA bug localization breakdown.

| Post-analysis technique | Exactly located | Candidate located | Completely missed |
|---|---|---|---|
| Data dependency (Sec. 4.2) | 14% | 74% | 12% |
| Control-flow (Sec. 4.3) | 96% | 2% | 2% |
| Load / Store (Sec. 4.4) | 98% | 2% | 0% |
| Decoding (Sec. 4.5) | 94% | 6% | 0% |

Table 5.3. Statistics for "Candidate Located" cases.

| Post-analysis technique | Number of candidates | | | |
|---|---|---|---|---|
| | Mean | Min. | Max. | Std.Dev |
| Data dependency | 6.3 | 2 | 34 | 7 |
| Control-flow | 5.3 | 2 | 10 | 4.2 |
| Load / Store | 2 | 2 | 2 | 0 |
| Decoding | 2.4 | 2 | 3 | 0.55 |

Our synthesis result (Synopsys Design Compiler with TSMC 0.13 microns library) shows that the area impact of IFRA infrastructure is 2% on Illinois Verilog Model [IVM] (an open-source RTL implementation of Alpha-like core) assuming 1MB on-chip cache, which is typical of the current desktop/server processors. The overhead is largely dominated by the circular buffers present in the FRS's, because of the absence of any global at-speed routing and simplicity of the FRS's control. For many designs, on-chip scan chains can be reused for shifting out FRS contents. All input signals to FRS's, including post-trigger signals, are not latency-critical and thus are pipelined. The FSMs inside FRS's consume less than 0.2% of the area according to synthesis results. Total information storage for all FRS's for the super-scalar processor used in this paper adds up to 50 Kbytes, which is a very small fraction of total on-chip storage (including caches and register files) for state-of-the-art processors.

## 6. RELATED WORK

Related work on post-silicon validation can be broadly classified into six categories: scan dump [Caty 05, Dahlgren 03], check-pointing with deterministic replay [Silas 03, Sarangi 06], embedded trace buffers for hardware debugging [Anis 07], on-chip program and data tracing [MacNamee 00], fault-tolerant computing [Austin 99, Lu 82, Oh 02], and on-line assertion checking [Abramovici 06, Bayazit 05, Chen08].

Debugging techniques using scan dump, checkpointing with deterministic replay, and embedded trace buffers require failures to be reproducible. Moreover, they require simulation for comparison of observed states against golden responses. If easy failure reproduction support is present, it will also help IFRA by allowing FRS's to record unlimited length of history through repeated recording and dumping.

On-chip storage of program and data traces [MacNamee 00], 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 software executed on the processor. However, they are fundamentally different because they target software debug running on correct hardware. Hence, they need to store very different form of information compared to IFRA.

The difference between IFRA and traditional fault-tolerant computing is that the latter mainly focuses on error detection and recovery, while IFRA focuses on bug localization. Thus, the ability of IFRA to not interfere with the system behavior (no code modification or resource conflicts) is essential.

On-line assertion checking techniques 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 post-analysis techniques supported by IFRA.

## 7. 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. The major novelty of IFRA is in the introduction of a high-level abstraction for bug localization through new low-cost hardware recorders hat record semantic information about instruction data and control flows concurrently in a system setup, and special analysis techniques that analyze the recorded data for localization after failure detection. These design and analysis techniques enable IFRA to overcome major post-silicon bug localization challenges: 1. It helps bridge a major gap between system-level and circuit-level debug; 2. Failure reproduction is not required; 3. Self-consistency checks associated with the analysis techniques eliminate the need for full system-level simulation.

IFRA raises several interesting research questions that can be explored in the future: 1. Sensitivity analysis and characterization of the inter-relationships between post-analysis techniques, architectural features, error detection mechanisms, FRS sizes and bug types; 2. Wider variety of post-triggers based on assertions, e.g., [Abramovici 06, Bayazit 05], and symptoms [Wang 04]; 3. Applicability of IFRA for homogeneous / heterogeneous multi-core systems, and system-on-chips (SoCs) consisting of non-processor designs; 4. Applicability of IFRA to directed diagnostic test generation and fault diagnosis.

## 8. REFERENCES


[Abramovici 06] Abramovici, *et al.*, "A Reconfigurable Design-for Debug Infrastructure for SOCs", *Proc. DAC,* 2006.

[Alpha 99] Alpha 21254 Microprocessor Hardware Reference Manual, July 1999.

[Ando 03] Ando, H., *et al.,* "A 1.3-GHz Fifth-Generation SPARC64 Microprocessor", *IEEE JSSC,* vol.38, no.11, pp. 1896-1905, Nov 2003.

[Anis 07] Anis, E. and N. Nicolici, "On using lossless compression of debug data in embedded logic analyzers", *Proc. Intl. Test Conf.,* 2007.

[Austin 99] Austin, T.M., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design", *Proc. Intl. Symp. on Microarchitecture,* 1999.

[Bayazit 05] Bayazit, A.A. and S. Malik, "Complementary Use of Runtime Validation and Model Checking", *Proc. Intl. Conf. on Computer-aided Design*, 2005.

[Caty 05] Caty, O, P. Dahlgren and I. Bayraktaroglu, "Microprocessor Silicon Debug based on Failure Propagation Tracing", *Proc. Intl. Test Conf.,* 2005.

[Chen 08] Chen, K., S. Malik, and P. Patra. "Runtime Validation of Memory Ordering Using Constraint Graph Checking". *Proc. Intl. Symp. on High-Performance Computer Architecture*, 2008.

[Dahlgren 03] Dahlgren, P., P. Dickinson and I. Parulkar, "Latch Divergence in Microprocessor Failure Analysis", *Proc. Intl. Test Conf.*, 2003.

[Heath 04] Heath M.W., W.P. Burleson and I.G. Harris, "Synchro-Tokens: Eliminating Nondeterminism to Enable Chip-Level Test of Globally-Asynchronous Locally-Synchronous SoC's", *Proc. Conf. on Design, Automation and Test in Europe,* pp1532-1546, 2004.

[IVM] http://www.crhc.uiuc.edu/ACS.

[Josephson 06] Josephson, D., "The Good, the Bad, and the Ugly of Silicon Debug", *Proc. DAC.*, 2006.

[Leon 06] Leon, A.S., B. Langley, and J.L Shin "The UltraSPARC T1 Processor: CMT Reliability", *Proc. Custom Integrated Circuits Conf.*, 2006.

[Livengood 99] Livengood, R. and D. Medeiros, "Design for (physical) Debug for Silicon Microsurgery and Probing of Flip-chip Packaged Integrated Circuits", *Proc.Intl.Test Conf.,* 1999.

[Lu 82] Lu, D.J. "Watchdog Processors and Structural Integrity Checking", *IEEE T COMPUT,* , pp.681-685, July 1982.

[MacNamee 00] MacNamee, C. and D. Heffernan, "Emerging On-chip Debugging Techniques for Real-time Embedded Systems", *Computing & Control Engineering Journal,* vol.11, no.6, pp.295-303, Dec 2000.

[Oh 02] Oh, N., S. Mitra and E.J.McCluskey, "ED4I: Error Detection by Diverse Data and Duplicated Instructions", *IEEE T COMPUT*, vol.51, no.2, pp.180-199, Feb 2002.

[Patra 07] Patra, P., "On the Cusp of a Validation Wall", *IEEE DES TEST COMPUT,* vol.24 no.2, pp.193-196, Mar 2007.

[Park 08] Park S., and S. Mitra., "IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization", *Technical Report, Stanford University*, 2008, url: http://www.stanford.edu/group/rsg_csl.

[Parker 03] Parker K.P., The Boundary-Scan Handbook, 3rd ed., Springer, 2003.

[Sanda 08] Sanda P.N. *et al.,* "Soft-error resilience of the IBM POWER6 processor", *IBM Journal of Research and Development*, vol.52, no.3, 2008

[Sarangi 06] Sarangi, S.R., B.Greskamp and J.Torrellas, "CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging", *Intl. Conf. on Dependable Systems and Networks*, 2006.

[Sarangi 07] Sarangi, S.R., *et al., "*Patching Processor Design Errors with Programmable Hardware", *IEEE MICRO,* vol.27, no.1, pp.12-25, Jan 2007.

[Silas 03] Silas, I., *et al., "*System-Level Validation of the Intel Pentium M Processor", *Intel Technical Journal*, May 2003.

[simplescalar] www.simplescalar.com.

[Trong 07] Trong, S.D., *et al., "*P6 Binary Floating-Point Unit", *Proc. Intl. Symp. on Computer Arithmetic*, 2007.

[Wagner 06] Wagner, I., V. Bertacco and T. Austin, "Shielding Against Design Flaws with Field Repairable Control Logic", *Proc. DAC*, 2006.

[Wang 04] Wang, N.J., *et al., "*Characterizing the effects of Transient Faults on a High Performance Processor Pipeline". *Intl. Conf. on Dependable Systems and Networks*, 2004.

[Yerramilli 06] Yerramilli, S., "Addressing Post-Silicon Validation Challenge: Leverage Validation & Test Synergy (Invited Address)", *Intl. Test Conf.*, 2006.