ABSTRACT

SRINIVASAN, VINESH. Phase II Implementation and Verification of the H3 Processor. (Under the direction of Dr.Eric Rotenberg).

With the 2D version of the H3 processor from phase I tapeout well into post silicon debug, this thesis effort focuses on the implementation of 3D version for the phase II tapeout. The main focus is on fixing the bugs identified in the phase I chip, netlist simulation based verification effort to identify new bugs and fixing them.

In addition to the 3D restructuring of the chip from Phase I tapeout, some new features have also been implemented for the 3D tapeout, which are also an integral part of this thesis. This includes, a new data cache that has been integrated to replace the OpenSparc T2 data cache from the phase I tapeout, virtual memory support for not flushing caches on migration and an invalidation based cache coherence mechanism in order to avoid the threads hitting on its own stale data after migration. In addition to the existing design, these new features are also extensively tested for corner cases.

Detailed analyses of the identified bugs, their fixes, stress tests to ensure the fixes work correctly are documented. New features were put under stress tests and some interesting bugs related to the invalidation based coherence mechanism in the two-core-stack are explained in detail. Apart from the verification tests, performance tests to measure novel features of the H3 processor such as Fast Thread Migration (FTM) latency and Cache Core Decoupling (CCD) advantages have been done. The benefit of not flushing caches on a migrate is also analyzed in detail. Finally, benchmark performance results from the H3 processor for some real world applications are shown to underline the benefits of a 3D heterogeneous multicore processor.

Phase II Implementation and Verification of the H3 Processor

by
Vinesh Srinivasan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2015

APPROVED BY:

_____	_____
Dr.Eric Rotenberg Committee Chair	Dr.W.Rhett Davis

_____
Dr.James Tuck

DEDICATION

To my guru, Prof. Venkateshwaran.

BIOGRAPHY

Vinesh was born in Chennai, India on 21st April, 1992. He earned his Bachelors of Engineering in Electronics and Communication Engineering from Anna University, Chennai, India in May 2013. Vinesh was part of the High Performance Computer Architecture group at Waran Research Foundation (WARFT, India) before joining NC State on fall 2013. Vinesh will receive his Master of Science degree in Computer Engineering with the defense of this thesis.

ACKNOWLEDGMENTS

Thanks to my advisor, Dr. Eric Rotenberg for his support and guidance right from the start of my time at NC State. I also want to thank him for advising me throughout the course of this year-long project. His commitment to achieve perfection both as a teacher and a researcher is inspiring. I immensely benefitted both from his classes and our research discussions at his office. I would like to thank Dr. James Tuck and Dr. Rhett Davis for agreeing to serve on my thesis committee and their valuable feedback.

Thanks to Rangeen and Anil, for helping me initially to get accustomed to the tools and infrastructure needed for research. Their valuable inputs and discussions helped me throughout the course of this thesis. I also would like to thank Elliott for his guidance related to this project. Without their help this thesis would not be possible. I enjoyed all the fun discussions with Anil which made our time at the lab memorable. I would like to thank Naren and Raghav for all their help, the driving classes, cooking recipes and more which made my life at Raleigh enjoyable.

Thanks to my dad Srinivasan and mom Vijayavalli as they gave me the freedom to make my own decisions and are always supportive no matter what I do. I am forever grateful to them for the sacrifices that they make in order to make my life better. I also would like to thank my little brother, Harish for always being there for me. Finally, I would like to thank Srividhya for her unconditional love and support to my decisions.

This research was supported by a grant from Intel Corporation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The instruction level parallelism (ILP) inherent in different programs and different phases of the same program varies. Thus the need to design diverse microarchitectures each specialized to suit different program phases arises. Microarchitectural diversity is of three types:

1. Single-ISA heterogeneous multicore processor where each core may differ in their superscalar structure dimensions and/or frequency.
2. Adaptive superscalar processor which is a single reconfigurable core
3. CPU + accelerators (ASICs, FPGA, GPGPUs)

1.1 H3 - 3D heterogeneous multicore processor

The H3 processor comes under the class of single-ISA heterogeneous multicore processor. H3 is a dual-core processor implemented using 3D die stacking where both the cores are integrated by face-to-face, microbump based buses. It consists of two core types named based on their peak instruction fetch rate. Two-wide OOO core, a high performance core that has bigger superscalar structure sizes to extract more ILP from the running program and a one-wide OOO core, a low power core that has smaller structure sizes more suitable for phases that less exploits the performance benefit of the big core. The rationale behind running on the little core is to reduce energy consumption with only a small slowdown compared to always running on the big core [1].

The little core utilization increases significantly when the migration intervals i.e. number of retired instructions while running on a core before migrating to the other, is lower [1]. This is due to the ability to exploit the short phases where the little core performance is

closer to the big core. But, fine grained thread migrations are more sensitive to migration overheads as the opportunity to exploit the short phases is lost if the overhead is high. Migration overheads come from the following two sources:

1. **Transferring architectural register state**: Architectural register state includes the architectural registers, program counter etc. The latency for transferring the architectural register state from the source core to the destination core is a major source of migration overhead.
2. **Migration induced misses and mispredictions**: After migration, if the core misses a cache block that exist in the previous cache but not in the current cache it is called a migration induced miss. This can occur due to two reasons.

   i) If there are two threads running, then one thread can evict blocks that might be useful for the other thread after migration.

   ii) Stores on the other core can cause invalidations in the current core.

   Other speculative structures like the branch predictor need time to get trained as counter values can be evicted by the other thread. Migration induced cache misses include these extra misses incurred.

3D die stacking enables fine grained thread migration by overcoming these migration induced overheads [1]. H3 handles these migration induced overheads via two mechanisms, they are:

Fast Thread Migration (FTM):

FTM handles the overhead of transferring architectural register state from the source core to destination core. FTM is a single-cycle swap of the architectural register state contents from

one core to the other. It is made possible by using a dedicated structure that holds the architectural register state called the "Teleport Register File" (TRF) and face-to-face buses that connect each bitcell of the TRF to the other core's TRF as shown in figure 1.1.

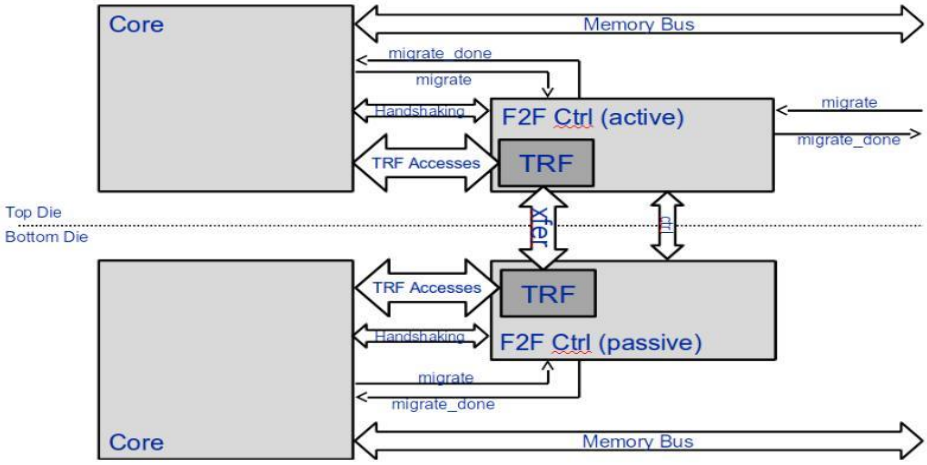

Figure 1.1 Fast Thread Migration Design in H3 [3]

The TRF registers, F2F logic are in a separate module called F2F control which also has the core suspend and resume handshaking. Although the swap can occur in one cycle, the cores require some cycles for consolidating the architectural register state from the Physical Register File (PRF) to the TRF using special instructions that does the move from the PRF to the TRF and vice versa. Along with the register values, the program counter is also moved to the TRF. Including consolidation, FTM has around 100 cycle latency for transferring architectural register state from one core to the other.

Two types of migrations can occur in the H3 design:

1. **Global Migrations:** Hardware-triggered migrations that are based on an external interrupt.

2. **Local Migrations:** Software-triggered migrations by embedding the migrate instruction along with the program code, which is executed by the processor.

Cache Core Decoupling (CCD):

Cache Core Decoupling (CCD) tries to reduce the overhead caused by migration induced cache misses. CCD enables each core to access the other core's L1 instruction and data cache. Figure 1.2 shows the CCD logic located at the fetch and memory state of the pipeline that connects each core to the other core's instruction and data caches respectively. This enables the threads to freely migrate from one core to another without having to switch caches. There can be different modes of operation with CCD. The threads can be mapped to the best core, cache throughout its execution. Or, the threads can choose to migrate cores but not caches thus avoiding migration induced cache misses.

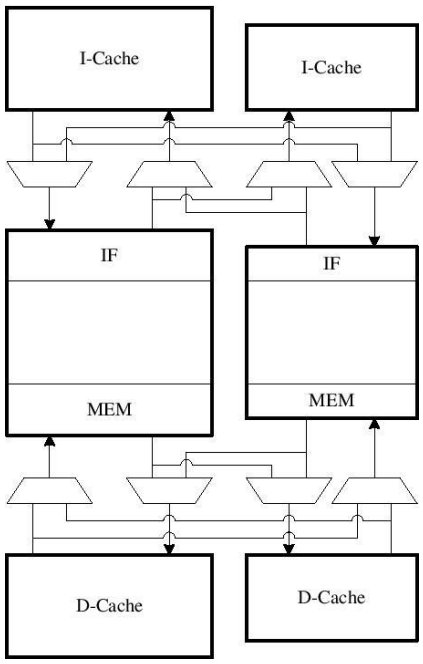

Figure 1.2 Cache Core Decoupling design in H3 [1]

1.2 H3 Tapeout: Phase I (2D version)

Phase I of H3 involved building a 2D version of the two-core-stack. The 2D version is mainly to test the design which included the FabScalar generated cores with compiled memories for instruction and data cache, FTM, CCD implementations, and also to work out the physical design flow. In addition to the two-core-stack, the phase I design also included a two-wide debug core. The debug core had synthesized scratchpads for instructions & data and full scan. The purpose of designing the debug core is to test the core without risks caused by the synthesized RAMs, and the highly complex OpenSparc T2 D$ which the two-core-stack was using. The phase I design was taped out on May 2013. The tapeout was done using IBM 8RF (130 nm) PDK and ARM standard cells, pads and memory compilers.

Post-silicon Tasks:

The debug core which was tested using a chip-on-board setup was successful in performing the liveness test and went onto retire millions of instructions of microbenchmarks. The two-core-stack in its Quad Flat Package was surface mounted onto the 4-layer PCB. The PCB was connected to Xilinx ML605 FPGA board as a mezzanine card using an LPC connector. The testbench, which also acted as L2 for servicing L1 instruction and data cache misses was synthesized to the FPGA. With this setup in place, more bugs were discovered in the chip during post-silicon debug process which are documented in detail in the WARP submission [2]. Chapter 2 describe the fixes that are made for each of these bugs in detail.

1.3 H3 Tapeout: Phase II (3D Version)

The phase II design of H3 involves implementing a 3D version of the two-core-stack. Phase II also includes fixing the bugs identified in the post-silicon debug process of phase I,

identifying and fixing newer bugs, implementing newer features that did not make it into the phase I tapeout, all of which is documented in this thesis. In phase II, the two chips are vertically stacked and bonded with face-to-face microbump based buses. Conventional top metal layer pads cannot be used because the two chips are bonded to each other. Hence, we are planning to use Ziptronix 3D process of implementing pads on the metal 1 layer of the top (flipped) chip. The pads are exposed by thinning the upper chip's substrate and back-etching through the substrate to reach the pads. The chip will then be wirebonded on to the package. Since the top chip has all the pads allotted for the two-core-stack, the bottom core needs to route its I/O through the top core to reach to the pads.

Figure 1.3 shows the arrangement of the two-core-stack, the two-wide core and its corresponding FTM logic (Teleport top) is located in the top tier, the one-wide core and its FTM logic (Teleport bottom) is located in the bottom tier. The teleport units in both the cores interact using the F2F-FTM bus interface, which is as wide as the TRF registers in the cores to enable single cycle data transfer. CCD control orchestrates the core-cache mapping for both the cores. It's a separate control module that handshakes with the suspend and resume logic in both the tiers.

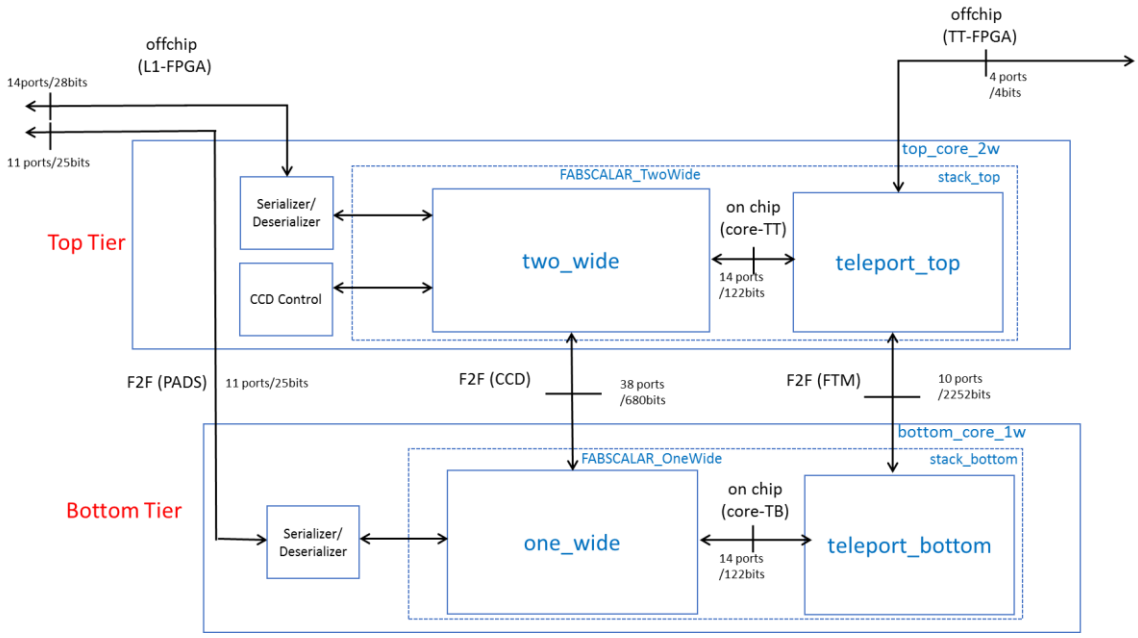

Figure 1.3: H3 Phase II design block diagram

The cores access the remote core's instruction and data caches using the F2F-CCD bus interface. The serializer/deserializer module is an asynchronous FIFO that does the job of narrowing the wide bus from the core to 8-bit core-to-mem bus, thus reducing the number of pads needed for the two-core-stack. An arbiter (not shown in the figure) arbitrates the 8-bit core-to-mem bus among three clients. They are, the instruction cache request, load request, and store request. As it is seen from the figure, the bottom tier's I/O bus has to route through the top tier using the F2F-pads interface to reach to the pads.

1.4 Contributions for Phase II:

This section briefly outlines my contributions made for the Phase II tapeout that are discussed in detail in the future chapters in this thesis.

1.4.1 Bug Fixes:

1. ***Post-Silicon phase I bugs & fixes:*** The post-silicon bugs identified during the phase I tapeout is documented in detail along with their fixes. Some of the bugs which we were able to reproduce in RTL simulations were reproduced and fixed. Netlist simulation was used to fix the rest of the bugs.

T2 Data Cache Bug: One among the critical bugs identified during netlist simulation was the Opensparc T2 data cache bug. The T2 design explicitly used latch based indexing for its tag and data RAMs. On a load fill, due to hold violation in the address latch the tag RAM was not written. In addition, the use of latches for indexing was found in other parts of the design including the store buffer. Thus we decided to replace the T2 data cache with an in-house designed data cache. The efforts related to the new data cache integration are outlined in the new features section.

2. ***Netlist Simulation*****:** Along with the phase II design, the synthesis scripts should be updated to compile two separate designs instead of one top level design. The changes to the synthesis scripts, sdf annotation, modifying the testbench to interact with the netlist instead of the RTL are discussed in detail.

1.4.2 New Features:

1. ***3D Restructuring:*** After phase I tapeout, the restructuring for phase II involved, splitting the top level module of the 2D structure into top and bottom tiers, moving the common logic related to the CCD control to the top tier. More effort was spent on modifying the I/O for the bottom tier to face-to-face buses and routing them through the top tier to reach to the pads. The complexity arises mainly from the huge number of F2F signals. The different classes of F2F signals are FTM, CCD and pads.

2. ***I/O Classification and Routing:*** Each class of I/O, interfaces to a different module in each tier hence needed careful classification and routing compared to the 2D design. The F2F buses were annotated to differentiate them from the I/O to ease physical design process of routing the F2F signals to the face-to-face microbump based buses and I/O to pads.

3. ***New Data Cache Integration:*** One of the major changes in the phase II design over phase I is the new data cache. The T2 data cache used compiled RAMs hence the pipeline was modified to suit the synchronous read/writes. The new data cache with its memory synthesized to flip-flops has asynchronous reads and synchronous writes. The pipeline had to be modified to accommodate this change. Further, the T2 data cache arbitrated load and store requests to the memory internally which eased top level design in the phase I tapeout. Arbitration had to be done at the top level module for phase II as the new data cache did not do the load/store request arbitration internally.

4. ***Virtual Memory Support:*** Another major change in phase II tapeout is the design of a 1-bit virtual memory mechanism. The two-core-stack design is a multiprogrammed

environment hence there is no memory sharing as in a multithreading environment. The caches in the phase I tapeout were virtually indexed and virtually tagged and during a migration caches were always flushed as the other program's address space is different. The L2 always maps the cache to the correct address space after migrations. This was easy to implement as it did not require any virtual-physical memory translation.

For the phase II tapeout, the caches are made virtually indexed but physically tagged this enables us to not flush caches during migrations. The virtual to physical translation is a 1-bit thread-id that maps the cache block to its corresponding address space in the memory.

5. ***Cache Coherence:*** Although there is no memory sharing, when we are not flushing caches, cache coherency among the caches is still required because a thread can load a data from memory, migrate from one core to the other, commit to the same memory, migrate back and will receive a stale data. An invalidation based cache coherency protocol has been implemented to avoid this scenario in phase II. It required changes to the L2 for monitoring the tag banks of the caches to generate invalidations and also to the data cache to receive these invalidations and act upon them.

1.4.3 Verification:

The verification effort consists of different classes of test cases. Each of the classes focus on testing different aspects of the design. The tests are classified as:

1. ***Stress tests for bugs and fixes:*** These tests focus on the phase I bug fixes and stressing them to ensure correct functionality in the phase II design. Detailed explanations regarding the test run and the aspects these tests are covering are shown with the help of test counters.

2. ***Stress test for new features:*** Tests for the new features include tests for the new data cache, virtual memory and cache coherence support. Also, new bugs found during these stress tests and their fixes are discussed in detail.

3. ***Performance tests***: These tests are to measure the performance of novel aspects of the phase II H3 design. Tests designed for measuring the FTM latency, CCD benefits, not flushing caches benefits were executed to make sure they match with the expected performance from the chip. These test strategies along with the results is discussed in detail.

4. ***SPEC benchmarks:*** Commercial workloads were run on the design to show the real world application of the H3 processor and graphs showing these performance results are shown and discussed.

Chapter 2 discusses the list of bugs found and fixed from the phase I tapout and the netlist simulation effort. Chapter 3 explains all the new features of the phase II tapeout over phase I. The verification effort - tests run for finding bugs, stressing fixes, new features, the tests to measure FTM, CCD and no-flush benefits are listed in Chapter 4. Chapter 4 also discusses the performance results based on SPEC benchmarks and microbenchmarks run on the design. Chapter 5 summarizes the thesis outcomes.

CHAPTER 2

BUG FIXES

A comprehensive list of bugs found in H3 phase I and their work-around are presented in the WARP submission [2]. This chapter explains in detail some of the bugs in the design found in the post-silicon debug process of the phase I chip that are fixed as part of this thesis's efforts. Some of the bugs which were reproducible in RTL simulations were fixed. Other bugs were investigated using netlist simulation and fixed. This chapter will also discuss the netlist simulation process in detail and how it was useful in investigating bugs that was observed on the chip but not in RTL simulations.

2.1 Load Wakeup Bug:

Bug Description: Load wakeup bug was caused by a misplaced `ifdef in the RTL which was exercised only in simulation and not during synthesis. In the fabscalar pipeline, once a load is issued for execution, its writeback packet is created with information related to the load i.e. its destination register ID, Activelist ID etc. The valid bit of the packet is set based on both the load's validity and data valid that comes after the load has hit in the cache. For a load that misses in the cache, its writeback packet consists of all information about the load with its valid bit set as '0'.

The bypass packet for wakeup consists of register tag, data and valid which is created from the writeback packet in the writeback stage of the memory lane. Whenever an invalid writeback packet arrives, its bypass packet register tag is filled with the default wakeup tag P0. Since, in PISA ISA, the register P0 is always set to 0, no instruction can write to P0, hence instruction dependent on P0 already have their ready bit set. This fact was used in the

issue stage wake-up mechanism for invalid bypass packets, as all instructions can be checked only for a match with the register tags and not the valid bit of the bypass packet. This crucial part of packing the bypass packet with P0 was inside the `ifdef, which got omitted during synthesis. So despite the bypass packet not being valid the register tags from the writeback packet was broadcasted, thus causing early wakeup of the load dependent instruction that already reached the issue queue. Although, future instructions that are just getting renamed get their ready bits from the PRF, once they reach the issue queue, matches with the register tag every time the load replays thus causing early wakeups throughout the load's miss penalty.

Fix: There are two fixes that are made to ensure that the bug is fixed.

1. First is removing the misplaced `ifdef from the RTL to make sure that the netlist and RTL simulation matches. All other instrumentations were also removed to make sure that the RTL simulation was not running based on any instrumentation that will not be used in synthesis. This ensures that the bypass packet wakeup tags were filled with P0.
2. The second part of the fix is in the issue stage wakeup logic of the pipeline. Earlier only the register tag match from the bypass packet was used to wake up dependents. Now as an additional check, the valid bits of the bypass packet are also checked to qualify the match vectors. With this check, even though the register tags of the bypass packet matched with the source registers of the instructions in issue queue, the valid bits will determine whether the match vectors qualify for waking up the instructions. Once the match vectors qualify they are sent to the select logic of wake up.

2.2 T2 D-Cache Bug:

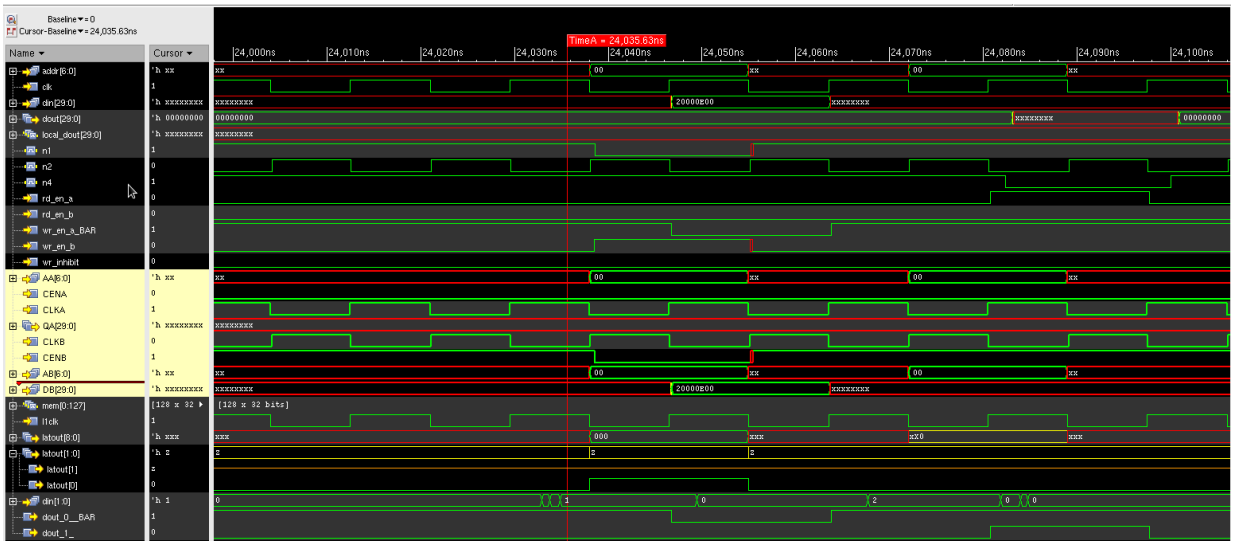

Figure 2.1 Waveform showing the hold time violation in T2 Data Cache

Bug Description: The T2 data cache used for the phase I tapeout had compiled RAMs for tag and data banks. Whenever a load misses in the cache, a request is sent to the memory and when the fill arrives, the tag bank needs to be written. The T2 data cache uses a highly custom latch based design for writing RAMs. The index/address gets latched into a positive level sensitive latch and the tag goes through a flip flop. Both the edges of the clock are used to write into the RAMs.

The waveform figure 2.1 shows the output of the latch "latout" for the address and "din" shows the data to be written into the tag RAM. The I/O of the tag RAM is highlighted for clarity. The signals AA, CENA and CLKA belongs to the read port, AB, CENB, CLKB,

DB belongs to the write port of the RAM. The CLKB signal is a 180-degree phase shifted version of CLKA. The positive level of CLKA is used to latch the address, enable which is shown as input AB, CENB (active low) to the RAM. The next posedge of CLKA is used to latch the data which is input DB to the RAM. The RAM is expected to get written on the posedge of CLKB. But due to hold violations at the latch, the address AB becomes 'x' before the posedge of CLKB causing the fill to be incomplete. The output of the RAM QA is shown.

Fix: This style of using latched to write the RAMs were found prevalently in other places of the design including the data RAM and store buffer write logic. In addition, the highly structured design of the T2 data cache made debugging difficult. Hence to fix this issue, the T2 data cache was replaced with an in-house developed data cache. The efforts related to replacing the data cache are described in detail in Chapter 3. Chapter 4 discusses stress cases run on the new data cache to make sure other problems related to the data cache do not surface in phase II.

2.3 False BTB hit bug:

Bug Description: The BTB of the two-core-stack was implemented using compiled RAMs, including the valid bits, hence it is not resettable. Hence, after a thread migrates from a source core, it's BTB is not resetted. When a new thread migrates to this core if the PCs of the new thread matches with the stale data in BTB it may generate a hit. To tackle this scenario, in the predecode stage, all instructions opcode, taken target, and immediate fields are decoded and compared with the information obtained from the BTB. In case of genuine branches, based on the type of the branch and predicted direction, its target or sign extended

immediate is compared with the BTB taken PC and branch type. If it fails to match, then it is considered as a BTB miss even though it hit in the BTB, thus forcing recovery in case of predicted taken branches. In case of other instruction types hitting in the stale data, their opcodes are used at predecode to generate a false BTB hit signal that forces recovery. As both BTB miss and false BTB hits forces recovery, they were handled in a similar manner in the fetch stage 2. Fetch stage 2 also handles allocating CTIQ slots for control instructions which will be updated later when they execute. Since instructions that generate BTB misses are genuine branches they were allocated CTIQ entries. Erroneously, the same logic was applied for false BTB hits causing wrong allocation of CTIQ entries. This resulted in wasted CTIQ entries as the instructions that false BTB hits will never clear their CTIQ entry. Eventually, core runs out of CTIQ entries and deadlocks.

Fix: The false BTB hit logic detects the invalid entries in the BTB efficiently. Hence, the BTB was left unchanged from phase I, with the valid bits still part of the RAMs. The CTIQ allocation logic in fetch stage 2 has been fixed to not allocate CTIQ entries for false BTB hits. The CTIQ entry is allocated based on a control instruction vector that points to the control instructions in the current fetch bundle. Instructions that are BTB misses are genuine control instructions that are either missed in the BTB or hit on a wrong entry. Hence, the control instruction vectors for BTB misses are set in fetch stage 2 after predecode. This code previously was setting the vector for false BTB hits, causing wrong CTIQ entry allocation. This setting of control instruction vector is blocked for false BTB hits now. Chapter 4 discusses in detail about the stress tests done for the false BTB hit detection logic and also for the fix.

2.4 CTIQ Full Bug:

Bug Description: The local migrate instruction that is executed along with the program code at retirement, triggers the FTM logic for state transfer. Once the migration flag goes high, the fetch stage is stalled to make sure no further instructions are fetched. But at the intermediate time between fetch and retirement of the migrate instruction, the pipeline fetches past the migrate instruction that alters the pipeline stages that is not squashed during migrate. One such stage is the CTIQ allocation stage, where new control instructions, when fetched are allocated CTIQ entry that are later cleared after execution. But control instructions that are fetched past the migrate instruction, only gets allocated CTIQ entries and are not cleared. As the thread keeps migrating from one core to the other, there is a lost CTIQ entry for every migration. Eventually, after the thread has migrated 32 times the CTIQ, which is sized at 16 entries for the two-wide core, gets filled up. And at the 33rd migration the core stalls as CTIQ is full.

Fix: This bug can be fixed both when the thread migrates from a core or a new thread migrates to this core. We chose to do the latter as it involved lesser modifications to the pipeline. The fix is done when the new thread resumes execution at the core. After the move from TRF registers is complete, the core sends a resume signal to begin fetch. This signal was also used to reset some of the pipeline stages. Unfortunately, the fetch stage 2 missed resetting causing the CTIQ bug. Now the resume signal resets all the pipeline stages including fetch 2 and dispatch which was missed out during phase I tapeout. A detailed stress test at Chapter 4 tests this fix further by doing multiple migrations with control instructions to ensure that flush after migrations solves the issue.

2.5 CCD Bug:

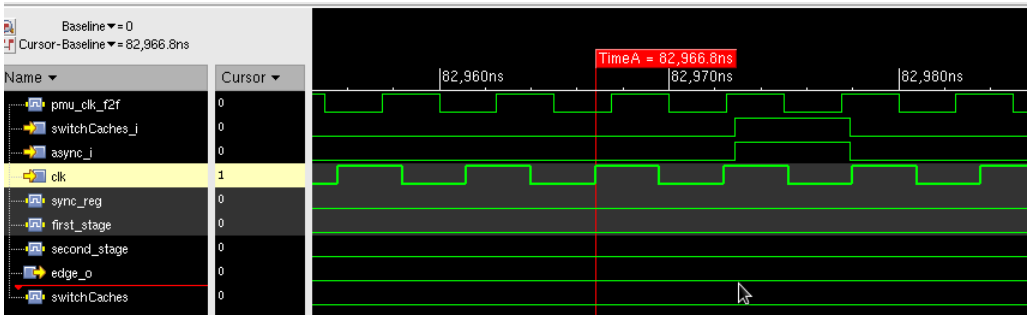

Figure 2.2 Waveform showing the CCD Bug

Bug Description: The CCD control state machine is responsible for the select signal to the muxes that maps the core either to the remote or to the local caches when CCD is enabled. They work based on the F2F clocks as the logic is common to both the cores. During first migration, the barrier signal from the cores reaches the control state machine which indicates that the cores have stalled fetch and are ready for migration. Once barrier signals of both cores are high, the "switch_caches" pulse is sent, which is latched by the cores. The F2F controller on seeing the barrier does the architectural register state transfer and sends the resume signal to the state machine. This resume is received by the CCD state machine and sent few cycles after the "switch_caches" pulse to ensure that the core cache mapping is complete. Based on the switch caches signal and external CCD configuration pin, cores get mapped to one of the caches.

This switch caches pulse generated at the F2F clock frequency crosses clock domains and is received by a double flip-flop synchronizer in the cores. During phase I testing, as the F2F clock was set slightly faster than the core clocks, this pulse transmitted from a fast to

slow clock domain is missed by the double flip-flop synchronizer causing wrong core-cache mapping. Thus, CCD will fail after an arbitrary number of migrations. Figure 2.2 shows one such instance where the incoming pulse is missed by the double flip-flop synchronizer.

Fix: The CCD control logic was organized by having all the clock domain crossing signals into a single module. The "switch_caches" signal logic which was earlier generating a pulse is now changed to a control signal that is high after the first migration and throughout the phase during which threads can access the remote cache. This way, the signal will not be missed by the cores and based on the CCD configuration pin the cores can choose to make the core access the remote caches. All signals that crosses clock domains are tested for various core-F2F clock combinations to ensure this bug do not appear in phase II. The test is shown in Chapter 4.

2.6 CTIQ Performance Bug:

Bug Description: After a branch is predicted, its predicted direction, prediction counters is packetized along with the instruction to fetch stage 2. In fetch stage 2 all control instructions are allocated CTIQ entries. During allocation, the branch's prediction counter is also written in the CTIQ. Later, when the branch executes, its next PC, control type, the actual direction taken is written into the entry allotted. Once execution is complete, the branch's prediction counters allocated earlier, are now updated with the actual direction taken and then written to the branch predictor's counter table. Whenever CTIQ is full the tail points to the head entry which is an in-flight branch. Incoming instructions are prevented from getting allocated this entry by checking the CTIQ full signal. But the prediction counters of the incoming

instruction got written into this head entry, thus causing wrong counter updates to the branch predictor.

Fix: The control vector input signal to the CTIQ shows how many control instructions are in the current fetch bundle that require a CTIQ entry. This control vector was previously used as the write enable to the CTIQ prediction counter RAM. This is now fixed by qualifying the input control vector with the CTIQ full signal. When CTIQ is full the write enable to the RAM is disabled. And only when CTIQ is not full, the input control vector directly acts as write enable to the counter RAM.

2.7 Netlist Simulation:

Netlist simulation helped identify three of the bugs from the phase I tapeout. In addition, the new features added were also tested using netlist simulation to ensure correct functionality. The design for phase II involved synthesizing two separate designs for the top and bottom tiers; hence the scripts were restructured to include sub-modules that come under each tier separately and were synthesized. Once the netlist is generated, back annotating Standard Delay Format (SDF) to the netlist is necessary to add timing information to the netlist for simulating it. The cell names are mapped to their corresponding timing information at the SDF and are back annotated. Since some of the sub modules are named identical across both the tiers, there were some conflicts during SDF back annotation. Though the top level modules are given unique names, different instances of the sub modules with common names across the tiers were getting annotated with the same timing information, hence SDF annotation failed. In order to provide a one-on-one relationship between the sub-modules and

its corresponding SDF, the names of all the modules across both the tiers were uniquified during synthesis.

After synthesis and SDF annotation, the testbench had to be modified to suit the netlist as some of the 2-dimensional arrays are now made one-dimensional and some signals are optimized out during synthesis. Hence, previous methods to dump traces for verification used in the simulation are no longer useful. New methods to dump traces are written for the synthesized netlist. In addition, the F2F controller design had a clock mux that had a huge fan out of driving all the TRF flip flops. Since a combinational logic is inserted in the path of the clock, synthesis does timing analysis on it which resulted in SDF with huge delays associated to the mux. Even though this can be fixed in the physical design process by instantiating explicit clock muxes to ensure no timing on these paths, it stalled the netlist simulation process. This was later fixed by setting zero delay on these paths using "set_annotated_delay" constraints during synthesis.

CHAPTER 3

NEW FEATURES

This chapter discuss the new features that are added to phase II design that did not make it into the phase I tapeout. The major changes made to the design are, the replacement of the T2 data cache with the new data cache, support for virtual memory to not flush caches and cache coherency support. These three changes are discussed in detail in this chapter and the verification effort of these new changes is discussed in Chapter 4.

3.1 Data Cache:

The bugs found with the T2 data cache and the reason for replacing it is discussed in detail in chapter 2. This section discusses the new data cache and the related efforts in integrating them to the two-core-stack design.

The new data cache is a 2KB, direct mapped cache. It supports hit under miss and it has one MSHR, hence capable of handling one outstanding miss at a time. The cache is a write-through cache, hence store requests on commit goes to the memory first before updating the cache. Only after the store acknowledgement returns to the cache, and if the store hits, it is updated in the cache. The major difference between the new data cache and T2 is the asynchronous read, synchronous write interface of the new data cache over synchronous read/write of the T2. The new data cache is designed for the tag and data banks to be synthesized to flip flops instead of RAMs as it was with T2.

In phase I, the pipeline has been heavily modified to support the synchronous read/write of the T2. The pipeline register in between the address generation and the load store unit of the pipeline known as the “Agen-Lsu” stage was moved inside the load store

unit. This was done to make the output of the Agen directly feed the data cache and the Agen-LSU pipeline register. So that, the 2 cycle latency of the data cache matches with the rest of the logic related to store queue accessing, load violation detection logic etc. These changes to the pipeline are now modified to make the output of the Agen-LSU feed both the dcache and store queue as the dcache output comes at the same cycle as the request. Similarly, other logic related to store queue match and writeback are adjusted for the new data cache.

In addition to the changes in the pipeline, the new data cache required arbitration between the load and store requests going to the memory. The T2 data cache arbitrated these requests internally hence making top level arbitration between instruction and data cache request for the single 8-bit core-to-mem bus easier. Now, there are 3 clients for the top level arbiter. They are I-cache, load, and store requests. The arbitration at the core side was made complicated as the packets should not only be arbitrated for a single bus but also should be differentiated from one another at the memory side depacketizer. This is achieved by using the packet ids of the packetizers.

Figure 3.1 shows the arbitration of the 3 packets through an arbiter to the core-to-mem bus. The packetizer ID which is extended to two bits for the three packetizers is embedded in the header of the packets. On the memory side, the core-to-mem bus is fed to three depacketizers. The depacketizer makes use of the ID to differentiate between the packets. Similar to the request side, there are 3 responses, the instruction, data packetizers arbitrate for the mem-to-core bus. The store acknowledgement reaches the core through a pulse synchronizer.

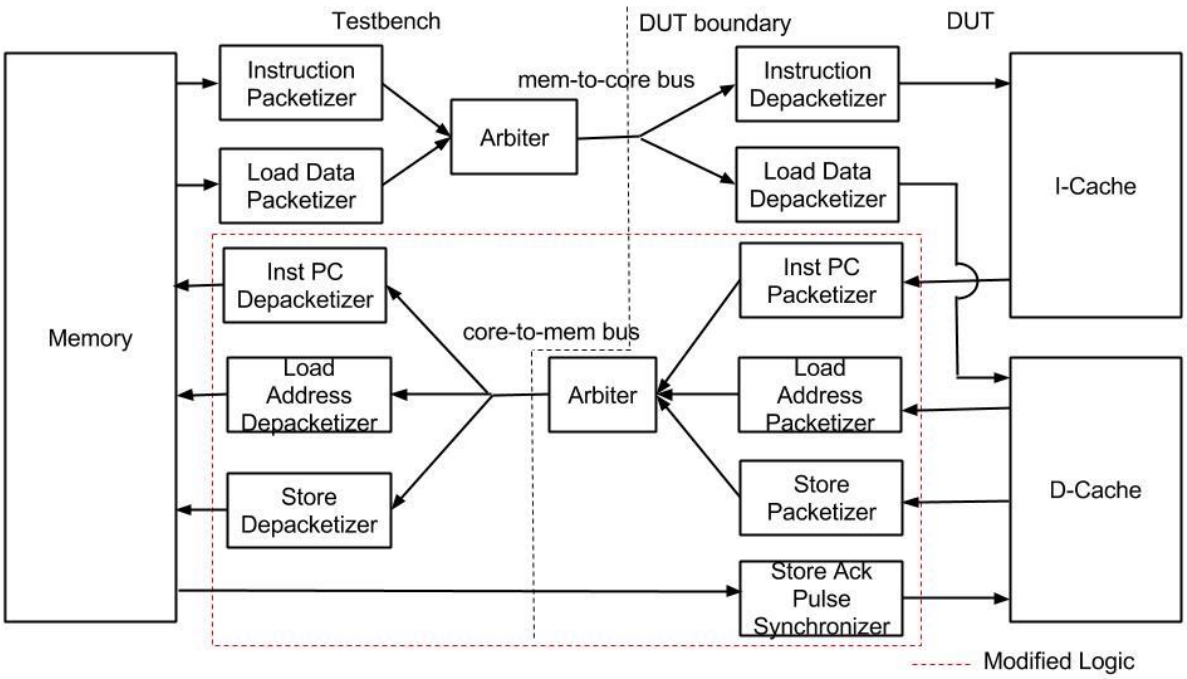

Figure 3.1 Top level organization of the L1 Caches-Memory Interface in H3

3.2 Virtual Memory:

The H3 heterogeneous multicore processor is a multiprogramming environment where at most two programs occupy the memory. One-half of the memory is allocated to a single program hence there is no memory sharing. In phase I design, the caches on the cores are virtually indexed and virtually tagged hence are always flushed during migrations as the other program's address space is different. An external signal monitors the execution of each program and maps their requests to the correct part of the memory. This causes performance degradation when we have CCD disabled. The degradation not only comes from the fact the

core has to start off with cold data caches when it migrates to the core again but also that it loses its instruction cache contents which if not flushed can save on crucial I-cache misses.

In order to not flush caches, the cores need to be aware of its physical memory space; hence a virtual to physical address translation is necessary. Since at most we can have only two programs running on the H3 processor, the virtual to physical memory mapping is a one-bit thread ID. This translation from virtual to physical address happens during cache access where the uppermost bit of the tag is embedded with the thread ID. Thus, the caches are made virtually indexed but physically tagged. This way there is no need to flush the caches when programs migrate and also the requests to the memory have the thread ID embedded in them, thus no external signal is necessary to map the block to the physical address location.

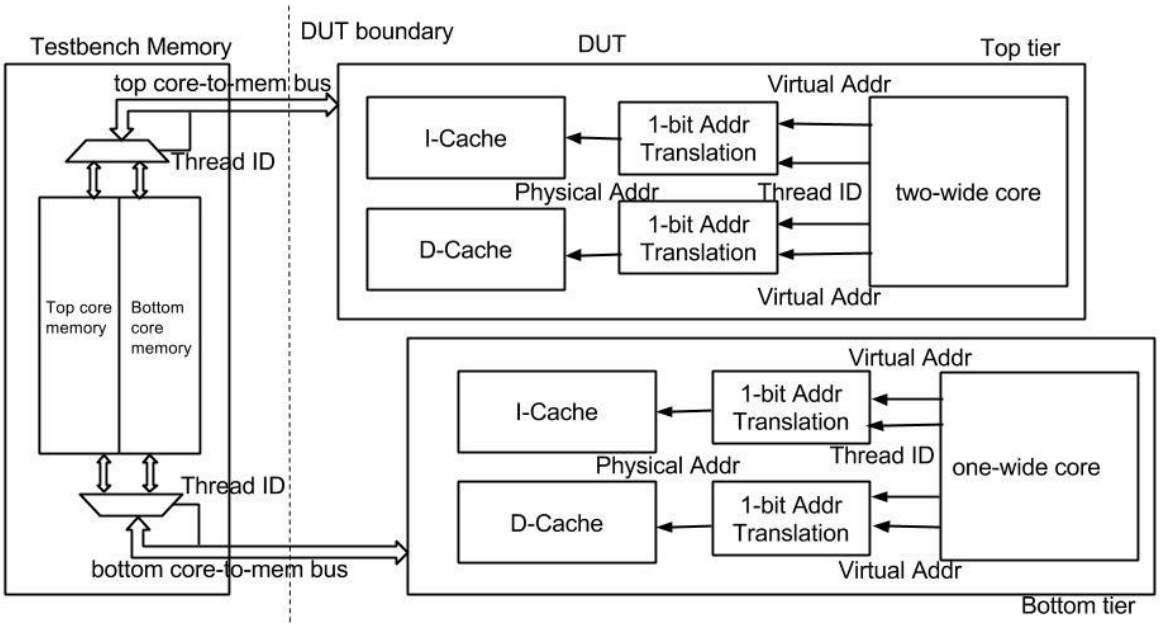

Figure 3.2 Virtual to physical address translation and memory interface in H3

Figure 3.2 shows the two cores and their virtual to physical address translation methodology. The caches use physical address for tags and thus based on the thread ID, the packet can be demuxed to one-half of the memory. The program can freely migrate from one core to the other without the need for flushing caches.

3.3 Cache Coherence:

Virtual memory ensures that when a new program migrates to a core it will not hit on the old program's cache contents thereby enabling not flushing caches. But not flushing caches can cause one additional problem. Say that program A running on core A loads memory block from address X, migrates to core B without flushing core A's cache and does a store to the same address X, now program A decides to migrate back to core A and when it loads the same memory block from address X it may hit on the stale data. This problem is solved by having cache coherence among the L1 caches. Since the caches are write through, a modified form of MSI coherency protocol with only two states (valid and invalid) is implemented. The testbench memory stores copies of the tag RAM of both the data caches. This way all memory blocks cached in each tier can be tracked and invalidations can be sent to the one tier when writes happened to the same memory block in the other tier.

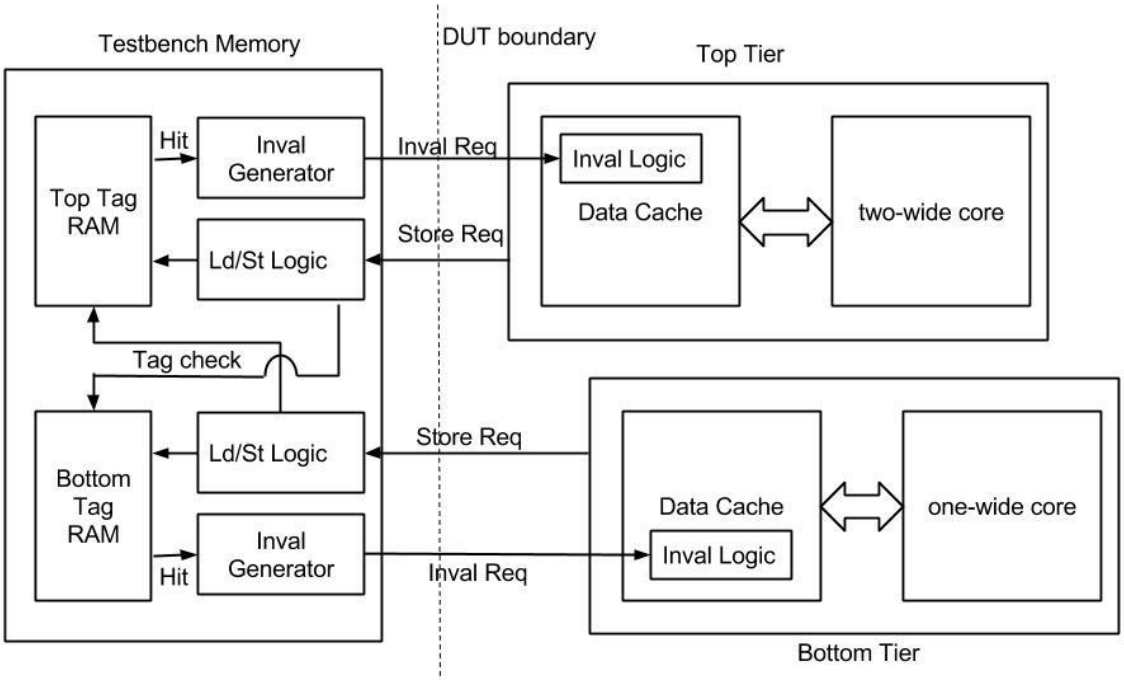

Figure 3.3 Cache Coherence Design in H3

The cache coherency model not only requires book-keeping at the memory level but also the data caches should be equipped with invalidation request detection logic. This logic is shown in the figure as the “Inval logic” inside the data cache. Whenever a data cache sends a store request to the memory, the store logic accepts the store request and then sends the store address to the other core’s tag RAM via the tag check interface. If it detects a hit, then the “Inval generator” uses the address to send an invalid request to the data cache and invalidates the tag in its own RAM. A control packet at the head of the invalidation packet differentiates invalidates from fill responses from the memory. This way the invalidate packets can use the same interface used by the load fill response.

CHAPTER 4

VERIFICATION

The verification effort related to phase II covers the following:

1. Stress tests for bugs found during phase I and its fixes
2. Tests for new features that are implemented in phase II
3. Performance tests for novel features related to FTM, CCD and not flushing caches
4. SPEC and microbenchmark results

All the tests used in this chapter are written and compiled using “Duct-Tape” an in-house developed assembly compiler. Each class of test cases are explained in detail with the test run, coverage counters and bugs found in the following sections.

4.1 Stress Test for Bugs and Fixes:

4.1.1 Load Wakeup Test:

The load wakeup bug is fixed by two modifications as explained in chapter 2. This test focuses on stressing the fixes and also in creating different dependence chains for the load. A loop, starting with the load instruction followed by dependents is created and the loop address iterates word by word. Every time a load misses the whole memory block which consists of four words is brought into the cache. Hence, there is a miss for every three hits in this loop. This test creates a combination of load hits and misses thus checking for early wakeup of dependents that are already in the issue queue. The second aspect of this test is varying the dependency chains of the load for making sure future incoming instructions that get their ready bit from PRF are not woken up until the load hits in the cache. This is tested by using the internal loop that iterates randomly based on the load value thus creating

random dependency chains. This will create a varying number of load-dependent instructions that reach the issue queue at different cycles. Every time the load replays and misses the register tag needs to use the P0 tag for not waking up dependents. Hence, both the PRF ready bits and register tags during load replays are verified. The test used and the counts regarding the test is shown in the figure and Table 4.1 respectively.

```
odd = 0x55555555
even = 0xaaaaaaaa
max = #512
cond = #0
laddr = @l
sum = #0

while (max) {
   lw lval, #0[laddr]
   bar = odd & lval
   while(cond)
   {
   sum = sum + lval
   cond = cond - #1
   }
   foo = even & lval
   result = bar | foo
   cond = lval & 0x7
   laddr = laddr + #4
   max = max - #1
 }
 wave
```

Figure 4.1 Load wakeup stress test

Table 4.1 Load wakeup stress test measurements

Instructions committed	10045
Control Instructions	2812
Load Instructions	512
Dcache misses	129
Pass/Bugs Found	Passed

4.1.2 Data Cache Test:

```
max = #511
  flag = #1
  while (flag) {
     flag = #0
     addr = @data
     ii = #0
     cond = ii < max
     while (cond) {
        lw val1, #0[addr]
        lw val2, #4[addr]

        comp = val2 < val1
        if (comp) {
           flag = #1
           sw val2, #0[addr]
           sw val1, #4[addr]
        }

        addr = addr + #4
        ii = ii + #1
        cond = ii < max
        delay = val2 & 0xf
        while(delay)
        {
          nop
          delay = delay - #1
        }
     }
  }
```

Figure 4.2: Data cache stress test

Table 4.2 Data Cache stress test measurements

Instructions committed	8980400
Load Instructions	524286
Store Instructions	261633
Data Cache miss	1152
Store Queue Hits	76089
Store Buffer Hits	107575
Pass/Bugs Found	Passed

The data cache stress test checks for all basic functionalities of the data cache including some special cases such as store-load forwarding from the store queue, store buffer forwarding. This test case is a bubble sort microbenchmark modified to test various forwarding scenarios. In general, bubble sort has two loads and two stores every loop thus is a data cache intensive microbenchmark. But, since the loads of each iteration happens immediately after the stores there are more chances of these loads hitting in the store queue than in the store buffer or data cache. A random delay loop based on the load values is used to delay the next iteration loads to create different forwarding cases. The test case used is shown in figure 4.2, the table shows the counts of number of loads and stores executed and out of which how many such loads hit in the store queue and the store buffer is also shown.

Store Buffer Size Bug:

Previous test checks only loads and stores that have the same sizes. While running load stores with different sizes a bug was found which is described in this section. The bug was found when we load a particular word, store a new word to the same address and load a byte from the same address. The load byte was getting the old value from the cache instead of getting the new store value. This bug occurred because the store-load forwarding mechanism in the fabscalar pipeline is conservative by not forwarding value for loads that do not match the store's size. Instead, these loads are considered violated and are made to replay. When the replay happens, the store reaches the store buffers. The store buffers do not check for partial matches and hence the load gets the stale value from the cache. This is now fixed by checking for partial matches at the store buffer and when such a match occurs the load is not allowed to hit in the cache. This makes the load to replay thus finally getting the correct value after the store completes. This new fix is also stress tested by the following microbenchmark in figure 4.3.

```
ii = #0
max = #512
addr = 0x10000000

while (max) {
   lw val, #0[addr]
   lw val1, #4[addr]

   sw val1, #0[addr]
   ii = val & 0xf
   while (ii)
   {
     ii = ii - #1
   }
   lb one, #0[addr]
   lb two, #1[addr]
   lb three, #2[addr]
   lb four, #3[addr]

   addr = addr + #4
   max = max - #1
 }
 wave
```

Figure 4.3 Store buffer size stress test

Table 4.3 Data cache load/store size stress test measurements

Instructions committed	19119
Load Instructions	3072
Store Instructions	512
Data Cache miss	129
Load Violations	23
Pass/Bugs Found	Passed

The test creates the same scenario that caused the bug and also the internal delay loop changes the time at which the load bytes reach the cache. Thus some of them may end up as load violations which are indicated in the table, other loads might reach the store buffer but will be made to replay and get the data after the store completes. The alternate case when there is a store to a byte and load to a word is also addressed using the same method of cancelling the load's hit was also verified.

4.1.3 False BTB hit Test:

The false BTB hit bug was fixed by blocking the CTIQ pushes in the case of false BTB hits. Both the false BTB detection logic and blocking of CTIQ is critical for this bug to be fixed. Both those aspects are tested in this stress test. The BTB is first presetted with valid entries with tags matching the PCs of the test to be run. The LFSR microbenchmark is used for testing the false BTB hit detection logic. This benchmark has a good mix of CTI and non-CTI instructions thus creating enough false BTB hits to verify whether the program doesn't fill up the CTIQ. Other genuine CTIQ instructions should commit properly thus checking whether the blocking is done only for false BTB hits and not genuine control instructions which are allocated CTIQ entries.

```
seed = 0xcafebeef
mask = 0x1
ii = 0x0
max = 0x003fffff
cond = ii < max

while (cond) {
   tap1 = seed & mask

   tap2 = seed >> #1
   tap2 = tap2 & mask

   tap22 = seed >> #21
   tap22 = tap22 & mask

   tap32 = seed >> #31
   tap32 = tap32 & mask

   new = tap1 ^ tap2
   new = new ^ tap22
   new = new ^ tap32

   new = new << #31
   seed = seed >> #1
   seed = seed | new

   ii = ii + #1
   cond = ii < max
}
addr = 0x00600000
sw seed, #0[addr] % final seed should be 0xae56deb8
wave
```

Figure 4.4 False BTB hit stress test

Table 4.4 False BTB hits stress test measurements

Instructions committed	67108927
Control Instructions	4194305
False BTB hits	62914589
Pass/Bugs Found	Passed

The program executes close to 67 million instructions out of which 63 million instructions are non CTI instructions that wrongly hit in the BTB, hence are tagged as false BTB hits. Other control instructions come to around 4 million. The test passed generating the correct final seed value.

4.1.4 CTIQ full Test:

The CTIQ fill bug was related to fetching past the migrate instruction where branches get allocated CTIQ entries that are never cleared. The CTIQ was resetted on a resume when a thread migrates to the core again to reclaim the lost entries. This fix is stress tested here by migrating more often to the core where CTIQ entries are lost to make sure the lost entries are regained every time a migrate happens. This is done by using a tight loop with local migrations that ping pong the thread from one core to the other multiple times. This loop also contains the loop handling branches that will be fetched past the migrate instruction every time the thread runs on the two-wide core due to the higher fetch rate of the core. Thus every time this scenario occurs, the resume signal should be capable of clearing the lost entries thus prevent the CTIQ from getting filled up. The stress test used is shown below. On every iteration, the register values are moved to the TRF using the m2t_trf instruction and a migrate happens. After migration, the core uses the mf_trf instructions to move the TRF values back to its architectural register and then resumes. Since the loop body is small, there are chances of instructions getting fetched past the migrate causing the wrong CTIQ entry allocation. Now after the migration is complete the cores resume execution using a resume signal. The CTIQ is cleared at that point to recover lost CTIQ entries.

```
data = #1
$r5 = #1
$r6 = #1
$r7 = #1
$r8 = #1
ii = #0
cond = ii < #10000
while (cond){
   m2t_trf $r1, $r2
   m2t_trf $r3, $r4
   m2t_trf $r5, $r6
   m2t_trf $r7, $r8

   migrate

   mf_trf $t1
   mf_trf $t2
   mf_trf $t3
   mf_trf $t4
   mf_trf $t5
   mf_trf $t6
   mf_trf $t7
   mf_trf $t8

   data = data + $r5
   data = data + $r6
   data = data + $r7
   data = data + $r8
   ii = ii + #1
   cond = ii < #10000
}
```

Figure 4.5 CTIQ full Stress Test

Table 4.5 CTIQ full stress test measurements

Instructions committed	210079
Control Instructions	20002
Migrations	10000
Pass/Bugs Found	Passed

4.1.5 CCD Test:

The CCD control signals bug was created when CCD is enabled and the core-F2F clocks are asynchronous to each other. The “switch_caches” signal responsible for CCD was crossing from a fast to slow clock domain causing the problem. This bug was fixed by revamping the CCD control logic and holding the “switch_caches” signal high throughout the phase where the core uses the remote cache to make sure the core-cache mapping is established properly. This aspect was stress tested by using the same test for the CTIQ stress test downsized to 100 migrations as shown in the figure 4.6, but with CCD enabled. Thus all cache requests should be to the same core in which execution started. And, the other clock domain crossing signals like the barrier and resume signals are also stress tested here by varying the core-F2F clock relations. The test was done for the following clock relations between the core-F2F modules as shown in table 4.6 and the test passed generating the correct data value and all core-to-mem requests were observed to be from the top core where the execution has started.

```
data = #1
$r5 = #1
$r6 = #1
$r7 = #1
$r8 = #1
ii = #0
cond = ii < #100
while (cond){
   m2t_trf $r1, $r2
   m2t_trf $r3, $r4
   m2t_trf $r5, $r6
   m2t_trf $r7, $r8

   migrate

   mf_trf $t1
   mf_trf $t2
   mf_trf $t3
   mf_trf $t4
   mf_trf $t5
   mf_trf $t6
   mf_trf $t7
   mf_trf $t8

   data = data + $r5
   data = data + $r6
   data = data + $r7
   data = data + $r8
   ii = ii + #1
   cond = ii < #100
}
```

Figure 4.6 CCD Stress Test

Table 4.6 CCD stress test clock relations

Core clock period (ns)	F2F clock period (ns)	Pass/Fail
10	10	Pass
12	10	Pass
14	10	Pass
15	10	Pass
20	10	Pass
25	10	Pass
30	10	Pass
10	12	Pass
10	14	Pass
10	15	Pass
10	20	Pass
10	25	Pass
10	30	Pass

4.2 Stress Test for New Features:

The new features implemented for phase II were also stress tested. The data cache which is part of both bug fixes and new features is already covered in the previous section. This section covers the other two major features and also few other features like the performance counters and the new wave instruction are also tested here.

4.2.1 Cache coherence:

The bubble sort microbenchmark which by itself a memory intensive test has been modified to stress invalidation logic. The benchmark does two loads and two stores every loop. The loop is modified to migrate after each iteration thus the loads from next iteration should get the correct value from the stores instead of the stale value in its own cache. This is done by invalidating the old value of the loads before the thread migrates to the other core. This occur every iteration as the thread migrates after performing the two loads and stores to the other core.

```
max = #511
flag = #1
while (flag) {
   flag = #0
   addr = @data
   ii = #0
   cond = ii < max
   while (cond) {
      lw val1, #0[addr]
      lw val2, #4[addr]

      comp = val2 < val1

      if (comp) {
         flag = #1
         sw val2, #0[addr]
         sw val1, #4[addr]
      }

     m2t_trf $r1, $r2
     m2t_trf $r3, $r4
     m2t_trf $r5, $r6
     m2t_trf $r7, $r8

     migrate

     mf_trf $t1
     mf_trf $t2
     mf_trf $t3
     mf_trf $t4
     mf_trf $t5
     mf_trf $t6
     mf_trf $t7
     mf_trf $t8
     mf_trf $t9
     mf_trf $t10

      addr = addr + #4
      ii = ii + #1
      cond = ii < max
   }
}
```

Figure 4.7 Invalidation logic stress test

Table 4.7 Invalidations Test counters

Top Core Load Instructions	262654
Top Core Store Instructions	131073
Top Core Data Cache miss	209965
Top Core Invalidations	81279
Bottom Core Load Instructions	261632
Bottom Core Store Instructions	130560
Bottom Core Data Cache miss	81921
Bottom Core Invalidations	81283
Pass/Bugs Found	Passed

This test initially revealed an interesting bug related to lost invalidation requests from the memory. As we discussed in one of the previous sections, the resume signal after migration resets most of the structures in order to restore the destination core to a pristine state before resuming execution. This resume signal was also resetting the depacketizer modules that are responsible for receiving the load fill/invalidations from the memory. This resulted in lost invalidations which caused the loads to hit on the stale data in the caches. This can also cause problems when CCD is enabled as load fills can be dropped by the depacketizer causing miss handling structures inside the data cache that are waiting for the response to create deadlock scenarios. Hence, this was fixed by not resetting the depacketizer when flush is disabled or CCD is enabled.

Thread Synchronization Issue: The lost invalidations bug showed us the significance of invalidations reaching the core cache before the first cache access. The threads are not made

to wait for the invalidations before migration due to performance reasons; the migrate can happen once the final store has been acknowledged by the memory. This can create problems when the invalidations are delayed and fails to reach the core before the first cache access; hence it creates a thread synchronization problem as the threads need to synchronize with its own requests reaching the other core before migrations. In case of global migrations, this risk is reduced as the migration penalty which comes mainly from moving the architectural register state that is all 34 registers, and the PC to the other core is high which gives plenty of time for the invalidations to reach the other core before the thread migrates. In case of local migrations which are software triggered migrations, the migration latency can be less. As the software knows the number of registers to be moved and hence can reduce the time taken for architectural register state transfer. This study was done to identify the least amount of registers needed to be transferred for which the invalidations will reach the destination core safely before the thread does. The test used for this study is shown in figure 4.8.

The number of registers to be moved was brought down iteratively observing the time interval between the invalidations reaching the other core and the first load access. Another source of latency of migration apart from the architectural register state transfer comes from the pipeline refill. It is the time taken by the destination core to fill its pipeline from fetch through memory stage where the cache access occurs. It was observed that this latency delays the load access by significant number of cycles. After iteratively trimming down the number of registers moved, finally only the address for the loads and stores was transferred from one core to the other. This 1-register architectural register state transfer passed with a 2-3 cycle interval between the invalidation packets and load access in the destination core. In

this study the memory is assumed to be clocked at the same rate as the cores. If this clock relation is changed, by making the memory slower the test fails, thus showing the breaking point up to which we can be assured of the invalidations reaching the cores before the loads.

```
laddr = @l          % $r2 = 0x10000000
saddr = 0x20000000
lw lval, #0[laddr]  % $r1 = 0x0000000a

cond =  #1
while (cond){

  lval = lval + #1
  cond = #1

  sw lval, #0[saddr]  % $r1 = 0x0000000b
  m1t_trf $r1

  migrate

  mf_trf $t1

  lw lval, #0[saddr]  % $r1 = 0x0000000b
  cond = #1
  }
}
```

Figure 4.8 Invalidation request vs. FTM latency

4.2.2 Virtual Memory:

Virtual memory implementation can be tested by running two threads simultaneously on the two-core-stack without flushing the caches. This creates scenarios where the memory blocks from both the threads co-exist in the caches and the translation methodology should ensure that one thread does not hit on the other thread's memory block. This theory is tested by running two SPEC benchmarks bzip and vortex on the two-core-stack simultaneously using

global migrations. The threads migrate after a fixed time interval specified by an external counter, which is now set at 16K cycles for this test, without flushing its caches and should successfully complete thus ensuring that the memory space for both the threads has been handled properly. The run successfully completed committing 400K instructions from bzip and 300K instructions in vortex. Table 4.8 shows the measurements from the run.

Table 4.8 Virtual Memory Test counters

Top core committed instructions	409503
Top core control instructions	84210
Top core ld instructions	112766
Top core st instructions	33245
Top core dc miss	6641
Top core ic miss	6148
Top core invalidations	508
Bottom core committed instructions	366180
Bottom core control instructions	74322
Bottom core ld instructions	97338
Bottom core st instructions	31105
Bottom core dc miss	5647
Bottom core ic miss	5832
Bottom core invalidations	527

4.2.3 Configuration Pins:

There are two external configuration pins to disable/enable CCD and flush-caches. The CCD pin controls the core-cache mapping, by acting as the select pin for the CCD muxes. The flush-caches pin controls resetting the caches on every migrate. The CCD configuration pin is given a higher priority; the flush-caches pin only takes effect when CCD is disabled. For example: when CCD is enabled, the caches should not be resetted irrespective of the flush caches pin. Even though they are separate pins they both can be checked for correctness by a single entity i.e. invalidations. The flush caches pin when enabled should always flush caches; hence there should not be any tag matches in the other core when two threads are migrating. Similarly when CCD is enabled irrespective of the flush caches pin, there should not be any invalidation happening as the cores always use the same cache and there cannot be any tag match in the other cache. Both these aspects are tested here by running bzip and vortex SPEC benchmarks on the two cores.

Table 4.9 flush caches enabled test counters

Top core committed instructions	431907
Top core control instructions	88452
Top core ld instructions	117125
Top core st instructions	34912
Top core invalidations	0
Bottom core committed instructions	382028
Bottom core control instructions	76987
Bottom core ld instructions	99954
Bottom core st instructions	33397
Bottom core invalidations	0

Table 4.10 CCD enabled Test counters

Top core committed instructions	410048
Top core control instructions	84273
Top core ld instructions	112340
Top core st instructions	33275
Top core invalidations	0
Bottom core committed instructions	380070
Bottom core control instructions	77107
Bottom core ld instructions	100796
Bottom core st instructions	32827
Bottom core invalidations	0

The table 4.9 lists measurements when the two threads are run with CCD disabled and flush caches enabled. In this case both the caches are flushed on a migrate by both the threads hence there are zero invalidations generated. Similarly table 4.10 lists measurements for CCD enabled, which also shows zero invalidations as the same caches are accessed by the threads throughout the run. This ensures correctness of the configuration pins.

4.2.4 Performance Counters:

Set of 32-bit counters monitor performance counts in both the tiers. These counters track critical measurements from the chip that can help to determine migrations decisions and also to track the performance of each thread on a core. These counters can be streamed out serially via a 1-bit signal to the FPGA on a sample request. Once a request is reached, the current values of the counters are pushed into an FIFO and are streamed out as 1-bit values to the FPGA. These counters take around 500 cycles to be read. This counter was tested by running an array-sum microbenchmark with known measurements and after execution the performance counter values were verified with the counts expected and they matched. The table 4.11 shows the measurements of the microbenchmark. The test is shown in figure 4.9.

Table 4.11 Performance Counter measurements

Instructions committed	3152
Control Instructions	514
Load Instructions	512
Store Instructions	1
Mispredictions	3
Data Cache miss	129
Instruction Cache misses	25
Load Violations	0
Issue Queue full	0
Active List full	3745
Load queue full	0
Store Queue full	0
Instruction buffer full	3222
Pass/Bugs Found	Passed

4.2.5 Wave Instruction:

The wave is a new instruction that is used by the chip to interact with the outside world. The instruction toggles a 1-bit signal that goes directly out to the FPGA. A simple microbenchmark that migrates the toggle instruction between both the cores was done. The test result waveform is shown in Figure 4.10.

```
total = #0
    max = #512

    addr = @data
    ii = #0
    cond = ii < max
    while (cond) {
        lw val, #0[addr]
        total = total + val
        addr = addr + #4
        ii = ii + #1
        cond = ii < max
    }

addr = 0x00600000
sw total, #0[addr] % total should be 0x563b105d
```

Figure 4.9 Performance counters Test

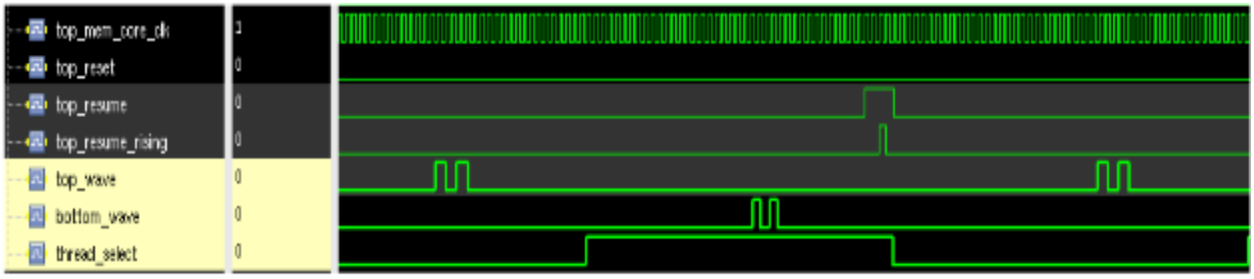

Figure 4.10 Waveform for the wave test

4.3 Performance Tests:

Performance tests include tests for all the novel features of H3 phase II. The tests used in this section were all done in a pin-accurate fashion, as the goal is to redo these tests in the chip in-order to measure performance results.

4.3.1 Fast Thread Migration Latency

The feature to perform thread migrations with reduced latency is one of the novel features of H3. This feature is measured here with the help of a test case and its results are discussed. The test makes use of the wave instruction as a method of communicating the completion of a migrate. The test case is a simple while loop that has the wave instruction in it. The thread is made to migrate from one core to the other for every migration. The loop iterates 100 times; hence 100 wave instructions are counted to measure the completion of 100 migrations. The loop doesn't include any additional arithmetic instructions, load/store instructions as the goal is to measure FTM latency apart from other execution latencies. The test used for the measurement is shown in figure 4.11. The sample test shows a 2 TRF-register move loop.

The test was run with flush caches disabled; hence the thread might suffer instruction cache misses the first two migrations. After the second migration, there are no misses in the instruction cache. For this study, the FTM latency for 1 migration is identified by taking an average among 100 migrations to compensate for the initial latency increase due to instruction cache misses. The PC instruction is moved by default hence is not part of the source code. In a real-time system, the number of registers to be moved will be decided by the compiler during runtime. Here for experimentation, the test is iteratively done by increasing the number of registers to be transferred during each migration.

```
$pc = 0x4000

mem (0x4000) {
cond: $r1
ii:   $r2

ii = #0
cond = ii < #100
while (cond){
 wave
 m2t_trf $r1 $r2
 migrate
 mf_trf $t1
 mf_trf $t2

 ii = ii + #1
  cond = ii < #100
  }
  wave
}
```

Figure 4.11 Test for FTM latency measurement

Latency to move only the PC, up till move of the maximum registers PC+34 architectural registers is shown in the figure 4.12. In the graph, zero-register-move indicates that only the PC is moved. The rest of the counts shown are PC+n register moves.

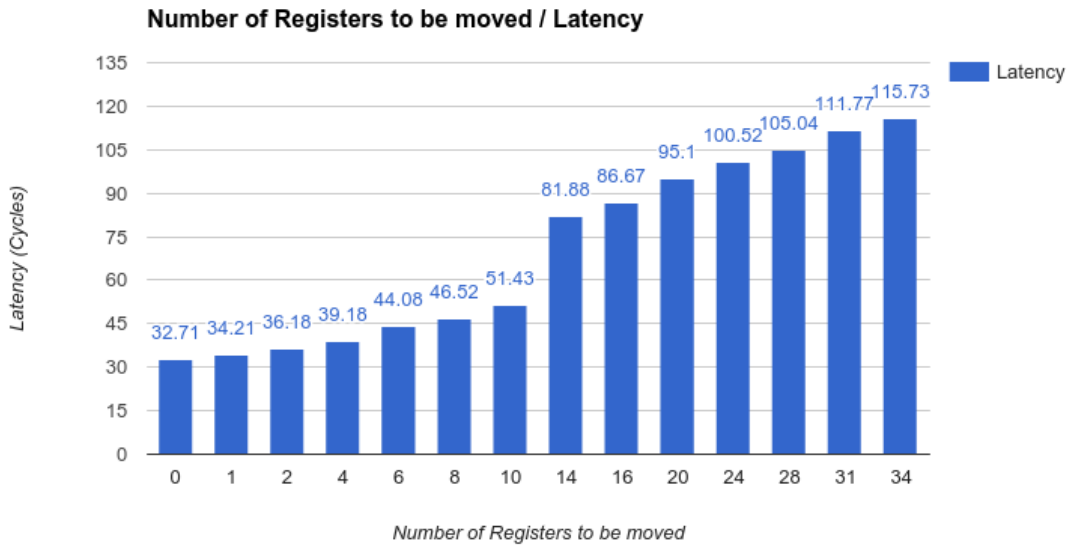

Figure 4.12 FTM latency measurements

As it is shown in the graph, the least amount of state transfer that involves only PC transfer takes around 33 cycles. While the maximum possible transfer PC+34 registers takes around 116 cycles. The latency mainly comes from filling the pipeline stages after a migrate, and executing the move-to and move-from instructions. The swap takes one cycle irrespective of the number of registers to be moved as it uses the wider face-to-face buses.

4.3.2 CCD Performance:

The benefit of CCD comes from the ability to access the same cache after migrations. CCD effectively gives the same number of cache misses as when we always execute in OOO processor without migrations. Although, there can be a few additional misses in case of CCD because the move from, move to TRF instructions might miss in the cache every time during migrate. The graph in figure 4.13 compares the data cache miss rate for the SPEC

benchmarks across CCD, Flush, no-flush. The benchmarks were run for 10M instruction commits with an external migration interrupt every 16K cycles causing the thread to migrate. Miss rate is calculated from the total number of load accesses and data cache misses in this 10M window. Similarly the graph in figure 4.14 compares the instruction cache misses across the three configurations. All threads start off at the bottom core; hence with CCD enable they access their bottom core throughout execution.

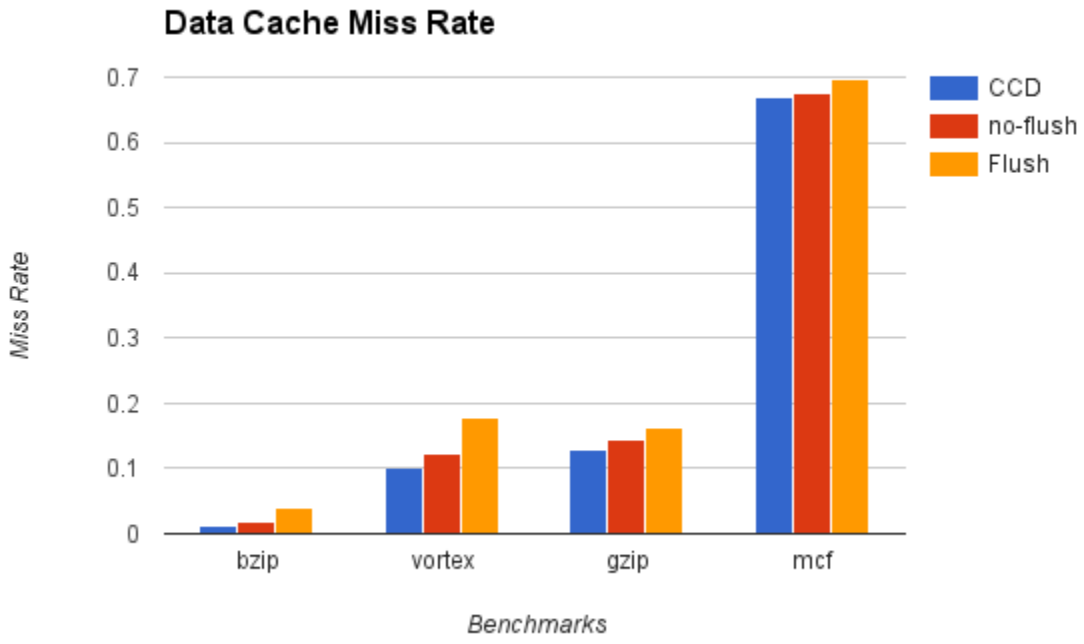

Figure 4.13 Dcache Miss Rate across benchmarks

As expected miss rates with CCD is significantly lower than the other two configurations. Bzip showed the least amount of misses among the 4 benchmarks with only around 3.9% of misses even for the always flush configuration. With CCD misses came down to 1.3%.

Vortex and gzip showed around 17% misses for the always flush, with CCD it came down to 10% and 12% respectively. Mcf showed the most misses even for the configuration with CCD enable. This is because of the random memory access patterns in mcf in addition with the direct mapped data cache that could result in a lot of conflict misses. Hence the benefit we get from CCD is minimal. It's around 66% with CCD enable and around 70% with always flush.

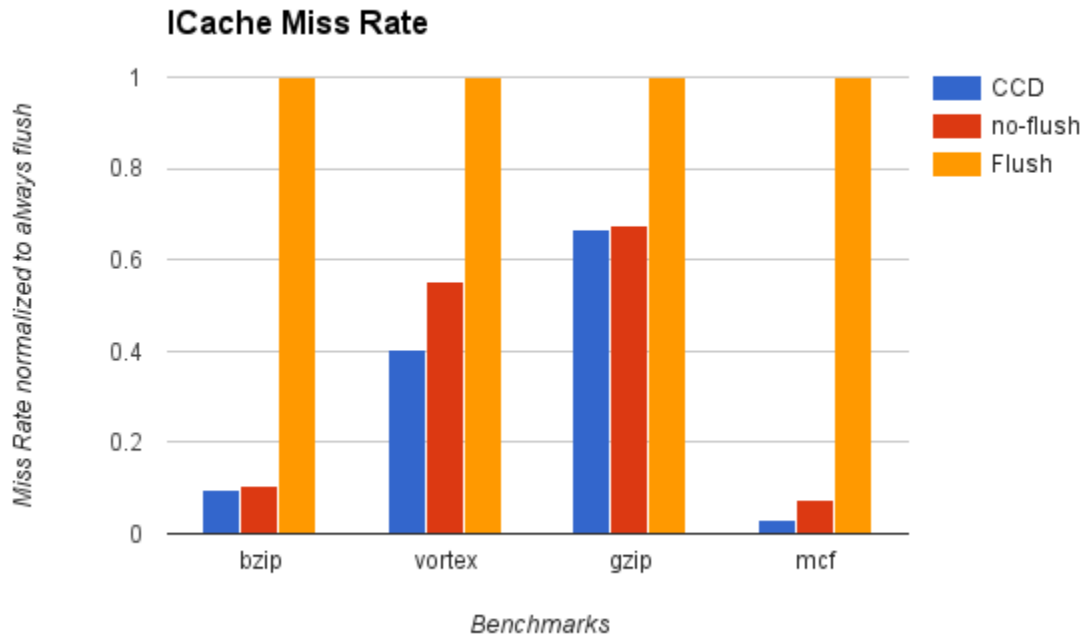

Figure 4.14 I-Cache Miss Rate across benchmarks

The graph 4.14 shows the instruction cache miss rate of CCD and no-flush normalized to always flush. The saving up of I-cache misses is a significant feature of CCD. Since the initial I-cache misses can effectively reduce the advantage of migrations in the case of always

flush mode, the ability to access warm instruction caches can boost performance with CCD enabled. CCD always reduces I-cache misses by more than 60% except for benchmarks like gzip where it was 66% the miss rate as always-flush.

4.3.2 Not-flushing Caches Performance:

The graphs 4.13 shows the data cache misses without flush and it falls in-between CCD and always flush for all the benchmarks. This is because of two reasons - the initial few migrations during which no-flush does not provide any benefits as both the caches are cold. Even after the initial migrations, the caches can still suffer from misses due to invalidations. There are a significant number of invalidations happening due to the large number of store instructions being executed. This is seen more often in gzip where the miss rate with no-flush is 14.3% with always flush at 16.2%, than benchmarks like vortex and bzip where no-flush 12%, 1.8% performs almost as same as CCD which has 10% and 1.3% miss rates respectively.

The I-cache misses shown in 4.14 shows more positive trend with miss rates of no-flush closer to CCD than always-flush. This is because there cannot be any invalidation triggered misses in case of I-cache, and also since these programs tend to execute the same loop body and can hit more often in I-cache even after executing few thousands of instructions in the other core. The miss rate is always around 10% of the I-cache miss rate as it is with CCD.

4.4 Benchmark Results:

This section shows the IPC results of the H3 processor for 4 SPEC benchmarks and a couple of other microbenchmarks. All these benchmarks were simulated for around 10M instruction

commits before measuring their IPC value. The graph 4.15 shows the IPC results for the benchmarks across the three configurations.

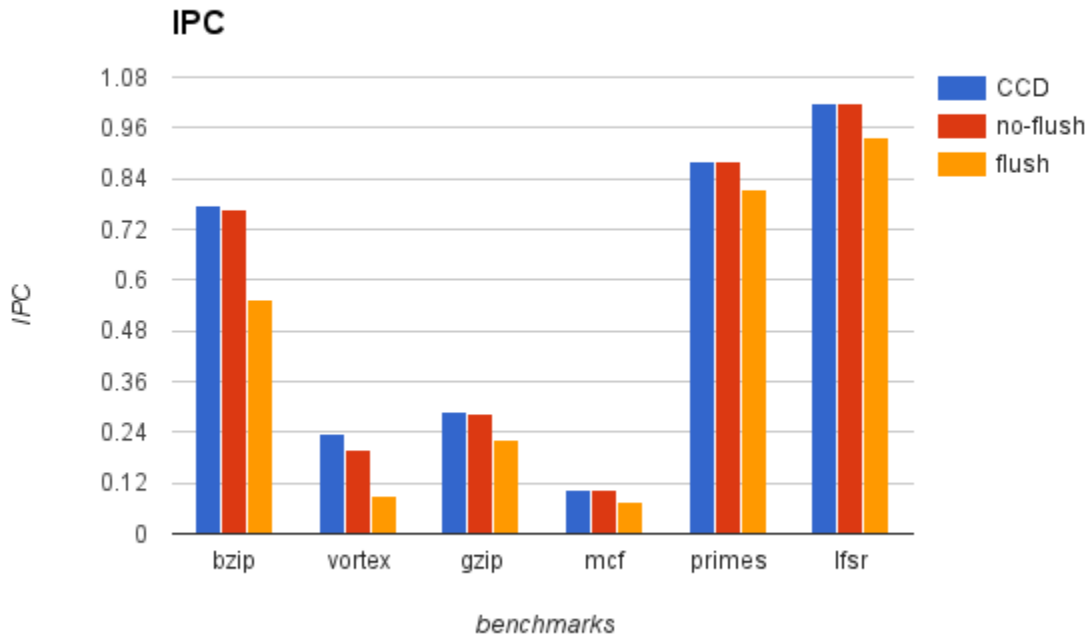

Figure 4.15 IPC across benchmarks

The IPC results for the three configurations are compared in the graph. CCD does the best across all the benchmarks irrespective of the number of load/store instructions in each benchmark. This is because of the low instruction and data cache misses in case of CCD. No-flush slightly trails behind CCD which can be attributed to the higher data cache miss rates compared to CCD. Although the IPC of no-flush is closer to CCD, it is to be noted that there is only one thread running, hence there are only invalidations triggered misses. If there are two threads running then the IPC of the no-flush is expected to come down as the other

thread will replace useful memory blocks from the cache. The no-flush benefit mainly comes from the low instruction cache miss rates which can be evidently seen from the IPC results of the microbenchmarks prime and lfsr. Both these microbenchmarks have no loads/stores and have a program working code that can easily fit in the cache. Hence, for these workloads no-flush performs as good as CCD. Always-flush tends to do perform poor even for the microbenchmarks, due to the constant flushing of caches causing the small working set to be loaded into the instruction cache every time the thread migrates.

CHAPTER 5

SUMMARY

The implementation details of the fixes, bugs caused them are documented in a detailed manner hence can be used for future reference during the bring up phase of the phase II chip. New features were implemented in the design and their functionalities were stress tested. Performance tests showed the expected outcome from both the already implemented novel features like CCD and FTM and also from the new features like virtual memory and cache coherence. New data cache has been successfully integrated and stress tested. Finally the IPC results of the SPEC benchmarks is shown to verify the working of the two-core-stack with an external interrupt based migration under all three configurations. The obtained IPC results match the expected trend; hence this design can now be improved upon further to implement other novel features after the phase II tapeout.

REFERENCES

[1] E. Rotenberg, B. H. Dwiel, E. Forbes, Z. Zhang, R. Widialaksono, R. Basu Roy Chowdhury, N. Tshibangu, S. Lipa, W. R. Davis, and P. D. Franzon. Rationale for a 3D Heterogeneous Multi-core Processor. Proceedings of the *31st IEEE International Conference on Computer Design (ICCD-31)*, pp. 154-168, October 2013.

[2] E. Forbes, R. Chowdhury, B. Dwiel, A. Kannepalli, V. Srinivasan, Z. Zhang, R. Widialaksono, T. Belanger, S. Lipa, E. Rotenberg, W.R. Davis, P.D. Franzon. *Experiences with Two FabScalar-Based Chips. WARP 2015, 6th Workshop on Architectural Research Prototyping.*

[3] Intel H3 Project: Thread Migration Design Review Presentation, Elliott Forbes

[4] SolvNet Synopsys: https://solvnet.synopsys.com/