POST-SILICON BUG LOCALIZATION IN PROCESSORS

TECHNICAL REPORT V2.0

ABSTRACT

For complex integrated circuits, pre-silicon verification alone is inadequate in ensuring that manufactured chips do not contain logic and electrical bugs. Post-silicon validation, which operates samples of manufactured chips in application environments to validate correct behaviors across specified operating conditions, is essential. According to industry reports, post-silicon validation is becoming very expensive. A major bottleneck in post-silicon validation is the bug localization step which involves identifying hardware bug locations and short functional stimuli that can expose detected bugs. For example, it may take several days to weeks to localize an electrical bug that may arise due to incorrect interactions between a design and the operating conditions.

This report presents IFRA (Instruction Footprint Recording & Analysis), a new technique for post-silicon bug localization in processors, which overcomes cost and scalability challenges of existing techniques. During normal operation of a processor in a post-silicon validation setup, special on-chip recorders collect information about flows of instructions through the processor and what the instructions did as they passed through various design blocks. Upon system failure, such as a crash, the recorded information is scanned out and analyzed offline using special self-consistency-based analysis techniques to localize hardware bugs. IFRA provides two major benefits over traditional techniques: (1) it does not require bugs to be reproduced at the system-level and (2) it does not require system-level simulation. Evaluation of IFRA on an open-source microarchitectural simulator modeling Alpha 21264 demonstrates high bug localization accuracy (96%) at low area overhead (1%).

Applying IFRA to new microarchitectures can be challenging because it requires some degree of manual effort. This report presents a new BLoG (Bug Localization Graph) technique which is a step towards automated application of IFRA. Evaluation of BLoG-assisted IFRA on an industrial microarchitectural simulator modeling Intel® Core™ i7, a state-of-the-art complex commercial processor, demonstrates its effectiveness (90% bug localization accuracy) and practicality.

TABLE OF CONTENTS

LIST OF TABLES

Number *Page*

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

1.1 POST-SILICON VALIDATION BACKGROUND

With increasing chip complexity and shortening time-to-market, chip developers are facing progressively more difficult challenges in ensuring their chips to be free of hardware bugs before shipment. Hardware bugs are detected either before chip fabrication, during *pre-silicon verification*, or after fabrication, during *post-silicon validation.* Pre-silicon verification [Clarke 99][Dill 98][Schelle 10] checks the design for its correct functionality using simulation / emulation techniques or using formal methods. Post-silicon validation involves operating manufactured chips in actual application environments to check for their correct behaviors across specified operating conditions (*e.g.*, voltage, temperature and frequency).

Table 1.1 compares and contrasts the two phases of chip development. Accessing internal signals of a physical chip is difficult as it is done indirectly through package pins or through special-purpose built-in hardware (*e.g.*, Boundary-scan JTAG interface [TI 97]). Pre-silicon verification is free of this problem because any arbitrary signal can be probed in a software environment. Fixing physical bug is expensive and can cost multiple silicon re-spins, which involves redesigning the mask (costing $1 million for sub-90nm technology [Ying 05]) to remanufacture the chip with corrections.

Table 1.1. Pre-silicon verification vs. post-silicon validation.

	Pre-silicon Verification	**Post-Silicon Validation**
Signal access	(+) Easy	(-) Limited
Bug fixes	(+) Cheap	(-) Expensive
Electrical bugs	(-) Difficult to model	(+) Already present
Problem detection speed	(-) Slow	(+) Fast

Although pre-silicon verification is essential, and has traditionally been the main tool for detecting hardware bugs, post-silicon validation is becoming increasingly important for the following three reasons:

1) With increasing chip complexity, several electrical interactions are becoming significant, e.g., signal integrity, cross-talk and power-supply noise, temperature effects, etc. Accurate modeling of all these physical effects is usually very difficult during pre-silicon verification;
2) With increasing design complexity (e.g., prevalence of adaptive power management), simulations and emulations are not fast enough (orders of magnitude slower than physical chip [Krupnova 04]) to explore many possible configurations and corner cases;
3) With increasing process variations, designing for the worst case leads to unacceptable power and/or performance. Chips not designed for the worst case must be validated after fabrication [Yerramilli 06].

According to recent industry reports, post-silicon validation is becoming significantly expensive. Intel reported a 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 post-silicon validation costs are rising faster than the design costs.

Post-silicon validation involves four steps:

1) *Bug detection*: Detecting a problem by running a test, such as end-user applications or functional tests, until a *system failure* (chip in a system returns a fatal exception or stops functioning altogether).
2) *Bug localization:* Localizing the problem to a small region from the system failure, *e.g.*, a bug in an adder inside an ALU of a complex processor. The stimulus that exposes the bug, *e.g.*, the particular 10 lines of code from some application, is also important.

3) *Bug root-cause*: Identifying the root cause of the problem. For example, an electrical bug may be caused by power-supply noise slowing down a circuit path resulting in an error at the adder output.
4) *Bug fixing*: Fixing or bypassing the problem by microcode patching [Chang 07] [Goddard 95] [Sarangi 07][Wagner 06], circuit editing [Livengood 99], or, as a last resort, re-spinning using a new mask.

Josephson [Josephson 06] points out that the second step, bug localization, dominates post-silicon validation effort and costs. Hence, this report focuses on bug localization.

1.2 BUG LOCALIZATION BACKGROUND

The bottleneck in post-silicon validation can be further narrowed down by considering the types of bugs being localized. There are two types of bugs that design and validation engineers worry about:

1) *Logic bugs* (also called *functional bugs)*: Bugs caused by design errors. While many functional bugs get detected during pre-silicon verification, a small percentage of them get exposed during post-silicon validation due to increasing design complexity and design schedule constraints.
2) *Electrical bugs* (also called *circuit marginalities):* Bugs caused by the interactions between the design and the electrical effects, such as cross-talk and power-supply noise. Such bugs generally manifest themselves only under certain operating conditions (temperature, voltage, frequency). Examples include setup and hold time problems.

There are two major challenges faced by most bug localization techniques used today that limit their scalability for future systems:

1) *System-level failure reproduction*: When a failure is detected while using a chip in a system, most existing techniques require the failure to be reproducible. Failure reproduction involves returning the chip to an error-free state, and re-executing

the failure-causing stimulus (including test segment, interrupts, and operating conditions) to reproduce the same failure. Unfortunately, many electrical bugs are hard to reproduce [Patra 07] due to presence of asynchronous I/Os, and multiple clock domains. Techniques to make failures reproducible [Heath 04][Sarangi 06][Silas 03] are often intrusive to system operation, and may not expose bugs.

2) *System-level simulation*: Most existing techniques require golden responses, *i.e.*, correct signal values for every clock cycle for the entire system (*i.e.*, the chip and all the peripheral devices on the board) to compare against the signal values produced by the chip being validated. System-level simulation is generally 7-8 orders of magnitude slower than actual silicon [Bentley 01][Nakamura 04] and in addition, expensive external logic analyzers are required to record all signals values that enter and exit the processor through external pins [Silas 03].

Due to these factors, a logic bug typically takes hours to days to be localized vs. an electrical bug that requires days to weeks and more expensive equipments [Josephson 01]. Hence, this report focuses on localization of electrical bugs.

1.3 CONTRIBUTIONS

The major contributions of this report are:

1) **Bug localization technique overcoming existing challenges**: This dissertation presents *IFRA* (*Instruction Footprint Recording and Analysis*), which is a new technique for localizing electrical bugs in processors without requiring system-level failure reproduction and system-level simulation. During normal operation of a processor in a post-silicon validation setup, special on-chip recorders collect information about flows of instructions through the processor and what the instructions did as they passed through various design blocks. Upon system failure, the recorded information is scanned out and analyzed offline using special self-consistency-based analysis techniques to localize hardware bugs. Evaluation of IFRA on an open-source microarchitectural simulator modeling Alpha 21264

[Digital 99] demonstrates high bug localization accuracy (96%) at low area overhead (1%).

2) **Framework for implementing IFRA on different microarchitectures:** Applying IFRA to new microarchitectures can be challenging because it requires some degree of manual effort. This report presents *BLoG* (*Bug Localization Graph*), a new framework that enables systematic implementation of IFRA and automatic execution of IFRA for different microarchitectures.
3) **BLoG-assisted IFRA for a commercial Intel processor**: We use the BLoG framework to implement IFRA on an Intel® Core™ i7 processor [Casazza 09] and demonstrate the effectiveness of BLoG-assisted IFRA using an industrial-grade microarchitectural simulator extensively used during product development. BLoG-assisted IFRA achieves high bug localization accuracy of over 90%.

1.4 OUTLINE

Chapter 2 describes hardware support required for IFRA and various analysis techniques used to localize bugs from the recorded data. It also presents simulation results obtained from an open-source microarchitectural simulator modeling Alpha 21264. Chapter 3 presents the BLoG framework and evaluation results of BLoG-assisted IFRA for a commercial Intel processor, followed by conclusions in Chapter 4.

CHAPTER 2. POST-SILICON BUG LOCALIZATION IN PROCESSORS USING IFRA

This chapter targets localization of electrical bugs in processors using a technique called *IFRA*, an acronym for Instruction Footprint Recording and Analysis. We did not investigate IFRA's applicability to logic bugs, because there is little consensus about models of logic bugs [ITRS 07]. Fig. 2.1 shows a post-silicon bug localization flow using IFRA. During chip design, a processor is augmented with low-cost hardware recorders for recording instruction footprints. *Instruction footprints* (or simply, *footprints*) are compact pieces of information describing the flows of instructions (*i.e.*, where each instruction was at various points in time), and what the instructions did as they passed through various design 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.

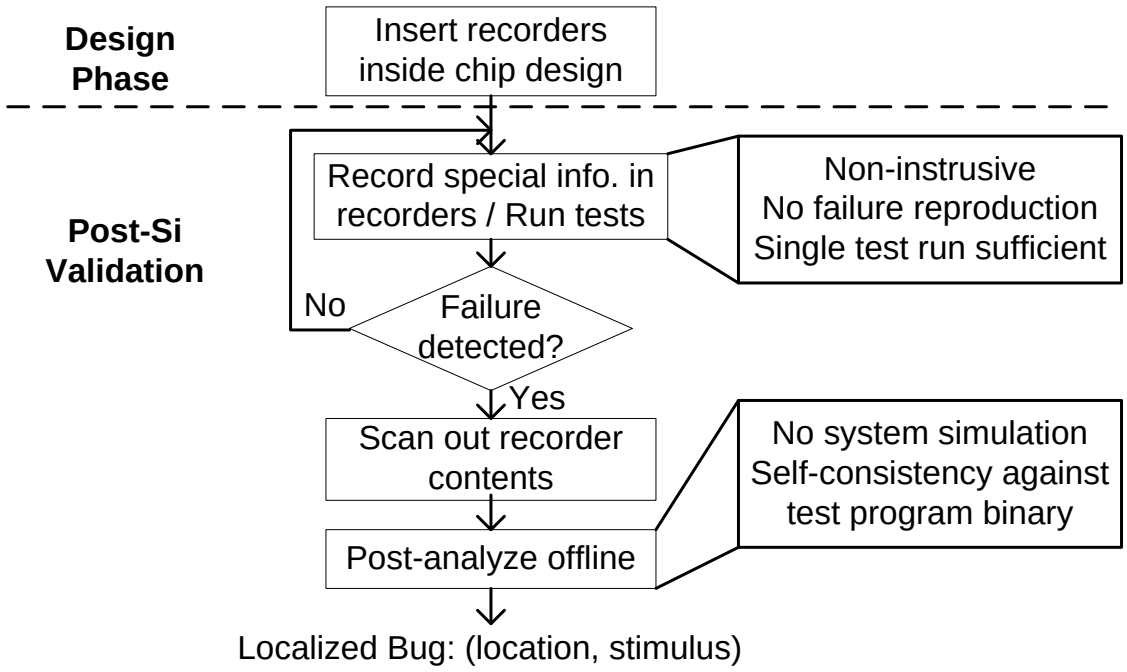

Fig. 2.1. Post-silicon bug localization flow using IFRA.

Upon detection of a system failure, the recorded footprints are scanned out through a Boundary-scan JTAG interface. IFRA uses special techniques (Sec.2.2) 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 purposes.

Scanned-out footprints, together with the test program binary executed during post-silicon bug detection, are post-processed off-line using special analysis techniques (Sec. 2.3). These techniques identify both location (*e.g.,* the instruction queue control unit, scheduler, forwarding path, and decoder) and clock 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. The analysis techniques do not require any system-level simulation because they rely on *self-consistency-based checks* that inspect for the existence of contradictory events in scanned-out footprints with respect to the test program binary.

Once a bug is localized using IFRA, existing circuit-level debug techniques [Caty 05][Josephson 06] may be used to 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 [Abramovici 06]) and control mechanisms (*e.g.,* clock manipulation [Josephson 06]) to the localized block or its neighboring microarchitectural block(s).

Sec. 2.1 describes a processor model, which IFRA is going to target. Sec. 2.2 describes hardware support required for IFRA, while Sec. 2.3 describes post-analysis performed on the scanned-out footprint. Sec. 2.4 presents simulation results, followed by related work in Sec. 2.5, and conclusions in Sec. 2.6.

2.1 TARGET PROCESSOR MODEL

There are three features and four assumptions on the processor model that IFRA is targeting. The features demonstrate generality of IFRA, while the assumptions describe traits of our particular model that are shared with modern superscalar processors [Shen 05].

Feature 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). 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 (reorder buffer) in order. The maximum number of instructions-in-flight, *n*, equals the number of ROB entries in a superscalar processor. We are targeting out-of-order processors, but for simpler in-order processors (*e.g.*, ARMv6, Intel Atom, SUN Niagara cores), the entire IFRA-based analysis can be significantly simplified.

Feature 2: Multiple clock domains: For the execution stage, the granularity of individual clock domains can be as fine as to include a single type of functional units, but not finer. 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. For the rest of the pipeline stages, the granularity of individual clock domains can be as fine as a single pipeline stage, but not finer.

Feature 3: Dynamic voltage and frequency scaling: Individual clock domains can undergo independent dynamic voltage and frequency scaling.

Assumption 1: Branch misprediction handling: Mispredicted branch instructions are detected by branch units in the execution stage and the corresponding pipeline flushes (process of removing instructions in the middle of execution to enforce a change in control flow) 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 the head of the ROB (*i.e.,* it is the instruction's turn to commit).

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 the occurrence, the processor stops fetching new instructions and allows the instructions that are already in the pipeline to commit.

Assumption 4: Fatal exception handling: Once an instruction with a fatal exception reaches the head of the ROB, the processor is halted without any pipeline flushes.

Assumption 5: Test program binary: Binary is statically linked so their PCs (Program Counter) can be inferred after system failure. Otherwise, we would increase total storage from 60 Kbytes to 76 Kbytes to record instruction words in the recorders (more details in Sec. 2.2.1)

2.2 IFRA HARDWARE SUPPORT

We use an Alpha 21264-like 4-way superscalar processor model [Digital 99] to explain the IFRA recording infrastructure. The shaded parts in Fig. 2.2 indicate the three hardware components of IFRA:

1) A set of distributed instruction footprint recorders (denoted by 'R' in Fig. 2.2) 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:

1) The ID corresponding to the instruction.
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 IFRA hardware. Electrical bugs affecting only IFRA hardware structures may result in two outcomes:

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 footprints 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.

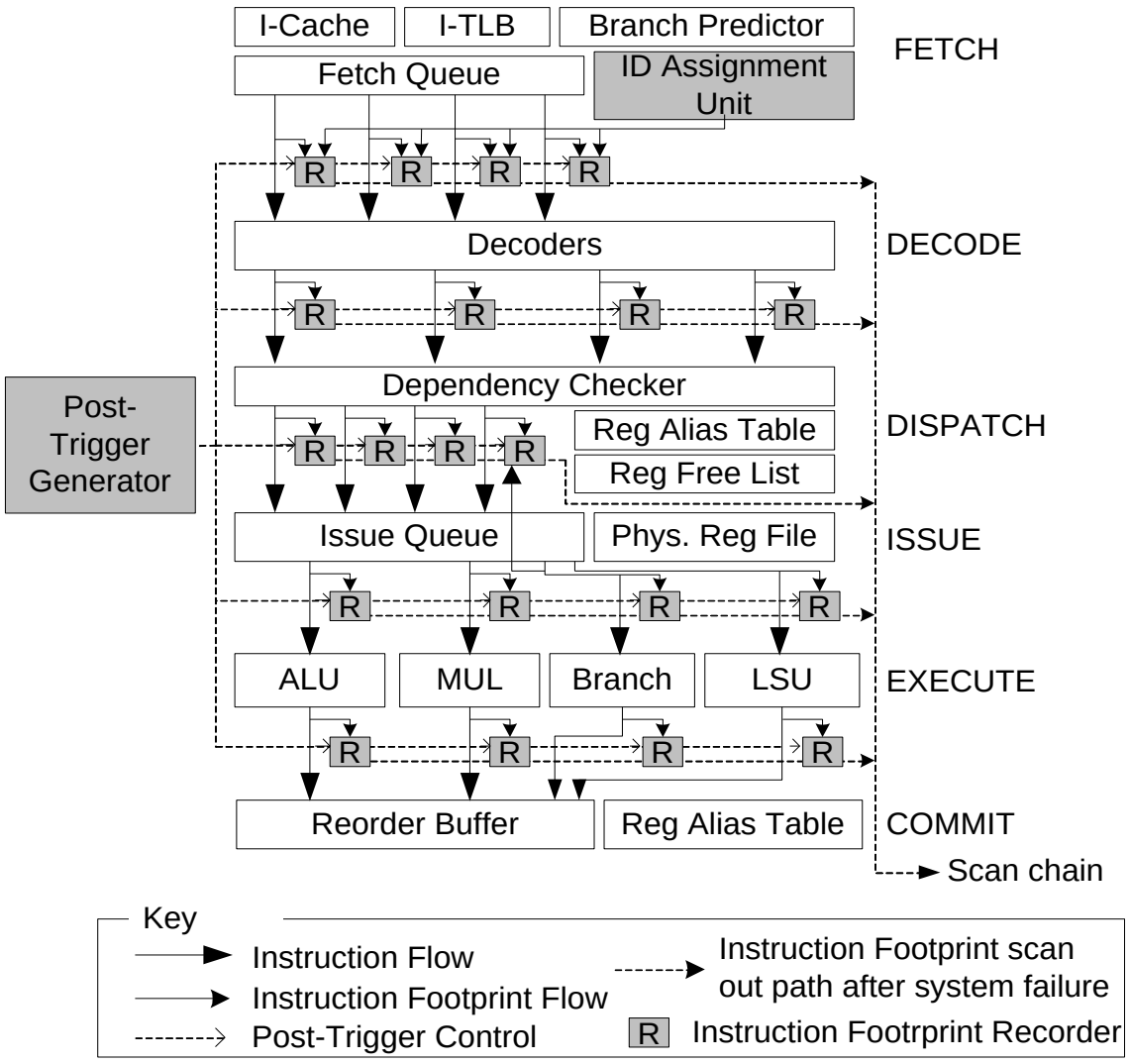

Fig. 2.2. Superscalar processor augmented with recording infrastructure.

2.2.1 INSTRUCTION FOOTPRINT RECORDER

As shown in Fig. 2.3, 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 during test run and is scanned out after a system failure. The control circuitry is responsible for four tasks:

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.

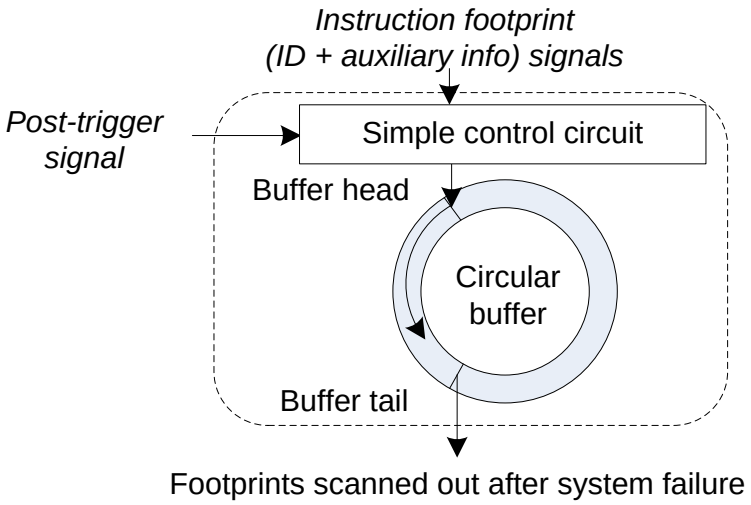

Fig. 2.3. Internal structure of a footprint recorder.

Table 2.1 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 bits), whether it uses a destination register (1 bit), and/or a second operand register (1 bit). The 2-bit and 3-bit 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 exceptions it caused.

Table 2.1. Auxiliary information for Alpha microarchitecture.

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 has an additional 8-bit ID (Sec. 2.2.2))			60 Kbytes	

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 [Wang 04], assuming a 2MByte 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 2.2) operate at slow speed, and a large portion of this routing reuses existing on-chip scan chains that are present for manufacturing testing. If on-chip trace buffers [Abramovici 06] already exist for validation purposes, they can be reused to reduce the area impact. Alternatively, a part of data cache may also be used [Agarwal 86] to reduce the area impact of IFRA.

2.2.2 ID-ASSIGNMENT UNIT

There must be a method of identifying which set of the recorded data, stored across multiple recorders, belong to a particular instruction that was in-flight in the processor. Each footprint can be tagged with an identifier, or an ID, but the choice of the ID assignment scheme is important.

Many possible simplistic ID assignment schemes exist, each with their own limitations. For example, a scheme can assign consecutive numbers in a circular fashion to each incoming instruction. However, if IDs are too short (*e.g.*, 8-bit IDs if there can be only 256 instructions in a processor at any point in time), then complex recorders are required that are capable of removing footprints in the event of a pipeline flush. Having wider IDs will solve this problem, but will suffer from large storage requirement (*e.g.*, 40-bit ID will increase the storage requirement to 160Kbytes from 60Kbytes). Another option that does not require complex recorders would be to use global timestamps in addition to the short IDs, but this scheme also suffers from large storage requirements. The schemes that do not require complex recorders suffer even more in the presence of multiple clock domains with voltage and frequency scaling (DVFS).

Using PC values as IDs does not work for processors supporting out-of-order execution. 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 below, uses $\log_2 4n$ bits, where n is the maximum number of instructions in a processor at any one time (*e.g.*, $n = 64$ for Alpha 21264).The first two rules assign consecutive numbers in a circular fashion to incoming instructions while the third rule allows the scheme to work (proof in Appendix A) under all the aforementioned circumstances: *i.e.*, for processors supporting out-of-order execution, pipeline flushes, multiple clock domains and DVFS. Instruction IDs are assigned to individual instructions as they exit the fetch stage and enter the decode stage

(Fig. 2.4). Since multiple instructions may exit the fetch stage in parallel at any given clock cycle, multiple IDs are assigned in parallel.

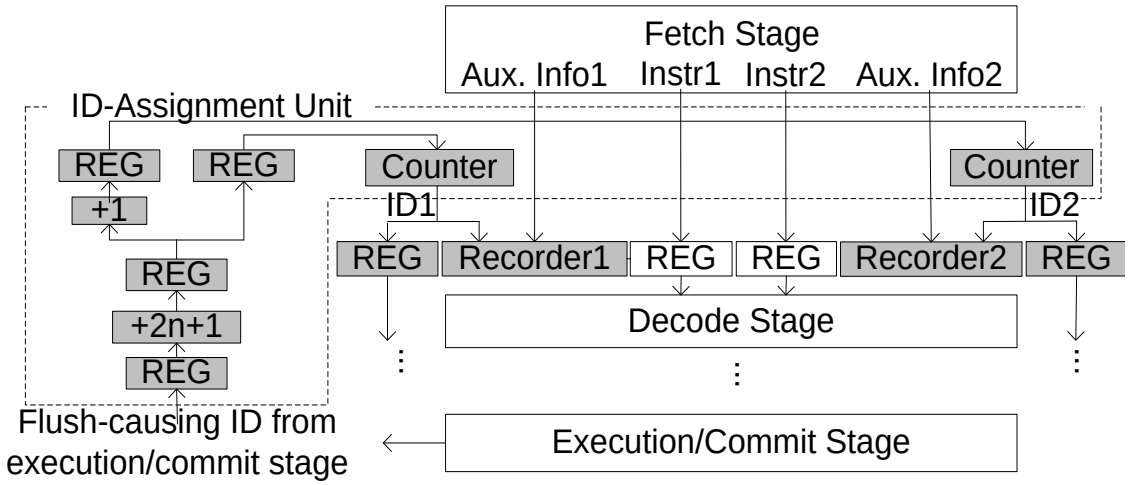

Fig. 2.4. ID-assignment unit for a 2-way processor.

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, ,.. , 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 (mod 4n), X+2 (mod 4n), …, X+q (mod 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 (mod 4n). As a result, ID of Y+2n+1 (mod 4n) is assigned to the first instruction that is fetched after the flush.

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

2.2.3 POST-TRIGGER GENERATOR

Suppose that a test program has been executing for billions of cycles and an electrical bug is exercised after 5 billion cycles from start. Moreover, suppose that the electrical bug causes a system crash after another 1 billion cycles. With limited storage, we are only interested in capturing the information around the time when the electrical bug is exercised. Hence, five billions of cycles worth of information before the bug occurrence may not be necessary. On the other hand, if we stop recording only after the system crashes, all the useful recorded information will be overwritten. Thus, we must incorporate mechanisms, referred to as *post-triggers*, for reducing *error detection latency*, the length of time between the appearance of an error caused by a bug and system failure.

Post-triggers targeting five different failure scenarios are listed in Table 2.2. A *hard post-trigger* fires when there is an evident sign of failure, and causes the processor operation to terminate. Classical hardware error detection techniques such as parity bits for arrays (*e.g.*, register file, ROB, register free list, and various queues) and residue codes for arithmetic units in ALUs and address calculators exist in several commercial processors [Ando 03][Sanda 08]. Fatal exceptions, such as unimplemented instruction exceptions, arithmetic exceptions and alignment exceptions are already present in most processors.

However, hard post-triggers mechanisms alone are not sufficient, *e.g.*, two tricky scenarios described in the last 2 rows of Table 2.2. These two failure scenarios may be detected several millions of cycles after an error occurs, causing useful recorded information to be overwritten even with the existing error detection mechanisms. Hence, we introduce the notion of *soft post-triggers*. A soft post-trigger fires when there is an early symptom of a possible failure. It causes the recording in all 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 pre-specified amount of time, the processor stops. If a 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 [Mahmood 88].

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. While running such tests, soft-triggers targeting segfault must be disabled so that recorders do not pause during the TLB servicing.

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 [Abramovici 06][Bayazit 05], software assertions, and symptoms [Wang 04] is a topic of future research.

Table 2.2. Failure scenarios and post-triggers.

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

2.3 POST-ANALYSIS TECHNIQUES

Once recorder contents are scanned out and formatted (Sec. 2.3.1) appropriately, footprints belonging to the same instruction (but in multiple recorders) are identified and linked together using a technique called *footprint linking* (Sec. 2.3.2) as shown in Fig. 2.5. The linked footprints are also mapped to the corresponding instruction in the test program binary using the PC values stored in the fetch-stage recorder. After which, two sets of self-consistency-based checks are run, inspecting for the existence of contradictory events in linked footprints with respect to the test program binary. A set of microarchitecture-independent checks, referred to as *high-level analysis* (Sec. 2.3.3), finds the first sign of an *inconsistency* in program execution. The information on the discovered inconsistency is passed along to the next step in the form of a <*location, footprint*> *pair* (Sec. 2.3.2.2). Starting from the inconsistency, the next step runs a set of microarchitecture-dependent checks, referred to as *low-level analysis* (Sec. 2.3.4), identifying a set *bug candidates*, which are also in the form of <location, footprint> pairs. From the bug candidates, corresponding exposing stimuli are derived (Sec. 2.3.5). The low-level analysis asks a series of microarchitecture-specific questions according to a manually-generated decision diagram. A technique for minimizing the manual effort is presented later in Chapter 3.

The post-analysis techniques rely on the concept of self-consistency. While such checks are extensively used in fault-tolerant computing for error detection [Austin 99][Lu 82][Oh 02][Siewiorek 98], we use them for bug localization. Such application is possible because, unlike fault-tolerant computing, the checks are performed off-line enabling deeper analysis for localization purposes.

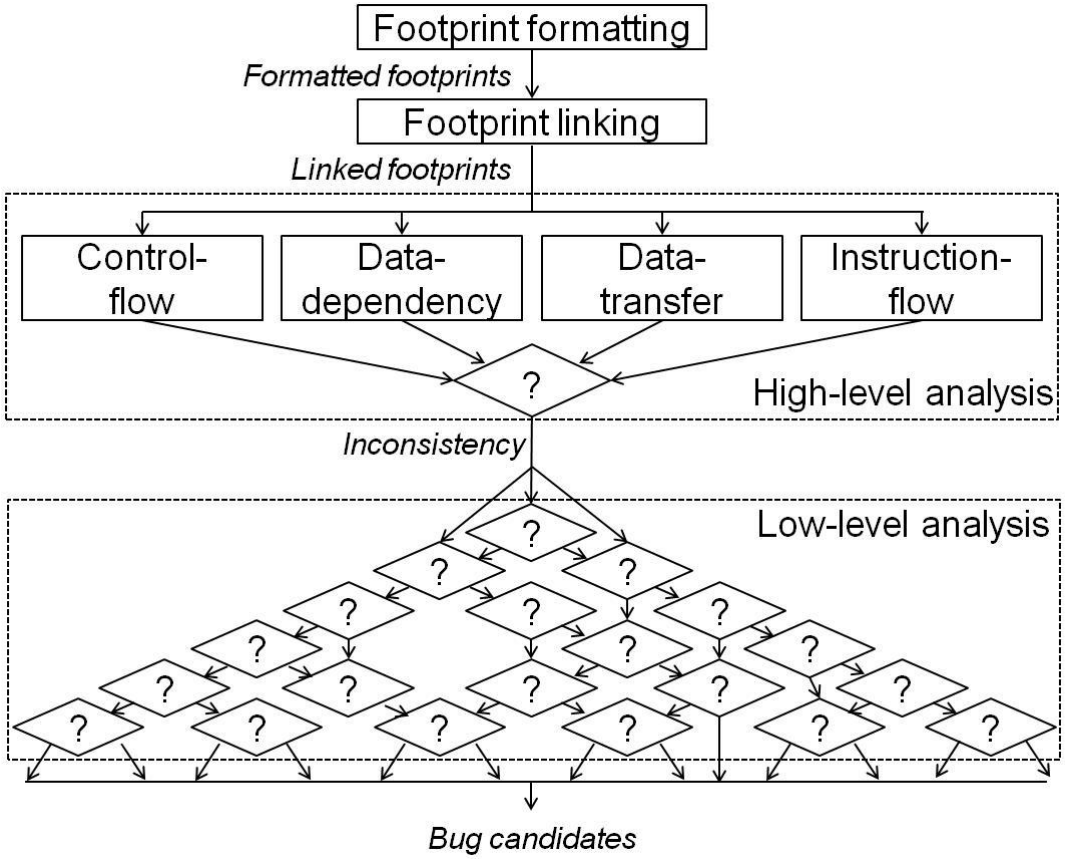

Fig. 2.5. Post-analysis summary.

2.3.1 FORMATTING SCANNED-OUT FOOTPRINTS

After scanning out footprints from individual recorders, the following six steps format the set of footprints before linking:

1) **Unwrap circular buffers:** The footprints scanned out from each circular buffer are unwrapped so that the youngest entry appears at the bottom and the oldest entry appears at the top (Fig. 2.6).
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. 2.6). Due to the 2nd feature of our processor model (Sec. 2.1), 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. 2.6). 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 $(wi+j)^{th}$ entry of the vector, for a w-way stage. Footprint matrices associated with out-of-order pipeline stages are retained.
6) **Augment fetch-stage footprint vector:** Footprint vector associated with the fetch-stage is augmented with an additional auxiliary information field to contain instruction words (Fig. 2.7). Since PC is stored in the fetch-stage recorder (Table 2.1), instruction word corresponding to each PC can be obtained from the test program binary.

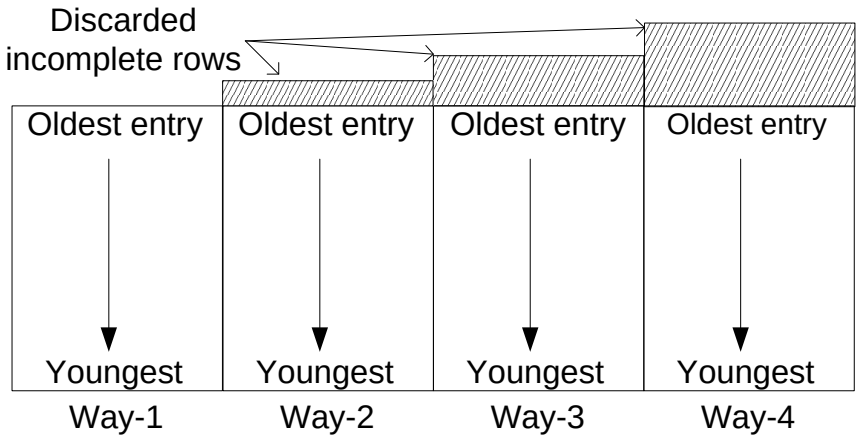

Fig. 2.6. Aligning four unwrapped circular buffers for a 4-way pipeline stage.

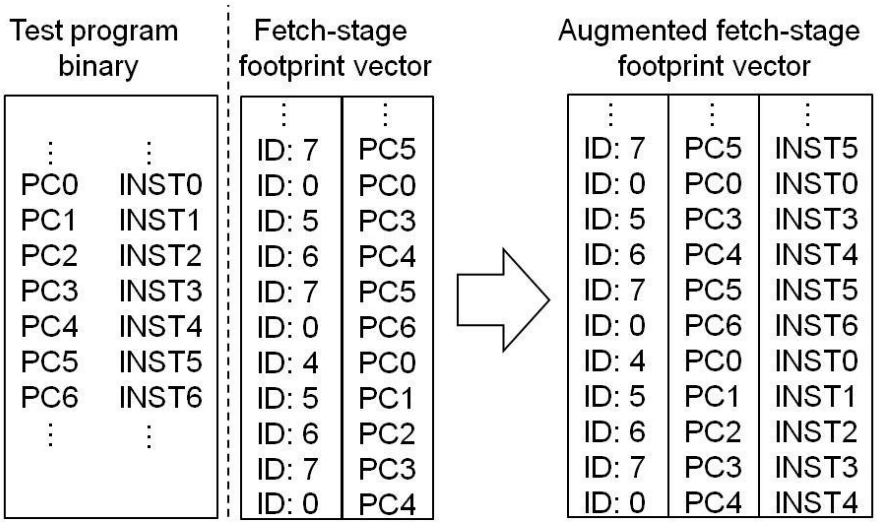

Fig. 2.7. Fetch-stage footprint vector augmented with instruction words.

2.3.2 FOOTPRINT LINKING

Fig. 2.8 shows part of a test program binary and the contents of three (out of many) recorders, right after they are scanned out. As explained in Sec. 2.2.2, since we use short instruction IDs (8-bits 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, in Fig. 2.8, ID 0 appears in three entries of the fetch-stage footprint vector, in two entries of the issue-stage footprint matrix (with a single column), and in three entries of the ALU footprint matrix (with a single column).

Footprint linking analyzes ID sequence to identify which of these ID 0s belongs to the same instruction. The ID assignment scheme presented in Sec. 2.2.2 enforces the following properties on the ID sequence (proof in Appendix A):

Property 1) All flushed instructions are uniquely identified by using Rule 3 of the ID assignment scheme;

Property 2) If instruction A was fetched before instruction B, and they both have the same ID, then A will always exit any pipeline stage (and leave its footprint in the corresponding recorder) before B does for that same pipeline stage.

In Fig. 2.8, using the first property, all flushed instructions with ID 0s are identified and discarded. Fig. 2.9 shows how the identification is done for the fetch-stage footprint vector. The example corresponds to a processor with n, the maximum number of in-flight instructions, equal to 2. The fetch stage processes instructions in program order. The first step is to identify breaks in consecutive IDs, *e.g.,* there is a break between the second youngest ID 0 and ID 4. The gap immediately indicates that the ID 4 belongs to a newly-fetched instruction after a pipeline flush. Subtracting $2n + 1$ from ID 4 identifies the ID of a flush-causing instruction. All IDs between the flush-causing (ID 7) and the newly-fetched (ID 4) correspond to flushed instructions. Similar analysis is done for all other breaks in the consecutive assignment to identify IDs corresponding to flush-causing, flushed, committed and uncommitted instructions. For pipeline stages that process

instructions out of program order (*e.g.*, issue- and execution-stages), identification of flushed instructions is more complex (full details in the Appendix A).

After the flushed instructions are identified and discarded, using the second property, the youngest instances of ID 0 across all vectors/matrices are linked together, followed by linking of the second youngest instances of ID 0, and so on.

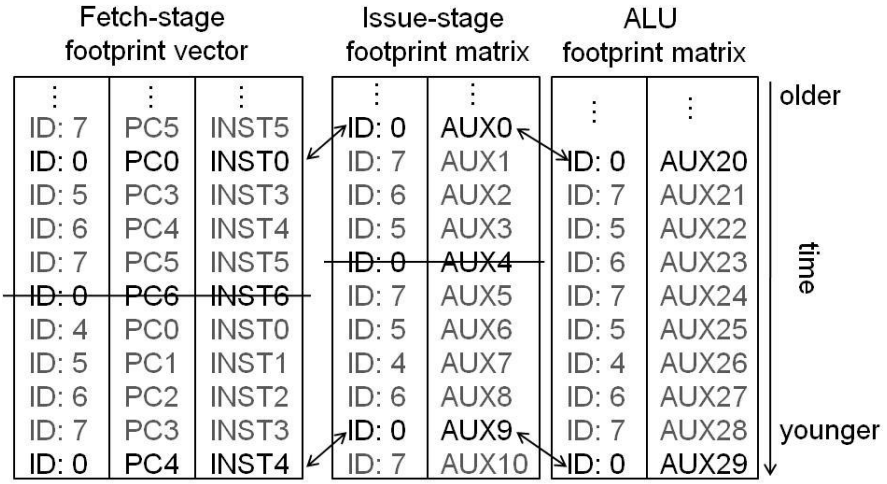

Fig. 2.8. Footprint linking, with a max number of 2 instructions in flight.

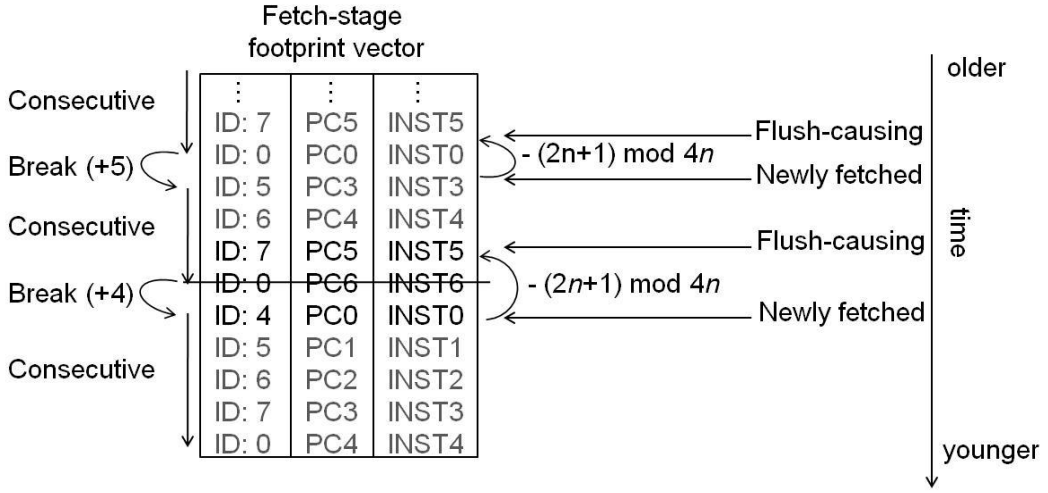

Fig. 2.9. Flushed / flush-causing instructions in fetch stage footprint vector.

Linked footprints provide the following information for each instruction that was in-flight in the processor:

1) Whether it was committed or uncommitted;
2) Whether it was flush-causing, flushed or neither;
3) The cycle when the instruction went through a particular pipeline stage relative to the cycle when the recording was stopped / paused for that stage;
4) The operation the instruction was performing at a particular pipeline stage.

2.3.2.1 Footprint Pointer

We define *footprint pointer*. A footprint pointer for a footprint vector/matrix points to a vector/matrix entry that contains a single footprint. When incrementing a pointer for a footprint matrix, the pointer accesses the matrix in row-major order (*i.e.*, left to right in a row and then from oldest to youngest row) as shown in Fig. 2.10.

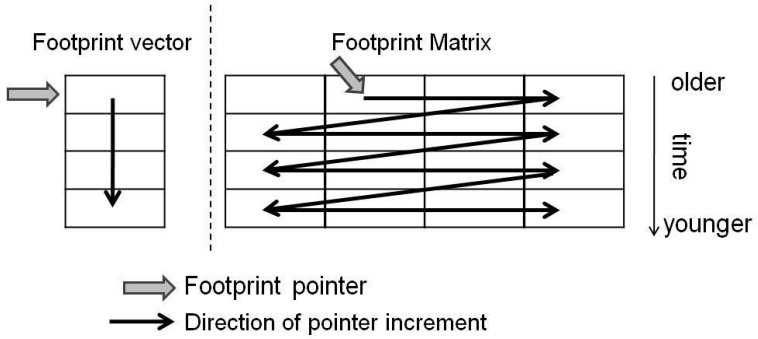

Fig. 2.10. Footprint pointer and direction of pointer increment.

A footprint pointer indicates a cycle within a pipeline stage, relative to the cycle when the post-trigger was recieved. For example, consider the fetch-stage footprint vector shown in Fig. 2.11. Compacted consecutive idle cycles are expanded back, as described in Sec. 2.3.1 and the youngest footprint is the one recorded just before receiving a post-trigger. Since each entry corresponds to a cycle within a pipeline stage, the difference from the tail pointer indicates the relative cycle from the cycle when the recording was stopped for that pipeline stage, as shown in the figure. If a single clock domain is used for the entire processor, as opposed to having a separate clock domain for each pipeline stages, then the footprint pointer would indicate a global cycle within the processor instead.

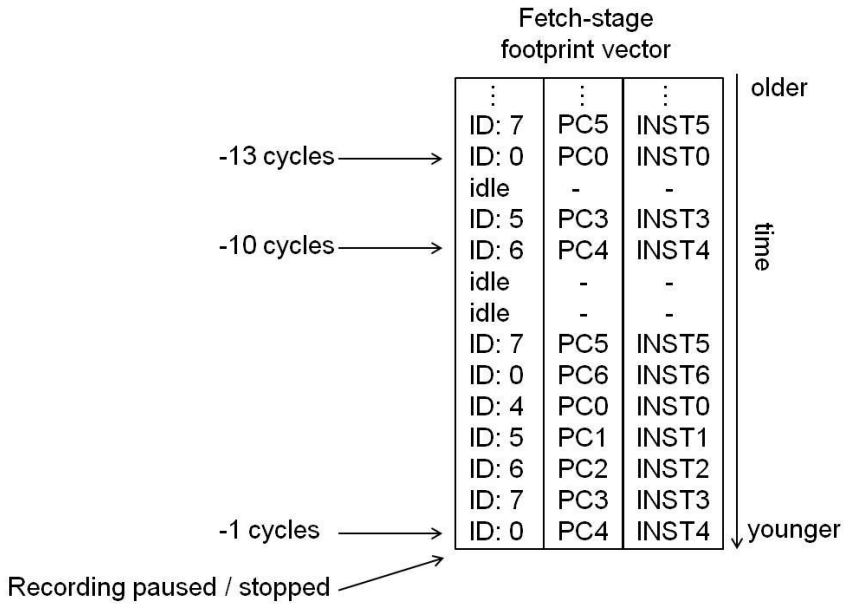

Fig. 2.11. Footprints indicating cycles within a pipeline stage.

2.3.2.2 <Location, Footprint> Pair

We define <location, footprint> pair. The location element of the pair indicates a design block containing an erroneous flip-flop affected by an electrical bug. The footprint element of the pair is a footprint pointer that indicates a cycle within a pipeline stage when the erroneous behavior occurred.

2.3.2.3 Follow_link() Operator

We define a new operator, *follow_link()*. It takes three inputs – source footprint pointer, source footprint vector, destination footprint vector – and returns a destination footprint pointer. While the source pointer points into the source footprint vector entry that contains a footprint belonging to a particular instruction, the operator returns a pointer into the destination footprint vector entry that contains the footprint belonging to the same instruction. A similar operator can be defined for footprint matrices, and we will be overloading the operator to include all combinations of footprint vector and matrices as source and destination.

2.3.2.4 Footprint Pointer Comparison Operator

We define a comparison operator for two footprint pointers, *P1* and *P2*, pointing at two different footprint vector/matrices. Denote the footprint vector/matrix pointed by *P1* and *P2* as *F1* and *F2*. The comparison *P1* > *P2* then becomes *follow_link (P1, F1,* fetch-stage) > *follow_link(P2, F2,* fetch-stage). The > operator can be replaced with any other relational operators (=, ≠, <, ≥, ≤,). For two footprint pointers pointing at the same footprint vector, perform numerical comparison of the pointers, and for the pointers pointing at the same matrix, perform numerical comparison of the row numbers.

2.3.3 HIGH-LEVEL ANALYSIS[1]

We have four high-level analysis techniques:

1) Data-dependency analysis;
2) Control-flow analysis;
3) Data-transfer analysis;
4) Instruction-flow analysis.

Each analysis technique is applied separately, starting from the oldest entry in the footprint vectors/matrices until any inconsistency is sighted. Each inconsistency is represented as a <location, footprint> pair (Sec. 2.3.2.2), where the location element of the pair dictates the entry point into the decision diagram used for the low-level analysis.

If only one of the analysis techniques identifies an inconsistency, then the corresponding entry point into the decision diagram is taken. If none of them discovers an inconsistency, then there is a default entry point into the decision diagram. If multiple of them identify inconsistencies, then there are two options:

1) Perform low-level analysis from each entry point corresponding to each inconsistency separately and then combine the results.
2) Since we are interested in the inconsistency that is closest to the electrical bug manifestation in terms of time, the reported inconsistencies are compared (as defined in Sec.2.3.2.4) to see which one occurred the earliest. The high-level analysis technique with the earliest occurring inconsistency then dictates the entry point into the decision diagram for the low-level analysis.

Our results are reported using the second option. Investigation of the first option is a topic of future research.

---

[1] The high-level analysis has been updated from the version shown in [Park 09] so that the result of the high-level analysis can be directly used by the BLoG framework presented in Chapter 3.

2.3.3.1 Data-dependency Analysis

This analysis technique verifies whether the instruction data dependency order [Shen 05] is preserved; *i.e.*, the analysis verifies that every committed instruction *A* is issued only after all instructions that would produce *A*'s operands finish execution. Let *B* be the producer instruction without loss of generality. Use the fetch-stage-footprint-vector to identify vector entries corresponding to *A* and *B* and denote them with two pointers *Pfa* and *Pfb*. Then perform *follow_link (Pfa,* fetch-stage, issue-stage) to obtain *Pia* to an issue-stage-footprint-matrix entry. Create new pointers, *Pic1…PicN,* where each pointer points to each of the committed footprints that are on the same row as the footprint pointed by *Pia*. *N* is the number of committed footprints on the same row, including the one pointed by *Pia*. Perform *follow_link* ("source pointer", issue-stage, execution-stage) by replacing "source pointer" with each of the *Pic1…PicN,* to obtain *Pe1 … PeN*. For each of *Pea1 … PeaN,* move the pointer by number of rows equal to the latency associated with the execution unit. For example, for an execution-stage footprint matrix corresponding to an ALU with 3-cycle latency, the pointer will be moved up by 3 rows. Then perform *follow_link (Pfb,* fetch-stage, execution-stage*)* to obtain *Peb*. If *Peb* is in the row below any of the *Pea1 … PeaN,* then there is a data dependency violation. A failed check returns <scheduler's issue-ready signal, *Pia*> as the <location, footprint> pair.

As an example, consider Fig. 2.12. Since instruction with ID 0 shown in the fetch-stage footprint vector 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 footprint matrices with single column each). 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 all the issue-stage recorders must be in a

single clock domain (feature 2 of Sec. 2.1), and thus know that the instructions with ID 3 and ID 5 must be issued at the same time (shown in the issue-stage footprint matrix). In this case, there is only two cycles difference between ID 3 and ID 0, 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.

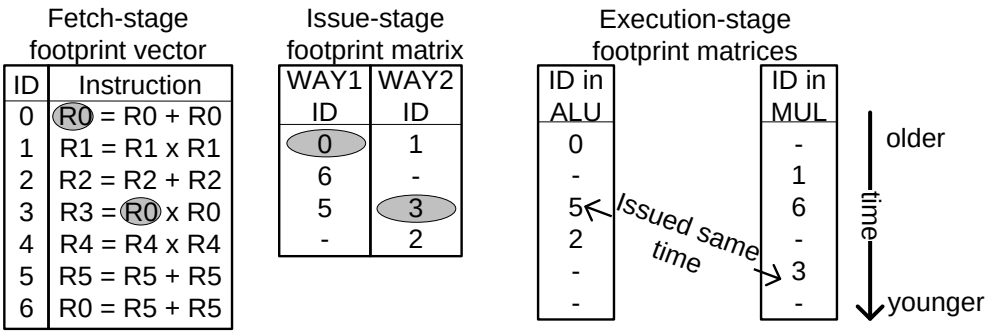

Fig. 2.12. Data-dependency analysis example.

2.3.3.2 Control-flow Analysis

In the program control flow analysis, four types of illegal transitions are searched in the PC sequence of committed instructions in the fetch-stage footprint vector, starting from the oldest PC:

1) The PC does not increment by +4 except in the presence of a control flow transition instruction (branch or jump);
2) The PC increments by +4 in the presence of unconditional transition instruction;
3) The PC neither increments by +4 nor jump to the correct target specified as the immediate in the presence of direct transition instruction (with target address not dependent on a register value);
4) The PC does not jump to an address that is part of the executable address space (determined from the program binary) in the presence of register-indirect transition instructions (with target address that depends on a register value).

With a violation in control flow from instruction A to instruction B, the analysis returns <Program counter register of branch unit, pointer to A's fetch-stage footprint vector entry> as the <location, footprint> pair.

2.3.3.3 Data-transfer Analysis

This technique verifies that a value loaded from a memory address matches with the value stored at that address. The check is performed on the load-store sequence obtained from load-store unit footprint matrix. The checks below 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) ... r[*a*](*n*).

A violation in Data-transfer analysis returns at least one and up to five of the following <location, footprint> pairs:

Pair 1) <Address output of store address queue in load-store unit, pointer to *w*[*a*] in load-store unit footprint matrix>;

Pair 2) <Address output of load queue in load-store unit, pointer to *r*[*a*](*k*) in load-store unit footprint matrix>, where *k* is determined during the checks;

Pair 3) <Data output of store data queue in load-store unit, pointer to *w*[*a*] in load-store unit footprint matrix>;

Pair 4) <Load data output of load-store unit, pointer to *r*[*a*](*k*) in load-store unit footprint matrix>, where *k* is determined during the checks;

Pair 5) <Memory, pointer to *w*[*a*] in load-store unit footprint matrix>;

Pair 6) <Memory, pointer to *r*[*a*](*k*) in load-store unit footprint matrix>, where *k* is determined during the checks.

There are three checks performed by the data-transfer analysis:

1) If for all *i, j, r*[*a*](*i*) = *r*[*a*](*j*) (*i.e.,* all loads were consistent) and for all *i, w*[*a*] ≠ *r*[*a*](*i*), and *n* > 1 then the multiple consistent loads suggest that the store address was incorrect. Return pair 1 listed above. In addition, there is possibility that the memory write/read was incorrect. Since we are only concentrating on processor bugs, this inconsistency is not passed onto low-level analysis, but is appended on

to the list of candidate location-footprint pairs returned by low-level analysis. The inconsistency location-footprint pair is the pair 5 listed above.

2) If $w[a] \neq r[a](i)$, and $n = 1$ then since there is only one load, a bug in the load cannot be ruled out. Return pair 1 and 5 as was done in the previous check, along with 2, 4, and 6, where $k = 1$.
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 $r[a](k) \neq w[a]$, pair 2, 4 and 6.

2.3.3.4 Instruction-flow 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.

Consistency among footprint vectors: Create pointers *P*1, *P*2, and *P*3 to the youngest footprints of the fetch, decode, and dispatch footprint vectors. If *P*1 points to an uncommitted footprint with ID *X*, verify that if P2 does not point to *X*, then *P*3 does not either. A failure returns an erroneous flush inconsistency whose inconsistent footprint is the next older flush-causing footprint in the dispatch footprint vector. If *P*1 points to a committed footprint *X*, verify that both *P*2 and *P*3 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 older entries of the respective footprint vectors. If a pointer cannot be incremented, stop.

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.

Consistency along the vector-matrix boundary (dispatch-stage vector and issue-stage matrix). Algorithms similar to that of Algorithm A in Appendix A are used to perform the following checks:

For each committed instruction in the issue-stage matrix, the algorithm checks that it appears in the dispatch-stage footprint vector as well. A failure returns a missing footprint inconsistency whose inconsistent footprint is *X*'s footprint in the issue-stage footprint matrix.

When performing Step 6 of Algorithm A.3 in Appendix A (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.

2.3.4 LOW-LEVEL ANALYSIS[2]

Given the inconsistency, in the form of a <location, footprint> pair, returned from the high-level analysis, the low-level analysis asks a series of microarchitecture-specific questions according to a manually-generated decision diagram (an example of which is shown in Appendix B). The location-element of the inconsistency decides the entry point into the decision diagram. While going through the decision diagram, the location and footprint elements are both updated to find the <location, footprint> pairs associated with the final bug candidates. The location element of a bug candidate indicates a possible design block containing the flip-flop that was first affected by an electrical bug, while the footprint element indicates the cycle in which the event took place.

2.3.5 BUG-EXPOSING STIMULUS

Bug-exposing stimuli are derived from footprint elements of bug candidates. Denote the footprint pointer of a bug candidate as *Pbug* and the footprint vector/matrix pointed by *Pbug* as *Vbug.* Performing *follow_link (Pbug, Vbug,* fetch-stage) returns a pointer partitioning the fetch-stage footprint vector into two parts. Older footprints correspond to the bug exposing stimulus, which is a trace of instructions leading up to the cycle in which the bug first caused an error in a flip-flop. Younger footprints correspond to trace of instructions responsible for propagating the error to an observable output.

---

[2] We only briefly mention the low-level analysis here, because Chapter 3 describes a framework for systematically constructing a low-level analysis decision diagram and a method for traversing it.

2.4 RESULTS

We evaluated IFRA by injecting errors into a microarchitectural simulator [Austin 02] augmented with IFRA. We used an Alpha 21264 configuration (4-way pipeline, 64 maximum instructions in-flight, 2 ALUs, 2 multipliers, 2 load/store units), which gave 200 different locations (excluding array structures and arithmetic units since errors inside those structures are immediately detected and localized using parity and residue codes, as discussed in Sec. 2.2.3). Each location has an average size equivalent of 10K 2-input NAND gates. Seven benchmarks from the SPECint2000 benchmark suite (bzip2, gcc, gap, gzip, mcf, parser, vortex) were chosen as validation test programs as they represent different types of workloads. Each recorder was sized to have 1,024 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 [McLaughlin 09].

Errors were injected in one of 1,191 flip-flops (Table 2.3). No errors were injected in structures protected with built-in parities/residues for error detection. Errors were injected in input / output registers and various control registers controlling the array structures. Pipeline registers in Table 2.3 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.

Table 2.3. 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

Upon error injection, the following scenarios are possible:

Case 1) The error is masked and causes no system-level effect [Wang 04].

Case 2) The error is silent in that it does not cause any post-trigger mechanism to trigger, but the program output is incorrect [Wang 04].

Case 3) Failure manifestation with short error detection latency, in which case the recorders successfully capture the history from error to failure manifestation (including situations where recording is paused upon activation of soft post-triggers).

Case 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 coverage of validation test programs and post-triggers, and are not the focus of this paper. Any error injection runs which does not result in the activation of any post-trigger within 100,000 cycles from the point of error injection was repeated with a new error injection run.

When case 3 occurs, IFRA-based post-analysis is performed to obtain a set of bug candidates in the form of <location, footprint> pairs. Each pair indicates a design block – which contains the flip-flop that is thought to be flipped in value – together with a cycle in which the bit-flip is thought to have occurred, relative to the cycle in which the system failure occurred.

Out of 100,000 error injection runs, 800 of them resulted in Cases 3 and 4. Table 2.4 presents results from these two cases. The "*exactly localized*" category represents the cases in which IFRA returned a single and correct <location, footprint> pair. The "*multiple candidates*" category represents the cases in which IFRA returned multiple bug candidates and at least one pair was fully correct in both location and in cycle. 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 2.4 that a large percentage of bugs were uniquely located to correct <location, footprint> pair, while very few bugs were completely missed, demonstrating the effectiveness of IFRA.

For "multiple candidates" cases, Table 2.4 also reports statistics on the number of possible candidates out of a total of 200,000 possible candidate <location, footprint> pairs. When IFRA identified multiple candidates, on average, it correctly dismissed 99.8% of the possible bug locations.

Table 2.4. IFRA bug localization summary.

Result category	**Percentage**
Exactly Localized	75%
Correctly Localized with multiple candidates	21% min. 2, avg. 6, and max. 34 candidates
Completely Missed	4%

2.5 RELATED WORK

Related work on post-silicon validation can be broadly classified into the following categories: formal methods [De Paula 08], embedded trace buffers for hardware debugging [Abramovici 06], on-chip program and data tracing [MacNamee 00], clock manipulation [Josephson 06], scan dump [Caty 05], check-pointing with deterministic replay [Silas 03][Sarangi 06], and on-line assertion checking [Abramovici 06][Bayazit 05][Chen 08]. Table 2.5 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 [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 executed software. If one can assume perfect hardware, capturing the signals at the asynchronous interfaces is sufficient to reconstruct all the internal signals using simulation [Xu 03]. 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.

On-line 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 post-analysis techniques supported by IFRA.

Table 2.5. IFRA vs. existing techniques.

<table>
<tr><th>Techniques</th><th>Intrusive?</th><th>Failure reproduction?</th><th>System-level simulation?</th><th>Area impact?</th><th>Applicability?</th></tr>
<tr><td>Formal methods</td><td>(+) No</td><td rowspan="6">(-) Yes</td><td>(+) No</td><td rowspan="2">(-) Yes</td><td rowspan="4">(+) SoC</td></tr>
<tr><td>Trace buffer</td><td>Depends</td><td rowspan="5">(-) Yes</td></tr>
<tr><td>Scan methods</td><td rowspan="2">(-) Yes</td><td>(+) No</td></tr>
<tr><td>Clock manipulation</td><td>(-) Yes</td></tr>
<tr><td>Program & data tracing</td><td>Depends</td><td rowspan="2">(+) No</td><td rowspan="2">(-) Processor</td></tr>
<tr><td>Checkpoint & replay</td><td>(-) Yes</td></tr>
<tr><td>Assertion checking</td><td>Depends</td><td>Depends</td><td>(+) No</td><td>(-) Yes</td><td>(+) SoC</td></tr>
<tr><td>IFRA</td><td>(+) No</td><td>(+) No</td><td>(+) No</td><td>(-) 1%</td><td>(-) Processor</td></tr>
</table>

2.6 CONCLUSIONS

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

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 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 multi-core systems, and system-on-chips (SoCs) consisting of non-processor designs.
2) IFRA does not currently support simultaneous multi-threaded processors [Shen 05].
3) Bugs that may only cause performance slowdown but not critical for correct program runs are not targeted.
4) Sensitivity analysis and characterization of the inter-relationships between post-analysis techniques, architectural features, error detection mechanisms, recorder sizes, and bug types are yet to be performed.

CHAPTER 3. APPLICATION OF IFRA USING BLoG

While the concept of IFRA is applicable to any processor microarchitecture, the manual effort required to correctly **hand-craft** a new decision diagram for a new microarchitecture limits the efficiency of IFRA. This chapter presents the *Bug Localization Graph* (*BLoG)* framework for systematically constructing and automatically executing IFRA's offline microarchitecture-dependent analysis, enabling IFRA to be applied to new microarchitectures with reduced engineering time and less expert knowledge.

A BLoG is an abstract graphical representation of a processor that exposes structural information (*i.e.*, connections between microarchitectural blocks) so that microarchitecture-dependent self-consistency checks can be performed on scanned-out footprints (*i.e.*, no system-level simulation) to localize bugs. A BLoG consists of a set of *BLoG nodes* and *edges*, representing various hardware blocks and inter-block connections, respectively (Sec. 3.1).

Using the BLoG framework, devising the microarchitecture-dependent low-level analysis is equivalent to constructing (Sec. 3.2) a special directional graph – the BLoG – and performing the analysis is equivalent to traversing the BLoG using special graph traversal rules (Sec. 3.3).

Fig. 3.1 shows the bug localization flow using BLoG-assisted IFRA vs. the original IFRA. After inserting recorders inside the chip, a BLoG is constructed using the chip's microarchitectural description. Then the same flow of the original IFRA is followed until the high-level analysis performed after a system failure. The high-level analysis returns an inconsistency (Sec.2.3.3), in the form of a <location, footprint> pair (Sec. 2.3.2.2), which is used together with the linked footprints (Sec. 2.3.2) to traverse the BLoG to obtain bug candidates, which is again, in the form of <location, footprint> pairs.

Sec. 3.1 introduces the components of BLoG. Sec. 3.2 and Sec. 3.3 describe BLoG construction and traversal methods respectively. Sec. 3.4 describes construction results for a complex commercial microarchitecture followed by related work in Sec. 3.5 and conclusions in Sec. 3.6.

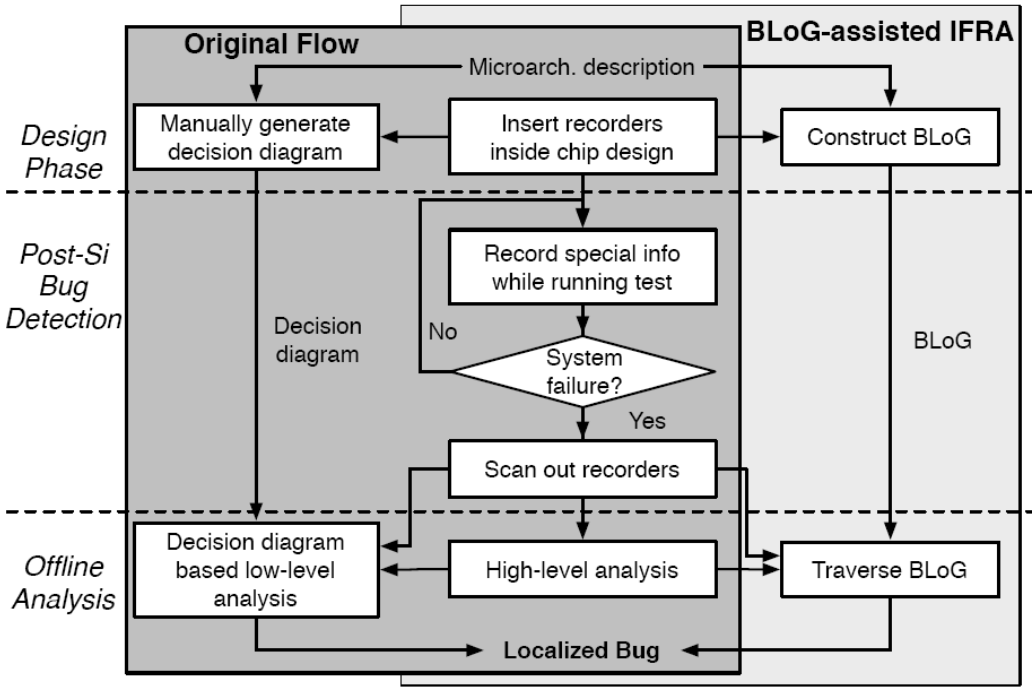

Fig. 3.1. Bug localization flow using BLoG-assisted IFRA vs. original IFRA.

3.1 BLOG COMPONENTS

A BLoG consists of a set of BLoG nodes and edges. BLoG nodes are abstract representation of hardware structures, capturing its functionality for the purpose of performing microarchitecture-dependent-self-consistency checks. BLoG edges are directional and are abstract representation of data or control signal values communicated between the BLoG nodes. The edges themselves do not represent any physical hardware and has zero delay. Bidirectional communication is modeled using two separate directional edges. Each node has one or more incoming and outgoing edges (*i.e.*, no sink or source).

3.1.1 BLOG NODE TYPES

BLoG nodes can be one of eight exclusive types (Fig. 3.2), which can be broadly classified into *storage types* (1-3) and *non-storage types* (4-8). Storage types model hardware structures having variable propagation delays from data entry and exit, while non-storage types model hardware structures having fixed delay, between data entry and exit, known at design time. We provide a brief explanation of each BLoG node type below.

1) **Random-access***:* Storage with index-based addresses (*e.g.*, register file, register alias table) for individual entries.
2) **Associative***:* Storage with associative access with explicit tags (*e.g.*, cache, branch target buffer, reservation station, TLB) for individual entries.
3) **Queue**: Storage with first-in-first-out entry management (*e.g.*, re-order buffer, load queue, store data queue, store address queue, instruction queue).
4) **Modifying**: Non-storage structure that modifies input values, producing different output values (*e.g.*, decoder, address generator, comparator, ALU).
5) **Connection**: Non-storage structure that propagates input values to output values without modification (*i.e..*, series of pipeline registers).

6) **Select**: Non-storage structure that takes two input values and chooses one as an output value (*e.g.*, forwarding path, register/immediate select, next-PC select, instruction select).
7) **Protected**: Modifying or Connection type with built-in error detection mechanisms such as parity bits for arrays and residue codes for arithmetic units. The Protected type is not necessary if such error detection techniques are not present.
8) **Default**: Any non-storage structure not included in the aforementioned types *(e.g.*, scheduler, load/store snoops, load replay handler).

All three storage types and the Connection type have a *Clear* input signal, which flushes the content of the hardware structures upon assertion. Each of the seven non-default types is associated with special rules that perform microarchitecture-dependent self-consistency checks, which will be described later in Sec. 3.3.2.

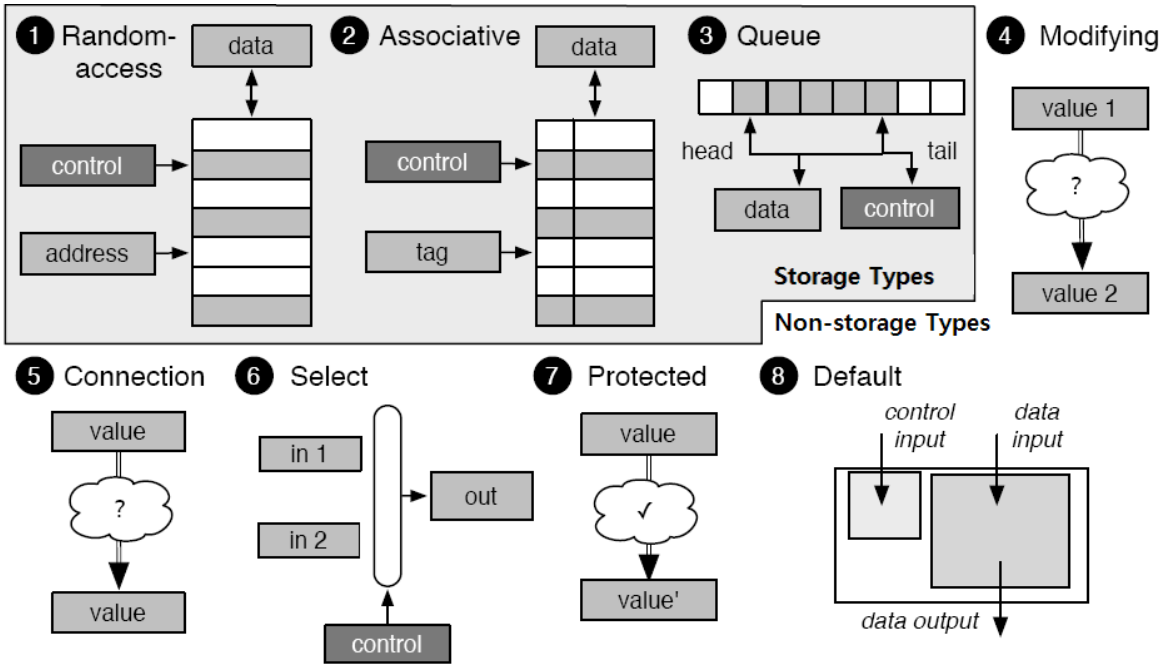

Fig. 3.2. Eight BLoG node types.

3.1.2 BLOG EDGE ATTRIBUTES

A BLoG edge has five attributes:

1) **A footprint vector/matrix**: specifies which vector/matrix to look up in order to obtain the data or control signal values on the edge;
2) **An *auxiliary-information-field selector***: specifies which auxiliary information field (Table 2.1, Table 3.2) in the vector/matrix to look up;
3) **A footprint pointer**: specifies which entry in the vector/matrix to look up;
4) **Set of <edge, *edge dependency*> pairs**: details described in a later paragraph;
5) **Data or control signal values**.

Fig. 3.3 describes the relationships between the attributes. The first, second and the fourth attributes are manually specified during BLoG construction. During BLoG traversal, footprint pointers are automatically obtained using the first, second and the fourth attributes, while the data or control signal values are automatically obtained using the first three attributes.

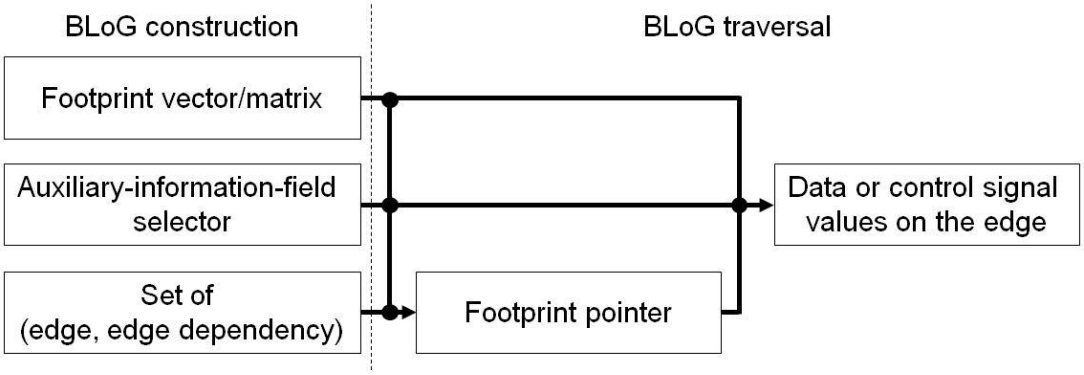

Fig. 3.3. Relationship between BLoG edge attributes.

When obtaining a footprint pointer for an edge during BLoG traversal, we come to a situation where we are given a footprint pointer for an outgoing edge of a node and have to derive the footprint pointer for an incoming edge of the same node. For this purpose, we will be using edge dependencies, which describe relationship between an outgoing edge of a node and an incoming edge of a node. There are six edge dependency types:

1) **Same instruction**: The data/control signal values on the outgoing and incoming edges belong to the same instruction (*e.g.,* a multiplexer takes opcode of an instruction as a control input value and selects either a register value or the immediate value of the same instruction);
2) **Same architectural register name**: The data/control signal values on the outgoing and incoming edges belong to the same architectural register (*e.g.,* a forwarding path takes a register value produced by one instruction and passes it on to another instruction that uses the same register name as an operand);
3) **Same physical register name**: The data/control signal values on the outgoing and incoming edges use the same physical register name (*e.g.,* ROB uses physical register name as index to access speculative register states);
4) **Same memory address**: The data/control signal values on the outgoing and incoming edges use the same memory address (*e.g.,* a cache uses memory addresses as tags to perform associative access);
5) **Pipeline flush**: The data/control signal value on the incoming edge is a pipeline flush event (*e.g.,* an instruction queue is flushed by a pipeline flush event);
6) **Default:** Any relationship not included in the aforementioned types, and includes the case where there are no relationships.

An <edge, edge dependency> pair for an edge *e*, consists of an outgoing edge from the node for which *e* act as an incoming edge, and one of the five edge dependency types. An edge could have multiple pairs. For example, in Fig. 3.4, edge *e2* has a single pair (<*e3, c23*>), while edge *e3* has two pairs (<*e4, c34*>, <*e5, c35*>). The Associative-type is the only non-default node type with multiple outgoing edges. The method for obtaining footprint pointers using these pairs is described later in Sec. 3.3.1.

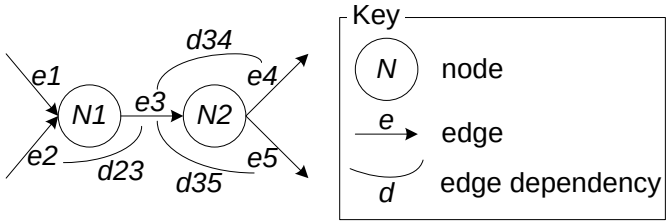

Fig. 3.4. Example <edge, edge dependency> pairs.

3.2 BLOG CONSTRUCTION

BLoG construction involves defining nodes and edges using the following two inputs:

- **Microarchitecture description of a processor**: It could be in the form of a microarchitectural block diagram (*e.g.,* from an architectural manual [Colwell 05]) or a language based specification (*e.g.,* EXPRESSION [Halambi 99], Generic netlist representation [Gorjiara 06]);
- **Recorder field description**: It specifies what each on-chip recorder is recording (*e.g.,* Table 2.1, Table 3.2).

Node Definition: Given chip design is manually decomposed with an objective of having as many partitions as possible, but with two constraints: (1) Most partitions should fall under the seven non-default types (Sec.3.1.1); (2) Most resulting inter-partition connections, which later become BLoG edges, should have at least one <edge, edge dependency> pair with non-default edge-dependency type (Sec. 3.1.2). More partitions enable fine-grained localization – but it may not be possible to derive all data or control signals on the edges due to reduced number of edges having at least one non-default-type edge dependency, which may reduce bug localization accuracy. After partitioning, each partition is assigned one of eight exclusive node types. Any partitions with variable propagation delays from data entry and exit will take one of the three storage types depending on how the entries are managed. For the rest, partitions with any protection mechanisms are assigned the Protected type, while partitions that consist of series of pipeline registers are assigned the Connection type. The Modifying and Select type are used for the remaining partitions with clear data inputs/outputs defined. All other partitions will fall under the Default type nodes.

Edge Definition: BLoG edges are defined between partitions according to inter-block connections. In addition, three out of five edge attributes (Sec. 3.1.2) are manually assigned in this phase: auxiliary information field selector, footprint vector/matrix and <edge, edge dependency> pairs.

Although the construction is described as an open-loop process, there are several possible places to introduce feedback. For example, there can be a feedback between BLoG construction and deciding what to record in each of the on-chip recorders. Selecting more signals to eliminate some of the default-type edge dependencies will improve bug localization accuracy at the expense of having more chip area impact. Another feedback can be introduced between node and edge definitions. Since one of the constraints on the node definition is that there should be few edges with default-type edge dependencies, partitioning can be changed so that the default-types go away.

Although hand-crafted new node or edge-dependency types can be used to replace default-type nodes and edge dependencies, it is still possible to traverse a BLoG without the aid. Sec.3.3.2 presents the traversal method in the presence of default type nodes and edge dependencies.

3.3 BLOG TRAVERSAL

Traversing a BLoG effectively performs microarchitecture-dependent-self-consistency checks on the recorded data. The traversal takes three inputs: the BLoG itself, linked footprints (Sec. 2.3.2), and an inconsistency found during the high-level analysis (Sec. 2.3.3). The inconsistency is in the form of <location, footprint> pair as was mentioned in Sec.2.3.2.2.

The location element of the pair decides the *starting edge*, which is an outgoing edge of a BLoG node, where the traversal begins. For example, consider Fig. 3.5, which shows a block diagram for a branch unit on the left, and a BLoG equivalent on the right. An inconsistency found during the control-flow analysis (Sec.2.3.3.2) would return a program counter register, which is associated with an outgoing edge of a Connection-type node shown in Fig. 3.5. The footprint element in the inconsistency is used as the footprint pointer for the starting edge.

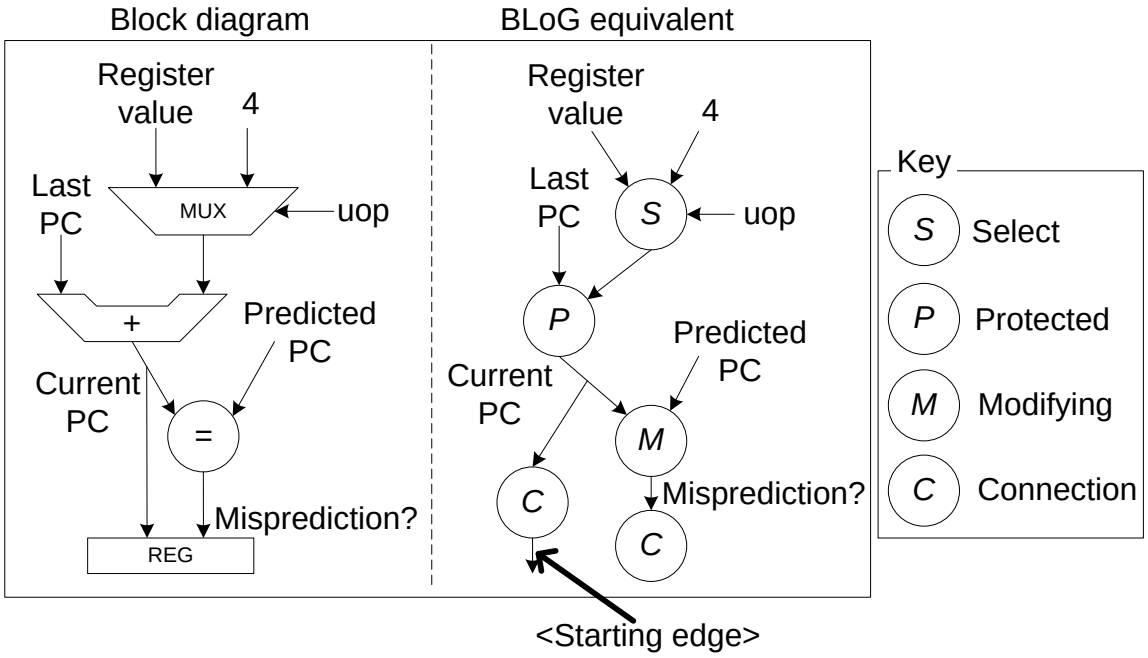

Fig. 3.5. Example starting edge for a control-flow analysis inconsistency.

As shown in Fig. 3.6, BLoG traversal begins by placing an *error label* on the starting edge. When an error label is placed on an outgoing edge, a node traversal takes place as summarized in Fig. 3.7. A set of *footprint-propagation rules* (Sec. 3.3.1) and *location-propagation rules* (Sec. 3.3.2) associated with that node are executed. A footprint-propagation rule, using the footprint pointer of the outgoing edge and <edge, edge-dependency> pairs associated with each incoming edge, derives the footprint pointers of all the incoming edges. Using these pointers, the data or control signal values are obtained by looking up in the footprint vector/matrices specified for each incoming edge. After which, the location-propagation rule decides one or both of the following:

- ***Error localization***: The error may originate from the node itself. The location-propagation rules further subdivide each node into one or more *localization regions,* each representing different component of the hardware structure modeled by the node (individual localization regions described in Sec. 3.3.2). If the error is from a region, it is marked with a candidate label and a footprint pointer, determined by the rules, is noted on the label.
- ***Error Propagation***: The error may originate from some other predecessor node. In this case, one or more new error labels are placed on the incoming edges of the node and a footprint pointer determined by the rules is noted on the label.

New error labels can be created on one or more incoming edges after a node traversal. The backward traversal continues by performing node traversal from each of the error labels on the node for which the incoming edges act as outgoing edges. When multiple error labels are present, breadth-first traversal is performed; error label with the oldest footprint pointer (as determined using the comparison operator defined in Sec.2.3.2.4) is propagated first. This algorithm holds even in the situations where multiple error labels are present across single or multiple outgoing edges of a single node.

A BLoG traversal terminates once one of the two stopping criteria is met:

1) The number of candidate label exceeds a pre-defined candidate limit;
2) The number of error label reaches zero.

After a BLoG traversal completes, all the candidate labels are returned to report the set of bug candidates in the form of <location, footprint> pairs as was done for the original IFRA.

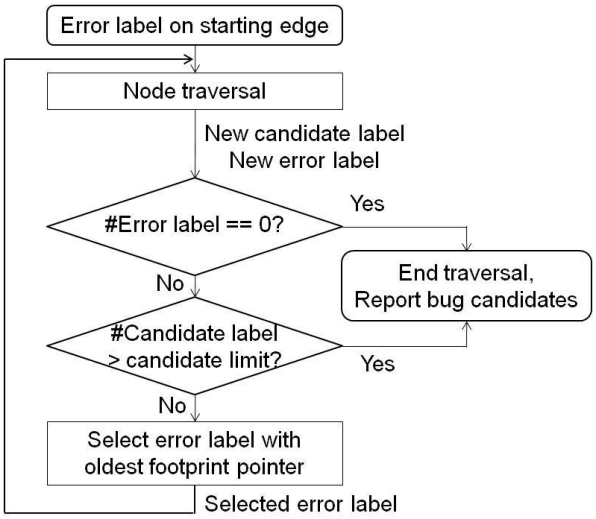

Fig. 3.6. BLoG traversal flow chart.

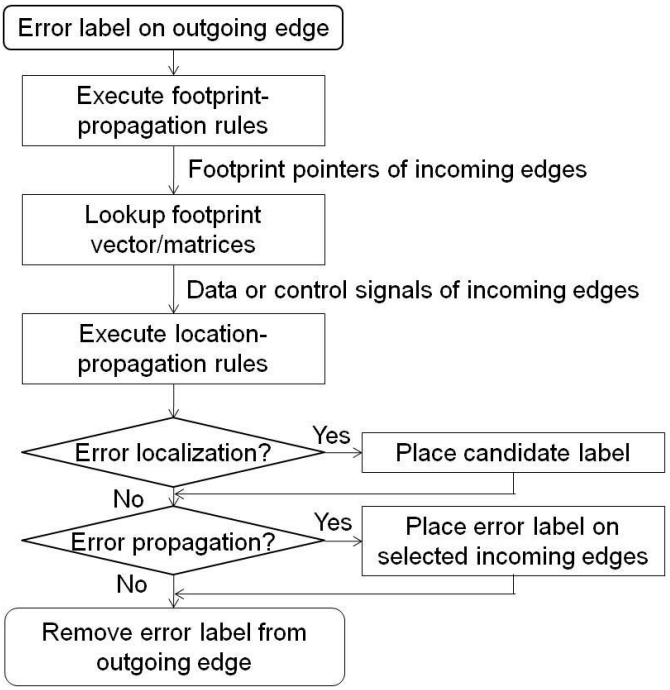

Fig. 3.7. Node traversal flow chart.

3.3.1 FOOTPRINT-PROPAGATION RULES

Footprint-propagation rules are used to obtain the footprint pointers for all the incoming edges when an outgoing edge already has a footprint pointer assigned. Each edge has five attributes (Sec.3.1.2): 1) a footprint vector/matrix; 2) an auxiliary-information-field-selector; 3) a footprint pointer; 4) set of <edge, edge dependency> pairs; and 5) data or control signal values. Note that only a single <edge, edge dependency> pair is relevant because only one of them specifies the edge dependency relationship with the outgoing edge under consideration. Denote the first and third attributes of an incoming edge as *Vi* and *Pi*, and those of the outgoing edge as *Vo* and *Po*. There are six footprint-propagation rules, each associated with a single edge dependency:

Same instruction: Perform *follow_link (Po, Vo, Vi)* to obtain *Pi.*

Same architectural register: Perform *follow_link (Po, Vo,* fetch-stage*)* to obtain a new pointer into a fetch-stage vector entry, *Pf*. Denote the architectural destination register used by the instruction pointed by *Pf* as *Rd*. Decrement *Pf* towards older entries until an instruction that uses *Rd* as its operand is found. Perform *follow_link (Pf,* fetch-stage, *Vi)* to obtain *Pi.*

Same physical register: The dispatch and allocation pipeline stages record the residues of physical register names for Alpha (Table 2.1) and Intel Core i7 microarchitecture (Table 3.2) respectively. When we are trying to compare residues, there is a possibility of aliasing. Thus, we return *N* number of *Pi's*, where *N* can be configured, so that multiple error labels are created to perform multiple traversals. For our purpose, we used $N = 5$. However, storing the entire physical register names, as opposed to residues, will eliminate this problem, at the expense of increasing the recorder storage requirement by less than 2%. The footprint-propagation rule starts by performing *follow_link (Po, Vo,* Dispatch/Allocation-stage) to obtain a new pointer, *Pa*. Denote the physical destination register used by the footprint pointed by *Pa* as *Rd*. Decrement *Pa* towards older entries until a footprint that use *Rd* as its operand is found. Perform *follow_link(Pa,*

Dispatch/Allocation-stage, *Vi)* to obtain the first *Pi*. Decrement *Pa* again towards older entries until a footprint that uses *Rd* as its operand is found. Perform *follow_link* again to obtain the second *Pi*. Repeat the *Pa* decrement and *follow_link* until *N* number of *Pi's* are found.

Same memory address: The memory-unit (in the execution-stage) footprint matrix contains the memory addresses as its auxiliary information. The footprint-propagation rule starts by performing *follow_link (Po, Vo,* memory-unit) to obtain a new pointer, *Pm*. Denote the memory address used by the footprint pointed by *Pm* as *MA.* Decrement *Pm* until a footprint that uses *MA* as memory address is found. Perform *follow_link (Pm,* memory-unit, *Vi)* to obtain *Pi.*

Pipeline flush: Since all the footprints corresponding to flush-causing instructions are discovered during footprint linking (Sec. 2.3.2), decrement *Po* until a flush-causing instruction is found. Perform *follow_link (Po, Vo, Vi)* to obtain *Pi.*

Default: Unless customized footprint-propagation rules are designed and used, assign NULL to *Pi* and state that no information is available. Methods for handling default-type edge dependencies are presented in the next section.

In case the end of recorded history is reached (*i.e.,* the footprint the rule is looking for was overwritten during the test run due to the limited storage) while performing *follow_link( )* or decrementing pointers, return NULL for *Pi* and state that the end of history is reached.

3.3.2 LOCATION-PROPAGATION RULES

Given a node with data or control signals on incoming and outgoing edges, location-propagation rules are responsible for propagating error labels from outgoing to incoming edge(s) and/or creating candidate labels on one of its localization regions. As a result, there are two possible scenarios:

- ***Error Containment*:** Given enough information, a location-propagation rule can have high confidence in localizing or propagating errors correctly. When such situation arises, all error and candidate labels on the BLoG are discarded except the ones produced by this particular rule.
- **Not enough information**: For the rest of the situations, we conservatively continue the node traversal, along with other error labels.

Each rule will require some, if not all, of the incoming edges to have their data or control signals. However, some incoming edges will not have any data or control signals because their footprint pointers have a NULL value. There are two possible reasons:

1) The end of recorded history is reached. In this case, any rule using this particular incoming edge will not produce any error or candidate labels. This scheme allows the rest of the error labels in the BLoG to complete their traversal while terminating the current node traversal.
2) Not all the edge attributes are specified or a Default-type edge dependency is used. In this case, we make a pessimistic decision. Any rule using this particular incoming edge will produce an error label on this edge with a footprint pointer obtained by following link from the pointer on the outgoing edge to the footprint vector/matrix assigned to the incoming edge. In addition, if the rule is capable of placing a candidate label, then the label is always placed with the pointer on the outgoing edge.

The set of location-propagation rules associated with each of the node types are presented next. Since the Protected type only requires small modification from the Connection or Modifying type, the rules for the Protected type are shown with the two types.

3.3.2.1 Connection Type

It has two incoming edges –input data (*I*), and clear signal (*C*) – and an outgoing edge, output data (*O*). Input and output data edges are assumed to be in a single clock domain. It models a series of pipeline registers that propagates values on the data input edge to the data output edge after some fixed number of cycles (*D*). When a clear signal is asserted, the contents of the pipeline registers are discarded. The node has a single localization region, representing the whole node. The location-propagation rules are shown below.

Location-propagation rule outcomes:

Outcome 1) Data output value does not match with data input value. Place a candidate label on the node with *Po* (defined later in the algorithm) as the footprint (Contained error).

Outcome 2) Clear signal did not discard the contents of the pipeline registers. Place a candidate label on the node with *Pc* as the footprint (Contained error).

Outcome 3) No error in the node, place an error label on the data input edge.

Location-propagation rule algorithm:

We denote the following:

- Footprint vectors/matrices assigned to *C, I, O* as *Fc, Fi, Fo*;
- Footprint pointers on *C, I, O* as *Pc, Pi, Po;*
- Data values pointed by *Pi*, *Po* as *Vi*, *Vo.*

The algorithm for executing the location-update rule is shown below.

Step 1) If ($Vi \neq Vo$) then return “Outcome 1”; Halt;

Step 2) If ((*follow_link (Pc, Fc, Fo) < Po)* and (*follow_link (Pc, Fc, Fi) > Pi))* then return “Outcome 2”; Halt;

Step 3) Return “Outcome 3”; Halt;

A Protected-connection-type node shares the same location-propagation rule as a Connection-type node, except that Step 1 is skipped, because “Outcome 1” cannot occur.

3.3.2.2 Modifying Type

It has an incoming edge – data input (I) – and an outgoing edge, data output (O). All edges are assumed to be within a single clock domain. It models a hardware structure that modifies its input data value into a different output data value. The node has a single localization region, representing the whole node. The location-propagation rules are shown below.

Location-propagation rule outcomes:

Outcome 1) Data output value was not correctly modified. Place a candidate label on the node with Po1 (defined in the algorithm shown below) as the footprint (Contained error).

Outcome 2) Data output value was not correctly modified, but do not know when the wrong modification happened. Place two candidate labels on the node with Po1 and Po2 (defined in the algorithm shown below) as the footprints (Contained error).

Outcome 3) No error found in the node, place an error label on the data input edge.

Outcome 4) Not enough information, place an error label on the data input edge and place a candidate label with Po as the footprint.

Location-propagation rule algorithm:

We denote the following:

- Footprint pointers on *I, O* as *Pi*1, *Po*1;
- New footprint pointers as *Pi*2, *Pi*3, *Po*2, *Po*3;
- Data values pointed by *Pi*1, *Pi*2, *Pi*3, *Po*1, *Po*2, *Po*3 as *Vi*1, *Vi*2, *Vi*3, *Vo*1, *Vo*2, *Vo*3.

The algorithm for executing the location-update rule is shown below.

Step 1) Assign *Pi*1, *Po*1 to *Pi*2, *Po*2 ;

Step 2) Decrement *Pi2* and *Po2*, in a circular fashion until *Vi2* is not an idle cycle and belongs to a committed instruction;

Step 3) If (*Pi*1 = *Pi*2) then "Outcome 4"; Halt;

Step 4) If (*Vi*1 = *Vi*2) and (*Vo*1 = *Vo*2) then "Outcome 3"; Halt;

Step 5) If (*Vi*1 = *Vi*2) and (*Vo*1 ≠ *Vo*2) then Goto Step 7;

Step 6) Goto Step 2;

Step 7) Assign *Pi*2, *Po*2 to *Pi*3, *Po*3;

Step 8) Decrement *Pi*3 and *Po*3, in a circular fashion until *Vi3* is not an idle cycle and belongs to a committed instruction;;

Step 9) If (*Pi*1 = *Pi*3) then outcome 4; Halt;

Step 10) If (*Vi*1 = *Vi*3) and (*Vo*1 = *Vo*3) then assign *Po*2 to *Po*1; "Outcome 1"; Halt;

Step 11) If (*Vi*1 = *Vi*3) and (*Vo*1 ≠ *Vo*3) then "Outcome 1"; Halt;

Step 12) Goto Step 8;

A Protected-Modifying-type node will only have "Outcome 3", *i.e.,* no error can occur in the node itself.

3.3.2.3 Select Type

A Select-type node consists of a multiplexer with logic driving the select decision (Fig. 3.8). The first rule checks whether the data output value matches any of the data input values. The second rule checks whether there are multiple instances of current control input values by searching footprints in the recorder assigned to the control input edge. If multiple instances are found, the rule checks whether same select decisions were consistently made at all times. If these two rules do not find a problem, then the error label is propagated to the control input edge and the selected data input edge (determined by the output value).

The algorithm is shown below. We denote the two data input edges as X and Y, the data output edge as Z, and control input edge as C. Denote footprint pointers assigned to C, Z, X, Y as Pc1, Pz1, Px, Py. Denote temporary footprint pointer variables as Pc2, Pz2, Pz3. Denote signal values obtained using Pc1, Pc2, Pz1, Pz2, Px and Py as Vc1, Vc2, Vz1, Vz2, Vx, Vy. Note that the decrement operator moves the pointer towards older entries in the recorder and then wraps around. There are three outcomes.

Outcome 1) Candidate label on the multiplexer with Pz1 as the footprint.

Outcome 2) Candidate label on the logic driving the select decision. For this case, there can up to two candidate labels with different footprints. The algorithm specifies the footprints.

Outcome 3) No error in the node. Create error label on the control input edge and the selected data input edge with Pc1 and Px as the footprints.

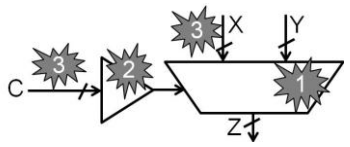

Fig. 3.8. Location-propagation rules for a Select-type node.

```
IF (Vz1≠Vx) AND (Vz1≠Vy) THEN Outcome 1;
ELSE{
  Pc2 = Pc1; Pz2 = Pz1;
  DO{ Decrement Pc2, Pz2, Px, Py;
      IF (Pc1=Pc2) THEN
      Outcome 2 with Pz1 and Outcome 3;
  }WHILE (Vc1≠Vc2);
  IF (Vz1=Vx AND Vz2=Vy) THEN{
    Pz3 = Pz2;
    DO{ Decrement Pc2, Pz2, Px, Py;
        IF (Pc1=Pc2) THEN
          Outcome 2 with Pz1 and Pz3;
    }WHILE (Vc1≠Vc2);
    IF(Vz1=Vx)AND(Vz2=Vy)THEN
      Outcome 2 with Pz1;
    ELSE outcome 2 with Pz3;}
  ELSE Outcome 3;}
```

3.3.2.4 Queue Type

A Queue-type node has two incoming edges – input data (*I*), clear signal (*C*) – and an outgoing edge, output data (*O*). When a clear signal is asserted, the contents of the queue are discarded. The contents of the queue are assumed to be protected using parity. The queue has *N* entries. The node has three localization regions: enqueue circuitry, dequeue circuitry and FIFO (first in first out) management. The location-propagation rules are shown below.

Location-propagation rule outcomes:

Outcome 1) Data output value does not match with data input value. Place a candidate label on enqueue circuitry and dequeue circuitry, each with *Pi* and *Po* (defined later in the algorithm) as their respective footprints (Contained error).

Outcome 2) Clear signal did not discard the contents of the storage. Place a candidate label on the FIFO management with *Pc* (defined later in the algorithm) as the footprint (Contained error).

Outcome 3) Data entry dropped due to a problem in FIFO management or a buffer overflow. Place a candidate label on FIFO management with *Pi* as the footprint (Contained error).

Outcome 4) Data entry created spontaneously due to a problem in FIFO management. Place a candidate label on FIFO management with *Po* as the footprint (Contained error).

Outcome 5) No error in the node, place an error label on the input data edge.

Location-propagation rule algorithm:

We denote the following:

- Footprint vectors/matrices assigned to *C, I, O* as *Fc, Fi, Fo*;
- Footprint pointers on *C, I, O* as *Pc, Pi, Po*;
- Data values pointed by *Pi, Po* as *Vi, Vo*;
- New integer variable as *count*.

The algorithm for executing the location-update rule is shown below.

Step 1) If ($Vi \neq Vo$) then return “Outcome 1”; Halt;

Step 2) If (($follow\_link\ (Pc, Fc, Fo) < Po$) and ($follow\_link\ (Pc, Fc, Fi) > Pi$)) then return “Outcome 2”; Halt;

Step 3) Assign 0 to *count*;

Step 4) Decrement *Po* until *Vo* is not an idle cycle and belongs to a committed instruction (as determined in Sec. 2.3.2);

Step 5) Decrement *Pi* until *Vi* is not an idle cycle and belongs to a committed instruction;

Step 6) Increment *count;*

Step 7) If *count* = N, return “Outcome 5”; Halt;

Step 8) If $follow\_link\ (Po, Fo, Fi) = Pi$ then Goto Step 4;

Step 7) If $follow\_link\ (Po, Fo, Fi) < Pi$ then return “Outcome 3”; Halt;

Step 8) If $follow\_link(Pi, Fi, Fo) < Po$ then return “Outcome 4”; Halt;

3.3.2.5 Random-access Type:

A Random-access type node has four incoming edges – read address (*Ar*), write address *(AW)*, write data *(W)* and clear signal (*C)* – and an outgoing edge, read data (*R).* We assume *Ar* and *R* to be in a single clock domain and *Aw* and *W* to be in a single clock domain. When a clear signal is asserted, the contents of the storage are discarded. The contents of the storage are assumed to be protected using parity. The node has three localization regions: read circuitry, write circuitry and clear handler. The location-propagation rules are shown below.

Location-propagation rule outcomes:

Outcome 1) Either data was written into a wrong address or, a wrong address was used in the first place. Create error label on *Aw* with footprint pointer *Paw* (Defined later in the algorithm), and create a candidate label on the write circuitry with footprint pointer *Pw* (Contained error).

Outcome 2) Either data was read from a wrong address or, a wrong address was used in the first place. Create error label on *Ar* with footprint pointer *Par* (Defined later in the algorithm), and create a candidate label on the read circuitry with footprint pointer *Pr* (Contained error).

Outcome 3) Clear signal did not discard the contents of the storage. Place a candidate label on the clear handler with *Pc* (defined later in the algorithm) as the footprint pointer (Contained error).

Outcome 4) Not enough information. Create error label on both *Aw* and *Ar* with footprint pointers *Paw, Par,* respectively. In addition, place a candidate labels on both the read and write circuitry with footprint pointers *Pw* and *Pr* as their respective footprint pointers.

Outcome 5) No problem found in the node. Create error label on W with Pw (defined later in the algorithm) as the footprint pointer.

Location-propagation rule algorithm:

We denote the following:

- Footprint vectors/matrices assigned to *C, W, R* as *Fc, Fw, Fr*
- Footprint pointers on *W, R, Aw, Ar,* as *Pw1, Pr1, Paw1, Par1;*
- New footprint pointers as *Pw2, Pr2, Paw2, Par2;*
- Data values pointed by *Pw1, Pr1, Par1, Paw1, Pw2, Pr2, Par2, Paw2* as *Vw1, Vr1, Var1, Vaw1, Vw2, Vr2, Var2, Vaw2;*

The algorithm for executing the location-update rule is shown below.

Step 1) Assign *Par1, Pr1, Paw1* to *Par2, Pr2, Paw2*;

Step 2) If ((follow_link (*Pc, Fc, Fr*) < *Pr*) and (follow_link (*Pc, Fc, Fo*) > *Po*)) then return "Outcome 3"; Halt;

Step 3) Decrement *Par2* and *Pr2,* until *Var2* is not an idle cycle and belongs to a committed instruction;

Step 4) If (*Par2* < *Paw1*) then Goto Step 7;

Step 5) If (*Var1* = *Var2*) then Goto Step 15;

Step 6) Goto Step 3;

Step 7) Increment *Paw2,* until *Vaw2* is not an idle cycle and belongs to a committed instruction;

Step 8) If (*Vaw1* = *Vaw2*) then Goto Step 11;

Step 9) If *Paw2* reaches youngest entry, Goto Step 11;

Step 10) Goto Step 7;

Step 11) Increment *Par2* and *Pr2,* until *Var2* is not an idle cycle and belongs to a committed instruction;

Step 12) If (*Par2* > *Paw2*) then assign NULL to *Pa2* and Goto Step 15;

Step 13) If (*Var1* = *Var2*) then Goto Step 15;

Step 14) Goto Step 11;

Step 15) Assign *Paw1* to *Paw2*;

Step 16) Decrement *Paw2* and *Pw2,* until *Vaw2* is not an idle cycle and belongs to a committed instruction;

Step 17) If *Paw2* reaches oldest entry, then assign NULL to *Paw2* and Goto Step 20;

Step 18) If (*Vaw1 = Vaw2*) then Goto Step 20;

Step 19) Goto Step 16;

Step 20) Execute rules in Table 3.1; Halt;

Table 3.1: Location-propagation rules for a Random-access type.

	***Par2* ≠ NULL**	***Par2* = NULL**
***Paw2* ≠ NULL**	If (*Vw1 = Vr1*) then outcome 5 If (Vw1 ≠ Vr1 = Vr2) then outcome 1	If *(Vw1 = Vr1)* then outcome 5 If *(Vw1 ≠ Vr1=Vw2)* then outcome 1 If *(Vw1 ≠ Vr1 ≠ Vw2)* then outcome 4
***Paw2* = NULL**	If (Vr2= Vw1 ≠ Vr1) then outcome 2	If *(Vw1= Vr1)* then outcome 5 If *(Vw1 ≠ Vr1)* then outcome 4

3.3.2.6 Associative Type

An Associative-type node has four incoming edges – read address (*Ar*), write address *(AW)*, write data *(W)* and clear signal (*C)* – and an outgoing edge, read data (*R).* The node has four localization regions: write circuitry, read circuitry, clear handler, tag handler. The location-propagation rules for an Associative-type node are very similar to that of Random-access type node with a difference. Whenever a candidate label is placed on the read/write circuitry for the Random-access type, the corresponding rules for Associative type places an additional candidate label on the tag handler.

3.3.2.7 Default Type

A default-type node has little localization capability: *i.e.*, given an error at the node's outgoing edge, the rules declare the node as a candidate by default and propagate the error to all incoming edges. There are three ways of handling Default-type nodes:

Method 1) Use it as it is;

Method 2) Merge with predecessor/successor nodes so that the merged node is of non-default type;

Method 3) Handcraft customized location-propagation rules by using parts of the decision diagram used in Appendix B.

As an example of a Default-type node with customized rules, consider the alignment checker mentioned in the decision diagram. It checks whether the load/store address of an aligned data access instruction is aligned (*i.e.*, whether the addresses are powers of two). On a failed check, it raises an alignment exception.

A Default-typed node is used to model the alignment checker. It has two incoming edges – memory address (*M*), aligned/unaligned access bit (*A*) – and an outgoing edge, the alignment exception signal (*X)*. For simplicity, we ignore the incoming edge with the "size" data. *X* is assigned the alignment exception field of the commit-stage footprint vector as edge attributes. *A* is assigned the aligned/unaligned access bit of instruction field of the fetch-stage footprint vector as edge attributes. *M* is assigned the memory address field of the load-store unit footprint matrix as edge attributes. All three edges have <*X*, "same instruction"> as their <edge, edge-dependency> pair. Below shows the possible outcomes and the customized location-propagation rule.

We denote the following:

- Footprint pointers on *M, A, X* as *Pm, Pa, Px;*
- Data values pointed by *Pm, Pa* as *Vm, Va/*

Location-propagation rule outcomes:

Outcome 1) Alignment exception was generated for an unaligned access. Either the alignment checker is faulty or the "unaligned" bit was incorrectly decoded into "aligned" bit. Place a candidate label on the node with *Px* and an error label on *A* with *Pa* (contained error).

Outcome 2) The alignment check is faulty. Place a candidate label on the node with *Px* (contained error).

Outcome 3) Memory address is incorrect. Place an error label on *M* with *Pm* (contained error).

Location-propagation rule algorithm:

Step 1) If *(Va = unaligned)* then return "Outcome 1"; Halt;

Step 2) If *Vm* is aligned then return "Outcome 2"; Halt;

Step 3) Return "Outcome 3"; Halt;

The location-propagation rules used for the alignment checker is highly context sensitive, and it is generally not applicable to other nodes, and hence it is categorized as a Default-type node. The Modifying type can be used to model the alignment checker, but it suffers from lower localization accuracy, which is apparent from the comparison of possible rule outcomes.

3.4 EVALUATION ON AN INDUSTRIAL SIMULATOR

We construct a BLoG for the Intel ® Core™ i7 processor, given the recorder field description shown in Table 3.2, and conduct an error injection campaign on an industrial-grade microarchitectural simulator to evaluate the bug localization capability of BLoG-assisted IFRA.

Table 3.2. Auxiliary information for Intel Core i7 microarchitecture.

Pipeline stage	**Auxiliary information**		**Number of recorders**	**Entries per Recorder**
	Description	Bits per entry		
Fetch	Instruction pointer (IP)	32	4	512
Decode	Ucode, bom, decoded results	24	4	1,024
Alloc	3-bit residue of register name	9	4	1,024
Schedule	4-bit residue of operands	8	4	1,024
IEU	4-bit residue of result	4	3	1,024
AGU/ MEM	4-bit residue of result; memory address;	36	2	1,024
Commit	Exceptions	4	4	1,024
Total storage required for all recorders: (Each entry has an additional 8-bit ID)			66KBytes	
Ucode(microcode), BOM (the beginning of each macroinstruction)				

3.4.1 SIMULATION FRAMEWORK AND METHODOLOGY

We use an industrial-grade, execution-driven, cycle-accurate IA-32 simulator that models an Intel® Core™ i7 processor. The simulator executes "Long Instruction Trace (LIT)"s, which consist of an architectural state snapshot and a list of system interrupts needed to simulate system events. The LIT includes an entire snapshot of memory and it can be used to execute down mis-speculated paths. Simulations are performed on a suite of 88 captured LITs sampled across several inputs and program phases of the 12 benchmarks in the Integer component of SPEC CPU 2006. We modify the original IFRA to use IA-32 in-built exceptions (#OF, #BR, #UD, #SS, #GP, #PF, #AC [Intel 08]).

The error injection campaign is conducted in a similar manner as described in Sec. 2.4. The only difference is in error injection sites, which are grouped into 32 categories (see Table 3.3), where each category consists of 1-400 bits that show similar error effects due to symmetry (*e.g.*, an error injected into the input of an ALU will exhibit identical behavior as those injected into the input of a multiplier). No errors are injected in array structures or arithmetic units, since they are assumed to be protected.

There are 160 BLoG nodes in total, of which 86% (83% of the total candidate locations) were non-Default types. Table 3.4 details the nodes by type and shows the maximum number of candidates for each node type. The 23 Default-type nodes were manually augmented with customized error localization/propagation rules adapted from the decision diagram in Appendix B. Since the Default-type varies from node to node, we report additional candidate statistics.

Table 3.3: Error injection sites.

Address entering ITLB	RAM control write/read circuit
Address entering AGU/DTLB	Conflict resolution (*e.g.*, MOB)
Address entering load buffer	Eviction handler
Address entering store address buffer	Replacement handler
Data entering store data buffer	Clearing queue
Data between RS output and EU	Queue pointer
Data between ROB/RRF and RS	Stall to path
Data between EU output and RS	Clear to path
Data from EU output to ROB/RRF	Decision of mux
Data exiting store data buffer	Stall generation
Data entering load buffer	Clear generation (ROB/JEU/BAC)
IP from branch predictor unit output	Cache miss handling 1
IP from instruction fetch output	Cache miss handling 2
IP from BAC	Updating branch target buffer
IP between JEU and next IP	Fetch to external bus
IP from JEU to branch predictor update	Instruction steering (+MS)

(TLB: translation look ahead buffer; AGU: address generation unit; MOB: memory order buffer; RRF: real register file; JEU: jump execution unit; BAC: branch address calculator; MS: microcode sequencer).

Table 3.4. BLoG node type distribution for Intel Core i7.

Node type	**Number of nodes**	**Number of localization regions per type**
Random-access	6	3
Associative	8	4
Queue	7	3
Modifying	22	1
Connection	46	1
Select	42	2
Protected	6	0
Default	23	Avg(2), min(1), max(4), std(1.1)
Total number of nodes: 160		
Maximum number of locations: 269		

3.4.2 RESULTS

Table 3.5 presents the results from over 30,000 error injection runs, of which 2,560 activated a post-trigger. The *exactly localized* category occurs when IFRA identifies a single and correct candidate. The *localized with candidates* category occurs when IFRA identifies multiple candidates, one on which is correct. The *completely missed* category occurs when IFRA does not identify the correct candidate.

For 56% of injected electrical bugs, IFRA pinpointed their exact <location, footprint> pair – composed 1 out of 270 localization regions and 1 out of over 1,000 cycles in which an electrical bug had caused an error in a flip-flop. For 34% of injected bugs, IFRA correctly identified their <location, footprint> pairs together with 5 other candidates on average (out of over 270,000 possible pairs). IFRA completely missed for 10% of injected bugs (*i.e.*, either the location or the footprint or both were incorrect).

The causes of the 10.4% complete misses are reported in Table 3.6. *Long latency* means that the 1024-entry recorders fail to capture the history from error to failure. *Candidate limit* means that a pre-defined candidate limit of 20 was reached before localizing the bug. *Wrong diagnosis* includes all other cases.

Table 3.5. BLoG-assisted IFRA bug localization summary.

Category	**Total (%)**
Exactly localized	55.6
Correctly localized with multiple candidates	34.0 (Avg. 6)
Completely missed	10.4

Table 3.6. Causes of complete miss.

Category	**Total (%)**
Long latency	32.9
Candidate limit (limit = 20)	20.7
Wrong diagnosis	46.8

Table 3.7 provides a summary of the manual effort reduced when using BLoG to design the low-level analysis compared to how it was originally done. The manual effort required for creating custom propagation rules for Default-type nodes and edge dependencies can be minimized by eliminating custom rules and pessimistically declaring an encountered node as candidate and propagating errors to its inputs. "Exactly localized" cases remain unaffected for errors originating from non-default-type nodes (86% of nodes). Remaining situations lead to increased candidate count; "completely missed" cases may increase when candidate count exceeds 20 (out of 200,000+). Alternatively, default-type node count may be reduced by adding more node and edge dependency types.

Table 3.7. Summary of manual effort reduced using BLoG.

Tasks	**Original IFRA**	**BLoG**
Node definition	Manual	Guided by node types
Edge definition	Manual	Guided by edge attributes
Designing location propagation rules for non-default-type nodes (86%>)	Manual	Automatic
Designing footprint propagation rules for non-default-type edge dependencies (95%>)	Manual	Automatic
Designing custom location propagation rules for default-type nodes (<14%)	Manual	Manual
Designing custom footprint propagation rules for default-type edge dependencies (<5%)	Manual	Manual

3.5 RELATED WORK

Related work can be largely divided into three categories: high-level test generation [Lee 94][Mishra 08][Tupuri 97][Utamaphethai 00][van Campenhout 99], circuit-level diagnosis [Caty 05][Tekumalla 01][Venkataraman 96][Yang 09] and fault-tolerant computing [Carretero 09][Austin 99][Lu 82][Oh 02].

Many BLoG abstraction concepts (*e.g.*, the datapath concept) are borrowed from the high-level test generation community, whose aim was to obtain architectural-level constraints for test generation purposes. Because our abstraction is purely for bug localization using self-consistency checks, our abstraction model (node types) is simpler and higher-level than the prior art.

Many concepts used for BLoG traversal are borrowed from circuit-level sequential diagnosis. There are many analogous counterparts. For example, the process of identifying a flipflop with an incorrect transition/value is akin to performing our high-level analysis to find the starting edge. Forward propagating known values to find transitions/values on all wires is analogous to our deriving information on each edge. Back-propagating transition/value through a netlist is akin to our BLoG traversal.

Finally, the propagation/localization rules for performing self-consistency checks have borrowed ideas from fault-tolerant computing. The chief difference is that they use self-consistency checks for the purpose of detection, rather than localization.

3.6 CONCLUSIONS

The BLoG framework enables the systematic construction and automatic execution of IFRA's offline analysis for bug localization, easing the application of IFRA to new microarchitectures. The BLoG framework is highly effective; BLoG used on an industrial simulator of a state-of-the-art processor yields 90% bug localization accuracy.

This work also introduces several interesting research directions, including:

- Automatically constructing BLoG from a language-based specification or an RTL description.
- Creating a new language for specifying microarchitecture for the purpose of BLoG construction.
- Automatically selecting what to record given a BLoG, borrowing concepts from work done at circuit-level [Ko 08].
- Supporting feedback between construction and traversal so that localization result can guide new partitioning.
- Introducing more node types to reduce the number of Default-type nodes.
- Combining probabilistic diagnosis to rank candidates.

CHAPTER 4. CONCLUDING REMARKS

Post-silicon bug localization of electrical bug is a major bottleneck in today's chip development. This report presents IFRA, a new technique for localizing electrical bugs in processors that overcomes the existing limitations by not relying on system-level failure reproduction and system-level simulation. In addition, the report presents BLoG, a new framework for minimizing the manual effort required to implement IFRA on new microarchitectures. High bug localization accuracy achieved on both the open-source simulator (96%) and the industrial-grade simulator (90%) demonstrates the effectiveness of the two techniques.

This work creates several interesting research directions:

- Application to system-on-chips (SoCs) consisting of non-processor designs. The footprint linking process, which is the basis for IFRA and BLoG, utilized instructions that were carrying information regarding their operations and operands/results along with them from entry to exit. It also used the fact that processors have a fixed limit on the maximum number of instructions in-flight and transfer data using register names and memory addresses. Finding similar features in SoCs may be the first step toward this objective.
- Application to multiple electrical bugs being activated within a small time frame. Current self-consistency checks assume rare occurrence of electrical bugs. The first step in the investigation could be to see whether using multiple inconsistencies discovered by the high-level analysis help in localizing multiple bugs. The next step could be to perform multiple BLoG traversals, each with different number of starting edges, and then intersecting the results. Extending this approach may provide a way to localize logic bugs as well.

REFERENCES

[Abramovici 06] Abramovici, M., *et al.*, "A Reconfigurable Design-for-Debug Infrastructure for SoCs," *Proc. Design Automation Conf. (DAC '06)*, pp. 7-12, July 2006.

[Ando 03] Ando, H., *et al.*, "A 1.3-GHz Fifth-Generation SPARC64 Microprocessor," *Proc. Design Automation Conf. (DAC '03)*, pp.702-705, June 2003.

[Agarwal 86] Agarwal, A., R.L. Sites, and M. Horowitz, "ATUM: a new technique for capturing address traces using microcode," *Proc. Intl. Symp. on Computer Architecture.(ISCA '86)*, pp. 119-127, May 1986.

[Austin 99] Austin, T.M., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. Intl. Symp. on Microarchitecture (MICRO '99)*, pp. 196-207, Nov. 1999.

[Austin 02] Austin, T., *et al.*, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, no. 2, pp 56-67, Feb. 2002.

[Bayazit 05] Bayazit, A.A., and S. Malik, "Complementary Use of Runtime Validation and Model Checking," *Proc. IEEE/ACM Intl. Conf. on Computer-aided Design (ICCAD '05)*, pp. 1052-1059, Nov. 2005.

[Bentley 01] Bentley, B., "Validating the Intel Pentium 4 Microprocessor", *Proc. Design Automation Conf. (DAC '01)*, pp.244-248, June 2001

[Carretero 09] Carretero J. et al. "End-to-End Register Data-Flow Continuous Self-Test", *Proc. Intl. Symp. on Computer Architecture (ISCA '09)*, pp. 105-115, Jun 2009.

[Casazza 09] Casazza J., "First the Tick, Now the Tock: Intel Microarchitecture (Nehalem)", Intel Corporation White paper.

[Caty 05] Caty, O., P. Dahlgren, and I. Bayraktaroglu, "Microprocessor Silicon Debug based on Failure Propagation Tracing," *Proc. Intl. Test Conf.*, pp. 293-302, Nov. 2005.

[Chang 07] Chang, K., I.L. Markov, and V. Bertacco, "Automating Post-Silicon Debugging and Repair," *Proc. Intl. Conf. on Computer-Aided Design (ICCAD '07)*, pp. 91-98, Nov. 2007.

[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*, pp. 415-426, Feb. 2008.

[Clarke 99] Clarke, E.M., D.A. Peled and O. Grumberg, "Model Checking", *Cambridge, MA[u.a.] MIT Press 1999, ISBN:* 0262032708 9780262032704

[Colwell 05] Colwell, R., et al., "Intel's P6 Microarchitecture," *Chapter 7 in Shen and Lipasti, Modern Processor Design*, New York: McGraw-Hill, 2005.

[De Paula 08] De Paula, F.M., *et al.*, "BackSpace: Formal Analysis for Post-Silicon Debug," *Proc. Formal Methods in Computer-Aided Design*, pp.1-10, Nov. 2008.

[Digital 99] Digital Equipment Corporation, *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[Dill 98] Dill, D.L., "What's between simulation and formal verification? (Extended abstract)", *Proc. Design Automation Conference (DAC '99)*, pp. 328-329, June 1999.

[Goddard 95] Goddard, M.D., and D.S. Christie, "Microcode Patching Apparatus and Method," *U.S. Patent 5796974*, Nov. 1995.

[Gorjiara 07] Gorjiara, B., M. Reshadi, and D. Gajski, "Generic Architecture Description for Retargetable Compilation and Synthesis of Application-Specific Pipelined IPs," *Proc. Intl. Conf. on Computer Design (ICCD '07)*, pp. 356-361, Oct. 2007.

[Halambi 99] Halambi A. et al. "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability", *Proc. Conf. on Design Automation and Test in Europe (DATE '99)*, pp.485, Mar. 1999.

[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 (DATE '04)*, pp. 1532-1546, Feb. 2004.

[ITRS 07] International Technology Roadmap for Semiconductors, 2007 ed.

[Josephson 01] Josephson, D., S. Poehlman, and V. Govan, "Debug Methodology for the McKinley Processor," *Proc. Intl. Test Conf.(ITC '01)*, pp. 451-460, Oct.-Nov. 2001.

[Josephson 06] Josephson, D., "The Good, the Bad, and the Ugly of Silicon Debug," *Proc. Design Automation Conf (DAC '06)*, pp. 3-6, July 2006.

[Intel 08] "Intel® 64 and IA-32 Architectures Sfotware Developer's Manual Volume 3A: System Programming Guide, Part 1," Order number: 253668-027US, Jul 2008.

[Ko 08] Ko, H.F. and N., Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," *Proc. Conf. on Design, Automation and Test in Europe (DATE '08)*, pp. 1298-1303, 2008.

[Krupnova 04] Krupnova, H., "Mapping Multi-Million Gate SoCs on FPGAs: Industrial Methodology and Experience", *Proc. Conf. on Design, Automation and Test in Europe (DATE '04)*, pp.21236-21243, Feb. 2004.

[Lee 94] Lee. J and J.H. Patel, "Architectural level Test generation for microprocessors", *IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems*, vol. 13, no. 10, pp.1288-1300, Oct. 1994.

[Livengood 99] Livengood, R.H., and D. Medeiros, "Design for (Physical) Debug for Silicon Microsurgery and Probing of Flip-chip Packaged Integrated Circuits," *Proc. Intl. Test Conf (ITC '99).*, pp. 877-882, Sept. 1999.

[Lu 82] Lu, D.J., "Watchdog Processors and Structural Integrity Checking," *IEEE Trans. Comput.*, vol.31, no.7, pp.681-685, July 1982.

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

[Mahmood 88] Mahmood A., and E.J. McCluskey, "Concurrent error detection using watchdog processors – a survey," *IEEE Trans. Comput.*, vol.37, no.2, pp. 160-174, Feb. 1988.

[McLaughlin 09] McLaughlin R., S. Venkataraman, and C. Lim, "Automated Debug of Speed Path Failures using Functional Tests," *VLSI Test Symp.(VTS '09)*, pp. 91-96, May 2009.

[Mishra 08] Mishra, P. and N. Dutt, "Specification-driven Directed Test Generation for Validation of Pipelined Processors," *ACM Trans. Des. Autom. Electron. System. (TODAES)*, vol. 13, no. 3, pp.1-36, July 2009.

[Nakamura 04] Nakamura, Y., *et al.*, "A Fast Hardware/software Co-verification Method for System-on-a-chip by Using a C/C++ Simualtor and FPGA Emulator with Shared Register Communication", *Proc. Design Automation Conference (DAC '04)*, pp. 299-304, June 2004.

[Oh 02] Oh, N., P.P. Shirvani, and E.J.McCluskey, "Control-flow Checking by Software Signatures," *IEEE Trans. Reliability*, pp.111-122, Mar. 2002.

[Park 09] Park S., T. Hong and S. Mitra, "Post-Silicon Bug Localization in Processors using Instruction Footprint Recording and Analysis (IFRA)", *IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1545-1558, Oct. 2009.

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

[Sanda 08] Sanda P.N., *et al.*, "Soft-error Resilience of the IBM POWER6 Processor," *IBM J. of Res. and Dev.*, vol.52, no.3, pp. 275-284, 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 (DSN '06)*, pp. 301-312, June 2006.

[Sarangi 07] Sarangi, S.R., *et al.*, "Patching Processor Design Errors with Programmable Hardware," *IEEE Micro*, pp.12-25, Jan. 2007.

[Schelle 10] Schelle G., *et al.*, "Intel Nehalem Processor Core Made FPGA Synthesizable", *Proc. ACM/SIGDA Intl. Symp. On Field Programmable Gate Arrays (FPGA '10)*, pp. 3-12, Feb. 2010.

[Shen 05] Shen, J.P., and M.H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, New York: McGraw-Hill, 2005.

[Siewiorek 98] Siewiorek, D.P., and R.S. Swarz, *Reliable Computer Systems – Design and Evaluation*, 3rd ed., Natick: A.K. Peters, 1998.

[Silas 03] Silas, I., *et al.*, "System-Level Validation of the Intel Pentium M Processor," *Intel Technology Journal*, vol.7, no.2., pp. 37-43, May 2003.

[Tekumalla 01] Tekumalla, R.C., S. Venkataraman, and J.G. Dastodar, “On Diagnosing Path Delay Faults in an At-Speed Environmetn”, *Proc. VLSI Test Symp. (VTS ’01)*, pp. 28-33, Apr 2001.

[TI 97] Texas Instruments, “IEEE Std 1149.1 (JTAG) Testability Primer”, 1997.

[Tupuri 97] Tupuri R.S., J.A. Abraham, “A Novel Functional Test Generation Method for Processors using Commercial ATPG”, *Proc. Intl. Test Conf. (ITC ’97)*, Nov. 1997, pp. 743-752

[Utamaphethai 00] Utamaphethai, N., R.D. Blanton and J.P.Shen, “A Buffer-Oriented Methodology for Microarchitecture Valdiation”, *J. Electron. Test*. 16, 1-2, Feb. 2000, pp.49-65.

[van Campenhout 99] Van Campenhout D., T. Mudge and J.P.Hayes, "High-Level Test Generation for Design Verification of Pipelined Microprocessors, " *Proc. Design Automation Conf. (DAC ’99)*, pp. 185-188, June 1999.

[Venkataraman 96] Venkataraman, S., I. Hartanto, and K. Fuchs, “Dynamic diagnosis of sequential circuits based on stuck-at faults”, *Proc VLSI Test Symp. (VTS ’96)*, pp.198, Apr. 19 96.

[Wagner 06] Wagner, I., V. Bertacco, and T. Austin, “Shielding Against Design Flaws with Field Repairable Control Logic,” *Proc. Design Automation Conf. (DAC ’06)*, pp. 344-347, July 2006.

[Wang 04] Wang, N.J., *et al.*, “Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline,” *Proc. Intl. Conf. on Dependable Systems and Networks (DSN ’04)*, pp. 61-70, June-July 2004.

[Xu 03] Xu M., R.Bodik, and M.D. Hill, “Flight Data Recorder for Enabling Full-System Multiprocessor Deterministic Replay,” *Proc. Intl. Symp. on Computer Architecture (ISCA ’03)*, pp. 122-133, May 2003.

[Yerramilli 06] Yerramilli, S., “Addressing Post-Silicon Validation Challenge: Leverage Validation & Test Synergy (Invited Address)”, presented at the *IEEE Intl. Test Conf (ITC ’06)*, Oct 25, 2006.

[Yang 09] Yang, Y-S., N. Nicolici, A.G. Veneris, “Automated Data Analysis Solutions to Silicon Debug”, Proc. Conf. on Design Automation and Test in Europe (DATE ’09), pp. 982-987, 2009.

[Ying 05] Geoffrey Ying, “Start at the top to reduce re-spins for analog-digital chips”, *Chip Design Magazine*, June/July 2005.

APPENDIX A: FOOTPRINT LINKING ALGORITHM

Footprint linking analyzes ID sequences to identify which of the footprints, stored across multiple recorders, belong to the same instruction. Appendix A provides the algorithm for performing footprint linking. Sec. A.1 elaborates on the features and assumptions on the target processor model, while Sec. A.2 presents algorithms for distinguishing footprints with identical IDs. Sec. A.3 presents algorithms for identifying footprints belonging to flush-causing, flushed, committed and uncommitted instructions.

A.1 ASSUMPTIONS ON THE TARGET PROCESSOR

The following consequences result from having the features and assumptions presented in Sec. 2.1.

1) **Maximum number of instructions in flight**: The maximum number of instructions-in-flight, n, equals the number of ROB entries in a superscalar processor.
2) **Mis-speculation handling**: A pipeline flush due to a mis-speculated instruction is only initiated after the corresponding branch instruction exits the execution stage.
3) **D-TLB miss handling**: An instruction with a D-TLB miss causes a pipeline flush only when the instruction reaches the head of the ROB. For the purpose of footprint linking, we consider the instruction to have committed, where in reality, it is discarded. As a consequence, if the instruction causing the DTLB-miss is assigned ID Y, the first instruction fetched after the resumption of IFRA recording will be assigned ID Y+2n+1 (mod 4n), similar to speculation handling. IFRA recording pauses between Y and Y+2n+1 because D-TLB miss is one of IFRA's soft-triggers.
4) **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 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.

5) **Fatal exception handling**: Once an instruction with a fatal exception reaches the head of the ROB, the post-trigger generator halts the processor. Thus, fatal exceptions do not cause pipeline flushes.

A.2 DISTINGUISHING FOOTPRINTS WITH 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 [Shen 05]. 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 Sec. A.3. The second step, described in this section, uses this information to order and link committed instructions in program order.

Algorithm A: 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.

Algorithm A.1: Linking - Steps 3 of Algorithm A 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.

Step 2): While the fetch-stage footprint vector pointer *P* does not point to a footprint marked as committed, decrement *P*. If *P* cannot be decremented, 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, decrement 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): Decrement *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 older 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 older 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 k^{th} younger entry either has the ID $X+k \pmod{4n}$ or $X+k+2n \pmod{4n}$, $k \in \mathbb{Z}^+$, 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 Sec.2.2.2; 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 k^{th} younger entry; thus, the k^{th} 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 k^{th} younger entry has an ID distinct from X, at any given time, where $1<=k<n$.

Proof: From Lemma 2.1, the k^{th} 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 [Shen 05], 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.

A.3 IDENTIFICATION OF UNCOMMITTED INSTRUCTIONS

Sections A.3.1 and A.3.2 detail 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.

A.3.1 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 illustrated by an example in Fig. A.1. Proofs for Theorem 3-5 that are behind the algorithm are presented afterwards.

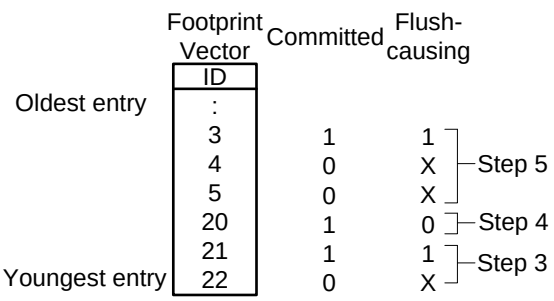

Fig. A.1. Example footprint vector with labels (*n=8)*.

Algorithm A.2: Labeling of a footprint vector

Step 1) Let *X* = ID of the youngest committed instruction obtained from the commit-stage recorder.

Step 2) Setup a pointer *P* to the youngest entry of the footprint vector.

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 decrement *P* to the next older entry. When *X* is encountered, label it as committed and flush-causing (Theorem 3). Before decrementing *P*, if *P* is pointing at the oldest entry of the vector, stop.

Step 4) While a break in consecutive assignment is not observed, label the footprint pointed to by *P* as committed and non-flush causing and decrement *P*. a break in consecutive assignment 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 older than P) is not +1 (mod 4n). The ID-jump indicates a pipeline flush (Theorem 4). Before decrementing *P*, if *P* is pointing at the oldest entry of the vector, stop.

Step 5) Let *W* = ID of the flush-causing instruction, obtained by subtracting 2*n*+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 decrement *P*. When *W* is encountered, label it as committed and flush-causing (Theorem 5). Before decrementing *P*, if *P* is pointing at the oldest 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 an 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, the instruction fetched next after X' will have ID $S=X+1+2m$ (mod $4n$) (mod $4n$) $m \in \{0,1\}$. 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 $X+1+k'+2nm$ (mod $4n$), $m \in \{0,1\}$, $1 <= k' < n$. Since all IDs in the ROB are of the form $X+k+2nm$ (mod $4n$), $1<=k<=n$ 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 older entry. Then X-W ≠ +1 (mod 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 $X-W \neq +1$ (mod $4n$), then Rule 3 of the ID assignment scheme (Sec. 2.2.2) 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 $X=W+2n+1$ (mod $4n$). 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 $S=W+1$ (mod $4n$). According to Lemma 2.1, instructions fetched after S would be assigned IDs $W+1+k'+2nm$ (mod $4n$) $m \in \{0,1\}$, $1 <= k' < n$; including S, these IDs are of the form $W+k+2nm$ $m \in \{0,1\}$, $1 <= k <= n$. We prove that $W+2n+1 - W+k+2nm \neq 1$ (mod $4n$) for all k between 1 and n:

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

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

$\rightarrow 2n+1-k-2nm' \neq 1 \pmod{4n}$, $m' \in \{0,1\}$

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

Also, $X=W+2n+1 \pmod{4n}$ 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 older 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 (mod 4n). The flush-causing instruction must have committed, and it appears in the footprint vectors as the youngest ID W older 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 of Sec. 2.1), $W=X-2n-1 \pmod{4n}$. 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 older than *X*. By Theorem 1, instructions are recorded in program order; thus the first instruction with ID *W* older than *X* is *X*'s corresponding flush causing instruction.

A.3.2 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 (Section A.3.1).

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 older 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 6) 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 row of the footprint matrix.

Step 2) While *P* does not point to an unvisited entry with ID *W*, decrement *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 $Y=P$, increment *Y* until an entry with ID=$W+2n+1 \pmod{4n}$ is found. The row of Y is the younger isolating row (Theorem 6). Decrement *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 $E=P$, decrement *E* until an entry with ID $X-n \pmod{4n}$ or $X-3n \pmod{4n}$ is found. This is the older isolating row (Theorem 7). Increment *E* until the end of the prior row. If the ID is not found, E = last entry of the oldest 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.

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 bottom and an *older isolating row* at the top. The older 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 youngest among the ID *X*s dispatched before the flushed ID *X*. Similarly, the younger 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 oldest among the ID *X*s dispatched after the flushed ID *X*. If one has the ability to find the older 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's 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).

A.3.2.1 Identification of 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 oldest row younger than W that contains an entry with ID W+2n+1 (mod 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 bottom row acts as the younger isolating row.

A.3.2.2 Identification of Older-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 older isolating row for X is the youngest row older than W that contains an entry with ID X-n (mod 4n) or X-3n (mod 4n).

We use Lemmas 7.1-7.5 to prove the theorem. If the ID *X-n (mod 4n)* or *X-3n (mod 4n)* cannot be found, an imaginary row above the top row acts as the older isolating row. If the recorders were large enough, the ID would have been found in an older row.

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

Proof omitted due to similarity with Lemma 2.1.

Lemma 7.2: If an instruction with ID W commits, then the kth older committed instruction, $k \in \mathbb{Z}^+$, has either ID W-k (mod 4n) or W-k-2n (mod 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 older than *W* are committed instructions. The k^{th} older entry represents the k^{th} older committed instruction and, by Lemma 7.1, either has ID *W-k* (mod 4*n*) or *W-k*-2*n* (mod 4*n*).

Lemma 7.3: Before the instruction with ID X enters the re-order buffer of size n, an instruction with ID X-n (mod 4n) or X-3n (mod 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 Algs. 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 1st through $(n-1)^{th}$ older 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* (mod 4*n*) or *X*-3*n* (mod 4*n*) that commits before ID *X* enters the ROB.

Lemma 7.4: The flushed instruction with ID X, if it exists, cannot co-exist with any of ID X-n (mod 4n) and X-3n (mod 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 (mod 4n) or X-3n (mod 4n).

Proof: Suppose *X*-*n* (mod 4*n*), rather than *X*-3*n* (mod 4*n*) 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* (mod 4*n*): *X*-*n*+*j* (mod 4*n*), or *X*-*n*+2*n*+*j* (mod 4*n*), where 1<=*j*<=*n*-1. However, none of them equal *X*. Now suppose *X*-3*n* (mod 4*n*) 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*-3*n* (mod 4*n*): *X*-3*n*+*j* (mod 4*n*), *X*-3*n*+2*n*+*j* (mod 4*n*), 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 Xs dispatched before the flushed ID X, does not coexist with any of X-n (mod 4n) and X-3n (mod 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 (mod 4n) or X-3n (mod 4n).

Proof: Suppose *X*-4*n* (mod 4*n*) 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-4*n* (mod 4*n*): *X*-4*n*+*j* (mod 4*n*), *X*-4*n*+2*n*+*j* (mod 4*n*), where 1<=*j*<=*n*-1. However, none them equal *X*-*n* (mod 4*n*) or *X*-3*n* (mod 4*n*).

APPENDIX B: LOW-LEVEL ANALYSIS DECISION DIAGRAM FOR IFRA

This section provides a brief description of the decision diagram used for the low-level analysis. The first eleven 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.

IF array error (→1)
IF arithmetic error (→2)
IF alignment exception (→10)
IF unimplemented instruction exception (→11)
IF integer overflow exception (→12)
IF deadlock (→21)
IF instruction access segfault (→13)
IF data access segfault (→14)
IF control-flow analysis violation (→13)
IF data-dependency analysis violation (→20)
IF data-transfer analysis violation (→21)

1) Error in array element OR (→3)
2) Error in arithmetic unit OR (→3)
3) Error in exception generation unit
4) Error in register value at the output of execution stage
   (→6) OR
   IF using arithmetic unit (→2)
   IF using load/store unit (→21)
   IF using complex ALU, then error in complex ALU

IF using branch unit (→2)

5) Error in speculative register alias table

(→16) OR (→15) OR (→18)

Similar analysis to data-transfer analysis, but address is architectural register names and data is physical register names.

6) Error in register value at the input of execution stage

Displacement selection multiplexor (→7) OR forwarding path (→8)

7) Displacement selection multiplexor

IF instruction is supposed to take immediate

IF operand residue doesn't match immediate residue

IF repeated inputs don't match output,

THEN error in multiplexor

ELSE Error in opcode or immediate (→17)

ELSE

IF immediate has been obtained from non-immediate field of the kn instruction

THEN opcode or immediate (→17)

ELSE physical register file (→5)

8) Forwarding path

IF data dependency analysis violation

THEN muxes + select signals

ELSE (→4) OR (→9)

9) Error in physical register file

(→4) OR wrong physical register name from RAT (→5) OR

Similar analysis to 5 except use register value instead of physical register name and use physical register name instead of architectural register name

10) Error in decoder part 1

(→6) OR (→3) 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) Error in decoder part 2

(→3) OR

IF exception at decode stage

Wrong instruction written from icache (parity protection) OR (→13) OR

IF fetched instruction doesn't 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 from output of decode to input of execute stage

12) Integer overflow

(→2) OR (→3) OR (→4)

13) Error in PC

IF control flow violation case 1

IF instruction went to branch unit

THEN opcode corruption (→17)

ELSE faulty nextPC select mux

IF control flow violation case 2

IF instruction went to non-branch unit

THEN opcode corruption (→17)

ELSE faulty nextPC select mux

IF control flow violation case 3,4

THEN (→4)

14) Error in address generator

(→6) OR (→2)

15) Error in architectural register name

Wrong decode or wrong decoded bits propagation (→16)

16) Wrong physical register name from register free list

17) Error in decoded bit propagation

IF decoded bits differ with re-simulated result THEN error

18) 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 older results are seen

THEN incorrectly initiated recovery

19) Error in architectural register alias table

Similar analysis to 5 but physical register name and architectural register names come from output of ROB

20) Error in scheduler

Scheduler array OR

IF ID duplication (from Instruction-flow analysis)

Incorrectly cleared issued bits in the array OR Queue pointer flip

IF ID disappearance (from Instruction-flow 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) Error in load/store unit (Data-transfer analysis)