Distributed Hardware Matcher Framework for SoC Survivability

Ilya Wagner
ilya.wagner@intel.com
Platform Validation Engineering
Intel Corporation

Shih-Lien Lu
shih − lien.l.lu@intel.com
Oregon Microarchitecture Lab
Intel Corporation

ABSTRACT

Modern systems on chip (SoCs) are rapidly becoming complex high-performance computational devices, featuring multiple general purpose processor cores and a variety of functional IP blocks, communicating with each other through on-die fabric. While modular SoC design provides power savings and simplifies the development process, it also leaves significant room for a special type of hardware bugs, *interaction errors*, to slip through pre- and post-silicon verification. Consequently, hard to fix silicon escapes may be discovered late in production schedule or even after a market release, potentially causing costly delays or recalls.

In this work we propose a unified error detection and recovery framework that incorporates programmable features into the on-die fabric of an SoC, so triggers of escaped interaction bugs can be detected at runtime. Furthermore, upon detection, our solution locks the interface of an IP for a programmed time period, thus altering interactions between accesses and bypassing the bug in a manner transparent to software. For classes of errors that cannot be circumvented by this in-hardware technique our framework is programmed to propagate the error detection to the software layer. Our experiments demonstrate that the proposed framework is capable of detecting a range of interaction errors with less than 0.01% performance penalty and 0.45% area overhead.

1. INTRODUCTION

In the last decade systems on chip (SoCs) have become an extremely popular type of digital design, and are widely employed today in millions of embedded and mobile devices. In addition to general purpose processor cores, SoCs feature multiple IP accelerator blocks, all integrated on a single silicon die. For example, Samsung S5PC100, a design comparable to that inside of iPhone 3GS, features an ARM processor core, a crypto engine, a set of hardware video, graphics and image codecs, and a variety of controllers for peripherals [11]. To communicate between the functional blocks SoCs typically employ on-die fabrics such a buses (AMBA AXI [2]) or point-to-point networks (Wishbone [10], OCP [9]). The modular architecture of SoCs brings several important benefits: First, it enables efficient integration of externally developed modules into the design. Second, it allows blocks to be powered down when not in use, thus lowering the energy consumption of the chip. Moreover, performance of IP accelerators typically exceeds that of general purpose processor cores for specific applications targeted by SoCs. However, the higher level of integration and diversification of components in an SoC makes validation and debugging of these devices extremely hard and increases the likelihood of errors escaping into production silicon. Since individual IP blocks are typically well validated in isolation, the majority of these errors are expected to arise due to unforeseen interactions between modules. Therefore, efficient survivability solutions are needed to detect and bypass such interaction bugs at runtime to lower the risk of a costly product recall or a release schedule slip.

Historically, companies employing SoCs in their products had significant control over both the hardware and the software stack and, therefore, could potentially implement software or firmware workarounds for late silicon bugs. For instance, the operating system of the device would be updated to check for certain outstanding accesses before scheduling DMAs or requesting data from I/O devices to ensure absence of offending interactions. Unfortunately, as the power of SoC-based devices increases, they begin to run progressively more complex software, including mainstream operating systems and third party applications. Consequently, SoC hardware vendors cannot rely on the traditional workarounds any longer and must look for techniques that enable patching of escaped bugs transparently, with respect to the operating system and application software.

1.1 Contributions

In this work we propose and evaluate a novel programmable framework for detection and correction of hardware errors that arise due to unforeseen interaction of accesses inside of an SoC. The framework is composed of distributed detectors, designed as a part of fabric interfaces of each SoC IP block. If an interaction error is discovered during post-silicon validation process or in a product deployed in the field, a patch describing the sequence of accesses that trigger the bug is generated and uploaded as firmware to the SoC. The firmware programs the detectors at system startup to monitor for the offending access sequence(s) and initiate recovery upon detection. For recovery we propose an innovative in-hardware scheme, which does not rely on costly state checkpointing and replay. Instead, the interaction of transactions in the offending sequence is altered in a predictable and deterministic manner, effectively bypassing the error. Conceptually, this can be thought of as programmable *time dilation* of transactions in the SoC fabric that eliminates buggy interaction between them.

We also demonstrate that our framework can be employed together with a software-based recovery solution, providing timely detection of error triggering events and invocation of appropriate handling routines. The messaging mechanism, that we use to warn the main processor core about a bug trigger in an individual IP block, is also applied to propa-

This material is based upon work supported by the National Science Foundation under grant No.0937060 to the Computing Research Association for the CIFellows Project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Computing Research Association.

gate information between detectors, enabling recognition of distributed trigger sequences. Furthermore, we discuss additional benefits that our proposed framework can provide in areas of post-silicon validation, debug, and runtime performance monitoring. Finally, we evaluate the performance and various overheads induced by this solution in a functional simulator. To the best of our knowledge this is the first work that investigates hardware survivability features designed specifically for SoCs.

The rest of the paper is organized as follows: In Section 2 we survey errors reported in public errata and overview several prior works on hardware survivability. In Section 3 we present the architecture of our framework and detail the detector design, time dilation error recovery and inter-detector messaging. Section 4 presents experimental evaluation of our approach and Section 5 concludes the paper.

2. BACKGROUND

To create an efficient solution for in-hardware SoC survivability we began by analyzing escaped errors reported in publicly available specification update documents. In addition to SoCs from Sun Microsystems and Texas Instruments [13, 14] we inspected three I/O controllers from AMD and Intel [1, 4, 7], and an integrated processor design [6]. In the latter we looked only at the errata located outside of the main IA32 core. The total 123 escapes that were listed in these sources were classified into the following categories:

- Not a bug - errata has no implication on the proper functionality of the device in the field. For example the escape 14 in [4] causes a wrong conflict code to be signaled on certain invalid operations, which has impact on correctness.
- Electrical error is an issue with analog signal properties, such as voltage levels (*e.g.* errata 8 in [7]) or timing (*e.g.* errata 98 in [1]).
- Functional bug category includes errors that hamper functionality of the circuit regardless of its operating mode. An example of that is uncore bug 4 in [6], which causes I/O controller to writeback invalid data for certain accesses.
- Mode errors, unlike functional bugs, occur only when the system is in a particular mode of operation, as was the case in errata 13 of [13], where corrupted data may be written to memory in a mode with credit overflow disabled.
- Interaction bug is a category of errors that involves interaction between accesses or different modules within a device. This includes, for instance bug 1.1.7 in [14], which manifests itself only when a pin polarity change command coincides with a receipt of a new request.

The breakdown of the categories is shown in Figure 1. As the figure demonstrates, the largest fraction (35%) of errors in the devices under survey was due to unforeseen interactions between modules of the design. Extrapolating this data to SoCs with dozens of IPs and complex communication fabric, we expect that interaction bugs will dominate the landscape of escaped errors in the future. Consequently, the industry and academia alike must begin to look into scalable solutions that combine efficient detection of offending interactions with low-overhead recovery techniques.

As we mentioned in the introduction of this work, software and firmware update is a viable patching solution for smaller SoCs, especially if the same entity controls the hardware and the software. Unfortunately, this assumption no longer holds for more complex designs of today: These products run under third party operating systems and contain a vast

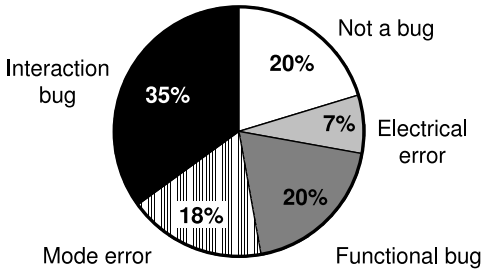

Figure 1: Survey of reported hardware escapes in SoCs, chipsets and embedded designs. We categorized errors reported in [1, 4, 6, 7, 13, 14] into five groups: Not a bug (does not affect in the field functionality), Electrical error, Functional bug, Mode error (manifested in certain modes only) and Interaction bug.

range of IPs bought from external vendors. Therefore, pure software techniques are becoming progressively less viable and SoC designers turn to in-hardware survivability.

In the mainstream processor domain survivability has been investigated in several research projects. DIVA [3] proposed using a small formally verified core in the "slipstream" of a high-performance pipeline. Thus, all execution results are checked and correctness is guaranteed, even with escapes in the main core. A similar solution was applied to the processor memory subsystem by Meixner *et al.* [8], who relied on in-hardware detectors and checkpointing for recovery.

Unlike the two solutions above, where detectors are fully specified in silicon, hardware patching techniques rely on flexible circuits that can be programmed in the field to monitor for specific error-related events and initiate recovery. For processor cores solutions were proposed by Sarangi *et al.* [12], as well as in our previous work [16], that observed the control state of the pipeline at runtime and changed the execution flow to bypass an error upon detection. Our later work [15] extended hardware patching to processor memory subsystem. However, all of the survivability research to date has been limited to mainstream processors and not SoCs, where the primary cause of errors is interaction of accesses. Furthermore, many of the prior recovery techniques rely on checkpointing and replay mechanisms, which are extremely hard to apply to SoCs. Finally, all of these schemes imply close integration of the detectors into the design, which may not be possible for IPs bought externally.

In contrast with the prior research, we concentrate specifically on *design error survivability for SoCs*, proposing a distributed hardware patching system that is non-invasive of IP blocks and has an efficient field-programmable mechanism to detect timed sequences of accesses. Moreover, we do not rely on costly checkpointing for recovery, instead attempting to distort interaction between accesses to an IP and make the offending sequence benign.

3. FRAMEWORK OVERVIEW

We overview our survivability framework based on the example of an SoC in Figure 2. The system there consists of a general purpose core, a graphics engine and a memory controller communicating over a high-performance bus, and several downstream blocks connected via point-to-point hierarchical network. A switch IP routes the traffic between the downstream modules and bridges both fabrics together.

The goal of our framework is to enable the hardware to monitor incoming and outgoing traffic for particular sequences of transactions, known to trigger escaped bugs. We

accomplish this with survivability detectors (matchers) located at the interfaces of IP blocks (shown in gray in the figure). Note that for the upstream shared bus only a single matcher may be adequate to observe all traffic. However, this design would not suit our time dilation recovery technique, described below, hence we employ multiple detector modules. In the downstream portion, the distributed nature of the point-to-point network also entails multiple matchers.

Matched sequences in our framework are not set in silicon, but are programmed in the field through an external interface or by software/firmware running on the GP core via IPs' control registers. The complexity and length of the sequence that a matcher is capable of detecting is dependent on the detector's physical area, and presents a design trade off. Matchers also must support wildcarded patterns to be able to efficiently detect broad classes of operations, *e.g.* accesses to a range of addresses. More complex and capable matchers will be able to identify offending sequences more precisely and have fewer false positive detections.

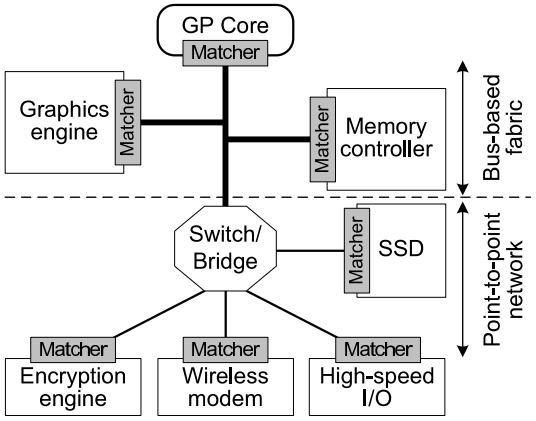

Figure 2: High-level architecture of the proposed framework. An example SoC consists of a high-performance bus connecting the general purpose core, memory and graphics, and a point-to-point network connecting other blocks through a switch/bridge. Proposed solution augments IP interfaces with programmable matchers (gray) that are capable of detecting timed transaction sequences and locking the corresponding IP's Rx port for error bypass.

For detection of distributed transaction sequences communication between matchers must be enabled. This can be done through in-band messages or by using a dedicated side-band channel. We use the former in our experiments in Section 4 to analyze the impact of inter-matcher traffic on the SoC performance. The latter design would trade those overheads for higher area penalty and design complexity. In addition to detection of distributed sequences, messaging allows us to combine in-hardware detection with software-based recovery: A matcher can now be programmed to send an interrupt message to the core, invoking an error-handling routine (see Section 3.2). Consequently, software patching becomes more efficient, since the GP core does need to monitor for errors and intervenes only when necessary.

Unfortunately, software-based recovery may entail multiple queries from the core to the buggy IP and, thus, may have noticeable performance overheads. More importantly, this scheme has some delay between the detection and the actual recovery, allowing the error to spread through the system. To alleviate these shortcomings we designed a hardware-only "time dilation" recovery technique. Recall, that we target errors that occur due to unforeseen interactions of accesses, thus, spreading out or delaying one of the operations in the offending sequence has the potential to eliminate the error. To this effect, the matcher has control of the receive port of the IP and locks it upon a match. We propose that port locks are controlled exclusively by timers or cycle counters and do not wait for an explicit unlock command. The rationale behind this is to prevent a possibility of circular wait between locked matchers, which leads to deadlock. While a time-based lock may have to seal the port longer than needed to bypass the error, it will not induce new uncertainties and error conditions in the system. We overview time dilation recovery in more detail in Section 3.3 below.

In Figure 3 we show the operation of our solution with two example escapes: In the first case the state of the memory controller is corrupted when a configuration access from the core (1) is followed, within a certain time, by a DMA from a graphics engine (2). To remedy the bug we program the controller's matcher to detect this timed access sequence and send a message (3) to the core to invoke a software handler. In the second example we assume that it is discovered that the encryption engine and the wireless modem, which reside physically close on the SoC die, interact through the power delivery network. To avoid data corruption due to voltage droop we must ensure that transactions in these two blocks are started some time apart. Patching this bug involves programming the engine's matcher to detect an incoming accesses (A) and to send a notification message (B) to the modem's detector. In turn, the matcher in the modem is programmed to wait for the message and delay subsequent transactions (C) with a time-based lock (D).

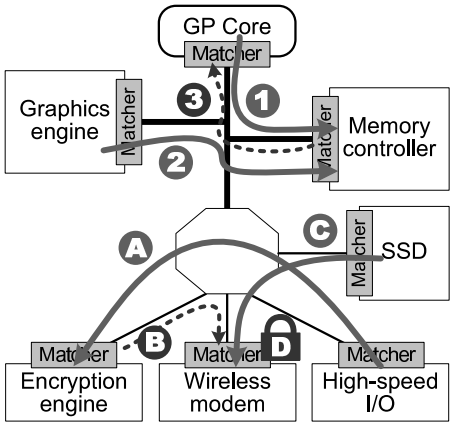

Figure 3: Examples of error detection and recovery with software intervention and time dilation. An escape in the memory controller is patched by programming its matcher to detect a triggering interaction between a core access (1) and graphics DMA (2), and to send a recovery request (3) to the core. In the second example a distributed interaction between the encryption engine and the modem is detected by first matching an access to the engine (A) and alerting the modem's matcher with message (B). Upon an interfering access (C) the Rx port of the modem is locked (D) and the interaction between (A) and (C) is distorted to bypass the error.

3.1 Detector Structure

A generalized architecture of a matcher is shown in Figure 4, where we mark all programmable fields with sold white rectangles. A matcher observes the values of transaction headers (1) and consists of multiple pattern entries (2), programmable by the system software/firmware or through an external interface, and control logic (3). Each entry has a decrementing counter (4) with a parallel load. Upon pattern activation the value of time-to-live, TTL (5), associated

with a pattern is loaded into the counter, which will decrement with every clock cycle until saturating at zero. The corresponding pattern is said to be active and participates in matching, while the counter is greater than zero, thus, with the help of TTL timing between accesses comprising an offending sequence can be specified and matched.

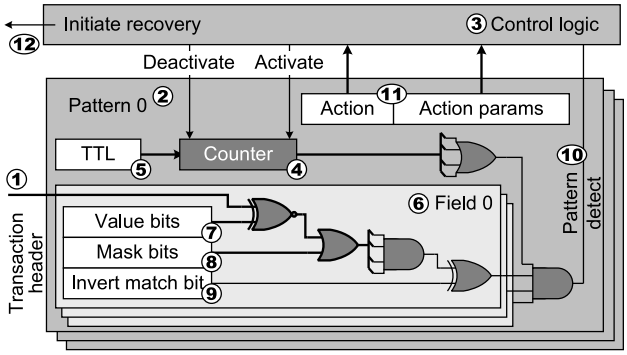

Figure 4: Structure of a detector. The detector compares transaction headers to multiple independent wildcarded patterns. Upon a match additional patterns can be activated or detection can be signaled to a hardware or software recovery engine.

Each pattern has multiple separate fields (6) that correspond to fields in transaction headers. For each N-bit field we allow specification of N fixed bits (7), N don't care bits (8) and a single invert match bit (9). Fixed bits must match those in the corresponding header field exactly, while don't cares match any bit value. Invert match bit, as its name implies, reverses the value of the field match signal, if set.

When all fields of an active pattern match the appropriate portions of a header, the detector's control logic is alerted. To allow for multiple simultaneous matches each entry's detect signal (10) is routed independently to the control logic block. It is then the job of the control logic block to take actions based on the values of the detect signals. With each pattern we associate programmable action and action parameters fields (11). These actions can include activation and deactivation of other patterns, initiation of time dilation recovery or signaling of sequence detection (12) to another detector or GP core. Detector control logic is also capable of handling multiple simultaneous matches - for mutually exclusive requests we use implicit priority of pattern IDs to determine the action to take. It is important to note that this matcher is not IP-specific, since it does not observe any internal signals and is restricted to the fabrics interface.

3.2 Messaging and Software-based Recovery

To enable our survivability framework to bypass errors at runtime we propose two recovery mechanisms. The first technique is structured around signaling of error detection in an IP up to the general purpose core and invoking a software error handler. This is implemented by injecting error-reporting messages into the fabric through the Tx port of the IP upon detection. The fields of the message indicate the error code to help the software select an appropriate handling routine. The handler can be supplied together with the matcher pattern to address a specific error, or it can be a generic routine that suspends the execution, queries the state of each component and resets or recovers the erroneous block. Since error-reporting messages are destined for the CPU core, their routing can be implicit, making this scheme lightweight and efficient. This approach is more flexible than the in-hardware solution described in the next section, since a software handler can perform a variety of recoveries depending on the particular error. The overhead of a recovery, on the other hand, is higher, since it involves propagating an alert message and multiple queries from the core. Consequently, this technique must be used judiciously and invoked only for rare bugs, for minimum impact on the performance.

3.3 Time Dilation Recovery

The second recovery scheme that we propose stems from the observation that a vast majority of interaction errors do not manifest themselves, if the timing of interfering accesses is changed even slightly. In the most pathological case it is fairly safe to assume that if transactions are made completely mutually exclusive, an IP would perform them properly, since the block was sufficiently validated. This perturbation can be done efficiently at the interface level of the IP by locking its Rx port and preventing transactions from entering the block and triggering the bug. Typically, sufficient buffering resources to implement this are already present in IPs, so only a few control signals are required. Note that time dilation is transparent to the software, since it is indistinguishable from contention and routing delays.

While time dilation could solve a large fraction of interaction errors, it must be implemented in a manner that does not introduce more severe issues, such as deadlock. We designed our framework with this particular goal in mind and made our locks exclusively time-based. The control logic of each detector is augmented with a counter that is set to the value of the pattern's action parameter (11 in Figure 4) if the pattern matches and its action is "Lock". After being set, the counter starts decrementing, releasing the lock upon reaching zero. No traffic flows through the port when it is locked and no new transactions are analyzed by the matcher. As a result, we do not rely on an explicit unlock event and eliminate the possibility of circular wait, thus protecting the system from deadlock. There are, of course, some escapes for which time locks are not as efficient as event-based ones. Yet, we believe that the danger of a more serious problem outweighs the efficiency advantages of the latter approach, so our time dilation locks are strictly counter-based.

3.4 Limitations and Extensions

There are several limitations to the approach proposed in this work, which must be kept in mind. First and foremost, this technique targets interaction errors specifically, and, therefore, may not be efficient, or even applicable, to issues such as functional bugs (see Section 2 for error classification). While our detectors are flexible enough to recognize accesses producing functionally invalid results, our time dilation technique is incapable of correcting them. Nevertheless, in these cases we can still resort to the software recovery scheme described in Section 3.2.

Secondly, it is possible that our matchers, restricted to the interface level of IPs, cannot precisely capture the nature of the interaction error without the insight into the block's internal state. Consequently, to bypass the bug, an overapproximated pattern must be used, causing the detector to initiate some unnecessary false positive recoveries and, thus, impacting processor performance. However, if only a small amount of time dilation is needed for error bypass, even high false positive rates will not hinder the system performance significantly, as we demonstrate in Section 4.3. Finally, delays introduced by time dilation may be undesirable in real-time systems, where timeliness of the response could

be more important than its correctness. In such applications the detectors can be selectively and dynamically disabled by the software via configuration register accesses.

Survivability is not the only application of our solution: It can also offer significant benefits to post-silicon validation and runtime performance monitoring. In the former field, observability of the on-die fabric is crucial for error isolation and reproduction, so matchers capable of detecting programmed sequence of transactions can be used as an on-die logic analyzers for the IP interfaces. Furthermore, our solution provides a methodology to bypass show-stopper post-silicon bugs, which otherwise may hinder the validation efforts. Finally, the system can configure unused patterns in matchers to monitor inbound transactions and pass this statistics to the processor core for performance optimization. Altogether, the proposed architecture offers significant value added in several phases of the design's life-cycle and, most importantly, provides a structured methodology for survivability-enhanced SoC design.

4. EXPERIMENTAL EVALUATION

In this section we overview our experimental evaluation setup and present several analyses of SoC performance overheads arising due to time dilation recovery.

4.1 Experimental Platform

For evaluation of our framework we developed a trace-driven SoC fabric simulator around an internal C++ PCIe library. While PCIe itself is not used in SoCs, functionally it represents a complex point-to-point hierarchical network with various types of accesses, which is likely to become the dominant communication medium of future high-end systems on chip. Consequently, lessons that we learned in this evaluation are not PCIe-specific, but can be generalized easily to any point-to-point fabric. Note that we did not need to model the exact behavior of the functional IPs, only their communication, since the simulator was capable of dynamically adjusting the trace timing depending on fabric resource usage, thus, being representative of the full system behavior.

Our simulator modeled an SoC with twelve IP blocks, including a root complex (GP core and memory controller), graphics and audio accelerators, crypto engine and several I/O controllers. The fabric was arranged as a hierarchical tree with high-performance components (graphics, crypto) residing closer to the root and on wider links. Fabric bandwidth tapered off closer to the third and fourth tier nodes, which modeled IPs with lower throughput requirements. We tested the system with twelve traces, each 10M cycles long, with bandwidth varying from 13.6 to 32 Gb/s. The traffic included large size DMAs, memory accesses, periodic message-signaled interrupts, and configuration operations.

4.2 Error Resilience Analysis

In our first study we analyzed the ability of our framework to protect an SoC from a variety of errors. We present the bugs we investigated in Table 1, listing the ID of each error, its name and the total number of times it was encountered throughout 120M-cycle simulation. In the last column we also show the duration of the time dilation lock that was needed for recovery. This value was dependent on the throughput and latency of IP blocks affected by the bug, as well as the worst case latencies in the fabric.

The errors were split into three groups. *Locally correctable* contained bugs, which could be bypassed with time dilation in a single IP block. *Globally correctable* represented distributed interaction bugs that could be bypassed by locking multiple IPs simultaneously. Finally, *software correctable* errors could not be avoided with the in-hardware technique, thus, their detection was signaled to the GP core. We investigated each error in isolation by programming the simulator to flag the interactions that would cause the bug to manifest.

Table 1: Error group, ID, name, total number of occurrences and maximum lock duration (in cycles) of analyzed design bugs.

	ID	Error name	No. of occur.	Lock cycles
Locally correctable	1	*crypto_rd+wr*	63,215	150
	2	*crypto_unaligned*	275	150
	3	*crypto_mutex*	145,407	150
	4	*rc_non_timer_msi*	103,517	250
	5	*rc_2msi_delay*	61,832	10
	6	*rc_rd+wr+msi_delay*	350	300
	7	*rc_crypto_rd+net_wr*	1,047	780
	8	*net_sata_rd+rc_wr*	818	1,700
	9	*rc_pwr_rd+cam_wr*	10	300
	10	*rc_2rd+msi*	24,353	1,600
	11	*sata_rd_crypto+wr_net_rd*	30	1,700
Globally corr.	12	*usb1_wr+net_wr_delay*	6,236	1,250
	13	*usb1_usb2_mutex_delay*	5,866	920
	14	*pwr_wr+graphics*	9	500
	15	*pwr_mutex*	1,473	1,300
S/W corr.	16	*sata_wr*	479	N/A
	17	*network_audio_wr*	8,328	N/A
	18	*graphics_rd_wr*	761	N/A

All bugs in Table 1 belong to the interaction error class, targeted by our framework, and some were taken from the errata we surveyed in Section 2. For others we modified reported escapes for more devastating effects; for instance, bug *3:crypto_mutex* is a scenario, where the crypto engine can handle only one transaction at a time. While this issue may not arise in a released component in reality, it helped us analyze pathological errors and worst case overheads.

This experiment showed that our proposed survivability framework was able to identify all instances of erroneous behavior and *was able to correct all bugs belonging to the first two groups using the time dilation recovery mechanism.* Furthermore, we were able to detect all occurrences of errors 16-18 and propagate an alert message to the root complex within 100 cycles. Overall, without our programmable detection and recovery framework these errors could have produced wrong results in individual IPs or could have corrupted the state of the entire SoC.

4.3 Detector False Positive Rate

In addition to initiating an in-hardware or a software-based recovery when an actual error trigger sequence occurs, our detectors may trigger an error bypass due to false positive matches. We computed the false positive rate from the total number of recoveries and the number of error instances shown in Table 1. As the gray bars in Figure 5 illustrate, false positive rate of our framework may be fairly high in some scenarios, however, it is important to keep the absolute number of error occurrences in mind. For instance, for bug *9:rc_pwr_rd+cam_wr*, which appeared only 10 times in 120M cycles of simulation, recovery was initiated 71 times, resulting in relatively high false positive rate (>85%), but without noticeable performance impacts. Similar situation occurred for bugs *11:sata_rd_crypto+wr_net_rd* and *14:pwr_wr+graphics*. To further analyze this, we plot-

ted the percentage of simulation cycles that had at least one time dilation lock in the system engaged (black bars in Figure 5). As the figure shows, in the majority of cases locks were active less than 10% of time, despite the false positive recoveries. Situations of high locking activity include high-frequency bugs 3 and 4, and bug 10, for which we have to engage a long latency lock (1,600 cycles).

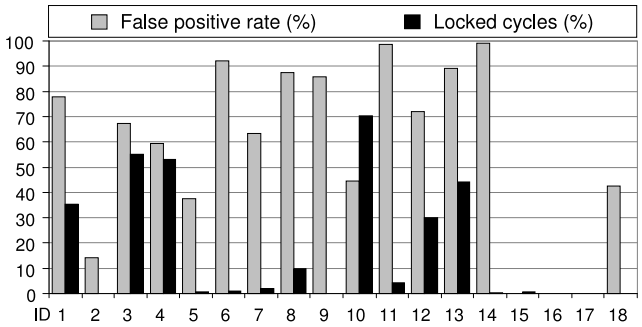

Figure 5: False positive detection rate and percentage of simulation cycles with recovery lock engaged. Gray bars show the false positive rate for each of the investigated errors, which was computed based on the number of error instances and the number of initiated recoveries. Black bars show the fraction of total cycles during which at least one recovery lock was engaged.

4.4 Time Dilation Recovery Overhead

In our third study we investigated performance overheads introduced by the time dilation solution. Since we lock IPs' interface ports upon detection of an offending sequence, completions of transactions are delayed, extending the execution. Comparing SoC performance with each of the errors in Table 1 to that of an ideal error-free system we discovered that *the performance penalty does not exceed 0.01%*. We attribute this negligible overhead to the bursty nature of the traffic and the fact that locking a single IP does not typically affect the rest of the system.

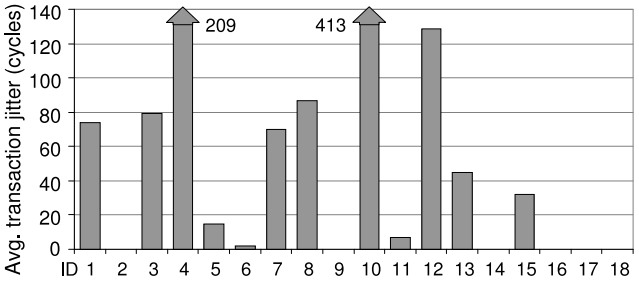

Figure 6: Transaction jitter due to lock-based recovery and injection of inter-detector messages. Jitter is computed as a difference, in clock cycles, between access completion times in an ideal error-free SoC model and in each of the buggy configurations.

In addition to the total overhead we measured transaction jitter, *i.e.* the difference of completion times of individual transactions between an error-free model and in the buggy SoCs. We graph the average jitter (in cycles) in Figure 6 for each of the erroneous designs. Note that jitter can be introduced by either time dilation locks or by inter-detector messages. As the figure demonstrates, for 11 of 18 errors the average jitter does not exceed 50 cycles. However, for some bugs, *e.g. 4:rc_non_timer_msi* and *10:rc_2rd+msi*, jitter is relatively high. We observed that these values are correlated with the error frequency and lock duration (discussed in Section 4.2) and the false positive rate (Section 4.3).

As another measure of the performance impact of our framework we calculated the amount of inter-detector traffic injected into the SoC fabric. We found it to be only 0.075% of the total traffic in the worst case, a negligible overhead.

4.5 Detector Area Overhead

Finally, we analyzed the area of our matching hardware, which is added to the SoC at design time. We looked at the sizes and locations of the matchers that would be needed to detect and correct *all* 18 errors in parallel. We assumed no sharing of entries between patterns belonging to different errors for worst case overhead calculation. The total area of the matching hardware that we computed with an internal estimation tool for Intel 45nm technology is 248,645 μm^2. Accounting for the control logic we estimate that the overhead of our solution is below 0.45% of the area of the 45nm Intel® Atom™ D410 processor [5]. For complex SoCs based on this core the overhead is expected to be even lower.

5. CONCLUSION

In this paper we presented a framework for detection and recovery from interaction errors, a major category of logic bugs in modern SoCs and embedded devices. Our solution is designed with minimum design invasiveness in mind and consists of multiple programmable detectors residing at the interfaces of individual IP blocks. These circuits can be configured to monitor for a variety of access sequences and invoke either the in-hardware time dilation recovery or signal error detection to the software level. As the results of our experiments demonstrate, our framework is capable of correcting a variety of escapes, while incurring less than 0.01% performance and less than 0.45% area overhead.

6. REFERENCES

[1] Advanced Micro Devices. *AMD Geode^{TM}CS5535 Companion Device Silicon Revision A3 Specification Update*, 2006.

[2] ARM Ltd. *AMBA Open Specifications*, 2010. www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[3] T. M. Austin. Diva: A dynamic approach to microprocessor verification. *Journal of Instruction-Level Parallelism*, 2000.

[4] Intel Corporation. *Intel®82574 Family Gigabit Ethernet Controller Specification Update*, 2010.

[5] Intel Corporation. *Intel®Atom^{TM}Processor D410 Specifications*, 2010. http://ark.intel.com/Product.aspx?id=43517.

[6] Intel Corporation. *Intel®EP80579 Integrated Processor Product Line Specification Update*, 2010.

[7] Intel Corporation. *Intel®I/O Controller Hub 10 (ICH10) Family Specification Update*, 2010.

[8] A. Meixner and D. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *Proc. DSN*, pages 73–82, 2006.

[9] OCP International Partnership. *Open Core Protocol Specification Release 3.0*, 2009.

[10] OpenCores Organization. *Wishbone B4 System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, 2010.

[11] Samsung Electronics. *Samsung S5PC100: ARM Cortex A8 based Mobile Application Processor*, 2009.

[12] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas. Patching processor design errors with programmable hardware. *IEEE Micro Special Issue*, Jan. 2007.

[13] Sun Microsystems. *Fire 2.0/2.1 Chip Errata*, 2007.

[14] Texas Instruments. *TMS320DM355 Digital Media System-on-Chip (DMSoC) Silicon Errata*, 2008.

[15] I. Wagner and V. Bertacco. Caspar: Hardware patching for multicore processors. In *Proc. DATE*, pages 658–663, 2009.

[16] I. Wagner, V. Bertacco, and T. Austin. Using field-repairable control logic to correct design errors in microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(2):380–393, Feb. 2008.