BLoG: Post-Silicon Bug Localization in Processors using Bug Localization Graphs

Sung-Boem Park[1,2] Anne Bracy[2,3] Hong Wang[2] Subhasish Mitra[1,4]
[1]Dept. of Electrical Engineering, Stanford University, Stanford, CA, USA
[2]Microarchitecture Research Lab., Intel Corporation, Santa Clara, CA, USA
[3]Dept. of Computer Science & Engineering, Washington University in St. Louis, St. Louis, MO, USA
[4]Dept. of Computer Science, Stanford University, Stanford, CA, USA

ABSTRACT

Post-silicon bug localization – the process of identifying the location of a detected hardware bug and the cycle(s) during which the bug produces error(s) – is a major bottleneck for complex integrated circuits. Instruction Footprint Recording and Analysis (IFRA) is a promising post-silicon bug localization technique for complex processor cores. However, applying IFRA to new processor microarchitectures can be challenging due to the manual effort required to implement special microarchitecture-dependent analysis techniques for bug localization. This paper presents the Bug Localization Graph (BLoG) framework that enables application of IFRA to new processor microarchitectures with reduced manual effort. Results obtained from an industrial microarchitectural simulator modeling a state-of-the-art complex commercial microarchitecture (Intel Nehalem, the foundation for the Intel Core™ i7 and Core™ i5 processor families) demonstrate that BLoG-assisted IFRA enables effective and efficient post-silicon bug localization for complex processors with high bug localization accuracy at low cost.

Categories and Subject Descriptors
B.8.1 Reliability, Testing and Fault-Tolerance
General Terms: Reliability, Verification.
Keywords: Silicon debug, post-silicon validation, IFRA, BLoG

1. INTRODUCTION

Post-silicon validation determines whether a set of manufactured chips functions correctly in actual application environments across a range of operating conditions. Post-silicon validation is becoming increasingly difficult and expensive [Abramovici 06, Patra 07]. Of the four major post-silicon validation steps – bug detection, localization, root-cause and fixing – the bug localization step dominates cost [Josephson 06]. Bug localization involves identifying the location of a detected hardware bug and a cycle in which the bug creates error(s). In particular, localization of *electrical bugs* – bugs caused by the interactions between a design and electrical effects (unlike *logic bugs* caused by design errors) – is extremely challenging [Josephson 01]. Localizing electrical bugs is challenging due to difficulties in reproducing observable failures caused by such bugs in a system setup (often referred to as *system-level failure reproduction*) and reliance on slow system-level simulation to obtain golden responses for comparison purposes.

Recently, we demonstrated a new post-silicon bug localization technique called *IFRA* (Instruction Footprint Recording and Analysis) [Park 09] for effective localization of electrical bugs in processors at low cost (< 1% chip area). IFRA does not require system-level failure reproduction or system-level simulation. It uses special hardware recorders (different from traditional trace buffers [Abramovici 06]) to collect special information (*instruction footprints*) on the last few thousand instructions that were fetched before the occurrence of a system failure. After a failure occurs, the instruction footprints are scanned out and analyzed offline using special self-consistency-based post-analysis techniques to localize the detected bug.

Although IFRA is applicable to any microarchitecture, the manual effort required to devise microarchitecture-dependent post-analysis techniques can be significant and could lead to an error-laden implementation. This paper introduces a new concept of *Bug Localization Graph* or *BLoG,* which overcomes this major challenge. The BLoG framework enables systematic construction and automated execution the post-analysis step of IFRA (Fig. 3.1).

We demonstrate the effectiveness of BLoG-assisted IFRA for the latest commercial processor microarchitecture from Intel, the Intel Nehalem [Casazza 09], which is the foundation of the Intel Core™ i7 and Core ™ i5 processor series consisting of more than a billion transistors.

The major contributions of this paper are:

1) Introduction of BLoG which overcomes the major barrier of manual implementation of microarchitecture-dependent post-analysis of IFRA. Hence, it enables IFRA to be systematically applied to new (and complex) microarchitectures with reduced engineering time and less expert knowledge resulting in significant productivity gains.

2) Demonstration of the effectiveness of BLoG-assisted IFRA in localizing electrical bugs with 90% accuracy using an industrial microarchitecture simulator modeling a complex state-of-the-art Intel Nehalem microarchitecture. Such high bug localization accuracy was achieved without requiring system-level simulation or failure reproduction.

The rest of this paper is organized as follows. Section 2 presents an overview of IFRA. Section 3 introduces the BLoG concepts. Sections 4 and 5 describe BLoG construction and usage during post-analysis, respectively. Section 6 presents BLoG-assisted IFRA results for the Intel Nehalem microarchitecture. Section 7 discusses related work followed by conclusions.

2. IFRA OVERVIEW

As discussed in Sec. 1, IFRA utilizes special hardware recorders, placed at each pipeline stage, to collect instruction footprints or simply *footprints* of instructions leaving the associated pipeline stage. Each footprint consists of an instruction ID and a set of auxiliary information (see Table 6.2) that describe the flow of the instruction through the processor and what the instruction did at each pipeline stage. (Note: given instruction will have multiple footprints in recorders belonging to multiple pipeline stages. The total distributed storage for footprint recorders amounts to 60KBbytes for an Alpha 21264-like processor). The recoding is performed in a circular fashion; only the last few thousand footprints are stored at any time. Special techniques for detecting

DAC'10, June 13-18, 2010, Anaheim, California, USA.

failure symptoms, referred to as *post-triggers* in [Park 09], enable short error detection latencies.

Upon detection of a problem using post-triggers, footprint recording is terminated and the recorded footprints are scanned out. Next, footprints are post-processed offline in three phases to localize the bug. First, *footprint linking* links together footprints belonging to the same instruction but stored across multiple recorders in a distributed fashion. A special scheme for assigning instruction IDs [Park 09] enables efficient footprint linking for complex processors supporting speculative and out-of-order execution with multiple clock domains. The linked footprints are also mapped to the corresponding instruction in the test program binary.

After footprint linking, two sets of *self-consistency-based checks* are executed (off-line) to identify any contradictory events in linked footprints with respect to the test program binary. Such checks do not require system-level simulation. A set of microarchitecture-independent checks, or *high-level analysis*, finds the first sign of an inconsistency in program execution. The inconsistency is in the form of a <*location, footprint*> *pair*. The location element of the pair specifies a hardware block and the footprint element is a pointer to an entry in one of the recorders. This pair serves as the starting point for a set of microarchitecture-dependent checks, or *low-level analysis*, which asks a series of microarchitecture-specific questions according to a **manually-generated** decision diagram (such a decision diagram for an Alpha 21264-like processor in presented in [Park 09]) to identify a set of bug candidates which are in the form of <location, footprint> pairs. The location element indicates the hardware block in which the electrical bug produces an error. The footprint element indicates a cycle in which the error occurred relative to the cycle when the post-trigger occurred. Once a bug is localized, existing circuit-level debug techniques [Caty 05, Josephson 06] can then identify the root cause of the bug.

3. BUG LOCALIZATION GRAPH (BLoG)

The BLoG framework, presented in this paper, provides a systematic approach for devising the low-level analysis of IFRA (Fig. 3.1). The BLoG is a directed graph with nodes and edges. A processor is partitioned into design blocks; each design block is represented as a *BLoG node*. Self-consistency checks are performed at BLoG nodes in a modular fashion using special rules associated with BLoG nodes. BLoG edges represent data or control signals communicated between nodes. The BLoG framework defines a set of *edge attributes* (Sec. 3.2) for obtaining edge signal values from recorder entries. It also defines a set of *node types* (Sec. 3.1) for executing self-consistency checks using the signal values on the edges.

Once a BLoG is constructed, post-silicon bug localization flow of the original IFRA is followed until high-level analysis is performed. The inconsistency returned from the high-level analysis initializes the BLoG (see Sec. 5). Low-level analysis is then performed by traversing the BLoG (Fig 3.1) using BLoG traversal algorithms (Sec. 5).

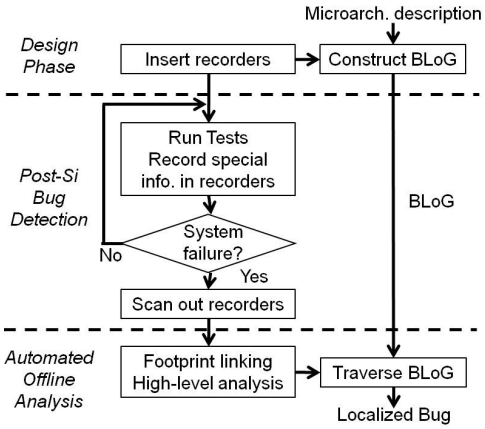

Figure 3.1. Bug localization flow using BLoG-assisted IFRA.

3.1 BLoG NODE TYPES

The BLoG framework defines seven node types, representing different "types" of processor design blocks (Fig. 3.2). A final default type is used for blocks not belonging to any of the seven types. Each node type has its own set of self-consistency checks (discussed in Sec. 5.2).

The types are broadly classified into storage and non-storage types. Storage types represent design blocks containing hardware structures with variable propagation delays from data entry to exit; non-storage types represent hardware structures with fixed delays between data entry and exit known *a priori*.

Node Type 1) Random-access: Storage with index-based entry management (e.g., register file, register alias table).

Node Type 2) Associative: Storage with associative entry management (e.g., reservation station, TLB).

Node Type 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).

Node Type 4) Transformation: Non-storage structure that modifies input values and produces different output values (e.g., decoder, address generator, comparator, ALU).

Node Type 5) Connection: Non-storage structure that propagates input values to output values without modification but after some fixed delay (i.e., series of pipeline registers).

Node Type 6) Select: Non-storage structure that takes two input values and chooses one as an output value according to control signal values (e.g., forwarding path, register/immediate select, next-PC select, instruction select).

Node Type 7) Protected: Transformation or Connection type with built-in error detection techniques 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.

Node Type 8) Default: Any non-storage structure not included in the aforementioned types (e.g., scheduler, load replay handler).

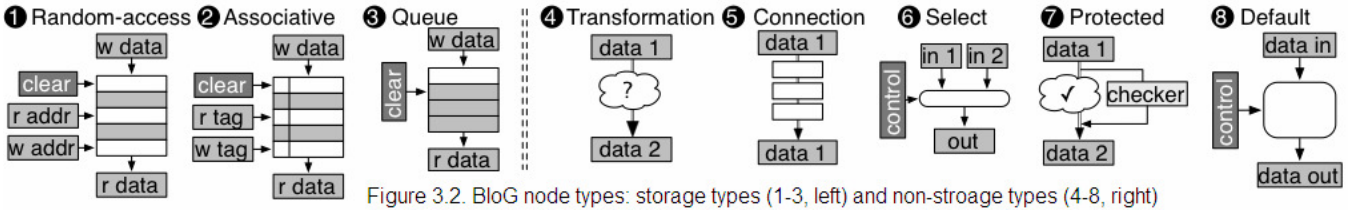

Figure 3.2. BloG node types: storage types (1-3, left) and non-stroage types (4-8, right)

3.2 BLoG EDGE ATTRIBUTES

Self-consistency checks, performed at each node, require signal values corresponding to each node's inputs and outputs. The edge attributes provide a framework for obtaining such values from the recorded footprints. Each edge has the following five attributes:

Attribute 1) Recorder: specifies which recorder to look up in order to obtain the data or control signal values for that edge.

Attribute 2) Auxiliary-information-field selector: specifies which auxiliary information field (e.g., in Table 6.2) in the recorder to look up for obtaining signal values.

Attribute 3) A footprint pointer: specifies which entry in the recorder to look up for obtaining signal values.

Attribute 4) Set of <outgoing edge, edge dependency> pairs: specify *edge dependencies* for a given incoming edge with respect to each outgoing edge.

Attribute 5) Data or control signal values

During BLoG construction, attributes 1, 2 and 4 are specified for BLoG edges. When inspecting a node during BLoG traversal, we start with a footprint pointer at an outgoing edge of the node. Using this pointer and the pre-specified attributes for the outgoing edge, we derive footprint pointers for the incoming edges of the node (Sec. 5.1). The corresponding signal values (in the entries pointed at by the footprint pointers) are obtained by looking up the specified auxiliary information fields of the assigned recorders (Attributes 1 and 2 of the incoming edges). To derive footprint pointers for incoming edges, we use edge dependencies. Each edge dependency describes a relationship between a single outgoing and incoming edge. The six edge dependency types in our BLoG framework are:

Dependency Type 1) Same instruction: signal values (attribute 5) on the outgoing and incoming edges specified in the pair belong to the same instruction (e.g., a Select-type node takes the opcode of an instruction as a control input and selects a register or immediate value for the same instruction).

Dependency Type 2) Same architectural register name: signal values on the outgoing and incoming edges specified in the pair belong to the same architecture register (e.g., a Connection-type node passes a register value produced by one instruction to another instruction that uses the same register name).

Dependency Type 3) Same physical register name: signal values on the outgoing and incoming edges specified in the pair belong to the same physical register (e.g., a Random-access-type node uses physical register name as index to access register value).

Dependency Type 4) Same memory address: signal values on the outgoing and incoming edges specified in the pair belong to the same memory address (e.g., an Associative-type node uses memory addresses as tags for associative access).

Dependency Type 5) Pipeline flush: signal value on the incoming edge is a pipeline flush event (e.g., a Queue-type node is flushed by a pipeline flush event).

Dependency Type 6) Default: Any relationship not included in the aforementioned types.

4. BLoG CONSTRUCTION

BLoG construction involves constructing nodes and edges using the following two inputs: 1) Microarchitecture description of a processor in the form of a microarchitectural block diagram from an architectural manual [Colwell 05] or a language-based specification [Halambi 99, Gorjiara 07] (e.g., similar to [Ketkar 09]); 2) Recorder field description (e.g., Table 6.2).

4.1 Node Construction

A given chip design is divided into as many partitions as possible, while further maximizing: 1) the number of partitions (BLoG nodes) that can be classified into one of seven non-default node types; 2) the number of resulting partition interconnects (BLoG edges) whose signal values can be obtained from recorded footprints. Although more partitions enable more fine-grained bug localization, fine-grained partitions without obtainable signal values may reduce bug localization accuracy.

Once formed, each partition is assigned one of eight node types, ideally a non-default node type. Partitions with storage are assigned types depending on how stored entries are managed. Partitions with protection mechanisms are assigned the Protected type. Partitions consisting of a series of pipeline registers are assigned the Connection type. The Transformation and Select types are used for the remaining partitions with data input/outputs defined. All other partitions belong to the Default type. Figure 5.1 shows an example for a simple branch unit.

4.2 Edge Construction

Once the partitions are defined, partition interconnects become BLoG edges. For each edge, attributes 1, 2 and 4 are assigned by inspection (Sec. 3.2).

5. BLoG TRAVERSAL

BLoG traversal takes four inputs: 1) a BLoG; 2) recorded footprints; 3) a *starting* edge, where BLoG traversal starts; and 4) a footprint pointer assigned to the starting edge. The last two inputs are obtained from <location, footprint> pair, returned by IFRA high-level analysis (Fig. 3.1). For example, if the high-level control-flow analysis (explained in [Park 09]) returns <PC register, 435th entry of fetch-stage recorder> as an inconsistency, then the resulting starting edge is the outgoing edge of the BLoG node corresponding to PC register (Fig. 5.1), and the resulting footprint pointer points to the 435th entry of the fetch-stage recorder assigned to that edge.

The BLoG traversal algorithm is shown in Fig. 5.2. After BLoG traversal terminates, all the candidate labels are returned to report the set of bug candidates in the form of <location, footprint>.

5.1 Footprint-propagation Rules

Footprint-propagation rules help obtain the footprint pointers of the incoming edges once an outgoing edge has an assigned footprint pointer. There are six footprint-propagation rules, each associated with an edge dependency. We present the rules corresponding to dependency types "Same architectural register" and "Default". The rest can be found in [Park 10].

We define a new operator, *follow_link()*, which takes three inputs – a source recorder, a source footprint pointer (points at the source recorder entry that contains a footprint belonging to a particular instruction), and a destination recorder – and returns a destination footprint pointer (points at a destination recorder entry that contains the footprint belonging to the same instruction as the source footprint pointer). The operation follows the links among footprint recorder entries established during footprint linking step.

Denote the first (recorder) and third (footprint pointer) attributes of the outgoing edge (whose footprint pointer has already been assigned) by *Ro* and *Po*, respectively, and the first and third attributes of the incoming edge (whose footprint pointer we are interested in finding) by *Ri* and *Pi*.

• **Same architectural register**: Perform *follow_link* (*Po, Ro,* fetch-stage recorder) to obtain a new pointer into a fetch-stage

recorder entry, *Pf*. Denote the architectural destination register used by the instruction pointed by *Pf* as *R*. Move *Pf* towards older entries until an instruction that uses *R* as its operand is found. Perform *follow_link* (*Pf*, fetch-stage recorder, *Ri*) to obtain *Pi*.

• **Default:** Unless customized propagation rules are designed for the default case, assign NULL to *Pi* and report that no information is available.

If the end of recorded history is reached (*i.e.*, the sought footprint was overwritten due to limited storage) while performing *follow_link()* or moving pointers, return NULL for *Pi* .

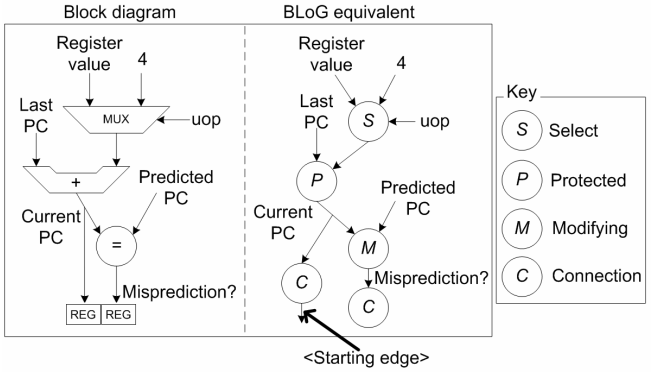

Figure 5.1. Example starting edge (without edge attributes).

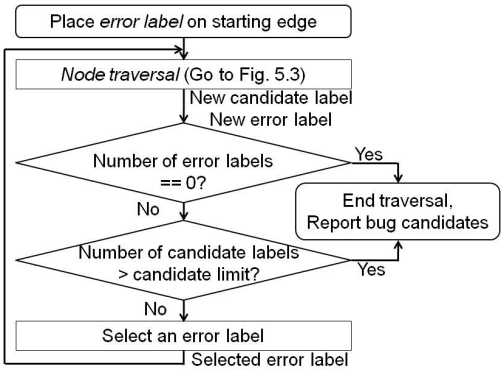

Figure 5.2. BLoG traversal algorithm.

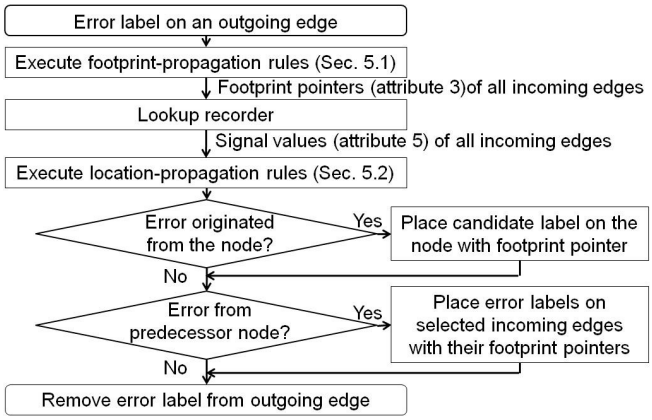

Figure 5.3. Node traversal algorithm.

5.2 Location-propagation Rules

Given a node with signal values on incoming and outgoing edges, location-propagation rules are responsible for:

1. Propagating error labels from outgoing to incoming edges; and/or

2. Creating candidate labels on one of the node's *localization regions* (individual regions discussed in Table 6.1)

Each rule requires signal values on some incoming edges, but values may not be present due to NULL footprint pointer. There are two possible reasons (Sec. 5.1):

• The end of history is reached. In this case, any rule using this incoming edge will not produce any error or candidate labels.

• Not all edge attributes are specified or a Default-type edge dependency is used. In this case, any rule using this incoming edge will produce an error label on this edge with a footprint pointer obtained by following the link from the pointer on the outgoing edge to the recorder assigned to the incoming edge (i.e., assuming same instruction). If the rule can place a candidate label, then this label is placed with the pointer on the outgoing edge.

Location-propagation rules for two node types are presented next. Rules for remaining node types appear in [Park 10].

5.2.1 Location-propagation Rules for a Select-type Node

A Select-type node consists of a multiplexer with logic driving the select decision (Fig. 5.4). 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.

Note that, executing these rules does not require any simulation.

```
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;}
```

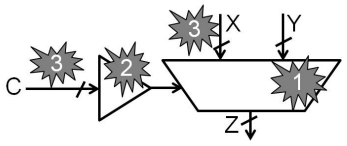

Figure 5.4. Location-propagation rule outcomes.

5.2.2 Location-propagation Rules for a Default-type Node

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 is.

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

Method 3) Create customized location-propagation rules.

6. RESULTS

We used the techniques in this paper to construct a BLoG for the Intel Nehalem microarchitecture and evaluated BLoG-assisted IFRA on an industrial-grade, cycle-accurate IA-32 microarchitecture simulator that is extensively used during product development.

The constructed BLoG contains 160 BLoG nodes, of which 86% are non-default types. Table 6.1 shows the number of nodes used and the localization regions for each type. The Default-type nodes were manually augmented (details in [Park 10]) with customized location-propagation rules adapted from the IFRA decision diagram in [Park 09].

Table 6.1. BloG node type distribution for Intel Nehalem.

Node type	Number of nodes	Localization regions
Random-access	6	Read/write circuitry, clear handler
Associative	8	Read/write circuitry, tag handler, clear handler
Queue	7	Enqueue/dequeue circuit, FIFO management
Modifying	22	Entire node
Connection	46	Entire node
Select	42	Multiplexer, logic driving select signal
Protected	6	No error occurs in the node (pessimistic)
Default	23	Avg(2), std(1), min (1), max (4)
Total number of nodes: 160		
Total number of locations: 269		

The IA-32 microarchitecture simulator was augmented with IFRA recording infrastructure (Table 6.2). We also included IA-32 in-built exceptions [Intel 08] as post-triggers for BLoG-assisted IFRA. While running programs from the SPECint 2006 benchmark suite, error injection campaign was conducted at error injection sites grouped into 32 categories (details in [Park 10]). Each category consists of 1-400 bits. To be pessimistic, no errors were injected in array structures or arithmetic units which are assumed to be protected using parity and residue codes, respectively. All bugs were modeled as single bit-flips at flip-flops to target hard-to-repeat electrical bugs. This is an effective model because electrical bugs eventually manifest themselves as incorrect values arriving at flip-flops for certain input combinations and operating conditions [McLaughlin 09].

Out of over 30,000 error injection runs, 2,560 runs resulted in a system failure. Figure 6.1 presents the localization results for those 2,560 runs. For over 90% of the cases, BLoG-assisted IFRA identified the correct candidate that matched both the location (1 out of 269) and cycle (1 out of over 1,000 footprints) of error injection. Out of these 90% cases with correct localization, BLoG-assisted IFRA returned a single correct bug candidate (i.e., <location, footprint> pair) for 62% of the cases (referred to as *exactly localized*) and returned multiple candidates (average of 6 out of over 269,000 possible <location, footprint> pairs) for the remaining 38% of the cases (referred to as *multiple candidates localized*). To be pessimistic, the *no localized* category includes any cases where either the location or the footprint is incorrect.

Table 6.2.Auxiliary information for recorder entries corresponding to various pipeline stages.

Pipeline stage	Auxiliary information: Description	Auxiliary information: Bits per entry	Number of Recorders	Entries per Recorder
Fetch	Instruction Pointer	32	4	512
Decode	Microcode, beginning of macroinstruction, 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 contains an additional 8-bit ID)			66KBytes	

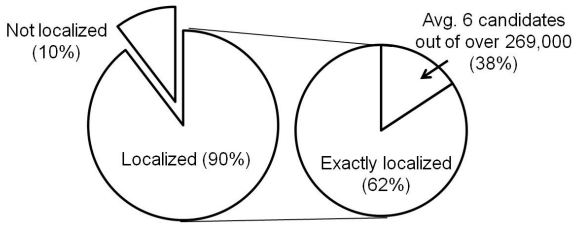

Figure 6.1. BLoG-assisted IFRA bug localization summary.

7. RELATED WORK

Comparison of IFRA vs. existing post-silicon bug localization techniques has been extensively discussed in [Park 09] and is not included in this paper. Unlike most existing techniques, BLoG-assisted IFRA overcomes major barriers because it does not rely on system-level failure reproduction or system-level simulation. Previous work related to the BLoG idea may be largely classified into two categories: high-level test generation (e.g., [Lee 94, Tupuri 97, van Campenhout 99]) and circuit-level fault diagnosis (e.g., [Caty 05, Venkataraman 96]). Table 7.1 presents a qualitative comparison of BLoG, which has been demonstrated for a large and complex design, vs. related techniques. While error detection techniques for processors in the context of fault-tolerant

computing (e.g., [Lu 82, Oh 02]) also use self-consistency checks, their main purpose is to detect errors only. The specific bug localization application enables us to perform deep self-consistency analysis (as presented in this paper) that is generally difficult and expensive to perform on-line during system operation for fault-tolerant computing.

Table 7.1. Comparison with existing techniques.

	High-level test generation	Circuit-level diagnosis	BLoG
Purpose	Test patterns	Defect diagnosis	Bug localization
Failure reproduce	N/A	(-)Required	(+) Not required
System-level simulation	(-)Required		(+)Not required
Abstraction	Arch./circuit	Circuit	Arch.
Granularity/ Complexity	Medium	High	Low

8. CONCLUSIONS

BLoG-assisted IFRA overcomes an outstanding challenge in post-silicon validation of processors: post-silicon bug localization (which is the most expensive and difficult step) without relying on system-level failure reproduction or system-level simulation. While we presented the idea of IFRA in the past, the major innovations in this paper is the idea of BLoG which enables systematic construction and automated execution of IFRA's post-analysis for bug localization purposes. Without BLoG, it would be difficult, if not impossible, to apply IFRA to new microarchitectures. The BLoG framework is highly effective and practical as demonstrated by high bug localization accuracy of 90% obtained for a large complex commercial microarchitecture – the Intel Nehalem design. The BLoG concept creates several interesting research directions:

- Automatically constructing BLoG from a language-based specification or an RTL description.
- Automatically selecting what information to record for a BLoG borrowing concepts from research done at the circuit-level [Ko 08].
- Enhancing the idea of BLoG for homogeneous / heterogeneous multi-core systems, and system-on-chips (SoCs).

9. ACKOWLEDGMENTS

The authors thank Joe Schutz, Shekhar Borkar, Per Hammarlund, Ronak Singhal, Mahesh Madhav, Nagib Hakim, Jag Keshava, Gadi Ziv, Doug Josephson, Priyadarsan Patra and Jim Schwartz of Intel for productive collaboration, guidance and support. This research is supported in part by the Semiconductor Research Corporation and the National Science Foundation. Sung-Boem Park was also partially supported by the Samsung Scholarship.

10. REFERENCES

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

[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.

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

[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*, pp. 356-361, Oct. 2007.

[Halambi 99] Halambi, A. et al. "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability," *Proc. Design Automation and Test in Europe*, pp.485-450, 1999.

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

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

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

[Ketkar 09] Ketkar, M. and E. Chiprout, "A Microarchitecture-based Framework for Pre- and Post-silicon Power Delivery Analysis," *Proc. Intl. Symp. Microarchitecture*, pp. 178-188, 2009.

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

[Lee 94] Lee, J. and J.H. Patel, "Architectural level Test generation for microprocessors," *IEEE Trans. CAD,* Vol. 13, No. 10, pp.1288-1300, Oct. 1994.

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

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

[Oh 02] Oh, N., P.P. Shirvani, and E.J. McCluskey, "Control-flow Checking by Software Signatures," *IEEE Trans. Reliability*, Vol. 51, No. 2, pp.111-122, March 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. CAD,* Vol. 28, No. 10, pp. 1545-1558, Oct. 2009.

[Park 10] Park, S., A. Bracy, H. Wang and S. Mitra., "BLoG: Post-Silicon Bug Localization in Processors using Bug Localization Graph," *Technical Report*, Stanford University, 2010, url: http://www.stanford.edu/group/rsg_csl.

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

[Tupuri 97] Tupuri, R.S., J.A. Abraham, "A Novel Functional Test Generation Method for Processors using Commercial ATPG," *Proc. Intl. Test Conf.*, pp. 743-752, 1997.

[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.*, pp. 185-188, 1999.

[Venkataraman 96] Venkataraman, S., I. Hartanto, and K. Fuchs, "Dynamic diagnosis of sequential circuits based on stuck-at faults," *Proc IEEE VLSI Test Symp.*, pp.198-203, 1996.