The Thesis Committee for Daniel Ruiz Santa Maria Certifies
that this is the approved version of the following thesis:

Identifying Post-Silicon Bugs and Their Root Causes Through a Hardware Introspection Engine

APPROVED BY

SUPERVISING COMMITTEE:

Mohit Tiwari, Supervisor

Andreas Gerstlauer

Identifying Post-Silicon Bugs and Their Root Causes Through a Hardware Introspection Engine

by

Daniel Ruiz Santa Maria

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2017

Acknowledgments

I would like to thank Jae Youl Kim for giving me the opportunity to do research during my graduate studies. I would also like to thank Samsung Electronics for the generosity of funding this research.

Identifying Post-Silicon Bugs and Their Root Causes Through a Hardware Introspection Engine

Daniel Ruiz Santa Maria, M.S.E.
The University of Texas at Austin, 2017

Supervisor: Mohit Tiwari

The goal of this project is to design, build, and evaluate new hardware mechanisms to debug post-silicon bugs in Systems-on-Chip (SoCs). Specically, we aim to accelerate the diagnosis of complex bugs such as deadlocks that are notoriously hard to identify using existing debugging mechanisms such as ARM CoreSight and hardware performance counters. We will design and evaluate programmable introspection mechanisms that will analyze streams of program and hardware-level trace data at test- and run-time, check correctness invariants, and generate event summaries that point to root causes of bugs.

This thesis describes an on-chip hardware introspection engine (HIE) that detects anomalous transactions and alerts the user of potential bugs that could lead to deadlock. The HIE is a device that attaches to a bus and snoops on request and response transactions and collects response latency metadata for the transactions it receives. From this metadata, HIE is able to evaluate the normal behavior of transactions and alert engineers when anomalous behavior is detected at run-time. The HIE also separates the metadata it collects for different address ranges, creating a local version of the memory map that allows easy integration into existing systems. Synthesis on a FPGA and simulation of the HIE show that minimal area overhead is required for implementation and 100% detection accuracy is achievable for deadlock scenarios. The concept of learning address ranges and collecting and analyzing metadata for these ranges can have many applications in different fields that leverage anomaly detection, i.e. security, debug, etc.

Table of Contents

Chapter 1

Introduction

Hardware-level bugs that manifest after synthesis are extremely challenging to detect and replicate. Determining the root cause of such bugs is even harder – for example, a typical post-synthesis bug can take a few hours to manifest and require over a month of effort to diagnose its root cause. This project aims to detect post-synthesis bugs quickly after they manifest in order to provide better diagnostic root-cause information.

Post-synthesis bugs are challenging to diagnose because they often have long dependency chains that were not explored in pre-fabrication testing and validation. For example, a data corruption in a cache can flow through the program, get loaded into the address register or the program counter, and lead to an illegal memory address or instruction value. As a result, the processor or state machine logic will output illegal bus-level transactions and cause some parts of the SoC to be deadlocked. By the time the actual error manifests, the program may have processed millions of instructions and the corrupted data may have been overwritten by a subsequent store instruction. Using ARM CoreSight, a designer can try to re-run the program and if the bug is reproducible, then try to analyze the extremely large volumes of data and use manual insight to identify the root cause. However, most of the challenging bugs are not reproducible in a deterministic manner and collecting hardware-level logs or running simulations until the bug manifests can take far too much memory and time.

We present a hardware introspection engine (HIE) – a hardware unit that is attached to the interconnect, snoops on all bus transactions among devices attached to the bus, and populates a trace buffer with transaction information that its logic identifies as anomalous. The HIE has multiple attributes that make it useful for debug and easy to deploy in a practical SoC. These attributes are listed below:

- HIE is an IP that connects to the bus and snoops on all transactions – as a result, using the HIE does not require all existing IP to be modified and limits debugging functionality to a small, bus standard compliant IP to be added to the SoC. This makes HIE practical to deploy.

- HIE uses statistics from transaction history to learn non-buggy behaviors of the SoC traffic – as a result, other IP in the system can be designed without interfacing with the HIE design team. Once the SoC components are synthesized and fabricated, post-synthesis test vectors can be used to generate the baseline inputs to train the HIE's anomaly detection mechanisms. When a bug occurs, the HIE will detect anomalies in bus traffic, record the source-destination-messages that are potentially relevant to the anomaly, and record these messages to a trace buffer that the debug engineer can read out.

- HIE adds little to no overhead to existing bus protocols or system flows. By learning the normal behavior of SoC traffic, HIE requires no programming from the host, and can begin learning SoC behavior after reset. Another benefit of learning SoC behavior instead of being programmed is that human error is avoided in the HIE.

In this thesis, an HIE device that detects potential deadlock transactions is emulated in Python with Gem5 traces and then realized on an FPGA inside a Freedom SoC. Gem5 is used to create trace data by running four SPEC2006 benchmarks and the Linux boot sequence itself to create normal transaction behavior for the HIE to learn on. The simulator is then injected with two deadlock scenarios (an invalid address transaction and a deadlocked peripheral device) to create a buggy trace that is later used in conjunction with the clean traces to evaluate the Python HIE model for accuracy and functionality.

The HIE design is then converted into RTL using Chisel HDL, integrated into a multicore RISCV SoC, and synthesized onto a Xilinx Ultrascale FPGA. The SoC is injected with a deadlocked peripheral device bug and Linux boot is ran to feed the HIE with clean transactions. The final design of the HIE shows 100% detection accuracy for all deadlock scenarios and adds an overhead of 17% more LUTs and 1% more Block RAMs to a SoC with two out-of-order cores that have 32KB instruction and data caches (104245 LUTS, 137 Block RAMs).

The rest of this thesis is laid out as follows. Section 2 will discuss the motivation behind this project and Section 3 will explain our proposed solution. The testing environment will be explained in Section 4 and the results of simulation are explained in Section 5. Section 6 elaborates on hardware requirements and results of the HIE and Section 7 concludes this thesis.

Chapter 2

Motivation: Post-Silicon Bugs and Existing Solutions

In this section, we categorize the most significant categories of post-silicon bugs and focus in on the bugs we specifically target. In practice, post-silicon hardware lockups can be classified as 'livelocks' or 'deadlocks'. Livelocks can be detected by software or can be easily fixed using existing debugger tools – software being available as a debugging option is crucial to solving livelocks. On the other hand, deadlocks caused by hardware design bugs do not allow engineers to use the core debug logic that is used commonly for software debugging. In this research, we focus on developing an IP to detect deadlocks – our target is finding the deadlock condition earlier and even finding main cause if possible. As a side effect, since anomaly detection is a general-purpose primitive, the system can be updated with new IP cores and the HIE will adapt its parameters to the new system.

2.1 Deadlock Scenarios

We classified deadlock scenarios into five cases based on the location of the cause.

1. Core bug. If a bug inside of the core causes deadlock, it is almost impossible to find the main cause from an SoC debug tool. Most CPU designs have their own debug feature that will help in system debug, but we are not going to handle core deadlocks in this research.

2. Load/Store unit bug. There are many transaction ordering conditions applied to the load and store units in each core – such as barriers, ordered transactions, hazard conditions, etc. Since bugs inside the host core cannot be found by SOC debug tools, these bugs are not targeted in this research.

3. Transaction missing due to incomplete address mapping in SOC. This is the most common design bug that causes system lockup. Each IP designer usually only focus on

their own dedicated address space, causing some address ranges to be left unhandled. These address ranges usually dont create problems if software works correctly or hardware doesnt make an unexpected transaction. When either of these happen, however, this bug becomes a difficult one to identify and pin-point the root cause for. Figure 2.1 shows an example of a request ignored by the bus arbiter. RAM is mapped to the address range 0x4000-0xFFFF and a peripheral device (PERI) is mapped to 0x0000-0x1FFF. If the CPU sends a request to address 0x3000, the arbiter will have no destination to send the transaction and will most likely drop the request. The CPU will not receive a response and will halt the progress of the executing program if a later instruction is dependent on the deadlocked transaction.

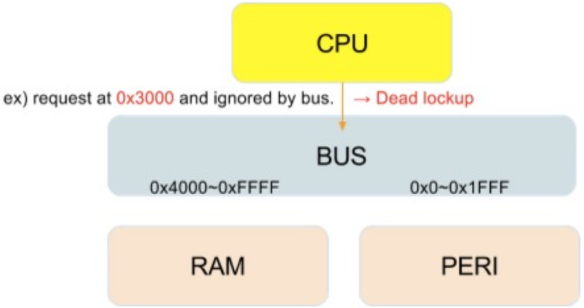


Figure 2.1: Deadlock caused by incomplete address mapping

4. Clock or power control logic bugs. The wrong sequence of clock or power control requests can cause system lockup as well. This could be caused by a missing signal or protocol handling error by software or hardware. Figure 2.2 shows a diagram of transactions sent out by the CPU during a clock gating sequence and a peripheral device does not respond to a request. There is a chance that the CPU sends clock gating requests out of order and the device stops responding because it entered a clock gated state sooner than expected. This phenomenon is almost the same as the incomplete address mapping case, but it is harder to find the root cause because it is difficult to reproduce cases that occur sporadically in executions with multiple masters.

5. Bus design error. Bus components themselves usually dont cause any problem, but bridges (especially an asynchronous bridge crossing), up or down sizers, and all other components which connect different types of devices are vulnerable to design bugs. For example, data is missed at an asynchronous crossing in specific timing corner cases, or an

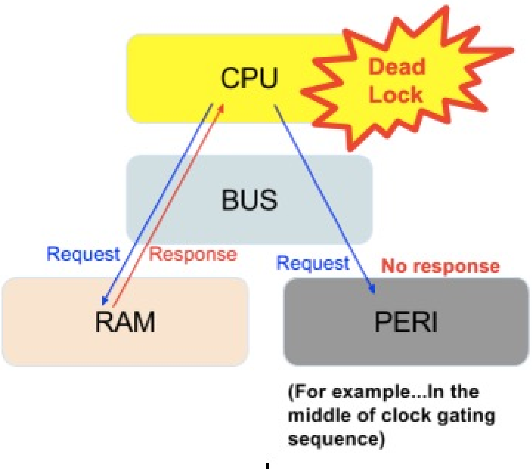


Figure 2.2: Deadlock caused by SoC

up or down sizer chops or merges data and the data size is no longer supported by other bus components.

This research covers the third and fourth scenarios where an unmapped address is accessed and a peripheral device does not respond to a request. The bus design error was not successfully integrated into the testing environment and was not covered in this research.

2.2 Existing Solutions

The inability to access internal signals inside an SoC limits the controllability and observability of logic to debug engineers. The most common design-for-debug (DfD) techniques are JTAG and embedded logic analyzers. JTAG is an IEEE standardized hardware interface that allows engineers to write and read flip-flop values in the internal logic of a circuit. This gives validation engineers the ability to initialize the state of a circuit, clock the circuit for one or more cycles with the initialized values, and then view the values of the flops in the circuit to analyze the behavior. JTAG is useful in sensitizing paths when a bug is reproducible, but when a bug occurs non-deterministically JTAG is not capable of storing the events that lead to the bug. Embedded logic analyzers (ELAs) provide a solution to the limitations of JTAG by storing a restricted group of signals before and after an event in embedded memory [1], but they are limited by the amount of data they can hold [9].

The combination of JTAG and ELAs increase the speed of bug diagnosis, but both of these techniques are still limited by the bandwidth they can drive off-chip.

Industry tools have evolved to allow engineers to have more observability when they debug SoCs. ARM Coresight, Xilinx Chipscope, and Intel Trace Hub are a few of the leading tools that allow the user to set a trigger event and capture various signals for debug. These tools add to JTAG and ELAs by giving the engineer the ability to reconfigure which signals are available for debug. Although these tools are very useful and have significantly helped in faster bug diagnosis, the increasing complexity of SoCs adds an overwhelming amount of data that engineers have to process manually.

Singerman et al. [10] and DeOrio et al. [5] propose using pre-silicon methodologies to better assist post-silicon debug. By using the observability of pre-silicon simulations, a database is created off-chip that stores transactions and events that occur in various scenarios in the design. Debug circuitry is inserted on-chip to capture the data that represents all the events that can take place. DeOrio et al. also claim that by having very specialized hardware, reproducing a bug is not required because most of the previous events are stored for the engineer to observe in the design. The results of these two designs are very promising and can greatly improve time-to-market by using pre-silicon observability to collect data on-chip that allows engineers to have a higher level of abstraction when debugging, but they can be limited by re-usability. Unless next generation products have only incremental changes, maintaining the on-chip circuitry and transaction databases will require development overhead that could actually the delay the product release. This is a trade-off that must be considered by SoC architects.

2.3 Opportunities for HIE

HIE provides a solution that learns the behavior of a SoC on-chip and provides transaction history to the debug engineer. HIE does not provide a self-contained solution to post-silicon debug, but when combined with existing DfD techniques, it can provide valuable insight to post-silicon engineers. The re-usability of HIE is only limited by the protocol interface of the bus it is attached to.

Chapter 3

Architecture of HIE

The HIE is an IP Core that attaches to the system interconnect using standard interfaces (AXI or Tilelink protocols). As a result, no other IP core in an SoC needs to be modified in order to be monitored by the HIE. Interestingly, ARM-based SoC tracing solutions today enable designers to get visibility into runtime executions. However, these solutions require the engineer to collect and process the trace data manually. Instead, HIE includes the logic required to process these traces at runtime and then creates warnings/alerts for the debuggers during post-silicon validation.

The HIE architecture consists of three modules called the Transaction buffer (XB), Range Entry Table (RET), and Trace Buffer (TB). Figure 3.1 shows a top level diagram of the HIE. These three components work together to learn the behavior of the transactions by matching responses with request, analyzing the response times of the new transactions with past behavior, and storing anomalous transactions for the engineer to debug. The following sections go into further detail about each module in the HIE.

3.1 Transaction Buffer

The Transaction Buffer (XB) is tasked with tracking the response time of each transaction. This block gets populated with requests and counts the number of cycles it takes to receive a response. After a response is matched with its request, the response is checked to see if it was sent with a response error. A transaction with a response error is sent to the Trace Buffer and not to the RET because a bad response could potentially skew the response times the RET is tracking, i.e. a response is received faster than usual because it was sent with an error. If the response does not have an error, the data stored in the XB is sent to the RET for response time analysis. Any anomalous behavior detected after a request is

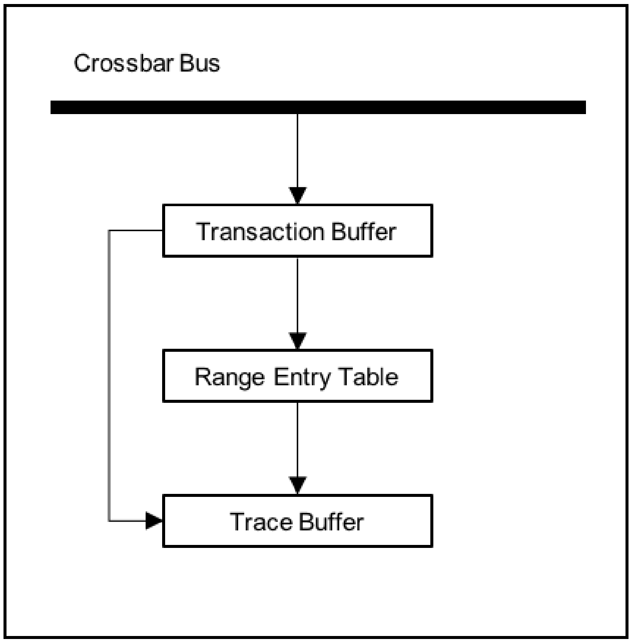


Figure 3.1: Top Level Diagram of the HIE

matched with a response is handled by the RET.

There is a chance that a request does not get a response back. Since the XB has limited space, requests that dont receive a response after a certain amount of time need to be removed. A parameterizable timeout value is used to invalidate entries in the XB once the counters reach this value. Once a request has timed out it will be added to the trace buffer as an anomalous transaction and the entry will be invalidated. Chapter 6 will discuss the timeout values chosen and the size of the XB. In the case that a request is matched at the same time it reaches the timeout value, the data will be forwarded to the RET instead of the TB. This is required because the TB will stop collecting transactions if a timeout is detected. Section 3.3 will explain the halting of trace data in more detail.

3.2 Range Entry Table

The RET is a special type of cache where each entry holds metadata (response time statistics) for the address range that it represents. This structure is similar to the Range Cache in [11] where special operations like insert, split and merge modify the data held in the table. The purpose of storing address ranges instead of point addresses is to try and learn the memory map of the SOC. The HIE needs to differentiate transactions targeting devices with slow response times from transactions that target devices with slow response times in order to to not flag every transaction sent to slow responding devices as anomalous. For example, if most transactions are routed to a DRAM controller and few transactions are routed to a USB device, the HIE should know that USB transactions take longer to respond than main memory transactions and are therefore behaving normally. This can be accomplished by storing response statistics for an address range as they are seen over time.

We chose to have the HIE learn the memory map instead of being programmed with the address space of different devices in order to provide an IP that has minimal overhead when incorporated into an existing system. This makes the HIE a simple plug and play device that makes no intrusion to existing flows in current and future SoCs. The HIE will also be free of programming error if it learns the memory map instead of being configured. If an engineer makes a mistake in the programming of the memory map, the HIE would learn on a bad memory map and would be useless for debug.

The RET tracks the response time averages of the different address ranges it detects. The HIE IP is placed at the outer boundary of the CPU, so the only commands tracked by the RET are read, write, and miscellaneous commands, where miscellaneous commands are anything other than the read and write opcodes. Therefore, each entry in the RET holds the average response time for three different categories of commands as well as a minimum variance for each category. Response time variance is necessary when a device can have variable response times due to events such as queueing latency or resource congestion. The RET must also have the most up-to-date response times to make accurate decisions with each transaction. Dynamic frequency scaling can change the response time of a device, so all of the commands are updated periodically by keeping a running sum of response times and response time variance. The Implementation section describes how the update is accomplished.

When the RET receives a transaction, it looks for the address of the transaction in its entries. There should be no overlap in the entries of the RET so the lookup will match with one entry (hit) or no entries (miss). The RET handles adding, updating, and removing entries with the special operations *insert*, *update*, *split*, *merge*, and *evict*. If the lookup results in a miss, then the *insert* operation adds the transaction data into the RET. If the result is a hit, then the RET will either *update* the entry that matched or it will *split* the entry into two entries with different statistics. When the RET becomes full, the *merge* and *evict* operations create space for future transactions. The following sections go into further detail for each operation.

3.2.1 Insert

The *insert* operation occurs when there is a RET miss. An entry is created that starts at the 4KB page of the transaction address and ends at the largest address possible in a 48-bit address space. For example, if the first transaction the HIE sees is for address `0x0000_0000_0100`, then the insert operation will create an entry with a start address of `0x0_0000_0000` (top 36 bits of the address) and an end address of `0xFFFF_FFFF_FFFF`. The read/write/misc averages and variances are initialized to a parameterizable value on entry creation, and the current transaction response time overwrites the initial value for its corresponding command. Initial values are needed because the RET would become fragmented very quickly if the average and variance fields are initialized to 0. Figure 3.2 shows an example of this fragmentation. When a read response for address `0x8000_0000` arrives, an entry is created ranging from `0x8000_0000` to `0xFFFF_FFFF_FFFF`. The initial values for write and invalidate are set to 0. When a write response matches with this range (`0x8200_1000` in Figure 3.2) its response time is checked with the entry's write average to determine if it is anomalous. Since the write average is set to zero, the HIE will determine the write transaction to be anomalous and will split the entry into two entries because it believes the significant difference in the write response times result in it being a different device. In reality, the write transaction should not split because it is for the same device but the initial value of the entry created the error in the HIE. The initial values chosen are explained in the Parameter Evaluation section.

Evictions of addresses can cause future transactions to miss in the RET. In this case the RET must find the largest address range possible for each new entry. A state machine iterates through the valid entries in the RET to find the next highest start address with respect to the current transaction. The entries must be sorted in order to know what ranges are present at the moment. The Implementation section will explain the logic inside this state machine. Once the next highest address is known, an entry is created starting at the transaction address and ends at the next highest start address - 1. By spanning as much of the address space as possible the amount of fragmentation in the RET is kept to a minimum.

3.2.2 Update

The update operation occurs when there is a hit in the RET and the transaction's response time falls within the variance of the matching entry's command. After a transaction is matched with an entry, the difference between the response time and entry's average is calculated and then squared to find the magnitude of the error. This value is then added to a field in the entry that holds the running sum of errors since the last periodic update. The response time is also added to a field that holds the running sum of response times since the last periodic update to calculate the current average response time. Finally, an update counter for the specific command the transaction was for is incremented. The Implementation section will describe the logic that calculates the squared error and the periodic update.

3.2.3 Split

If the address hits in the RET but is outside the variance of the matching entry, then a split operation is performed. A split can be the result of a transaction taking longer than usual due to queueing or bus congestion, or because the address is mapped to a device that has longer access time than that of the matching entry. In either case, the entry that resulted in a hit will be split into two entries with different average response times for that command. The new entry created from the split starts at the current transaction address and ends at the original end address of the entry accessed. The original entry has to end at the page right before the new entry to avoid any overlaps. For example, if the transaction address is `0x0_8500_0000` and hit inside the entry [`0x0_0008_0000`, `0x0_0008_FFFF`], the split would modify the existing entry to [`0x0_0008_0000`, `0x0_0008_4FFF`] and create a new entry with a range of [0x0_0008_5000, 0x0_0008_FFFF]. This new entry is initialized the same way an *insert* operation initializes an entry and is then added to the RET.

A transaction with a longer than normal response time is seen as a potential anomaly by the HIE. If an anomaly is flagged for every transaction that causes a split, then the HIE would create a lot of false alarms for the debug engineers. To minimize the number of false alarms, the HIE only flags transactions that took longer than AVG_ANOM_MULT times the average response time of the entry that matched. AVG_ANOM_MULT is a parameterizable constant and the evaluation section explains what value was chosen for the design.

3.2.4 Merge

Merge operations are performed when the RET has reached its max capacity. Transactions that took longer (or shorter) than usual to respond will cause entries to split in the RET, but because the HIE does not know the true memory map of the SoC, it is possible that the RET becomes full with entries that were created erroneously. These entries created through splits will eventually converge to the same average response time as the original entry, and therefore can be merged back into a single entry.

The first event that takes place in a *merge* is to sort all the entries in the RET. Once the order of the entries is determined, the RET decides whether to merge an entry based on three cases. The first case is if the average of one of the two entries falls within the bounds of the other entry. The second case is if an entry has had minimal access since it was created. This alleviates the scenario when one transaction took anomalously longer than usual to respond and a new entry was created for it, but subsequent transactions did not target this new entry again. Finally, if an address range has not been accessed in a long time it will be forced to merge. These three cases are listed below with more detail.

Case 1: The two entries being evaluated for a merge are tested to see if they lie within each other's bounds. The bounds are calculated the same way as in the update operation where the difference in the averages are tested to be within the variance of each entry. If the read, write, and misc. averages fall within either of the entries' variances, the entries are merged into one range spanning both of the entries. The entry with the higher address range is then invalidated in the RET.

Case 2: is necessary to reduce fragmentation in the RET. If neither of the entries fell within each other's bounds as described in Case 1, then evictions would be necessary to create space for new address ranges. This case is useful when a range splits due to one anomalous transaction and the newly created entry is not hit frequently. Using the same example above for the Split operation, the range [`0x8000_0000`, `0x8FFF_FFFF`] would split into [`0x8000_0000`, `0x84FF_FFFF`] and [`0x8500_0000`, `0x8FFF_FFFF`] if a transaction to `0x8500_0000` was seen as anomalous. If the range [`0x8500_0000`, `0x8FFF_FFFF`] is not accessed frequently then it will waste space and should be merged back into [`0x8000_0000`, `0x8FFF_FFFF`]. This case occurs when the LRU counter of an entry is greater than parameter LRU_THRESHOLD_MIN and if the entry has not gone through a periodic update.

Case 3: also reduces fragmentation. This case will force a merge to occur if an entry's LRU counter is above the parameter LRU_THRESHOLD_MAX. This case includes entries that have gone through a periodic update.

After an entry is merged, it is invalidated and the next highest address range is tested to be merged. If an entry cannot be merged then the next valid entry in the index table is tested with its next highest address for merging. This continues until all address ranges have been checked for merging.

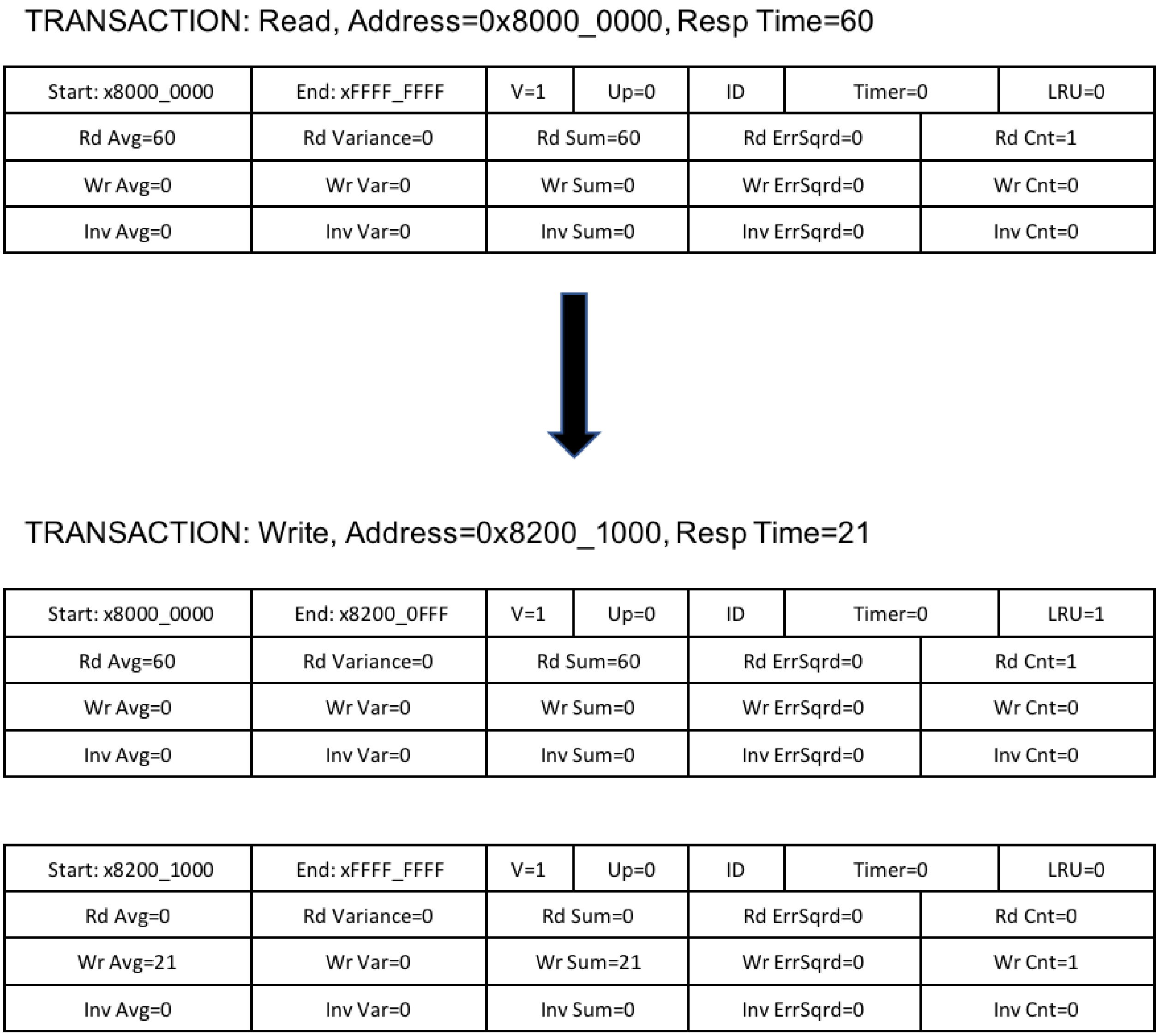
3.2.5 Evict

There are times when the RET is full and none of the entries can be merged. This can occur when all entries that have not been updated have not reached LRU_THRESHOLD_MIN and all entries that are updated are below LRU_THRESHOLD_MAX. This leads to an eviction of the least recently used entry. The RET becomes fragmented when entries are evicted and ends up leading to more evictions. Setting the LRU thresholds to lower values minimizes the evictions that occur over time.

3.3 Trace Buffer

The Trace Buffer holds transaction information that was deemed anomalous and raises an interrupt when an anomalous transaction is detected. There are three types of anomalies that the HIE will detect: a deadlock, a significantly long or short response time, and a response error. Deadlocks are detected when the XB counters timeout and data is sent to the TB. When this activity occurs, the Deadlock_error_interrupt signal is used to signal that a deadlock was detected. Long or short response times are detected by the RET and forwarded to the TB for future analysis. The Delay_error_interrupt signal notifies the user that this event occurred. These interrupts are raised after a transaction is loaded into the TB. The response error transactions that the Trace Buffer receives do not raise any interrupt flags. Since the request received a response, it will not cause deadlock and it is assumed that the original requester will know how to react to a transaction with the error field set.

The TB also stops keeping track of transactions after the deadlock_error_interrupt asserts. This prevents succeeding transactions from overwriting the anomalous transaction that caused the deadlock. The delay_error_interrupt does not halt the Trace Buffer because long response times for a few transactions might not lead to failure, and long response times occur more frequently than deadlocks.



TRANSACTION: Read, Address=0x8000_0000, Resp Time=60

Start: x8000_0000	End: xFFFF_FFFF	V=1	Up=0	ID	Timer=0	LRU=0
Rd Avg=60	Rd Variance=0	Rd Sum=60	Rd ErrSqrd=0	Rd Cnt=1		
Wr Avg=0	Wr Var=0	Wr Sum=0	Wr ErrSqrd=0	Wr Cnt=0		
Inv Avg=0	Inv Var=0	Inv Sum=0	Inv ErrSqrd=0	Inv Cnt=0		

TRANSACTION: Write, Address=0x8200_1000, Resp Time=21

Start: x8000_0000	End: x8200_0FFF	V=1	Up=0	ID	Timer=0	LRU=1
Rd Avg=60	Rd Variance=0	Rd Sum=60	Rd ErrSqrd=0	Rd Cnt=1		
Wr Avg=0	Wr Var=0	Wr Sum=0	Wr ErrSqrd=0	Wr Cnt=0		
Inv Avg=0	Inv Var=0	Inv Sum=0	Inv ErrSqrd=0	Inv Cnt=0		

Start: x8200_1000	End: xFFFF_FFFF	V=1	Up=0	ID	Timer=0	LRU=0
Rd Avg=0	Rd Variance=0	Rd Sum=0	Rd ErrSqrd=0	Rd Cnt=0		
Wr Avg=21	Wr Var=0	Wr Sum=21	Wr ErrSqrd=0	Wr Cnt=1		
Inv Avg=0	Inv Var=0	Inv Sum=0	Inv ErrSqrd=0	Inv Cnt=0		

Figure 3.2: Example of fragmentation due to initial values starting at zero

Chapter 4

Implementation of HIE

This chapter describes the microarchitecture and algorithms implemented inside HIE.

4.1 Transaction Buffer Design

The XB matches responses with requests to track response times. Each transaction is first decoded to find whether it is a request or response. Requests are added to the XB and a counter is triggered to count the number of cycles it takes to receive a response. Each request registered in the buffer is associated with a counter that increments at every clock cycle to get a measurement of the response time. These counters are cleared when the valid bit toggles from high to low and start incrementing when the valid bit is set. Responses can be received out of order, therefore a lookup process, similar to a cache lookup, is required to match responses with their request entries. For fast lookup times, a tag is created to match requests with their responses. Figure 4.1 shows the fields of a XB entry and the fields that are used to make the tag. The response opcode is used for matching to reduce the lookup time.

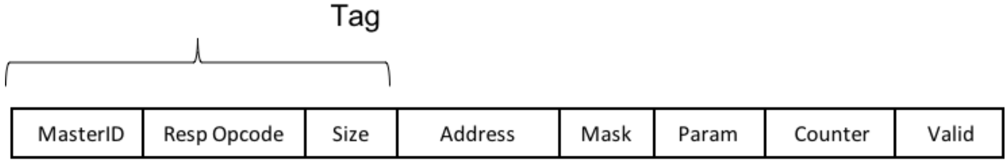


Figure 4.1: Transaction Buffer Entry

When a response arrives, the fields required to create the tag are concatenated, and a parallel search of this value is done with all entries in the buffer. When the matching request is found the valid bit is cleared, and the tag and other transaction fields are propagated to the RET. If an entry's counter reaches the TIMEOUT value, the entry is invalidated and forwarded to the Trace Buffer. Due to the fact that the XB has two separate data paths to the RET and TB, there is no contention if an entry times out and a response is matched in the same cycle. In the scenario that a response is matched with a request in the same cycle that a counter reaches the TIMEOUT value, the transaction data is sent to the RET to be determined if it is anomalous or not. Figure 4.2 shows a microarchitecture diagram of the Transaction Buffer.

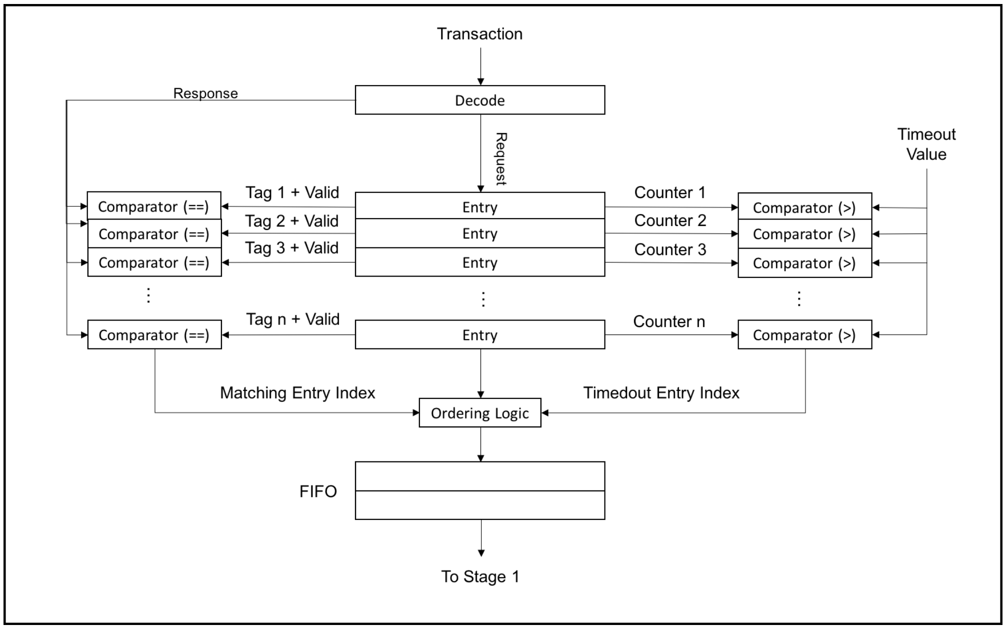
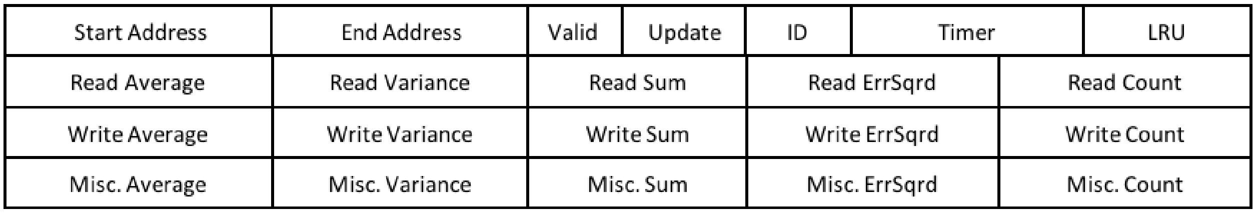


Figure 4.2: Transaction Buffer Microarchitecture Diagram

4.2 RET Entry Description

Figure 4.3 shows all the fields in an entry. The *start address* and *end address* fields are the values used to find a matching entry in the RET. A simple comparison of whether a transaction address lies within these fields leads to a hit in the RET. The Read/Write/Misc. *average* fields are used to track the average response times for that range. Response times can vary due to queueing latency and resource congestion, so the *variance* fields create bounds for the HIE to detect anomalies. Since we do not have unlimited memory to track the average response times of a range over the lifetime of an application, we need to periodically update the average response times to have the most recent data for each range. We use the *sum* fields to keep a running sum of response times and the *count* fields as the total number of transactions seen since the last update. The period for updating is parameterizable and the evaluation section explains what thresholds are best for detecting anomalies. The *errsqrd* fields have a similar function as the *sum* fields. This is the running sum of the squared error in response time of each transaction where the error is the difference between the range average and the current transaction response time. In order to avoid using a multiplier to compute the square, we use shift registers and shift the error value by log_2 of the next highest power of 2 number. For example, if the error is 30 cycles then the number thirty is shifted left by 5 bits, creating a value close to 30^2. With this technique, we sacrifice accuracy in the variance that is calculated but as the results show, this does not affect the end result. The *errsqrd* field is used to update the *variance* at the same time the *average* is updated. The *valid* bit is used to insert and evict data in the RET and the *update* field is used by the merge operation to minimize fragmentation (further explained in the next section). *ID* is used by the merge and insert operation to sort the entries in ascending order. The *LRU* field increments any time the RET is accessed to create new entry or to update an existing entry.



<table>
<tr><td>Start Address</td><td>End Address</td><td>Valid</td><td>Update</td><td>ID</td><td colspan="2">Timer</td><td>LRU</td></tr>
<tr><td>Read Average</td><td>Read Variance</td><td colspan="2">Read Sum</td><td colspan="2">Read ErrSqrd</td><td colspan="2">Read Count</td></tr>
<tr><td>Write Average</td><td>Write Variance</td><td colspan="2">Write Sum</td><td colspan="2">Write ErrSqrd</td><td colspan="2">Write Count</td></tr>
<tr><td>Misc. Average</td><td>Misc. Variance</td><td colspan="2">Misc. Sum</td><td colspan="2">Misc. ErrSqrd</td><td colspan="2">Misc. Count</td></tr>
</table>

Figure 4.3: RET entry fields

4.3 Sorting Logic

A state machine iterates through all the entries in the RET to order the transactions from lowest to highest. To avoid doing a $O(n^2)$ sort on the RET, N comparators and one index table are needed for N entries in the RET, where N is the max entries the RET can hold. The comparators are loaded with the end address of the entry being analyzed and the start address of all of the entries. The output of the comparators result in 1 if the end address is larger than the start address and 0 otherwise. The result of all the comparators are passed to an adder to sum how many entries represent a lower address range. After having this sum, the ID of the entry is stored in the index table, where the index is the sum of the adder minus 1. Figure 4.4 shows a brief example of sorting for merge in a 4-entry RET. The left input to the comparators is the End Address `0x8500_0FFF` and the other inputs are the start addresses of all the other entries. Three of the four comparators will be true ,so the output of the adder will be three. This sum is then subtracted by 1 and used as the index to the index table.

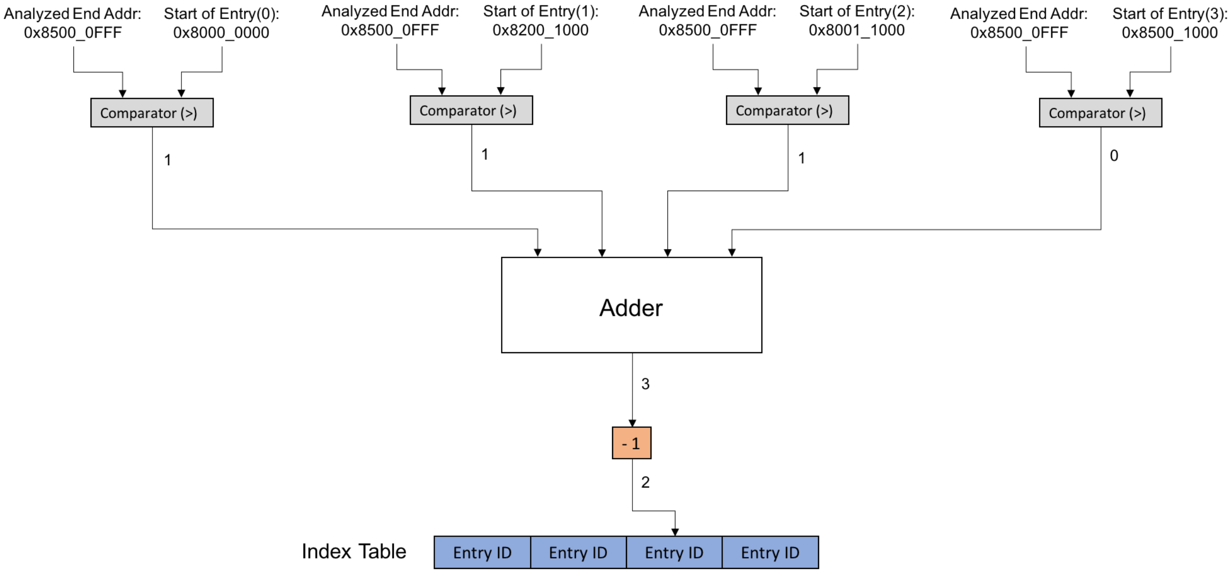

Analyzed End Addr: 0x8500_0FFF
Start of Entry(0): 0x8000_0000
Analyzed End Addr: 0x8500_0FFF
Start of Entry(1): 0x8200_1000
Analyzed End Addr: 0x8500_0FFF
Start of Entry(2): 0x8001_1000
Analyzed End Addr: 0x8500_0FFF
Start of Entry(3): 0x8500_1000
Comparator (>)
Comparator (>)
Comparator (>)
Comparator (>)
1
1
1
0
Adder
3
- 1
2
Index Table
Entry ID
Entry ID
Entry ID
Entry ID

Figure 4.4: Logic used to sort RET entries

Chapter 5

Experimental Setup

For proof of concept and quick prototyping, the HIE was first simulated using a python model and transaction data was generated with Gem5 [4]. Two bugs were injected into the simulation to create the deadlock scenarios and multiple SPEC2006 benchmarks were ran to create a wide variation of transaction data to test the HIE design. After this, the HIE RTL was developed and integrated into the Freedom U500 SoC and tested on a Xilinx VU190 Ultrascale FPGA. The following sections go into more detail on each of these environments.

5.1 Gem5 Test Bed

The Gem5 simulator was used because it can model a Realview ARM platform with up to 64 heterogeneous out-of-order cores and boot unmodified Linux using the AARCH64 (ARM 64-bit) ISA. The trace-based CPU model, event-driven memory system, and flexible addition of peripheral devices allows for the creation of transaction data from a complex system topology. Our simulation environment, shown in Figure 5.1, established 4 ARMv8 cores with 32K Bytes L1 Instruction Cache, and 64K Bytes L1 private data caches, and one 2M Bytes L2 shared cache. The CPU cores ran at 2Ghz and the crossbar bus clock ran at 1Ghz. All components attached to the crossbar run at the same speed as bus with some delay. The Linux kernel used was genericarmv8 3.16.0-rc6, and the benchmark programs ran were the SPEC2006 BZIP2, GCC, HMMR, and LibQuantum.

5.1.1 Bug Injection in Gem5

Gem5 has an event-driven memory system which includes caches, crossbars, and a DRAM controller model. To create a scenario where an unmapped memory address is ac-

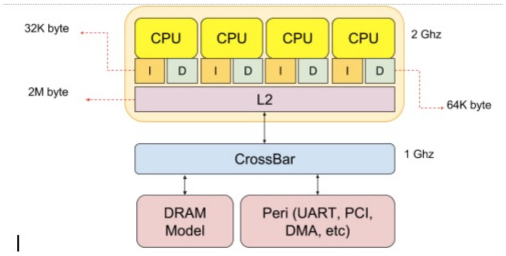


Figure 5.1: Architecture Diagram

cessed, a pre-defined ‘non-responsive memory transaction’ is created. An address is arbitrarily chosen and labeled as a ‘non-mapped-address’ to trigger the non-responsive transaction scenario. This address is seen at the crossbar and tagged as a ‘bug’, and then forwarded to its target destination. The slave will check the ‘bug’ tag and drop the request, skipping the response generation. Since a master (CPU in this case) will never get a response back from the slave, the request transaction will not be retired and the system deadlock situation is simulated. Once the bug is manifested in the simulation, the master (CPU) that generates the transaction will never retire the transaction, and the master will stop the execution of the running program. However, other masters (CPU and other components) can keep generating transactions because the system bus is still functional unless they generate a transaction that hits the bug again or make a transaction that has a hazard condition with the bug transaction.

The second deadlock scenario simulated was a non-responsive device bug. The Uart IP inside Gem5 is connected to the terminal (TTY device) and generates various interrupts during simulation, so a similar infrastructure of creating a ‘non-responsive transaction’ was used to simulate a system deadlock caused by the Uart. After a large arbitrary number of accesses to the Uart, the Uart IP sets the bug tag in the response transaction and the crossbar does not send the response back to the master (CPU). This bug causes the master to go into deadlock by not allowing it to retire the bug transaction, but it also causes the Uart to go into deadlock as well. The bug specifically targets a register inside Uart that clears its interrupts, so the Uart will stop generating interrupts after the bug is hit because the Uart logic believes there is a interrupt in flight for the rest of the simulation. This creates a deadlocked core because it never receives a response as well as an IP that is no longer functional for the rest of the system.

5.1.2 Bus Trace Monitor

To collect transaction data from Gem5, a bus debug monitor was used to create a log file of all the transactions sent through the crossbar. The bus monitor module was placed between the L2 cache and the crossbar component as shown in Figure 5.2. By having the bus monitor at this hierarchy, all device and memory transactions are captured in the trace file. The information collected by the monitor includes the transaction opcode, address, size, source ID, and cycle time.

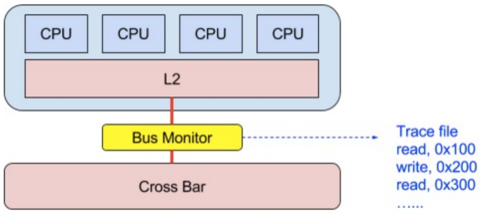


Figure 5.2: Bus Monitor

5.1.3 Creating the Trace Data

The trace data was created by running both the Linux boot sequence and SPEC2006 benchmarks before encountering the bug. This allowed the HIE model to learn the behavior of the SoC with at least 500,000 transactions to determine the bug.

The debug monitor began collecting transaction data for the Linux boot sequence from the start of execution. If the simulation was started with the deadlock scenarios compiled into the simulator, the memory bug would be triggered by DMA transactions and the system would hang before the boot sequence finished. To circumvent this issue, the Linux boot sequence was traced without either of the deadlock scenarios and a Gem5 checkpoint was created to save the state of the simulator after boot. Modifications can be made to the simulator after a checkpoint is created, and the state of the simulation can be restored from the checkpoint after the modifications are made. Once the simulation was saved in a checkpoint, the bugs were compiled separately into the design so they would not interfere with each other. As a result of compiling two different binaries, the traces were different for each bug after restoring from the checkpoint. The simulation and trace were started again after injecting each bug, and a custom program was executed to trigger the deadlocks. The purpose of the program was to create memory transactions through multiple memory allocation commands and eventually issue a request to the non-responsive address and trigger the memory bug. The program was also used to trigger the deadlock scenario for the uart bug by printing text to the terminal and generating a uart transaction every time a character was printed to the screen. After encountering each bug, the simulator was allowed to keep tracing transaction data for about 3ms. By not stopping the trace earlier, the scenario where the active cores would continue their normal operation while one core was deadlocked was simulated.

For the SPEC2006 benchmarks, the simulation started at the checkpoint after booting Linux and the debug monitor began collecting traces from the checkpoint until the simulation was stopped. Unlike the Linux boot sequence where the simulator had to be bug-free in order to not create a deadlock, these benchmarks ran with each bug pre-compiled into the simulator separately. The different binaries created with each bug created slightly different traces for each benchmark and provided more variety of transaction information to analyze the HIE model. After the benchmarks finished running, the custom program was executed to trigger the bugs and create the deadlock scenarios. In the same fashion as the boot sequence, these traces ran for approximately 3ms after hitting the bug to allow the active cores to continue processing instructions. Chapter 6 describes the results acquired from processing the trace data.

5.2 Freedom U500 Testing Platform

For RTL development, the Freedom U500 SoC from SiFive was selected because it is an open source SoC design that allows for easy configuration of cache sizes, number of cores, transaction data size, and attachment of peripheral components. Freedom U500 integrates the Rocket-Chip generator described in [2] and has undergone full functionality testing for academic and industry research. This SoC implements the RISCV ISA and is written in

Chisel HDL for quick development. The RISCV-toolchain also allows conversion of Chisel to Verilog, which allows for existing industry tools to be used for validation and testing. The HIE RTL was written in Chisel, integrated into Freedom, and then converted to verilog to be synthesized onto a Xilinx Ultrascale FPGA using the Vivado 2017.1 tool suite. Linux 4.6.2 was used to boot the SoC and was compiled with riscv64-unknown-linux-gnu-gcc version 6.1.0. The following sections go into further detail of the SoC design and configuration.

5.2.1 Architecture Configuration

The Freedom U500 SoC was configured to have two 64-bit Rocket cores with 32KB L1 instruction and data caches. The L2 cache has been removed by SiFive because of multiple bugs, and it is not used in this experiment. The processors and crossbar peripherals all run at 62.5 MHz and an asynchronous crossing is used to interface with the DDR4 memory controller. There are three devices (Bootrom, UART, GPIO) connected to the SoC in which one is used to generate the deadlock situation.

5.2.2 Bug Injection in Freedom SoC

A non-responsive peripheral bug was injected into the Freedom platform to create a deadlock scenario. This bug was created by adding a counter inside the UART to count the number of requests that are sent to the UART. A large arbitrary number of transactions was chosen (between 1 to 2 million) before the counter triggered the deadlock. Upon seeing the number of transactions we chose, the UART ties the valid signal for the response transactions to zero and the last request is never retired.

5.2.3 HIE Instantiation in Freedom

Unlike Gem5, there is no global crossbar in the U500 SoC where all transactions can be seen. The peripheral devices are connected to an MMIO crossbar and the DRAM memory is connected to a separate crossbar. We ran experiments with the HIE connected to the MMIO crossbar only and to both of the crossbars. By snooping at the port where the cores connect to each of these crossbars, the HIE can keep track of all the requests that go out of the cores. Freedom adds Tilelink Fragmenters throughout the crossbars and these modules take one request and split it into multiple requests. We chose to connect the HIE at the connection of the CPU cluster to each crossbar instead of the Fragmenters to avoid large code modification to the Freedom RTL.

5.2.4 Freedom Simulation

After synthesis and implementation, the HIE was tested on the Freedom U500 design by booting unmodified Linux. The uart bug was synthesized into the design from the beginning of boot and the print messages generated by the Linux kernel were used to generate uart transactions. The bug was encountered after about 1.5 million uart transactions. It was not possible to run the SPEC2006 benchmarks on the FPGA because the HIE detected a deadlock before the Linux boot sequence finished. This could be due to a bug in the design that was not covered during unit testing or if the SoC has a way to poll data and not expect a response for every request, i.e. polling for an Ethernet device that does not exist.

Chapter 6

Parameter Evaluation

As was mentioned earlier, the HIE requires the user to set parameters to get proper functionality from the HIE. This chapter explains the exact parameter values used and the size of the data structures in the HIE. From the trace data collected in Gem5, the parameters in the RET were varied in the python model to find the smallest values that would capture each bug. The graphs use a y-axis with logarithmic values because some benchmarks issued significantly more transactions during the experiments. The raw numbers made it difficult to visualize the data for the benchmarks that generated less transactions. All of the plots show the results of the evictions generated for the traces with the memory bug and uart bug injected. As was mentioned in section 5.1.3, the traces for each benchmark was different so the results for both of the bugs were analyzed to verify that the parameters were not skewed towards the transaction behavior of one of the bugs.

6.1 Transaction Buffer Size and Timeout Value

The Transaction Buffer is dependent on how many transactions can be in flight in a system. If no more than 32 transactions can be in flight at a time, then a XB of size 32 is enough for a system. Our implementation used a XB of size 64 for all of our experiments and a timeout value of 1500 cycles. In general, the smaller the timeout value is the faster the HIE can detect an anomaly, but setting the timeout value too small can create false positives by evicting requests from the XB that will get a response. We found that increasing the timeout value to more than 1500 cycles did not decrease the number of false positives produced.

6.2 LRU_THRESHOLD Values

The LRU threshold values were refined based on the periods where the HIE would evict the most entries from the RET. At the beginning of each trace, the HIE would evict entries with LRU fields that were greater than 3000, meaning more than 3000 transactions had been analyzed since this entry was last hit. After about a million transactions, the HIE became fragmented and was evicting entries with LRU fields set to around 100. To prevent the RET from being fragmented even further, the LRU_THRESHOLD_MIN parameter was varied from 25 to 100 to find the value that generated the least amount of evictions. Figure 6.1 shows the results for Linux Boot and four SPEC benchmarks for the Uart bug and Memory bug.

Smaller thresholds force the entries to merge at a faster rate, leading to less evictions. The final value used in our implementation was 25.

Using the minimum LRU Threshold of 25, the max threshold was tested using the values 100, 500, 1000, 1500, and 2000. We wanted the max threshold to be significantly larger than the minimum threshold so that the RET would not merge entries too soon after they had split. Figure 6.2 shows the number of evictions for these values. The results were similar to those of LRU_THRESHOLD_MIN where the smaller thresholds led to less evictions. The larger values allowed infrequently used entries to occupy space in the RET that could not be merged with the minimum threshold and had to be evicted to create space. A larger RET can help in minimizing evictions with the tradeoff of increasing area, but to minimize the area cost we chose the threshold of 100 for LRU_THRESHOLD_MAX in our design. These values remained the same with our RTL implementation of HIE.

6.3 Update Period

Using the LRU thresholds of 25 and 100, we iterated through values between 10 and 100 to find the optimal point to update RET entry averages and variances. Figure 6.3 shows the number of splits generated for all the benchmarks with both bugs. We compared the number of splits produced because the HIE will split the entries too frequently if the averages are not accurate and evictions will begin to occur. By updating the averages and variances

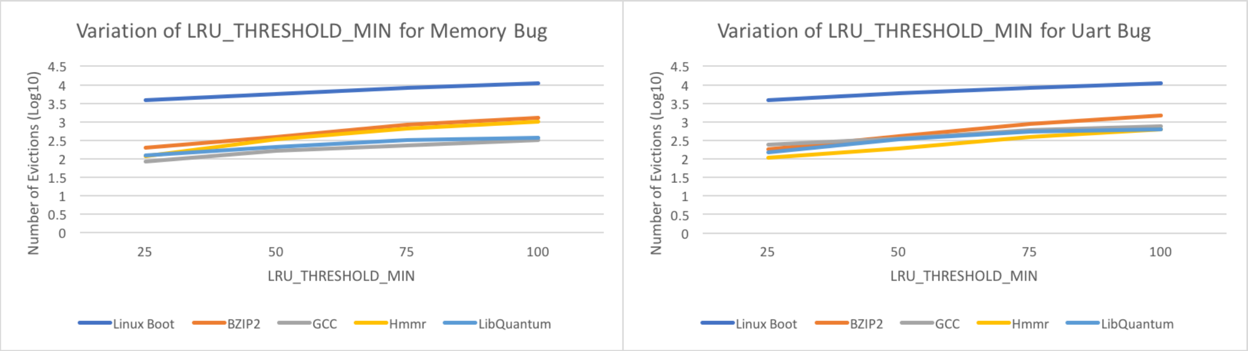


Figure 6.1: Evictions produced for different LRU_THRESHOLD_MIN values

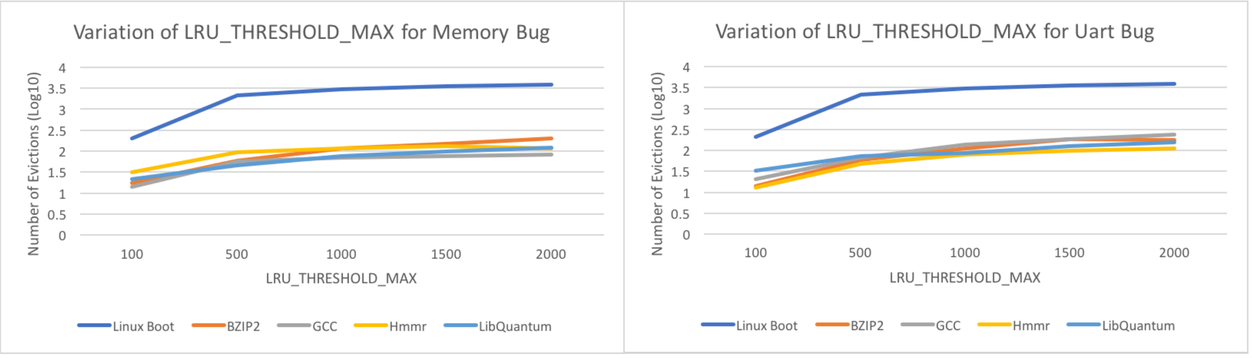


Figure 6.2: Evictions produced for different LRU_THRESHOLD_MAX values

at a faster rate, the HIE is able to learn the behavior of an address range faster and diagnose bugs faster. In both scenarios, the shorter update periods produced slightly less splits than longer update intervals. For this reason, we chose 10 in our final solution for the update period. In our HIE implementation, the value 16 was used instead of 10. This allowed us to do the division with shifting instead of having a lot of logic to division of numbers with non-powers of 2.

6.4 Range Entry Table Size

With the optimal values for LRU thresholds and update period given above, we then varied the size of the RET to find the smallest size that would yield the least amount of evictions and keep as much address range data in the RET. Figure 6.4 shows the results from this experiment. There were little to no evictions produced when the RET had more

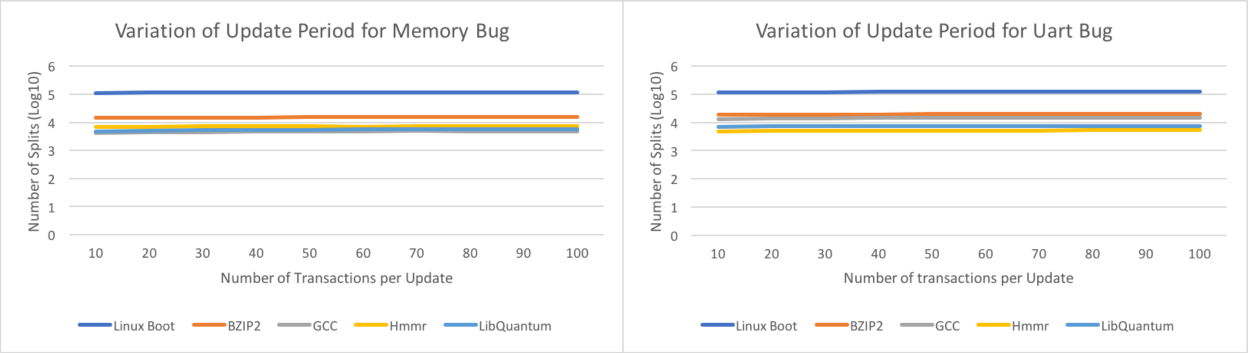


Figure 6.3: Splits produced for different update frequencies.

entries to use. The smaller RETs had less space to use and evicted entries when new data was found. The tradeoff we had to consider was the amount of area the larger RETs would require to implement with the amount of evictions that were generated. The 8-entry RET produced too many evictions to be useful in a system so this configuration was not useful. A 16-entry RET produced significantly less evictions than the 8-entry RET but still more than the 24-entry RET. All the RETs of 24 entries or more produced little to no evictions, so the 32 and 40 entry RETs were ignored and the two RET sizes left to consider were the 16 entry and the 24 entry. Our RTL implementation used the 16-entry RET because the larger RET caused the area of the HIE to grow significantly larger. The Area Cost section explains more on this.

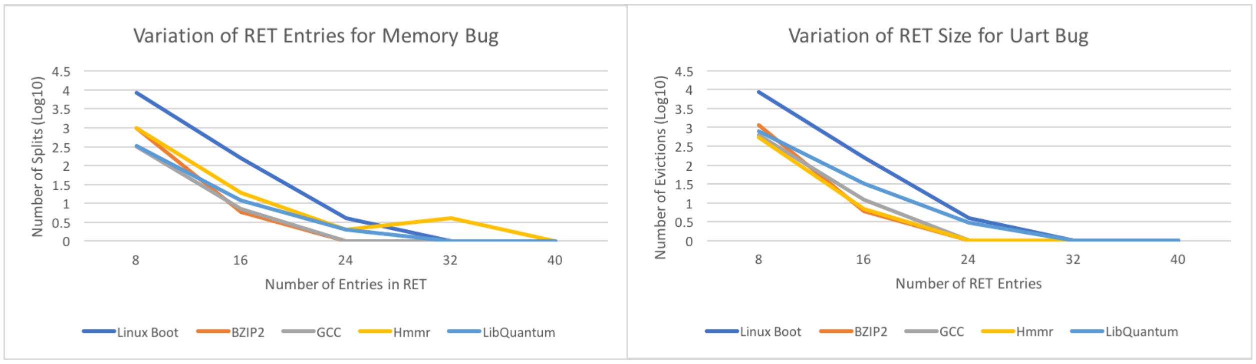


Figure 6.4: Evictions produced for different RET sizes

6.5 Initial Average and Variance

The initial values used for the Read/Write/Invalidate averages were chosen based on the statistics of the GEM5 simulator. GEM5 tracks the average latency of transactions sent to the crossbar and gives the average queueing latency and bus latency. By using the average access time as the average and calculating the variance to have a range that adds the queuing and bus latency, we were able to predict the response times of the transactions enough to minimize split operations. We used the same parameters from Gem5 in our Freedom implementation. Since there is no monitor on the FPGA, it is difficult to measure response times accurately. Finding accurate measurements would entail developing hardware that would act as a monitor and measure the response times.

Chapter 7

Results

The following sections evaluate the implementation details (area and delay costs) and the detection accuracy of the HIE in both the Gem5 and Freedom platforms. VCS simulations were ran to find the delay costs and Vivado synthesis/implementation reports were used to find the physical requirements of the HIE.

7.1 Detection Accuracy in Gem5

The HIE uses the AVG_ANOM_MULT parameter to label transactions that take significantly longer than usual as anomalous, and the value of this parameter had a great effect on the total alerts raised by the HIE. For the deadlock scenarios simulated, the HIE had a 100% true positive rate of detecting the deadlock for parameter values of 8 to 16. No false alarms were raised for deadlock scenarios in any of the simulations.

However, the HIE aims to alert the engineer of anomalous response behavior as well as deadlocks, and the AVG_ANOM_MULT parameter had it's largest effect on detection accuracy when reporting anomalous response latencies. The HIE found a few transactions that behaved anomalously during the simulations, and the rate of false positives grew significantly for smaller values of AVG_ANOM_MULT. Figure 7.1 plots the rate of false positives for the values 8 through 16. Flagging anomalies only when a transaction's response time was greater than 16 times the address range's average yielded the least amount of false alarms for all of the benchmarks and a few of the traces did not generate a false positive at all. The value of 16 filtered out almost all of the outliers in response times created from bus contention and resource congestion. High values for AVG_ANOM_MULT limited the precision of the anomalous behavior that was detected by the HIE, and values of 13 or higher made the design less sensitive to response time variation. Lower values detected smaller variations in response times but the number of false alarms grew significantly with values below 10, and this contradicted the goal of the HIE to mitigate the amount of data an engineer has to analyze to speed up the debug process. In order to have the most accurate and precise anomaly detection for all of the benchmarks, a value of 11 or 12 provided the best results.

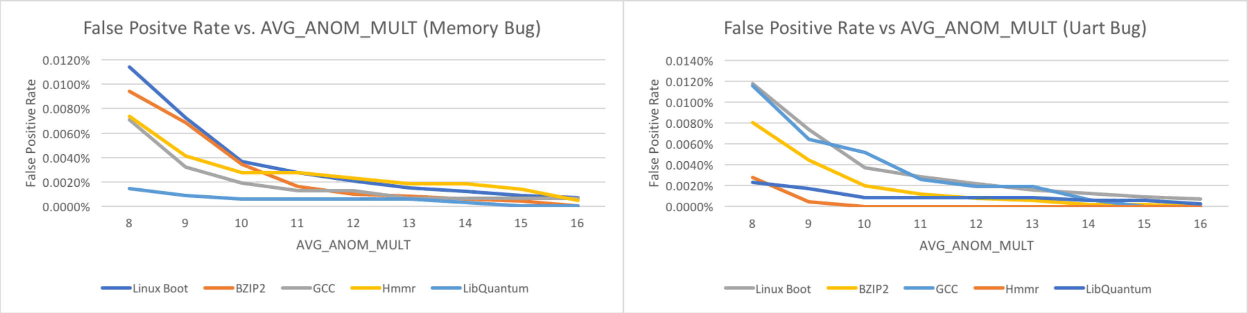


Figure 7.1: False Positive Rate of Anomalies Flagged by HIE for Different Multiplier Values

7.2 Detection Accuracy in Freedom

As was mentioned before, the HIE was instantiated to snoop on device traffic only and snoop on both memory and device traffic. When only device traffic was snooped, the HIE detected the bug 100% of the time and the detection window was 1500 cycles after the initial request occurred. The HIE halted the trace of other transactions into the Trace Buffer, and the anomalous transaction could always be found in the Trace Buffer. When both the memory and device crossbar traffic was input into the HIE, there was a 55% detection accuracy of the bug during Linux Boot on the FPGA. When the bug counter was below 1.5 million transactions, the bug was almost always detected. When the counter was set to values above 1.5 million, the HIE would register a memory transaction as a deadlock and stop the trace of instructions before the bug was actually hit. This could be caused by the initial parameters not being precise for the FPGA, and leading to a lot of stalls and some responses probably being dropped because the HIE is busy processing previous transactions. More analysis of the behavior of the memory transactions is required to avoid all of the false positives the HIE flags.

7.3 Hardware Requirements

The following sections describe the area cost and cycle time of implementing the HIE. The RTL was written in Chisel and the implementation was tested on a Freedom U500 SoC synthesized on a Xilinx Ultrascale FPGA.

7.3.1 Cycle Time

The HIE can take many cycles to process a transaction, and can lead to stalls to process new transactions. An Insert operation takes 18 cycles because it needs to sort the RET to find the largest range it can span and then add the range into the RET. Update and Split operations require three cycles to finish modifying the RET. A Merge operation can take up to 50 cycles to finish because it has to sort all the entries and then iterate through every entry and check the averages and variances in order from lowest to highest, taking two cycles for each entry. The Evict Operation takes 50 cycles because it is the last check done by the Merge Operation.

The worst case scenario where the RET would be busy while new transactions are ready to be processed is when a transaction is inserted and causes the RET to become full, and a merge operation is triggered to make space for new ranges. This process requires 50 cycles to finish before the RET is ready to process new transactions. A 16-entry fifo was the smallest that was able to capture the Non-responsive peripheral bug in the Freedom SoC.

7.3.2 Synthesis Report Analysis

Table 7.2 shows the RTL component breakdown of the Transaction Buffer created by Vivado. The 16 adders are used to increment the timers for each entry. The 36-bit registers are the Address fields in all the entries and the 13-bit registers are the Timer fields. The 8-bit registers are the Source field, the 4-bit registers are the Mask field, the 3-bit registers are for the Size, Opcode, and Param fields, the 2-bit registers are the Cmd_Type field, and the 1-bit registers are for the Valid bit in each entry. The 11-bit register is the response tag that is created when a response transaction is received. There are a few more registers than the 16 that are needed for each entry and this can be attributed to boilerplate logic in

Field	Size	Quantity	Total
Valid	1 bit	1	1 bit
Opcode	3 bits	1	3 bits
Source	8 bits	1	8 bits
Address	36 bits	1	36 bits
Mask	4 bits	1	4 bits
Size	3 bits	1	3 bits
Param	3 bits	1	3 bits
Timer	13 bits	1	13 bits
Cmd_Type	2 bits	1	2 bits
Total			**73 bits**

Table 7.1: Size of Transaction Buffer entry fields

the design (flops for input and output). The majority of the muxes are used to output the data in the Transaction Buffer that has matched with a response. The two input 36, 13, 8, 4, 3, 2, and 1 bit muxes are used to select the entry data that goes to the RET or to the Trace buffer. The 16 input 4-bit mux indexes into the highest available entry out to add new requests. Response opcode decoding is done by the 5 input 3-bit mux and finding the command type is done with the 6 input 2-bit mux. The 32 input mux is a valid signal that is sent to Trace Buffer. This mux has 32 inputs instead of 64 because a timeout error or a response error can send data to the Trace Buffer.

Table 7.3 shows the size of the fields in a RET entry for a 16-entry RET. Address ranges are represented at the page level so Start and End Address only require 36 bits of memory for a 48-bit address. Using these field sizes, one RET entry requires 51 bytes (407 bits) of memory. A 16-entry RET would then require a minimum of 816 bytes of memory for implementation.

Table 7.4 and Table 7.5 show the Vivado component breakdown of the RET. The 36-bit adders are used to calculate the new start and end addresses when a Split or Insert operation take place. 32-bit adders are used throughout the RET to increment LRUs and the Squared Error fields, and to calculate the bounds for considering splits and merges. Updating the running sum is done by the 2 input 13-bit adders and the 3 input 13-bit adders compare new transactions for long anomalous response times. The 16 input 5-bit

Component	**Number of Inputs**	**Output Size**	**Quantity**
Adder	2	13 bit	16
Mux	2	36 bit	32
Mux	2	13 bit	16
Mux	2	8 bit	32
Mux	2	4 bit	32
Mux	16	4 bit	1
Mux	5	3 bit	1
Mux	2	3 bit	96
Mux	6	2 bit	1
Mux	2	2 bit	32
Mux	2	1 bit	113
Mux	16	1 bit	4
Mux	32	1 bit	2
Mux	17	1 bit	1
Register		36 bit	18
Register		13 bit	17
Register		11 bit	1
Register		8 bit	18
Register		4 bit	18
Register		3 bit	54
Register		2 bit	17
Register		1 bit	21

Table 7.2: Vivado Component Report for Transaction Buffer

adder is used to sum the comparators for the index into the sort table, and the other 5-bit adders are used for tracking the periodic update, iterating through all of the entries when sorting, and subtracting 1 from the 16 input adder to have the proper bounds in the sort table. The 4-bit adders are used for the counters in the merge operation to skip over invalid entries.

The registers implemented are mostly all inside the RET. There are two 32-bit registers inside each RET entry (errSq, variance) for three different commands plus an LRU counter, and there are 16 entries which yield 112 32-bit registers for all the entries. There are two 36-bit register in every entry which comes out to 32 36-bit registers required for the RET. There are a few more 36-bit registers implemented because they are needed for storing the new start and end addresses when a Split or Insert is done. Each entry also has one register to keep track of the average and sum for each command, leading to one 13-bit and 17-bit register in each entry. The 5-bit register accumulate because of the three count fields in each entry and the 4-bit registers are used for the entry_id in each entry and to store the order of the RET in the sort table. The 1-bit registers are the valid bits and other flags used to select data out of muxes or to choose the next state.

The synthesis tool used over 9000 muxes to implement the logic inside the RET. This large amount of muxes is needed for all the possibilities that can occur when sorting the RET, and then accessing the RET based on that order. The amount of branching that the state machine has to do based on its calculations also adds to the sum of muxes. These signals are mostly bulked in the 2-input 1-bit mux quantity. As was mentioned in the Cycle Time section, stalls can occur in the RET logic. When the HIE was connected to the Memory and Peripheral Buses, it required a 16-entry Queue to be able to catch the bug. The Queue data is the inputs to stage 1 from stage 0, so each entry contains the opcode, source, address, mask, size, param, timer, and Cmdtype for each transaction. For a 16-entry Queue, each entry requires 72 bits, and in 16 entries, 144 bytes are needed for implementation. The synthesis tool implemented this queue as Block RAM on the FPGA.

A 128-entry Trace Buffer was used to capture the anomalous transactions. Since entries in the Trace Buffer had the same fields as the Transaction Buffer entry minus the

Field	**Size**	**Quantity**	**Total**
Start Address	36 bits	1	36 bits
End Address	36 bits	1	36 bits
Valid	1 bit	1	1 bit
Update	1 bit	1	1 bit
ID	4 bits	1	4 bits
LRU	32 bits	1	32 bits
Average	13 bits	3	96 bits
Variance	64 bits	3	192 bits
Sum	64 bits	3	192 bits
ErrSqrd	64 bits	3	192 bits
Count	5 bits	3	15 bits
Total			**797 bits**

Table 7.3: Size of RET entry fields

Component	**Number of Inputs**	**Output Size**	**Quantity**
Adder	2	36 bit	2
Adder	2	32 bit	19
Adder	2	17 bit	3
Adder	3	16 bit	3
Adder	3	13 bit	3
Adder	2	13 bit	3
Adder	2	5 bit	7
Adder	16	5 bit	1
Adder	2	4 bit	1
Register		36 bit	39
Register		32 bit	116
Register		17 bit	48
Register		16 bit	48
Register		8 bit	2
Register		5 bit	52
Register		4 bit	40
Register		3 bit	7
Register		2 bit	1
Register		1 bit	39

Table 7.4: Vivado Component Report for Range Entry Table

Component	**Number of Inputs**	**Output Size**	**Quantity**
Mux	2	36 Bit	1157
Mux	2	32 Bit	418
Mux	16	32 Bit	1
Mux	2	17 Bit	465
Mux	3	17 Bit	16
Mux	2	16 Bit	464
Mux	4	16 Bit	16
Mux	2	13 Bit	515
Mux	2	5 Bit	1013
Mux	3	5 Bit	16
Mux	4	5 Bit	4
Mux	16	4 Bit	2
Mux	2	4 Bit	31
Mux	5	4 Bit	2
Mux	2	3 Bit	13
Mux	6	3 Bit	1
Mux	3	3 Bit	1
Mux	2	2 Bit	231
Mux	2	1 Bit	4578
Mux	5	1 Bit	18
Mux	16	1 Bit	2
Mux	4	1 Bit	2
Mux	15	1 Bit	1
Mux	8	1 Bit	1
Mux	7	1 Bit	1

Table 7.5: Vivado Component Report for Range Entry Table Continued

Component	Number of Inputs	Output Size	Quantity
RAM		4608 bit	1
RAM		1024 bit	1
RAM		512 bit	1
RAM		384 bit	3
Mux	2	36 bit	1
Mux	2	8 bit	1
Mux	2	4 bit	1
Mux	2	3 bit	3
Mux	2	1 bit	8

Table 7.6: Vivado Component Report for Trace Buffer

timer and valid fields, each entry required 8 bytes (59 bits) of storage. This means the Trace Buffer would require a minimum of 1024 bytes of storage to be implemented. Table 7.6 shows the Component report for the Trace Buffer implementation. JTAG was not implemented in the design and in order to have the Vivado synthesize the Trace Buffer, the fields of each Trace Buffer entry was output to the top level of the HIE. The muxes implemented were used to select data from Stage 0 or Stage 2 to add to the Trace Buffer. The 4608-bit RAM stored the Address for the 128 entries and the 1024-bit RAM saved the Source fields of the anomalous transaction. For the Param, Size, and Opcode fields, Vivado created three 384-bit RAMs as storage and the Mask Field was stored in the 512-bit RAM.

7.3.3 Final Implementation Results

Table 7.7 shows the FPGA resource breakdown for the entire Freedom SoC, the HIE, and each of the components in the HIE. The HIE used 17% of the total LUTs used to implement the design and 10% of the total registers. The areas where the HIE required more hardware was with F7 and F8 muxes, using 36% and 79% of all the muxes respectively. Only one tile of Block RAM was used in the HIE implementation and this was used for the Trace Buffer.

Table 7.8 shows the resource utilization of the HIE and of one Rocket core. Each core required about 13503 LUTs, 9693 registers, 376 carry chains, 170 F7 muxes, 34 F8 muxes, 36 Block RAM tiles, and 8 DSP slices for implementation. The total LUTs used for the

Module	**CLB LUTs**	**CLB Registers**	**Carry8**	**F7 Muxes**	**F8 Muxes**	**Block RAM Tiles**	**DSPs**
Freedom U500	104245	77497	1692	3385	636	136.5	11
HIE	17682 (17%)	7702 (10%)	318 (19%)	1228 (36%)	505 (79%)	1 (1%)	0 (0%)
Transaction Buffer	531	703	32	0	0	0	0
Range Entry Table	16854	6954	282	1228	505	0	0
Trace Buffer	297	11	4	0	0	1	0

Table 7.7: HIE Utilization Report after Implementation

Module	**CLB LUTs**	**CLB Registers**	**Carry8**	**F7 Muxes**	**F8 Muxes**	**Block RAM Tiles**	**DSPs**
Freedom U500	104245	77497	1692	3385	636	136.5	11
Rocket Core	13503 (13%)	9693 (13%)	376 (22%)	170 (5%)	34 (5%)	36 (26%)	4 (36%)
HIE	17682 (17%)	7702 (10%)	318 (19%)	1228 (36%)	505 (79%)	1 (1%)	0 (0%)

Table 7.8: FPGA Resource Utilization for HIE and Coreplex

HIE amounted to 37% more LUTs than one single Rocket core, and the registers and carry chains were fewer but comparable. The most significant difference was the amount of Block RAM tiles required for implementation. One core needed 36 block RAM tiles while the HIE used only one. Each Block RAM tile in the Xilinx Virtex Ultrascale architecture has 36Kb of data storage [7] and this considerably increases the area of the core. In summary, the HIE uses 37% more logic than a single core, but only needs 3% of the total storage, making the HIE small and deployable in an SoC if the size of SRAM is larger than that of standard cells.

Chapter 8

Summary and Conclusion

The existing solutions for post-silicon bug diagnosis require the debug engineer to manually dissect the data collected through debug tools in order to find the root cause of a bug. As SoCs add more cores (masters) and devices, the manual analysis of debug data will increase the time to diagnose a bug and increase a product's time to market. The HIE design presented in this thesis presents a solution that will provide faster bug diagnosis by learning the response times of transaction data on-chip and flagging anomalous transaction behavior before trace data can be overwritten by the progress of different masters. By only having to adhere to the protocol behavior of the interconnect, the HIE has quick development time and adds little overhead to the design phase of a product.

The HIE is a step in the right direction to provide an on-chip debug solution that can streamline the debug effort for engineers working on post-silicon products. This current design adds a significant area overhead to the SoC, but there is room for improvement by changing the logic and using different components for implementation. If HIE can be deployed into an existing system and allow engineers to have a real-time diagnosis to deadlocks and other bugs, time to market can be greatly reduced and products can be sold faster. With the help of HIE, heterogenous SoCs can continue to grow in complexity, and either maintain or reduce the design effort of current SoCs.

Bibliography

[1] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A reconfigurable design-for-debug infrastructure for socs. In *Proceedings of the 43rd annual Design Automation Conference*, pages 7–12. ACM, 2006.

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[5] Andrew DeOrio, Jialin Li, and Valeria Bertacco. Bridging pre-and post-silicon debugging with biped. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 95–100. IEEE, 2012.

[6] Andrew DeOrio, Qingkun Li, Matthew Burgess, and Valeria Bertacco. Machine learning-based anomaly detection for post-silicon bug diagnosis. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 491–496. IEEE, 2013.

[7] Xilinx Inc. Ultrascale architecture configurable logic block, 2017.

[8] David Lin, Ted Hong, Yanjing Li, S Eswaran, Sharad Kumar, Farzan Fallah, Nagib Hakim, Donald S Gardner, and Subhasish Mitra. Effective post-silicon validation of system-on-chips using quick error detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1573–1590, 2014.

[9] Nikola Nicolici and Ho Fai Ko. Design-for-debug for post-silicon validation: Can high-level descriptions help? *IEEE International High Level Design Validation and Test Workshop*, November 2009.

[10] Eli Singerman, Yael Abarbanel, and Sean Baartmans. Transaction based pre-to-post silicon validation. In *Proceedings of the 48th Design Automation Conference*, pages 564–568. ACM, 2011.

[11] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. *Proceedings of the International Symposium on Microarchitecture*, November 2008.