RTL-Level
↳ RTL

Online Design Bug Detection: RTL~~-Level~~ Analysis, Flexible Mechanisms, and Evaluation

Kypros Constantinides† Onur Mutlu§ Todd Austin†

†Advanced Computer Architecture Lab
University of Michigan
{kypros, austin}@umich.edu

§Microsoft Research and Carnegie Mellon University
onur@{microsoft.com,cmu.edu}

Abstract

Higher level of resource integration and the addition of new features in modern multi-processors put a significant pressure on their verification. Although a large amount of resources and time is devoted to the verification phase of modern processors, many design bugs escape the verification process and slip into processors operating in the field. These design bugs often lead to lower quality products, lower customer satisfaction, diminishing brand/company reputation, or even expensive product recalls.

This paper proposes a flexible, low-overhead mechanism to detect the occurrence of design bugs during on-line operation. First, we analyze the actual design bugs found and fixed in a commercial chip-multiprocessor, Sun's OpenSPARC T1, to understand the behavior and characteristics of design bugs. Our RTL~~-level~~ analysis of design bugs shows that the number of signals that need to be monitored to detect design bugs is significantly larger than suggested by previous studies that analyzed design bugs at a high~~-~~level using processor errata sheets. Second, based on the insights obtained from our analyses, we propose a programmable, distributed online design bug detection mechanism that incorporates the monitoring for bugs into the flip-flops of the design. The~~y~~ key contribution of our mechanism is its ability to monitor *all* control signals in the design rather than a set of signals selected at design time. As a result, it is very flexible: when a bug is discovered after the processor is shipped, it can be detected by monitoring the set of control signals that trigger the design bug.

We develop an RTL~~-level~~ prototype implementation of our mechanism on the OpenSPARC T1 chip multiprocessor. We found its area overhead to be 10% and its power consumption overhead to be 3.5% over the whole OpenSPARC T1 chip. We show that the hardware substrate used to enable our proposal can also be used for enabling hardware defect detection. As a result, we propose a unified *bug detection* and *defect detection* solution that increases overall dependability during on-line operation.

but the area overheads are additive?
→ needed?

1. Introduction

The Challenges of Correct Design - The advent of chip-multiprocessing has led to unprecedented levels of chip integration. Today, most general purpose processor chips are equipped with multiple cores, multiple levels of coherent memory, on-chip interconnect networks, and memory and I/O controllers. ~~The~~ Complex interactions between these modules, as well as the complexity of the modules themselves, put a tremendous pressure in the verification of the system. Although the verification phase of modern processors can consume a large portion of the design cycle [3], require significant amount of resources [7], and utilize state-of-the-art verification techniques, *design bugs* (also known as errata, design defects, or design errors) still slip into the final products and *"buggy"* processors find their way into the field. This trend is clearly shown in ~~the study of~~ Figure 1. We studied the errata documentation of five recent Intel processors and ~~we~~ found that the rate of design bugs discovered after product release has more than doubled in the latest generation of processors. The graph

rewrite

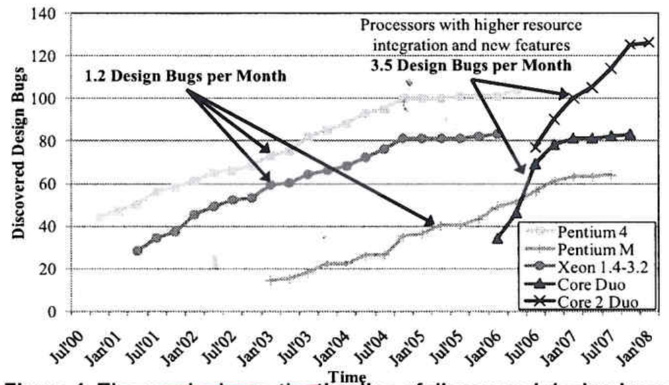


Figure 1. ~~The graph shows the~~ timeline of discovered design bugs over the lifetime of five Intel processors.

move to the end of previous sentence

shows the number of discovered design bugs over the lifetime of five Intel processors.[1] The Pentium 4, Pentium M and the Xeon 1.4-3.2 processors exhibit a similar trend with an average of 1.2 design bugs discovered per month during their lifetime. We suspect that the reason why the Pentium M processor had less design bugs than the other two processors is because it was based on the matured Intel P6 architecture. ?

disconnected. Not clear we are still talking about Fig 1.

The advances in silicon process technology that offered higher device integration and the power consumption challenges of complex single-core architectures led to the advent of chip-multiprocessing where two or more cores are plugged into a single chip. However, putting more cores into a single chip required the addition or adaption of other on-chip resources such as the memory subsystem for supporting on-chip shared/coherent caches and on-chip interconnection networks. At the same time, processors where augmented with new technologies such as virtualization, dynamic power management, and 64-bit extensions. As shown in the graph, this higher chip-level integration of resources and the addition of new features resulted in~~to~~ more design bugs. For example, although ~~that~~ the Core Duo dual-core processor was derived from the Pentium M single-core processor and had the same architecture, it exhibited a much higher rate of design bugs than its predecessor. This trend also holds with the newest generation of Intel's multi-core processors, the Core 2 Duo. Specifically, the design bug discovery rate of the two multi-core processors is 3.5 design bugs per month, almost triple that of their single-core predecessors. This trend is expected to ~~exacerbate~~ worsen in the future as technology scaling will allow for more diverse resources to be integrated into a single chip.

Why is Online Bug Detection ~~is~~ Needed? Today, design bugs are treated with *ad-hoc* heuristic techniques that seek to avoid the occurrence of design bugs through software and hardware configuration changes [16]. A common approach employed by such techniques to avoid the occurrence of design bugs is disabling some processor features that trigger the design bugs (*e.g.*, support for cache prefetch-

[1] The data is extracted from the processors' errata documentation [12, 11, 8, 10, 9].

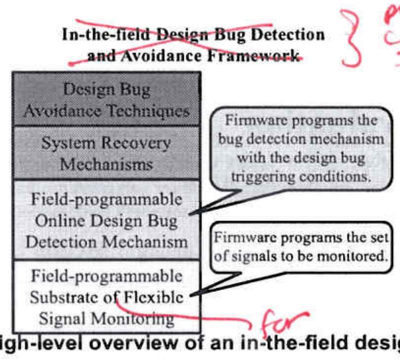

~~In-the-field Design Bug Detection and Avoidance Framework~~
put into the caption
Design Bug Avoidance Techniques
System Recovery Mechanisms
Field-programmable Online Design Bug Detection Mechanism
Field-programmable Substrate ~~of~~ for Flexible Signal Monitoring
Firmware programs the bug detection mechanism with the design bug triggering conditions.
Firmware programs the set of signals to be monitored.

Figure 2. A high-level overview of an in-the-field design bug detection and avoidance framework.

ing [16], dynamic power management [1], *etc.*). However, this often leads to reduced product quality/performance and lower customer satisfaction. Furthermore, when such workarounds are not possible, design bugs can lead to expensive product recalls [27] and a potentially diminishing brand/company reputation.

Augmenting a design with a mechanism that enables a systematic approach to detect and avoid design bugs after the product release and while the system is operating in the field can offer the following benefits:

1. Faster design cycle and time to market. Today, a significant fraction of the verification phase is spent to discover a very small number of design bugs [6]. This time can be saved by discovering and fixing that small number of design bugs in the field after product release.
2. Reduce the risk of expensive product recalls (and potentially damaged company reputations) due to *ad-hoc* heuristic techniques that might not be able to avoid a discovered design bug. A systematic online design bug detection technique increases the probability of successfully dealing with the design bug and avoiding expensive recalls.
3. Avoid potential impact~~s~~ to product quality and customer satisfaction due to the use of conventional techniques that disable design features to avoid design bugs. Instead, online design bug detection allows the system to operate with all its features enabled and recover the system only when the design bug occurs. Therefore, during bug-free execution the system is operating under its original specifications.

Online Design Bug Detection ~~&~~ and Avoidance - Figure 2 ok ✓ provides a high-level overview of our in-the-field design bug detection and avoidance framework which we will describe in this paper. The framework has four layers: 1) The bottom layer ~~that~~ provides a field-programmable substrate for flexible signal monitoring. This substrate is programmed by special firmware at system startup to select the set of signals that are required to be monitored for design bug detection. 2) A field-programmable design bug detection mechanism that checks if the monitored signals match with a bug triggering condition. The mechanism is programmed by special firmware at system startup with the bug triggering conditions. 3) A system recovery mechanism that rolls back the system state to the last correct state when a design bug occurrence is detected. 4) Design bug avoidance techniques that are activated after a design bug detection to guide execution around the bug triggering conditions and avert the design bug. *In this paper we focus on the first two layers and provide a novel mechanism for performing flexible signal monitoring and online design bug detection.*

1.1. Contributions of this Work

~~This work provides a solid foundation in the area of online design bug detection. Specifically,~~ this work makes the following contributions in the area~~:~~ of online design bug detection:

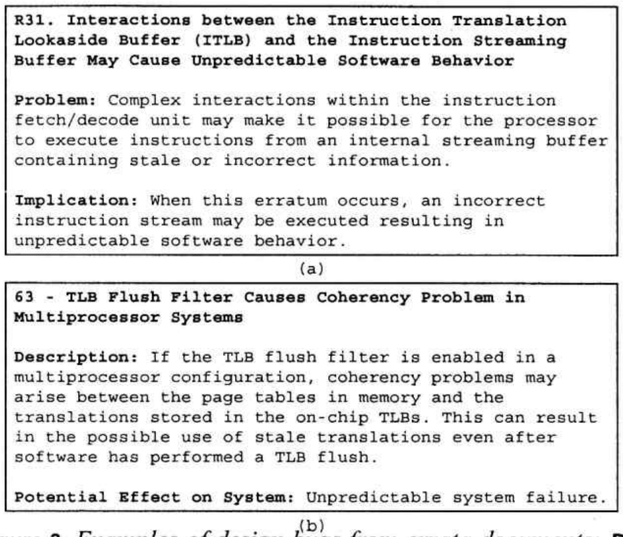
- ~~In this paper,~~ We further the understanding in online design bug detection by performing a rigorous analysis of the design bugs in the OpenSPARC T1 chip-multiprocessor, the open source version of Sun's Niagara processor. Unlike previous works that based their design bug analyses on high-level abstract descriptions of the processor errata documentation, our analysis is performed at the RTL-~~level~~ model of the design, thus enabling the extraction of low-level information directly related to the actual hardware implementation. To the best of our knowledge this is the first RTL-level design bug analysis performed ~~to date~~. With our low-level analysis, we found that the signal monitoring requirements of online design bug detection are significantly higher than the estimates of previous studies. As a result, the problem of detecting design bugs is more difficult and the solution is likely more hardware intensive than estimated by previous work. needed?
- Based on the insights obtained from our RTL-level analysis of design bugs, we propose a novel distributed online bug detection mechanism. Unlike the mechanisms proposed in previous work, our mechanism can concurrently monitor *all* the control signals in the design that can trigger a design bug. This feature waives the requirement of selecting the candidate monitor signals at design time as required by previously proposed mechanisms. Therefore, our mechanism has no restriction on the design bugs that can be detected. needed?
- We provide a detailed hardware implementation of our mechanism. Unlike previously proposed mechanisms that route selected signals from the source flip-flops to a centralized monitoring mechanism that checks for bug triggering conditions, our mechanism distributes the monitoring and checking process at the flip-flop level. The distributed checking results are aggregated using a hierarchical tree structure. This implementation offers low overhead and high runtime flexibility for our mechanism. needed?
- We show that our online design bug detection mechanism can be synergistically combined with a previously proposed online hardware defect detection mechanism. The combination implements a thorough solution that provides a high degree of both reliability and dependability to a system operating in the field. We show that the hardware used for the detection of design bugs can partially be used for detecting hardware defects, thereby amortizing the cost of both mechanisms.
- We provide an extensive evaluation of our mechanism. Specifically, we use as our test case design the OpenSPARC T1 chip-multiprocessor to develop an RTL-level prototype implementation and estimate its overhead in terms of silicon area and power consumption. needed?

2. Design Bug Analysis

We first analyze design bugs in a real processor to obtain insights into their characteristics and to develop a mechanism that can flexibly and efficiently detect the occurence of design bugs while the system is in operation.

2.1. Previous Design Bug Analysis Studies

The potential of augmenting future microprocessors with online design bug detection has led to a number of studies that analyzed the known design bugs that slipped into recent commercial microprocessors. The objective of these studies was to better understand and gain insights into the characteristics of the known design bugs in existing microprocessors, and extrapolate the expected characteristics of the design bugs of future microprocessors. These insights were later used to devise techniques that aimed to detect design bugs discovered after ~~the~~ product release and while the processors are already shipped and operating in the field. The main premise of these techniques is that most design bugs are detectable by tapping a set of hardware signals and checking for bug triggering conditions while the processor is operating.



```
R31. Interactions between the Instruction Translation
Lookaside Buffer (ITLB) and the Instruction Streaming
Buffer May Cause Unpredictable Software Behavior

Problem: Complex interactions within the instruction
fetch/decode unit may make it possible for the processor
to execute instructions from an internal streaming buffer
containing stale or incorrect information.

Implication: When this erratum occurs, an incorrect
instruction stream may be executed resulting in
unpredictable software behavior.
```

(a)

```
63 - TLB Flush Filter Causes Coherency Problem in
Multiprocessor Systems

Description: If the TLB flush filter is enabled in a
multiprocessor configuration, coherency problems may
arise between the page tables in memory and the
translations stored in the on-chip TLBs. This can result
in the possible use of stale translations even after
software has performed a TLB flush.

Potential Effect on System: Unpredictable system failure.
```

(b)

Figure 3. *Examples of design bugs from errata documents:* **Design bug descriptions from (a) the Pentium 4 errata sheet, and (b) the Opteron errata sheet. Both design bug descriptions are limited to a high-level description that is hard to associate with low-level details of the underlying hardware ~~implementation.~~** needed? shorter

Specifically, Avžienis et al. [2] analyzed the known design bugs in the Intel Pentium II since its initial release. More recently, Sarangi et al. [20] analyzed the design bugs in ten modern commercial microprocessors from Intel, AMD, IBM and Motorola, and Narayanasamy et al. [17] analyzed the design bugs in two microprocessors: Intel's Pentium 4 and AMD's Athlon 64. Another study by Wagner et al. [26] analyzed the design bugs in Intel StrongARM SA1100 and IBM PowerPC 750GX. The analysis in all of these studies was based on information extracted from the available microprocessor errata sheets [13, 1, 5]. An errata sheet is a document published and maintained by the microprocessor manufacturer to provide its customers with details about known microprocessor design bugs. The sheet provides an assessment of each design bug's severity, the degree to which it can affect a running system, a possible set of conditions that can trigger the design bug, any possible workarounds, and sometimes the company's intention to provide a fix in a future version of the product.

A major drawback of using the errata sheets to extrapolate statistics about design bugs is that the errata sheets commonly provide very high-level descriptions of the design bugs. Such descriptions provide little or no insight into the low-level details of the underlying hardware problem. An example description of a design bug listed in the Intel Pentium 4 errata sheet [13] is shown in Figure 3(a). This design bug is related to complex interactions between the processor's instruction translation lookaside buffer and the instruction streaming buffer that can result in the execution of an incorrect instruction stream with unpredictable software behavior. Using this description, it is very hard to accurately relate this design bug to the actual hardware implementation and reason about, for example, exactly what hardware signals (*i.e.*, wires) need to be monitored by an online design bug detection mechanism to effectively detect the occurrence of the design bug. Figure 3(b) shows another example design bug description, from AMD's Opteron errata sheet [1]. This bug is related to the translation lookaside buffer flush filter and can lead to unpredictable system behavior. Again, from this high-level description, it is impossible to infer the set of hardware signals that should be examined to dynamically detect its occurrence. Without knowing the set of hardware signals that needed to be monitored to detect the bug, it is impossible to design a mechanism that would detect the bug and to accurately estimate the hardware cost of such a mechanism.

Our Goal: In order to design a hardware mechanism that detects design bugs, the signals that affect the occurrence of each bug need to be known. Our goal in this section is to perform a more rigorous, lower-level (RTL~~-~~level) analysis of design bugs. Our purpose is to understand design bug characteristics at the register transfer level to (1) design a flexible mechanism that can detect known design bugs during online operation after the chip is manufactured, and (2) more accurately estimate the hardware cost of such a design bug detection mechanism. To this end, we first draw insights from our analysis of design bugs found and fixed in an existing commercial processor, Sun's OpenSPARC T1.

2.2. RTL~~-Level~~ Design Bug Analysis e.g.

We perform an RTL~~-level~~ design bug analysis in an attempt to bridge the gap between the high-level design bug descriptions provided by the microprocessor errata sheets and the low-level hardware implementation details needed to devise effective online design bug detection mechanisms. At the RTL level, the microprocessor design behavior is described in a hardware description language ~~(most commonly~~ Verilog or VHDL). This level is considered to be very close to the actual hardware implementation. The only design phases separating the RTL level with the actual hardware implementation are 1) logic synthesis, which generates the design's gate-level netlist and 2) place-and-route, which creates the transistor-level layout of the netlist. Therefore, the direct relation between the RTL level and the underlying implementation provides an adequate level of detail that allows the extraction of low-level design bug characteristics.

Our study focuses on the Verilog RTL source code of the OpenSPARC T1 chip-multiprocessor [23], the open source version of Sun's commercial UltraSPARC T1 (Niagara) chip-multiprocessor. Since no errata documentation is publicly available for the UltraSPARC T1 microprocessor, we focus on the actual design bugs found during the development of the OpenSPARC T1 and documented in the RTL source code. Specifically, when the designers corrected a design bug, they left the original buggy code in the RTL source file as a comment. Therefore, both the original erroneous implementation as well as the fixed implementation are available in the source code. As such, by examining these two implementations, it is straightforward to discover what hardware signals are involved in each design bug. Although these design bugs did not slip into the final product, we believe they share ~~very~~ similar characteristics with the design bugs that eventually slipped into the released version of the microprocessor with the exception of some differences which we discuss in the next section.

Methodology: We analyzed 296 design bugs that were documented in the Verilog source files of two OpenSPARC ~~core units~~. and corrected These bugs account for about 99% of all documented bugs in the OpenSPARC T1 ~~design~~ RTL. We classified these bugs into three major classes: 1) *Logic* design bugs, 2) *Algorithmic* design bugs, and 3) *Timing* design bugs. Later, in Section 3, we analyze the logic signals that need to be monitored to detect these bugs.

2.3. Classification of Design Bugs

Logic Design Bugs: This class of design bugs is characterized by erroneous logic in combinational circuits. A logic bug occurs because the designer formed an erroneous logic block; for example an AND gate could be used instead of an OR gate, or an inverted signal rather than the non-inverted one. The code segment presented in Figure 4, taken from the OpenSPARC T1 Verilog source files, illustrates an example of a logic design bug. The design bug is located in the core's trap logic unit (TLU) and is associated with the combinational logic that computes the control signal trap_to_redmode. The incorrect combinational circuit implementation is commented out in line 1106. The corrected combinational circuit implementation is shown in line

Correct Code

Buggy Code

```
Example 1 from Verilog file tlu_tcl.v

line 1089:   assign  intrpt_taken =
line 1090:               rstint_taken | hwint_taken | sirint_taken;
...
line 1105:   // modified for bug 3919
line 1106:   // assign        trap_to_redmode = trp_lvl_at_maxtlless1 & ~intrpt_taken;
line 1107:   assign  trap_to_redmode = trp_lvl_at_maxtlless1 & ~(rstint_taken | sirint_taken);
```

Figure 4. *Example of a logic design bug at the RTL level.*

1107. By examining lines 1089-1090, we notice that the signal replaced in the correct code (`intrpt_taken`) is computed by ORing three other signals. One of the three signals (`hwint_taken`) is not any more a source signal in the correct implementation. We observed that many logic design bugs cannot be fixed by simply redefining the logic between the source signals in the buggy implementation. Instead, it is very common that fixing the bug requires the addition or removal of signals to/from the buggy implementation (more than 95% of logic design bugs had this requirement).

This example demonstrates the amount of low-level information provided at the RTL-level that is missing from the design bug descriptions in the errata documentation. For instance, by observing the code segment associated with the design bug, it is very easy to find the set of hardware signals that activate the bug (*i.e.*, `trp_lvl_at_maxtlless1`, `rstint_taken`, `hwint_taken`, and `sirint_taken`). *In analyses solely based on errata sheets, this low-level information is abstracted away in the high-level design bug description and has to be inferred; a process that involves a high amount of uncertainty and inaccuracy.*

Algorithmic Design Bugs: This class covers major design bugs related to the algorithmic implementation of the design. These design bugs exhibit algorithmic deviations from the design specification and they usually require major modifications to be fixed. Figure 5 illustrates an example algorithmic design bug located in the load queue control logic at the core's load/store unit. This bug is due to an incorrect implementation of the round robin algorithm for selecting one of the four loads buffered in the load queue. To fix the incorrect round robin implementation described in module `lsu_rrobin_picker1`, a new module had to be implemented (`lsu_rrobin_picker2`). Unlike fixes for logic design bugs, fixes for algorithmic design bugs are not limited to combinational circuit modifications, rather they sometimes require multiple major modifications that can span the whole module.

Timing Design Bugs: This third class of design bugs ~~are~~ is associated with the timing correctness of the implementation. We have observed that most of these design bugs are cases where a signal needed to be latched a cycle earlier or a cycle later in order to keep the timing of signals correct in the design. An example of such a design bug is shown in Figure 6. This timing design bug is located in the queue data path of the core's load/store unit. As shown in the source Verilog code, the incorrect implementation at line 1248 assigns the value of the 48-bit `tlb_st_data_d1` bus to the `lsu_ifu_stxa_data` bus in the same cycle. However, as shown in lines 1239-1244, the correct timing of the data movement between the two buses requires the data to be latched for one clock cycle. We found that the most common fix for this class of design bugs is the addition or removal of flip-flops to adhere to the required timing constraints to keep the design correct.

2.4. Design Bug Type Distribution

After studying the OpenSPARC T1 source Verilog files [22] we found that almost all (~99%) of the documented design bugs are located in two units, the load/store unit (LSU) and the trap logic unit (TLU) [23], shown in Figure 7(a).[2] The LSU processes all data memory access instructions. It interfaces with all the functional units and it serves as the gateway between the SPARC core and the core-cache crossbar to the memory subsystem. The LSU also includes the core's data TLB and L1 cache. The TLU implements the SPARC core's trap and software interrupt handling logic. It supports six trap levels ranging from hypervisor and supervisor mode traps to user mode traps and is capable of handling up to 64 pending software interrupts per thread. In our study we analyzed a total of 296 design bugs documented in these two units.

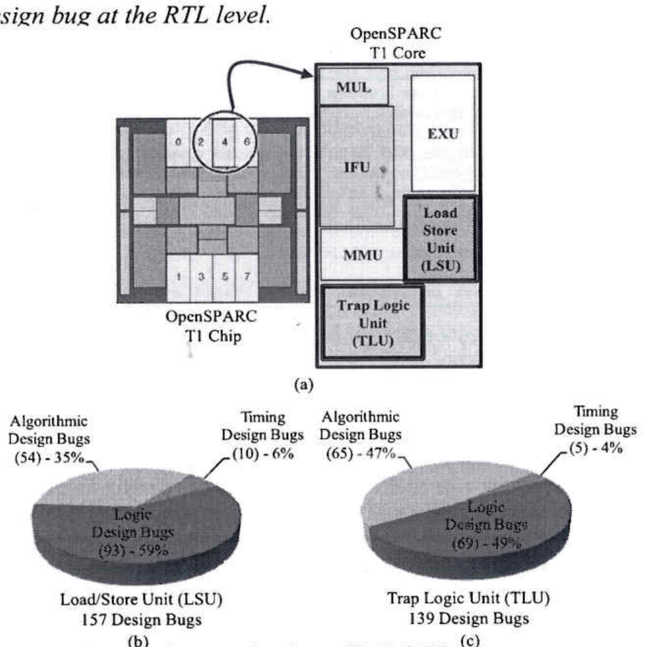


Figure 7. *Design bugs in the OpenSPARC T1 core:* **Part (a) shows the OpenSPARC T1 core. Parts (b) and (c) show the design bug distribution for the Load/Store unit (LSU) and the Trap Logic unit (TLU) respectively.**

shorten too long

Figure 7(b-c) shows the design bug type distribution. A large fraction of the documented design bugs in the two units belong to the logic design bug class, which accounts for 59% and 49% of the total design bugs for the LSU and the TLU respectively. The second most frequent design bug class is algorithmic design bugs, while timing design bugs are less frequent and account for only ~5% of the total bugs. The dominance of logic design bugs over the other two bug classes can imply that the process of implementing complex combinational logic is more prone to human error than implementing the algorithmic or timing specifications of the design.

Furthermore, as mentioned earlier in this section, these design bugs were discovered, fixed, and documented before the final tape-out of the design. As such, we expect them to have some differences with the design bugs that escape the verification phase and slip into the final product. We suspect that the algorithmic and timing design bugs

[2] The concentration of the bugs in these two units might stem from the logic complexity involved in these units. Alternatively, there is a likelihood that the verification methodology employed in the other units did not follow the same bug documentation approach, resulting in source files without any bug-related information in the other units.

```
Example from Verilog file lsu_qctl1.v

line 2993:    //bug4814 - change rrobin_picker1 to rrobin_picker2
line 2993:    // Choose one among 4 loads.
line 2994:    //lsu_rrobin_picker1 ld4_rrobin  (
line 2995:    //     .events          ({ld3_pcx_rq_vld,ld2_pcx_rq_vld,
line 2996:    //                 ld1_pcx_rq_vld,ld0_pcx_rq_vld}),
...
line 3007:    //     .se(se),
line 3008:    //     .so()
line 3009:    //  );
line 3010:
line 3011:    lsu_rrobin_picker2 ld4_rrobin   (
line 3012:         .events        ({ld3_pcx_rq_vld,ld2_pcx_rq_vld,ld1_pcx_rq_vld,ld0_pcx_rq_vld}),
...
line 3020:      .se(se),
line 3021:      .so()
line 3022:   );
```

Buggy Code

Correct Code

put into previous page

Figure 5. *Example of an algorithmic design bug at the RTL level*

```
Example from Verilog file lsu_qdp1.v

line 1228:    // Begin - Bug3487.
...
line 1239:    dff  #(48) ifu_std_d1 (
line 1240:            .din      (tlb_st_data[47:0]),
line 1241:            .q        (lsu_ifu_stxa_data[47:0]),
line 1242:            .clk      (asi_data_clk),
line 1243:            .se       (1'b0),      .si (),           .so ()
line 1244:            );
line 1245:
line 1246:    // select is now a stage earlier, which should be
line 1247:    // fine as selects stay constant.
line 1248:    //assign  lsu_ifu_stxa_data[47:0] = tlb_st_data_d1[47:0] ;
line 1249:
line 1250:    // End - Bug3487.
```

Correct Code

Buggy Code

Figure 6. *Example of a timing design bug at the RTL level.*

have a more severe impact on the design's correctness and therefore they have a higher probability of being discovered during the design verification phase. In contrast, because logic design bugs are isolated and localized to small combinational logic portions, they could be less likely to be discovered during the verification of the chip. This is because the erroneous effects of the logic design bugs either might not be exercised or might be masked before propagating to observable outputs during testing. For example, in order for the logic bug illustarted in Figure 4 to be active, the source combinational circuit must be set to specific values (which might be an infrequent combination of values). Furthermore, even in the case that a logic bug does get active, its effect can be cancelled out by subsequent logic or microarchitectural effects and ~~do not~~ not affect in any way the correct execution.[3] Based on this ~~observation~~ reasoning, the distribution of design bugs that actually slip into the final product ~~could~~ have fewer algorithmic and timing design bugs than the distribution shown in Figure 7(b-c) and the bugs in the shipped product might be heavily dominated by logic design bugs.

rewrite

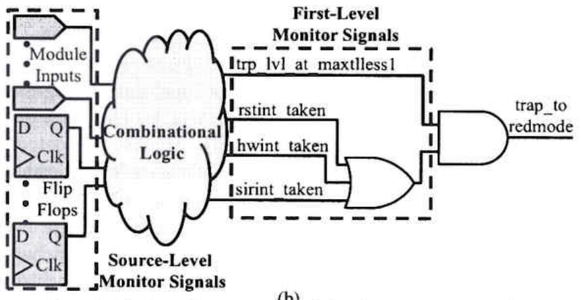
bugs may

3. Detecting Logic Design Bugs at Runtime

Although logic design bugs might be harder to discover than the other two design bug classes, we believe that once they have been discovered, it is much easier to detect their occurrence while the "buggy" microprocessor is in operation in the field. Their characteristic of being isolated in a combinational logic circuit portion makes it possible to deterministically detect their occurrence by monitoring the values of their source signals. To illustrate this concept, we consider the logic bug example shown in Figure 4. By computing the truth table of the buggy circuit (line 1106) and the correct circuit (line 1107), as shown in ~~the table of~~ Figure 8(a), we can infer that the design bug occurs when the source signals are set to a specific combination of values (shown in the shaded column of the table). Therefore, by monitoring the values of the bug's source signals it is possible to deterministically detect the occurrence of the specific design bug. In this work, we call this set of signals *first-level monitor signals* (*i.e.*, signals that directly determine the occurrence of the design bug). For this specific bug, the size of the first-level monitor signal set is 4 because there are 4 signals whose values directly determine the bug's occurrence.

Input Signals	trp_lvl_at_maxtlless1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	rstint_taken	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	hwint_taken	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	sirint_taken	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Output Signal	trap_to_redmode (buggy code)	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	trap_to_redmode (correct code)	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0

(a)



(b)

Figure 8. ~~*Logic design bug truth table & source signals.*~~ **Part (a): The logic bug shown in Figure 4 is triggered whenever its source signals take the values shown in the shaded column. Part (b): The source-level and first-level signals for the same logic bug.**

reduce

[3] This is similar to the masking phenomenon observed in glitches occuring in combinational logic due to soft errors.

active vs. exercised?

Although it is easy to find the set of first-level monitor signals to ~~at~~ in the RTL ~~level~~, these signals unfortunately might not exist in the lower transistor-level implementation due to the logic synthesis process and optimizations employed during the process of translating the RTL-~~level~~ implementation to ~~the~~ gate-level and then transistor-level implementation [24]. Thus, there is not a guaranteed one-to-one mapping between signals in the RTL-~~level~~ and signals in the transistor-level implementations. However, the logic synthesis process maintains a one-to-one mapping of the state-holding elements (*e.g.*, flip-flops) and module-level primary inputs/outputs[4] between the RTL-level and transistor-level implementations [24]. To effectively detect the occurrence of a logic design bug ~~in~~ the transistor-level hardware implementation, we need only to trace back the combinational logic that feeds the first-level monitor signals to a set of signals that are directly connected to either 1) state-holding elements or 2) primary inputs of the module. We call this set of signals *source-level monitor signals*. This process is illustrated in ~~at~~ Figure 8(b). Monitoring the source-level monitor signal set of a design bug allows the detection of the bug. Note that it is very easy to construct a truth table using the source-level monitor signals instead of the first-level monitor signals to understand which combination of the values assigned to source-level signals would exercise the design bug.

To determine the number of signals required to be monitored to detect the occurrence of logic design bugs, we measured the first-level and source-level signals of the 162 logic design bugs located in the LSU and the TLU units. ~~The chart of~~ Figure 9 shows the cumulative distribution of the logic design bugs versus the first-level and source-level monitor signal set sizes in the LSU and the TLU units. We observe that 97% of the logic bugs located in the LSU and 92% of those located in the TLU have a source-level monitor signal set size that is smaller than 64 signals. This means that for detecting any *single* bug that is within the aforementioned percentage, at most 64 signals need to be monitored. Table 1 focuses on the number of source-level signals to be monitored to detect single bugs

explain Table 1 briefly first

An interesting observation ~~from Table~~ 1 is that the average set size of source-level monitor signals per logic bug is about double the size of the first-level monitor signal sets. Notice that the size of the first-level monitor signal set determines the minimum number of RTL-level signals required to be monitored to precisely detect the occurrence of a certain bug, given that those signals exist in the actual hardware implementation, and can be probed. On the other hand, the size of the source-level monitor signal set determines the number of transistor-level signals required to be tapped to detect a bug, given that design flip-flops and module inputs can be probed. Furthermore, the average number of source-level monitor signals per logic design bug is 17 and 24 for the LSU and the TLU units respectively (The minimum and maximum set sizes are presented as well). Hence, the detection of an average design bug requires 17 to 24 transistor-level signals to be monitored. 7 in TLU

The total amount of tapped signals can be small if there is a high degree of source signal sharing between multiple design bugs. To quantify this, we studied the amount of sharing between the 162 logic bugs covered by our study. We found that the sharing between the source-level monitor signal sets is about 65% on average (68% in LSU and 64%). This means that 65% of the signals that belong to the source-level monitor signal set of a logic design bug also belong to the source-level monitor signal set of at least one other logic design bug. Furthermore, each logic design bug has on average 6-9 signals in its source-level monitor signal set that are unique, *i.e.*, they do not belong to the source-level monitor signal set of any other logic design bug. This result implies that the discovery of a new design bug requires the monitoring of an additional 6-9 signals, on average, that

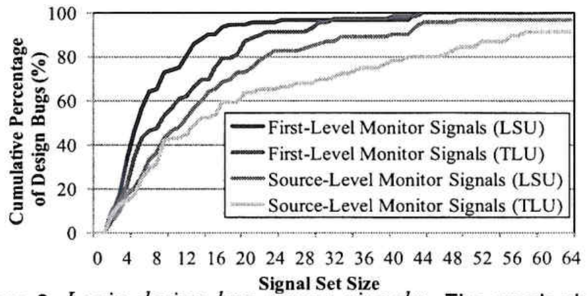


Figure 9. *Logic design bug source signals:* **The graph shows the cumulative distribution of logic bugs versus the first-level and source-level monitor signal set sizes for the LSU and the TLU units.** better and shorten

Metrics	LSU	TLU
Min./Average/Max. number of first-level monitor signals per logic design bug	2/8/43	2/12/44
Min./Average/Max. number of source-level monitor signals per logic design bug	2/17/97	2/24/89
Source-level monitor signal sharing among different design bugs	68%	64%
Average number of unique source-level monitor signals per logic design bug	6	9
Unique source-level monitor signals (for all logic design bugs)	516	602

Table 1. *Logic design bug statistics:* ~~**The table shows statistics regarding the first-level and source-level monitor signals that need to be monitored to detect the logic design bugs.**~~

have not been previously monitored for any other bug, thus increasing the total number of tapped signals.

In order to detect all the 162 studied logic design bugs, 516 and 602 unique source-level monitor signals need to be monitored in the LSU and the TLU modules respectively. Note that these numbers are much higher than previous work estimates that used high-level errata documentation to analyze design bugs. Specifically, the study in [20] reports that on average, for the ten processors studied, only 210 signals need to be monitored to detect all design bugs in all modules of a processor, with the maximum requirement out of the ten microprocessors being 260 signals. The study in [17] reports that monitoring only 41 signals is adequate to detect the occurrence of 43 out of the 63 known design bugs in the AMD Athlon 64 and AMD Opteron microprocessors. In contrast, our study shows that 1118 signals need to be monitored to detect 162 bugs in two modules of the SPARC core. We believe this discrepancy stems from the attempt in previous studies to infer low-level hardware implementation information from the high-level, abstract information provided in the microprocessor errata documents. By studying the documented design bugs at the lower RTL level, we found that the signal monitoring requirements of online design bug detection ~~are~~ significantly higher than the estimates of these previous studies. As a result, the problem of detecting design bugs is more difficult and the solution is likely more hardware intensive than estimated by previous work. ok ✓

3.1. Insights from RTL-Level Design Bug Analysis

In summary, our RTL-level design bug analysis provides the following conclusions and insights:

1. The design bugs documented in the Verilog source files of the OpenSPARC T1 chip-multiprocessor can be classified into three major classes based on their characteristics: logic, algorithmic, and timing design bugs (Section 2.3).
2. Logic design bugs outnumber the documented design bugs of the other two design bug classes. Furthermore, they ~~are expected to~~ might

remove space

[4]In this work we consider a module to be a Verilog design module in the RTL code.

dominate the distribution of design bugs that escape the verification phase and slip into the final product (Section 2.4).

3. Because they only affect combinational logic, the occurrence of logic design bugs is more readily detectable while the system is in operation. ~~Furthermore,~~ This can be done deterministically by monitoring a set of source-level signals.
4. The number of signals that need to be monitored to detect the occurrence of logic design bugs is significantly higher than estimates provided by previous work. ~~Furthermore,~~ The discovery of a new design bug requires the monitoring of additional 6-9 signals, on average, that have not been previously monitored for any other bug.

These conclusions and insights call for a mechanism capable of concurrently monitoring a large number of different signals scattered in the design and thus providing an effective and efficient substrate to perform online detection of the occurrence of logic design bugs. In the rest of the paper we describe our proposal for developing such a mechanism.

4. Distributed Online Bug Detection

In this section we present our distributed online design bug detection mechanism. ~~In~~ Section 4.1 ~~we~~ gives an overview of our mechanism and ~~in~~ Section 4.2 ~~we~~ describes how it can be implemented in hardware. Details regarding the system integration of our mechanism are provided in Section 4.3.

4.1. Overview of Online Design Bug Detection

Figure 10 illustrates the high-level architecture of our online design bug detection mechanism. The mechanism is characterized by two phases: 1) the initial setup of the mechanism, and 2) the cycle-by-cycle operation where design bugs are detected while the system is operating in the field.

4.1.1. Initial Setup Process The first step of the mechanism's setup process is the determination of the triggering conditions for each design bug in the system. The design bug triggering conditions are characterized by (1) ~~the set of~~ the bug's source-level monitor signals and (2) their values that would activate the bug. The design bug triggering conditions of each bug are determined by the system engineers after performing the bug analysis process presented in Section 3.

Bug Signatures: Once bug triggering conditions are determined, they are represented by a structure called a *bug signature* (step 1 in Fig. 10). Conceptually, the bug signature is a list of all the signals in the system. From that list, the bug's source-level monitor signals are marked with the value they need to take to trigger the bug, while non-source signals are marked with a `don't care` value (X) indicating that their values are irrelevant to the bug activation. The bug signature can be considered as a representation of the system state that would activate the design bug. Each design bug can have multiple bug signatures due to multiple possible combinations of triggering conditions.

System Bug Signature: The next step in setting up our design bug detection mechanism is the generation of the *system bug signature*. The collection of bug signatures of all design bugs are merged together to form the *system bug signature* (step 2 in Fig. 10). The system bug signature constitutes a representation of all the conditions that can trigger any individual design bug in the system. The process of merging multiple bug signatures into the system bug signature is detailed in Section 4.2.2.

Bug Detection Segments: The system bug signature is subsequently encoded into a binary representation, partitioned into segments, and loaded into the mechanism's *bug detection segments* (step 3 in Fig. 10). The bug detection segments are field programmable structures each associated with a part of the system state (*i.e.*, the system's flip-flops). Each bug detection segment is loaded with the part of the system bug signature corresponding to its part of the system state. The loading of the bug detection segments is done by firmware that have access to the segments' field programmable resources. During system operation, the bug detection segments compare the system state to the system bug signature and generate match/mismatch signals.

Segment Match Detection Tables: The source-level signals of a design bug can be located only in some of the bug detection segments. Therefore, each bug is associated with a *segment match detection entry* that indicates which lower-level segments need to match the system bug signature with the system state for the bug to be detected. In essence, the system bug signature summarizes all the triggering conditions from all bugs whereas each segment match detection entry demultiplexes them to enable the detection of individual bugs. The segment match detection entries are loaded into the field-programmable segment match detection tables by firmware (step 4 in Fig. 10).

4.1.2. Cycle-by-cycle Operation and Design Bug Detection

Flip-Flop Level Checking: Once the initial setup of the mechanism is done by the firmware, the remaining task of the mechanism is to check if the system steps into a bug triggering state while it is operating. To check this, each bug detection segment compares its portion of the system bug signature to the system state and generates a segment-wide match/mismatch signal (step 5 in Fig. 10). → foreach bug???

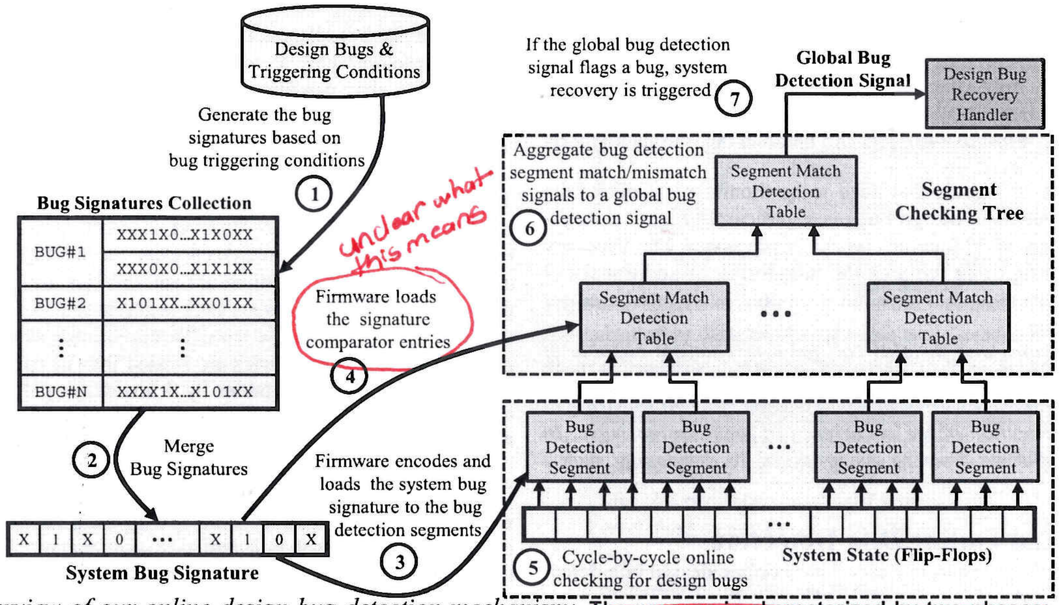
Segment Checking Tree: The detection of each individual bug usually requires only a subset of all the bug detection segments in the design to match their portion of system bug signature with the system state. For each bug, this information is encoded into a segment match detection entry. However, the subset of segments that are sensitive for the detection of an individual bug might be scattered in different areas of the chip. To aggregate the match/mismatch signals of all the segments on the chip, our mechanism employs a distributed *segment checking tree*. Each node in the segment checking tree has a *segment match detection table* that is populated with the segment match detection entries of each bug that has sensitive segments connected to the tree node. These entries are loaded during the initial setup phase by firmware (step 4 in Fig. 10). During system operation, if the match/mismatch signals of the underlying segments match with one of the node's segment match detection entries this indicates that the local triggering conditions of a design bug within that node are met. In a similar fashion, each level of nodes in the tree generates a match/mismatch signal and feeds the upper level of nodes (step 6 in Fig. 10). If a match signal propagates all the way from the bug detection ~~segments~~ segment level to the top level of the tree, this indicates that the triggering conditions of a specific design bug are met for the whole chip and a *global bug detection signal* is set. This process is illustrated in detail with an example in Section 4.2.3. The bug is subsequently flagged to the *bug recovery handler* (step 7 in Fig 10).

do we need to define "sensitive"?

Design Bug Recovery Handler: If a bug is flagged by the global bug detection signal, the design bug recovery handler recovers the system into the last validated system state. Execution is then restarted and guided by design bug avoidance techniques so that the design bug is averted. Since our focus is on detection, we leave the design of the handler to future work.

4.2. Hardware Implementation

4.2.1. Bug Detection Flip-Flops In our mechanism, the system bug signature and its comparison with system state is distributed to the flip-flop level. This is achieved by augmenting the system flip-flops with extra logic for storing the system bug signature and comparing it to the system state. Figure 11(a) shows a system flip-flop augmented with these extensions. The non-shaded logic comprises a scan flip-flop, the common type of flip-flop used in most modern processors to enable scan-in and scan-out functionality to facilitate manufacturing testing using Automatic Test Pattern Generation [14, 28]. The system ~~flip-flop~~ portion is used for holding the system state, while the scan portion is used to scan-in test patterns and scan-out test responses. In current move to previous page



Design Bugs & Triggering Conditions
Generate the bug signatures based on bug triggering conditions
1
Bug Signatures Collection
BUG#1 XXX1X0...X1X0XX
XXX0X0...X1X1XX
BUG#2 X101XX...XX01XX
BUG#N XXXX1X...X101XX
2
Merge Bug Signatures
X 1 X 0 ... X 1 0 X
System Bug Signature
Firmware loads the signature comparator entries
4
unclear what this means
Firmware encodes and loads the system bug signature to the bug detection segments
3
If the global bug detection signal flags a bug, system recovery is triggered
7
Global Bug Detection Signal
Design Bug Recovery Handler
Aggregate bug detection segment match/mismatch signals to a global bug detection signal
6
Segment Match Detection Table
Segment Checking Tree
Segment Match Detection Table
Segment Match Detection Table
Bug Detection Segment
Bug Detection Segment
Bug Detection Segment
Bug Detection Segment
5
Cycle-by-cycle online checking for design bugs
System State (Flip-Flops)

Figure 10. *Overview of our online design bug detection mechanism:* ~~The process is characterized by two phases:~~ 1) initial setup (steps 1-4), and 2) online bug detetcion (steps 5-7). single line ct any?

designs, the scan portion is utilized only during the manufacturing testing phase and stays idle during processor operation. During processor operation, our mechanism uses the scan portion in combination with an extra *bug detection portion* to store the system bug signature. The scan portion is used to indicate whether the specific flip-flop belongs to a bug's source-level monitor signal set. If the scan portion is set to 1 the flip-flop is indicated as a bug source signal, otherwise the flip-flop's value is irrelevant to the activation of a design bug. In the former case, the value that will activate the design bug is stored in the bug detection portion. The box at the top of Figure 11(b) illustrates the three encoding rules used to binary map the system bug signature to the bug detection portion (shaded box) and the scan portion (white box).

any
(this is not for a specific bug, so "any" is better)

If
~~In the case that~~ the scan portion is set, the value of the system flip-flop is compared to the value of the bug detection portion to check if there is a match between the system state and the system bug signature. In our mechanism, flip-flops are grouped into *bug detection segments* to simplify checking; the comparison result is *ORed* with the comparison result of the previous flip-flop to generate a segment match/mismatch signal. The signal is propagated to the next flip-flop (0 indicates a segment match and 1 indicates a segment mismatch). A bug detection segment consists of multiple bug detection flip-flops connected together in a serial fashion (this is analogous to scan segments in scan chains).

Figure 11(b) demonstrates the system bug signature binary encoding process with an example 8-bit system bug signature. It also demonstrates how the bug detection segment signals are generated for two different scenarios. Once the system bug signature is encoded and loaded into the bug detection and scan portion, the checking is partitioned into two 4-bit bug detection segments. In the first scenario, the system state matches with the system bug signature and the segment bug detection signals are both set to `zero` indicating that the bug is activated. In the second scenario, the second bit of the system state does not match with that of the system bug signature and therefore the bug detection signal of the particular segment is set to `one` indicating that the bug is not activated.

4.2.2. Merging Individual Bug Signatures to System Bug Signature

in

In this section we describe the technique we employ to merge multiple bug signatures to generate the system bug signature. First, we merge all the bug signatures related to a single design bug

single

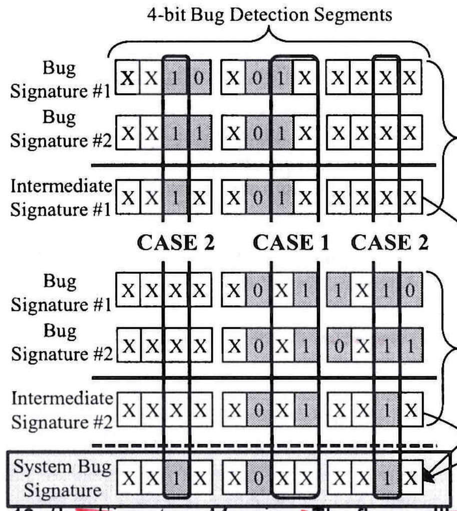


Figure 12. ~~*Bug Signature Merging.* The figures illustrates a technique for merging multiple individual bug signatures to generate a single system bug signature.~~ Merging bug signatures into system bug signature. single line

into an intermediate bug signature. To do ~~that~~ so, for each bit location we check the values of all bug signatures. If the bit takes the value of `zero` in some signatures and the value of `one` in others, then a `don't care` (X) value is assigned to the merged intermediate bug signature since for that signal either value can lead to a bug triggering combination. If the value of the bit is constant for all signatures then that value is assigned to the merged intermediate bug signature. This technique is illustrated in Figure 12 for two example design bugs.

For merging the intermediate bug signatures of multiple bugs to generate the system bug signature, we employ a slightly different technique. Again, if a bit location takes both values (`one` and `zero`) among different intermediate signatures then it is marked with a `don't care`. The difference from the previous technique is that now it is possible for a bit location to have a `zero` or a `one` value in the intermediate signature of one bug and a `don't care` value in the intermediate signature of another. This case is treated differently depending on the status of the remaining signals in the bug detection segment:

put this after Figure 10

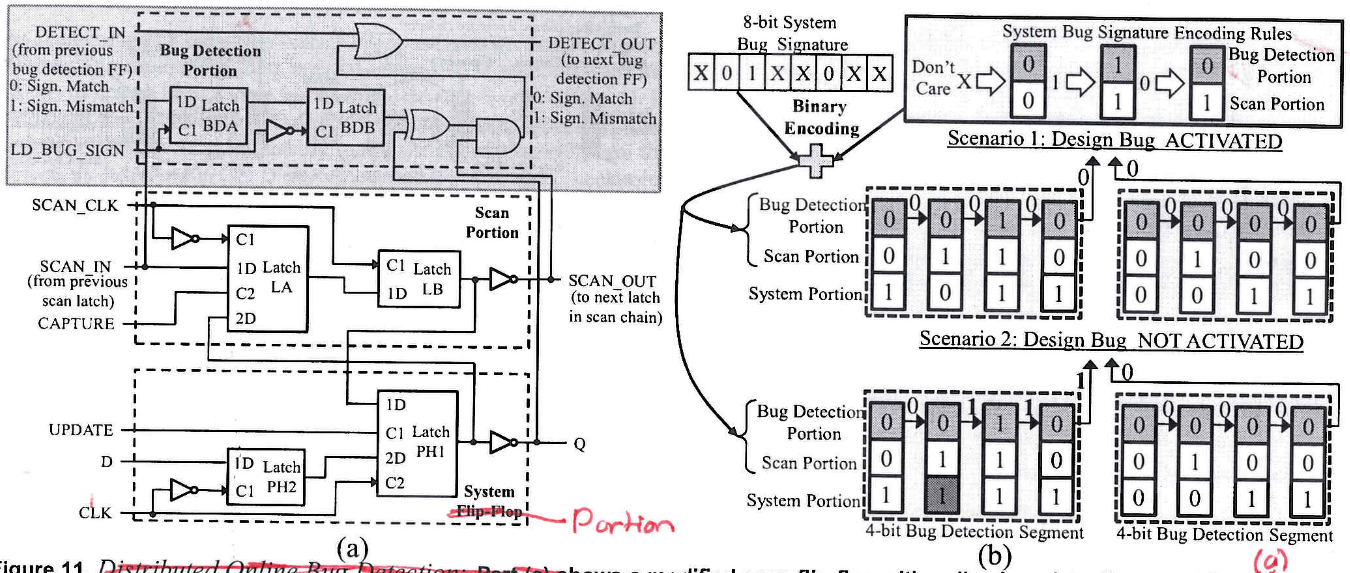

DETECT_IN (from previous bug detection FF) 0: Sign. Match 1: Sign. Mismatch
Bug Detection Portion
DETECT_OUT (to next bug detection FF) 0: Sign. Match 1: Sign. Mismatch
1D Latch C1 BDA
1D Latch C1 BDB
LD_BUG_SIGN
SCAN_CLK
SCAN_IN (from previous scan latch)
CAPTURE
C1 1D Latch LA C2 2D
C1 Latch 1D LB
Scan Portion
SCAN_OUT (to next latch in scan chain)
UPDATE
D
CLK
1D C1 Latch 2D PH1 C2
1D Latch C1 PH2
Q
System ~~Flip-Flop~~ Portion
(a)
8-bit System Bug Signature
X 0 1 X X 0 X X
Binary Encoding
System Bug Signature Encoding Rules
Don't Care X
Bug Detection Portion
Scan Portion
Scenario 1: Design Bug ACTIVATED
Bug Detection Portion
Scan Portion
System Portion
Scenario 2: Design Bug NOT ACTIVATED
4-bit Bug Detection Segment
4-bit Bug Detection Segment
(b)

Figure 11. ~~*Distributed Online Bug Detection.*~~ ~~Part (a) shows a~~ modified scan flip-flop with online bug detection capabilities (a) ~~, Part (b) shows an~~ example of an 8-bit bug signature encoding and checking (b). } single line

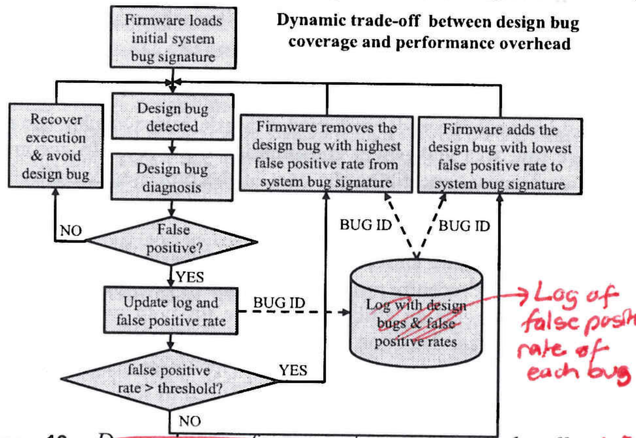

Dynamic trade-off between design bug coverage and performance overhead
Firmware loads initial system bug signature
Design bug detected
Design bug diagnosis
Recover execution & avoid design bug
Firmware removes the design bug with highest false positive rate from system bug signature
Firmware adds the design bug with lowest false positive rate to system bug signature
BUG ID
BUG ID
NO
False positive?
YES
Update log and false positive rate
BUG ID
Log with design bugs & false positive rates
false positive rate > threshold?
YES
NO
Log of false positive rate of each bug

Figure 13. ~~*Dynamic performance/coverage trade-off:*~~ A ~~flowchart of a scheme that~~ dynamically changes ing the set of covered design bugs to regulate the false positive rate and ~~system~~ performance overhead.

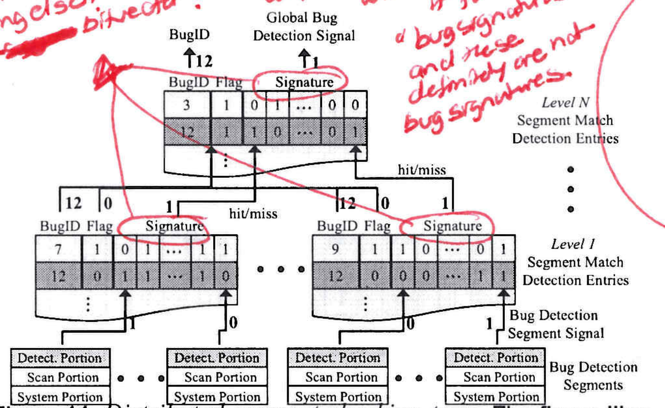
- *CASE 1:* Consider the two righmost bits of the middle bug detection segment of Figure 12. They both have the value of one in one of the itermidiate bug signatures and a `don't care` value in the other. Since the whole bug detection segment needs to match to trigger a bug and both bugs have other source signals in this bug detection segment (the second source signal with the value `zero`), the specific source signal is assigned a `don't care` value so that it will not prevent the detection of any of these two particular design bugs. spelling
- *CASE 2:* Now, consider the third bits of the leftmost and the rightmost bug detection segments. Again, in one of the intermediate signatures they have the value of `one` while in the other they have a `don't care` value. However, in this case no other source signal in the bug detection segments is shared between the two bugs. ~~This means that the segments are associated~~ only with one design bug. Therefore, the source signals can be set to `one` in the system bug signature because only a single bug requires the particular segment to match its portion of system bug signature with the system state to detect the bug activation.

False Positives - Notice that our mechanism uses `don't care` values to merge multiple bug triggering conditions and multiple bug signatures. This approach relaxes the bug triggering conditions and can result in *false positives*, that is, non-errant conditions which initiate an innocuous recovery sequence. However, since the technique only relaxes the triggering conditions, it cannot exhibit *false negatives*, that is, discovered design bugs with installed signatures that do not successfully initiate recovery. This is a very important property, since it guarantees that the system will not experience the effects of a specific design bug once the bug is covered by the mechanism.

However, the presence of false positives can adversely impact the performance of the system if too many false recovery alarms are issued. Since the false positive triggered rate highly depends on the dynamic system conditions and workload, we propose a dynamic scheme for trading off design bug coverage with system performance. Figure 13 gives a high-level overview of this scheme. At system start-up, firmware a loads into the mechanism the initial system bug signature that covers all design bugs. A design bug detection is followed by diagnosis process that determines if it is the design bug detection is correct or a false positive. If the detection is correct, the system execution is recovered and the design bug is averted using design bug avoidance techniques. If the detection is a false positive, then the false positive rate of the specific design bug is logged using the bug's ID tag and the system's false positive rate is calculated. The system's false positive rate is then compared with a predefined threshold. If the system's false positive rate is larger than the threshold, the design bug with the highest false positive rate is removed from the set of covered design bugs and firmware regenerates and loads into the mechanism the new system bug signature. On the other hand, if the system's false positive rate is smaller than the threshold, the design bug with the lowest false positive rate is added to the set of covered design bugs, and again firmware regenerates and loads into the mechanism the new system bug signature. can be cut

The predefined threshold can be adapted dynamically based on the requirements of the running applications. For example, a performance-critical application with low dependability requirements can set the threshold low, while a dependability-critical application can set it high. Furthermore, this scheme can be optimized to achieve the optimum trade-off between design bug coverage and performance overhead due to false positives. ~~however~~ Unfortunately, the exploration and evaluation of this technique requires extensive simulation of real workloads on a low-level (RTL-level) model of the design that would simulate our hardware signal monitoring and bug detection mechanism. To the best of our knowledge, such a simulation infrastructure is not available can be tightened

* IMPORTANT
Can we call this something else? For example, match-[illegible] bitvector? "Signature" is confusing because we already use it for "bug signatures" and these definitely are not bug signatures.



BugID
12
Global Bug Detection Signal
1
BugID Flag Signature
Level N Segment Match Detection Entries
hit/miss
Level 1 Segment Match Detection Entries
Bug Detection Segment Signal
Detect. Portion
Scan Portion
System Portion
Bug Detection Segments

Figure 14. *Distributed segment checking tree:* ~~The figure illustrates the distributed generation of a global bug detection signal through the segment checking tree.~~

example

This font size is too large.

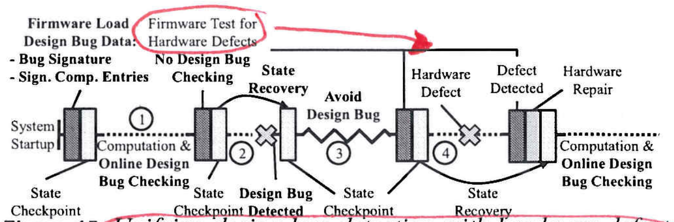


Figure 15. *Unifying design bug detection with hardware defect detection:* ~~The figure shows the synergistic execution timeline of the two mechanisms.~~

in our research community. ~~Currently, this is a work in progress and~~ Therefore, we leave this exploration and evaluation for future work.

4.2.3. Segment Checking Tree Implementation

In our mechanism, the bug detection segment signals are aggregated to generate one global bug detection signal through a hierarchical tree structure, the segment checking tree. The implementation of this structure is shown in Figure 14. Each leaf node of the structure is connected to a set of bug detection segments. For each bug that has source signals located in bug detection segments assigned to a leaf node, a *segment match detection entry* is allocated ~~for that bug~~ in that node. Each segment match detection entry indicates which subset of the node's bug detection segments ~~need to~~ match the system bug signature to trigger the given bug through the `Signature` field.

Confusing
This is not bug signature

Each entry also has a `Bug ID` and a `Flag` field. The `Bug ID` field indicates the bug associated with the specific entry, while the `Flag` field indicates whether the specific bug has source signals that are mapped on a different leaf node. For example, the design bug with the ID tag 12, has source signals both in the leftmost and in the rightmost leaf nodes of the tree. Therefore, it is allocated a segment match detection entry in each of those nodes with the `Flag` field set to `zero`. On the other hand, the design bugs with ID tags 7 and 9 have source signals limited only to one leaf node and this is indicated by having their `Flag` field set. A bug that has its `Flag` field set means that if the `Signature` field of that particular bug matches with the values of the underlying bug detection segments, then no further checking is required (since the bug's signals are limited only to that node) and the bug is flagged, along with its ID tag, through the tree to the top level *global bug detection signal*. Notice that if two bugs are flagged in the same cycle (*e.g.*, bugs 7 and 9), only one of them will be flagged to the top level and the decision will be arbitrary based on the implementation. However, due to the rare occurence of design bugs, we don't expect two design bugs to be triggered in the same cycle.

~~The example of~~ Figure 14 illustrates the detection of the bug with the ID tag 12. In the specific example, the values generated by the underlying bug detection segments match with `Signature` fields of bug 12 in both leaf nodes. Since the `Flag` field is set to `zero`, the bug is not flagged and the hit/miss signal from the leaf nodes are passed to the upper level. When the node hit/miss signals reach the top level nodes of the tree, ~~we notice that~~ the values match with the bug's `Signature` entry and therefore the global bug detection signal is set to `one`, triggering the design bug recovery process, and the bug ID tag is passed to the bug recovery handler.

4.3. System-Level Integration

The proposed bug detection mechanism described ~~in the previous subsection~~ require~~s~~ two additional critical functionalities to provide a complete online bug detection solution:

1. **In-the-Field Programmability:** The system bug signature and the data that need to be stored in segment match detection entries are dynamic and change as new design bugs are discovered or old bugs get fixed. This part of the design needs to be field-programmable and upgradable by special firmware developed and distributed by ~~the~~ microprocessor vendors ~~companies~~.
   reduce distance
2. **Recovery Support:** The detection of the occurrence of a design bug is only the first step in providing a solution to the problem. Further action is required to avert the design bug and avoid ~~any bug effects from~~ corrupting the execution. This is commonly achieved through recovery support where the system state recovers to the last validated/correct state [18, 21] and execution is guided from there in a way that the design bug is averted [19, 26, 17].

In addition, to be widely adopted, a bug detection mechanism needs to have low area and power consumption overhead. To accomplish this, we propose amortizing the cost of the proposed logic by using it for other purposes than solely bug detection (*e.g.*, hardware defect detection).

Unifying Online Defect Detection and Design Bug Detection - We found that ~~a~~ our recently proposed mechanism for detecting hardware defects can provide an efficient substrate for both requirements [4]. This online defect detection technique introduces the Access/Control Extension (ACE) framework to provide firmware access to the processor's scan state. This functionality is used to load ATPG test patterns and test the underlying hardware through specialized firmware. The ACE framework also uses a tree structure to access the scan state, similar to the tree structure used in our work to maintain the segment match detection entries and perform distributed bug checking (see Figure 11(b)). Since the ACE framework can read/write to any of the tree nodes and any scan flip-flop in the design, it can also be used to program the segment match detection entries in our distributed bug checking tree and load the bug signature at the flip-flop level. We believe that this framework can be easily adapted to provide our mechanism programmability through firmware with minor engineering changes. ~~We refer the reader to [4] for a detailed description of the ACE framework.~~

It would be good to condense this even more

Checkpointing & Recovery: The same framework also employs coarse-grained checkpoint and recovery techniques, based on previous work [18, 21], to provide recovery from hardware defects. We believe ~~that~~ these checkpoint and recovery techniques can offer an effective substrate to provide recovery support to our mechanism. By rolling back the system state to the last validated and correct system state, execution can be guided by design bug avoidance techniques in a way to avert the design bug. Several design bug avoidance techniques have already been proposed in the research literature [19, 26, 17]. Any further advancement toward this direction is left for future work. ~~Our major focus in this papers is on detecting design bugs flexibly.~~

System-Level Operation: Furthermore, the two mechanisms can work together synergistically and provide a collective solution for reliable and dependable computing. Figure 15 shows the synergistic execution timeline of the two mechanisms. At system startup special firmware uses the ACE framework to load the bug signature and the segment match detection entries needed for online design bug detection. During a checkpoint interval, execution is guarded from the effects of design bugs by our online bug detection design (phase 1). If no design bug is detected, at the end of the checkpoint interval special firmware uses the ACE framework to test the underlying hardware for defects as described in [4]. If the test succeeds a new ~~state~~ checkpoint is taken. If, during the checkpoint interval, a design bug is detected by our mechanism, the system state is rolled back to the last checkpoint (phase 2) and bug avoidance techniques are employed to avoid the design bug (phase 3). 4

If a hardware defect manifests during a checkpoint (phase 4), the defect is detected at the end of the checkpoint and after system repair the system state is rolled back to the last checkpoint for re-execution as described in [4]. Notice that the use of the tree resources and the scan state is mutually exclusive by the two mechanisms. Our online design bug detection mechanism utilizes these resources during a checkpoint interval, while the hardware defect detection mechanism utilizes the resources at the end of a checkpoint interval. Hence, the cost of the tree is amortized between ~~the two different schemes,~~ bug detection and defect detection.

I think we can remove this. Good overview, but occupies space.

5. Experimental Evaluation

~~In this section we experimentally evaluate the effectiveness of our technique. We also present its overhead in terms of silicon area and power consumption. We first start by describing our experimental evaluation methodology in Section 5.1. Next, in Section 5.2, we explore the trade-off between the silicon area required by our design bug detection mechanism and the provided design bug coverage. In Section 5.3 we estimate the power consumption of our technique. Finally, in Section 5.4 we present the area and power overhead of a unified online design bug and hardware defect detection mechanism.~~

5.1. Experimental Methodology

The case study design used for the experimental evaluation of our mechanism is the OpenSPARC T1 chip-multiprocessor, the open-source version of Sun's Niagara (UltraSPARC T1) processor [22]. We choose this design because the OpenSPARC T1 chip-multiprocessor targets commercial applications such as database and web servers where system correctness is of paramount importance. We believe ~~that~~ such systems constitute ideal candidates to employ our mechanism to provide the required correctness guarantees. ~~Furthermore, the~~ The OpenSPARC T1 is a full-system multiprocessor design implementing the 64-bit SPARC V9 architecture. It contains eight 6-stage pipelined in-order cores, each with 8KB L1 data-cache, 16KB L1 instruction-cache and full hardware support for four threads. The eight cores are connected through a crossbar to a unified 3MB L2 cache and a shared floating-point unit. The chip also includes four memory controllers and an input/output bridge [23].

RTL Implementation: To make an accurate assessment of our mechanism's requirements in silicon area and power consumption, we developed a detailed RTL-level model of our mechanism in Verilog. Specifically, in our prototype we implemented 1) the bug detection flip-flops that hold the bug signature and compare it with the system state, 2) the segment checking tree with a parameterized number of segment match detection entries per tree node, and 3) the ACE-based field programmable framework that loads through firmware the bug signature and the segment match detection entries. Our implementation covers all ~~the~~ modules in ~~the~~ OpenSPARC T1 ~~chip~~ except the memory cache, data and tag arrays (we don't expect logic design bugs to be located in ~~such~~ regular and meticulously optimized ~~data and tag~~ arrays).

Logic Synthesis and Tools: We used the Synopsys Design Compiler to perform logic synthesis on the RTL code of the OpenSPARC T1 ~~design~~ and our mechanism. ~~The~~ logic synthesis mapping is done

Methodology/Tools Used	Design Components
Synopsys Power Compiler	1) SPARC Cores, 2) Crossbar, 3) FPU, 4) Misc. Units (I/O Bridge, DRAM Controllers, Control & Test Unit) 5) ACE Framework, 6) Online Design Bug Detection Mechanism
CACTI 4.2	1) L1 Inst. & Data Caches, 2) L2 Cache
Taken from [15]	1) I/O Pads, 2) Wires & Repeaters

Table 2. *Power consumption estimation methodology.* ~~**The table lists all the major OpenSPARC T1 components and the methodology/tools used to estimate the associated power consumption.**~~

using the Artisan IBM 0.13um standard cell library. The resulting gate-level netlists of the OpenSPARC design and our mechanism provided a common substrate to accurately estimate the silicon area and power consumption overhead ~~of our mechanism~~ on the whole OpenSPARC design.

Power Consumption Estimation Methodology: To evaluate the power consumption overhead of our mechanism, we first estimated the power consumption of the baseline OpenSPARC T1 design that lacks the extra hardware required by our mechanism. We calibrated the estimated power consumption with actual power consumption numbers provided by Sun for each module of the chip [15]. After we validated our power estimates for the baseline OpenSPARC T1 design, we estimated the additional power required by our mechanism. Table 2 shows the major design components of the OpenSPARC T1 and the methodology/tools we used to estimate their power consumption. We estimated the power consumption of the majority of the OpenSPARC T1 modules using the Synopsys Power Compiler (part of the Synopsys Design Compiler package). The analysis uses the power consumption values of the Artisan IBM 130nm standard cell library, characterized at typical conditions of 1.2V (Vdd) and 25C average temperature. The average transistor switching activity factor was set to 0.5 transitions per cycle. correct? For modules dominated by SRAM structures, such as the on-chip caches, where logic synthesis and power analysis using the RTL code is inefficient[5], we used existing tools designed specifically to characterize SRAM modules. needed? To estimate the power consumption of the L1 and L2 caches, we used the CACTI 4.2 tool [25], a tool with integrated performance, area, and power models for memory cache structures.

This methodology is sufficient to estimate the power consumption of most of the chip's logic modules. However, there are parts of the design whose power consumption cannot be accurately estimated with these tools. These include 1) numerous buses, wires, and repeaters distributed all over the design, which are very hard to model accurately using the Design and Power Compilers, unless the design is fully placed and routed, 2) I/O pads of the chip. In order to estimate the power consumption of these two parts, we used values from the reported power envelope of the commercial Sun UltraSPARC T1 design [15].

False Positives & Execution Overhead: Our mechanism affects ~~the~~ system performance only in the case of a wrongly initiated system recovery sequence due to a false positive design bug detection. In Section 4.2.2, we provide a scheme that dynamically trade-offs design bug coverage with performance overhead reduction by regulating the system's false positive rate of design bug detections. Although, we believe that this scheme can be optimized to obtain a favorable trade-off ~~with a~~ with high design bug coverage and low performance overhead, ~~the~~ further exploration and evaluation of this technique requires extensive simulation of real workloads on a low-level (RTL-level) model of the

[5] In logic synthesis memory elements are synthesized into either latches or flip-flops. Therefore, SRAM macro-cells are implemented using memory compilers instead of using the conventional logic synthesis flow.

needed?

Chip Submodule	Data Signals	Control Signals
SPARC Core (x8)	15632 (79.06%)	4140 (20.94%)
CPU-Cache Crossbar	27283 (98.69%)	362 (1.31%)
Floating-Point Unit	4054 (87.75%)	566 (12.25%)
Control & Test Unit	2325 (55.29%)	1880 (44.71%)
Input/Output Bridge	10251 (95.14%)	524 (4.86%)
DRAM Controller (x4)	13449 (94.70%)	752 (5.30%)
Total	222765 (84.95%)	39460 (15.05%)

* I like this font :-) for captions

Table 3. ~~The table shows the~~ fraction of data and control signals ~~for all the modules~~ in the OpenSPARC T1 ~~chip.~~

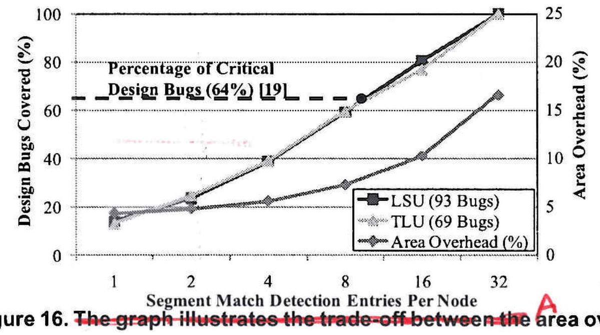


Figure 16. ~~The graph illustrates the trade-off between the~~ A area over~~head of our mechanism and the provided~~ versus design bug coverage.

design that would simulate our hardware signal monitoring and bug detection mechanism. To the best of our knowledge such a simulation infrastructure or methodology is not available in our research community. ~~Currently, this is a work in progress and~~ Hence, we leave this ~~exploration and~~ evaluation for future work.

5.2. Area Overhead and Design Bug Coverage

Control vs. Data Signals - After synthesizing the OpenSPARC T1 chip we found that there are about 262K flip-flops in the design. We also found that providing monitoring and bug detection capabilities for all these signals results in~~to~~ prohibitive area overhead (~69%). However, we observed that the majority of these flip-flops serve as buffers to data busses or data registers, and only a small fraction of them are control signals. Furthermore, after analyzing the source signals of the logic design bugs studied in Section 3, we found that all of the bug source signals were control signals, and no logic design bug had a source signal that was part of a data bus or a data register. After this observation, we partitioned the flip-flops of the OpenSPARC T1 design into *data* and *control* signals. Table 3 shows the fraction of data and control signals for all ~~the~~ modules in the OpenSPARC T1 ~~chip~~. For the whole chip, only 39K flip-flops drive control signals, accounting for 15% of all ~~the~~ flip-flops in the design.

* Did we say this in Section 3? I think this is important. It should perhaps be in Section 3.1.

Our prototype implementation taps all 39K control in signals be in the OpenSPARC T1 design. This means that each of these flip-flops is augmented with the extra bug detection logic shown ~~at~~ in Figure 11(a). The area overhead of this flip-flop augmentation is estimated to 3%. ~~The~~ In flip-flops are grouped into 8-bit bug detection segments and connected to a four-level segment checking tree structure (shown ~~at~~ in Figure 14). The area overhead of the tree structure depends on the number of segment match detection entries per tree node. The number of design bugs that can be covered by our mechanism also depends on the number of segment match detection entries per tree node, raising an engineering trade-off between area overhead and bug coverage.

Area Overhead vs. Coverage - The graph of Figure 16 illustrates this trade-off based on the 162 logic design bugs located in the SPARC core's LSU and TLU units ~~and~~ studied in Section 3. The graph depicts the percentage of design bugs covered (left Y-axis) and the area overhead (right Y-axis) versus the number of segment match detection entries per tree node. When the tree nodes are equipped with 32 entries,

This table should come after Figure 17

Mechanism	Flip-Flops Covered	Area Overhead	Power Overhead
Online Design Bug Detection (16 seg. comparator entries per tree node)	39K Flip-Flops	10.26%	3.5%
Online Hardware Defect Detection	262K Flip-Flops	5.8%	4%
Online Design Bug Detection + Online Hardware Defect Detection	39K Flip-Flops (bug detection) 262K Flip-Flops (defect detection)	15.15%	6.8%

Table 4. The table shows the cost of the combined defect detection and design bug detection mechanism.

our mechanism can cover all the 162 studied design bugs with an overall area overhead of 17%. Fortunately, not all design bugs are critical to functional correctness and need to be covered. Sarangi et al. [20]~~,~~ studied the errata documentation of ten modern microprocessors and found that, on average for all the studied processors, ~~only~~ 64% of the design bugs are critical to functional correctness. The remaining 36% of the design bugs were found to be non-critical to the correctness of the system and commonly located into modules such as performance counters, error reporting registers, or breakpoint support [20]. Furthermore, as stated in the errata documentation, the majority of these non-critical design bugs are not even planned to be fixed ~~in new~~ releases of the processors [13, 1, 5]. In ~~From the graph of~~ Figure 16, we can observe that 16 segment match detection entries per tree node provide a design bug coverage of 80% that is much higher than the typical fraction of critical design bugs ~~in the processors that need to be fixed~~. *This design configuration leads to a silicon area overhead of 10% of the whole OpenSPARC T1 design.*

no italics needed here

5.3. Power Consumption Overhead

Employing the methodology described in Section 5.1, we estimated be the power envelope of the baseline OpenSPARC T1 chip, without the additional hardware required by our mechanism, to 56.3W. Our estimate of the OpenSPARC T1 power is within 12% of the reported power consumption of the commercial Sun Niagara design [15]. ~~The pie chart of~~ Figure 17 shows the power consumption for our enhanced OpenSPARC T1 design including our online design bug detection mechanism. The power envelope of the enhanced design is 58.3W. From this, a total of about 3.4% (1.96W) is devoted to the extra hardware required by our mechanism. Specifically, 1.5% (0.87W) is consumed by the 39K bug detection augmented flip-flops, 1.3% (0.74W) by the segment checking tree (each node has 16 segment match detection entries), and 0.6% (0.35W) by the ACE-based field programmable framework. therefore *The overall power consumption overhead of our mechanism over the baseline ~~power envelope of 56.3W~~ is about 3.5%.*

This can be removed if we need space

no italics

5.4. Overhead of ~~Unified~~ Design Bug & Defect Detection

(or try "Unified Design Bug & Defect Detection Overhead")

As illustrated in Section 4.3, our online design bug detection mechanism can be easily coupled with an online ACE-based hardware defect detection mechanism ~~to provide a unified high dependability solution for both design bugs and hardware defects~~. ~~The~~ Table 4 presents the silicon area and power consumption overhead of the unified mechanism. The estimated silicon area overhead of the unified mechanism is 15.15% and its power consumption is 6.8%. *Based on these numbers, we believe that the coupling of the two mechanisms provides an attractive low* overhead *overhead solution for high dependability computing.*

no italics needed

* can you make these bold or in italics? to distinguish that they are our components?

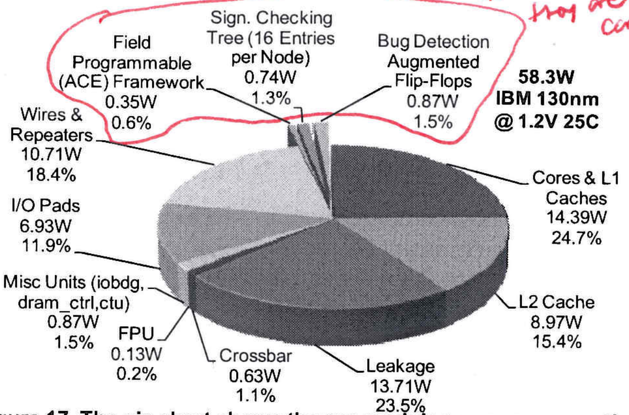


Figure 17. ~~The pie chart shows the per module power consumption of the OpenSPARC T1 chip enhanced with our online design bug detection mechanism.~~ Power consumption overhead

6. Related Work

Design Bug Analyses: Our online design bug detection mechanism is based on insights from this work and previous design bug analyses that characterize the known design bugs of existing processors. Section 2.1 provides a discussion on previous design bug analyses and how our RTL-level design bug analysis differs from those previous works~~, provides new insights, and pushes forward the understanding of processor design bugs~~.

Online Design Bug Detection: Recently, studies have brought to attention the increasing rate of discovered design bugs in modern processors [20, 17, 26]. These works suggest employing in-the-field design bug detection and recovery as an approach to mitigate the negative effects of design bugs. As with our mechanism, online design bug detection is facilitated by a signal monitoring substrate. Our work differs from these previous works in the following ways:

1. **Higher Flexibility:** In all these works, the signal monitoring substrate is limited to a small set of signals selected at design time when the design bugs are still unknown. This constitutes a major limitation of these previously proposed mechanisms. Specifically, if a design bug is discovered after product release and its bug triggering conditions involve signals not included in the set of signals selected to be monitored by the substrate, the occurrence of the design bug cannot be detected effectively by those mechanisms. This leads to greatly inflexible design bug detection mechanisms whose ~~that their~~ effectiveness depends on decisions made at design time based on assumptions regarding the set of signals that would be involved in ~~yet~~ unknown design bug triggering conditions. Our online design bug detection mechanism addresses this limitation with a novel field-programmable substrate capable of monitoring *all* ~~the~~ control signals in the design that can trigger a design bug. This capability waives the requirement of selecting the set of signals to be monitored at design time and allows this decision to be made after product release ~~and~~ when the design bugs and their triggering conditions are known. Italicize all
2. **Synergy with other Mechanisms:** We show that our mechanism can synergistically share its resources with other mechanisms to amortize their cost and provide a complete high-dependability solution. In contrast, the mechanisms proposed in previous works are specialized for only design bug detection.
3. **Extensive Evaluation:** We extensively evaluate the area cost and power consumption of our mechanism through an RTL-level prototype implementation based on a commercial chip-multiprocessor.

not sure if this is a major difference?

Online Hardware Defect Detection: Our mechanism relies on a combination of field programmable flip-flops and a field programmable tree structure to provide a flexible and efficient in-the-field signal monitoring and bug detection. ~~We found that a recently proposed approach of online hardware defect detection relies on the same combination of field programmable resources [4]. The specific technique employs an Access/Control Extension (ACE) framework that allows firmware to read and write from and to the design's flip-flops. This enables the dynamic loading of automatically generated test patterns into the system state to test the hardware for defects.~~ We believe that the two techniques can be combined seamlessly to provide a thorough solution that offers a high degree of both reliability and dependability to a system operating in the field. ~~Furthermore, the hardware used by our design bug detection mechanism can be used also for hardware defect detection, thereby amortizing their cost over multiple applications.~~

Short version: This support is similar to the support needed for flexible online defect detection techniques [4]. As we showed in section 4.3 defect detection and bug detection can be combined seamlessly and the hardware cost of the tree structure can be amortized.

7. Conclusions ~~& Future Directions~~

This paper provided a rigorous analysis of processor design bugs in the RTL of a commercial microprocessor, Sun's OpenSPARC T1 chip. Our low-level analysis of design bugs concluded that the signal monitoring requirements of online design bug detection are significantly higher than the estimates of previous studies. We believe that this discrepancy stems from the attempt in previous studies to infer low-level hardware implementation information from the high-level, abstract information provided in the microprocessor errata documents.

Based on the insights obtained from our rigorous deisgn bug analysis, this paper also proposed a novel distributed online bug detection mechanism. The proposed mechanism is able to flexibly monitor *all* control signals~~, which might participate in the triggering of a bug~~. This approach enables flexibility in bug detection because, unlike previous proposals, it does not rely on the successful selection of relevant signals at design time~~, rather, signals that affect possible bug trigger conditions maybe monitored as needed~~. Instead, any signal that can participate in the exercising of a bug can be monitored as needed. design ~~Italicize~~ Italicize

We evaluated the cost of our mechanism based on a detailed RTL ~~level~~ prototype implementation. The silicon area overhead incurred by our mechanism is 10% of the whole OpenSPARC T1 chip, whereas the power consumption overhead is only 3.5%. We showed that the hardware cost of the proposed technique can be amortized by combining it with a previously proposed online hardware defect detection mechanism that relies on similar field-programmable resources.

We believe that the processor design bug analyses presented in this paper provide an important step in the understanding of design bugs and the requirements to detect, tolerate, and avoid them. The proposed online design bug detection mechanism is the first fruit of that understanding. We hope the analyses and frameworks provided in this paper will lead to other, similar techniques that will enable the building of systems with high levels of dependability during in-the-field operation. ~~In our current and future work, we intend to follow two~~ directions. First, to evaluate the effects of RTL-level design bug detection mechanisms on the performance of real applications, we need to bridge the gap between high-level architectural simulation and low-level RTL simulation. We are exploring the development of a simulation infrastructure that would enable extensive simulation of real workloads on RTL-level models of commercial designs such as the OpenSPARC T1, in order to evaluate our mechanisms more extensively. Second, we are planning to rigorously investigate effective and efficient design bug avoidance and recovery techniques to be used in conjunction with the proposed distributed online bug detection mechanism.

REMOVE. I think these are GREAT research directions that we should continue! But, we can save a lot of space and do not need to tell the world yet :-)

References

[1] Advanced Micro Devices. Revision Guide for AMD Athlon 64 and AMD Opteron Processors, Pub. No. 25759 Rev. 3.75, Feb. 2008.

[2] A. Avžienis and Y. He. Microprocessor entomology: A taxonomy of design faults in COTS microprocessors. In *DCCA-99*, pages 3–24, 1999.

[3] F. Bacchini, R. Damiano, B. Bentley, K. Baty, K. Normoyle, M. Ishii, and E. Yogev. Verification: what works and what doesn't. In *DAC-41*, 2004.