Master’s Thesis

Design and Implementation of an Efficient and Realistic Cooperative Core Architecture

Supervised by
Associate Professor Kenji Kise

Presented by
Tomoyuki Nagatsuka

Department of Computer Science
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

January 2013
Abstract

The number of cores on a chip increases in recent years. In order to improve performance in the many-core era, we should utilize all cores on a chip effectively. However, it is difficult to parallelize programs so as to utilize all cores, and single-thread regions remain as bottlenecks.

To solve these bottlenecks, cooperative core architectures is proposed. Cooperative core architecture can accelerate single-thread execution by fusing some narrow-issue cores into a wide-issue core. It can also balance single-thread performance and multi-thread performance by fusion and split during execution.

We have proposed CoreSymphony architecture that is one of the cooperative core architectures. In this paper, we design and implement efficient and realistic CoreSymphony and run it on FPGA. Then, we clarify the performance and the hardware budget of CoreSymphony.
# Table of Contents

**Abstract**

1. **Table of Contents**

2. **List of Figures**

3. **List of Tables**

4. **List of Listings**

1. **Introduction**
   1.1 Introduction to CoreSymphony
   1.2 Contributions
   1.3 Organizations

2. **Cooperative Core Architecture**
   2.1 CoreSymphony
   2.2 Other Cooperative Core Architectures
      2.2.1 Core Fusion
      2.2.2 Anaphase
      2.2.3 Composable Lightweight Processors
      2.2.4 Voltron
      2.2.5 Fg-STP
      2.2.6 CoreGenesis
      2.2.7 MorphCore

3. **Baseline Processor**
   3.1 Instruction Fetch, Instruction Decode
   3.2 Register Rename
6 Verification

6.1 Verification on Simulation

6.1.1 Verification Method

6.1.2 Generating Answer Traces by Software-based Simulator

6.1.3 Generating Log Files by Icarus Verilog

6.1.4 Converting Log File to PC/RSLT trace

6.1.5 Taking Difference between Answer Traces and Converted Log Files

6.2 Verification on FPGA

7 Evaluation

7.1 Evaluation on Software-based Simulator

7.1.1 Evaluation Environment

7.1.2 Performance Improvement by Fusion

7.1.3 Comparison of Steering Algorithms

7.1.4 Broadcast Width Sensitivity

7.1.5 Remote Write back Width Sensitivity

7.1.6 Re-order Buffer Distribution Overhead

7.1.7 In-order State Distribution Overhead

7.2 Evaluation on Xilinx ISE

7.2.1 Evaluation Environment

7.2.2 Register Map Table

7.2.3 Physical Register File/Logical Register File

7.2.4 Instruction Window

7.2.5 Re-order Buffer Distribution

7.2.6 Total Resources

7.2.7 Maximum Frequency

8 Conclusion

Acknowledgements

Bibliography

Publications
List of Figures

2.1 Conceptual diagram of 2-core fusion CoreSymphony ........................................ 4
2.2 Conceptual diagram of Core Fusion ................................................................. 5
2.3 Block diagram of Anaphase .................................................................................. 6
2.4 Conceptual diagram of Composable Lightweight Processors ............................. 6
2.5 Block diagram of Voltron .................................................................................... 7
2.6 Block diagram of Fg-STP .................................................................................... 7
2.7 Conceptual diagram of CoreGenesis .................................................................. 8
2.8 Conceptual diagram of MorphCore .................................................................... 9

5.1 Cooperative instruction fetch mechanism with Local Instruction Cache ............. 21
5.2 Case 1 of Local I-$ line filling ............................................................................ 22
5.3 Case 2 of Local I-$ line filling ............................................................................ 23
5.4 Steering results of three steering algorithms ...................................................... 25
5.5 Steered-leaf vector determination logic ............................................................. 27
5.6 Steering vector determination logic ................................................................... 28
5.7 Cooperative renaming with GRMT ...................................................................... 29
5.8 ROB distribution ............................................................................................... 34
5.9 In-order state distribution ................................................................................... 36
5.10 Block diagram of 1-entry, 1-operand of instruction window of CoreSymphony ... 37

5.11 nRTW Memory by Replication ......................................................................... 38
5.12 nRMW Memory with LVT ................................................................................ 39
5.13 Remote Write back Queue ................................................................................. 40
5.14 Block diagram of CoreSymphony architecture ................................................. 41

7.1 Performance improvement obtained by fusion ................................................... 50
7.2 Comparison of three steering algorithms ........................................................... 51
7.3 Broadcast Width (BW) sensitivity ....................................................................... 51
7.4 Remote write back Width (RW) sensitivity ........................................................ 51
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.5</td>
<td>ROB distribution overhead</td>
<td>52</td>
</tr>
<tr>
<td>7.6</td>
<td>In-order state distribution overhead</td>
<td>52</td>
</tr>
<tr>
<td>7.7</td>
<td>FPGA resources of RMT</td>
<td>58</td>
</tr>
<tr>
<td>7.8</td>
<td>Detailed resources of RMT of CoreSymphony</td>
<td>58</td>
</tr>
<tr>
<td>7.9</td>
<td>FPGA resources of PRF and LRF</td>
<td>59</td>
</tr>
<tr>
<td>7.10</td>
<td>FPGA resources of instruction window</td>
<td>59</td>
</tr>
<tr>
<td>7.11</td>
<td>FPGA resources of ROB</td>
<td>60</td>
</tr>
<tr>
<td>7.12</td>
<td>Detailed resources of ROB of CoreSymphony</td>
<td>60</td>
</tr>
<tr>
<td>7.13</td>
<td>FPGA resources of a core</td>
<td>61</td>
</tr>
<tr>
<td>7.14</td>
<td>Detailed resources of a core of CoreSymphony</td>
<td>61</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Hardware complexity of CoreSymphony ............................................. 42

7.1 Baseline processor configuration ...................................................... 50

7.2 Number of entries of GROB, LCB, and RCB against number of entries of LROB .................................................... 55
## List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Program for test on simulation</td>
<td>44</td>
</tr>
<tr>
<td>6.2</td>
<td>Answer trace of program counter</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>Answer trace of logical destination register and result</td>
<td>44</td>
</tr>
<tr>
<td>6.4</td>
<td>Execution Log File</td>
<td>45</td>
</tr>
<tr>
<td>6.5</td>
<td>Shell script to convert log file to traces</td>
<td>46</td>
</tr>
<tr>
<td>6.6</td>
<td>Program for test on FPGA</td>
<td>47</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction to CoreSymphony

Performance on a single core reaches its limit, and chip multiprocessors (CMPS) that have multiple cores on the same chip have become a mainstream. Current CMPS with between two and eight cores achieve high performance by executing a number of threads in parallel. The number of cores on a chip is expected to increase towards many-core era with the development of semiconductor technology [1].

On the multi-core/many-core era, Amdahl’s Law [2] indicates that performance improvement of multi-threaded programs on CMP is restricted by execution time of single-threaded region. We can reduce execution time of multi-threaded region by parallelizing programs into a number of threads. However, it is difficult to eliminate the single-threaded region on complicated and large programs like SPEC CPU, and the execution time of single-threaded region remains as a bottleneck. Therefore, instead of reduction of single-threaded region, we should improve the single-thread performance.

In order to improve single-thread performance, heterogeneous multi-core architectures [3, 4] have been proposed. These architectures have cores of various sizes on a single chip. Large (fast) cores execute single-threaded regions, and small (slow) cores execute multi-threaded regions. The disadvantage of heterogeneous multi-core architectures is that the number of large cores and the number of small cores is determined at the processor design time. Parallelism varies depending on workloads. Therefore, these architectures cannot use all cores effectively.

To solve these problems, cooperative core architecture is proposed. Cooperative core architecture is based on a homogeneous architecture that uses narrow-issue cores. Cooperative core architecture can balance single-thread performance and multi-thread performance by fusion and split cores. It can accelerate single-thread execution by fusing some narrow-issue cores into a wide-issue virtual core. It can also accelerate multi-thread execution by splitting
a single wide-issue core into some narrow-issue cores.

In this paper, we design and implement efficient and realistic CoreSymphony architecture \cite{5, 6, 7} that is one of the cooperative core architectures and evaluate its performance. It can keep core modularity by forbidding most of communication of control signals from front-end. It adopts an out-of-order core as baseline processor in order to use conventional architecture technologies. It can realize cooperative execution without binary modification for keeping binary compatibility.

1.2 Contributions

The research in this paper makes the following contributions:

- Proposing efficient and realistic CoreSymphony, the cooperative core architecture that forbids most of communication from front-end, adopts an out-of-order core as a baseline, and does not need binary modification.
- Clarifying the problems associated with microarchitecture in order to realize CoreSymphony.
- Designing and implementing CoreSymphony to solve these problems.
- Showing possibility of the CoreSymphony architecture.

1.3 Organizations

This paper is organized as follows: Chapter 2 explains CoreSymphony and clarify differences between CoreSymphony and other cooperative core architectures. Chapter 3 shows the baseline processor that is used to construct CoreSymphony. Chapter 4 shows the challenges for realizing CoreSymphony and briefly explains the approaches to overcome these challenges. Chapter 5 describes the design and implementation of CoreSymphony. Chapter 6 presents the verification of the implementation by Verilog simulator and FPGA. Chapter 7 presents the evaluation of CoreSymphony. Finally, Chapter 8 concludes this paper and shows future work.
Chapter 2

Cooperative Core Architecture

We call an architecture that balances single-thread performance and multi-thread performance by fusion and split of cores cooperative core architecture \cite{8, 9, 10, 11, 12, 13, 14, 15, 16}.

In this chapter, we briefly explain CoreSymphony and clarify differences between CoreSymphony and other cooperative core architectures.

2.1 CoreSymphony

CoreSymphony belongs to the cooperative core architectures. Cooperative core architectures have three important requirements:

**Keeping core modularity.** On CMPs, core modularity is important, because it improves design productivity and reduces bad influences of fault cores. Therefore, it is undesirable to centralize the controls like a clustered architecture \cite{17, 18}.

**Keeping continuity of architecture technology.** In order to fuse cores, it is necessary to modify the architecture of conventional core, and it complicates core architecture. To decrease the core complexity and increase the feasibility, it is desirable to keep continuity of conventional architecture technologies.

**Keeping binary compatibility.** ISA modification is an effective way to decrease hardware complexity of cooperative core architectures. However, if we modify ISA dramatically, we cannot use state-of-the-art compiler technologies. Therefore, it is important to keep binary compatibility.

To keep core modularity, CoreSymphony forbids the most of communication from frontend, and all control signals are generated on each core. To keep continuity of architecture technology, CoreSymphony adopts an out-of-order core as a baseline, and tries to minimize core modification and core complexity. Additionally, to keep binary compatibility, CoreSym-
CoreSymphony tries to realize fusion of cores adopting conventional ISA.

CoreSymphony is able to fuse up to four 2-issue cores, and the fused cores execute programs as a single wide-issue core. The wide-issue core employs instruction-level parallelism (ILP) and is able to execute programs faster than a baseline core. Figure 2.1 shows a conceptual diagram of CoreSymphony. In this figure, for simplicity, we show 2-core configuration, on which two 2-issue cores are fused into a single 4-issue core.

I-$, decode, rename, steering, instruction window, physical register file, and load/store unit have cooperativeness. The numbers of effective sizes of the cooperative modules are increased by fusion. Inter-core data forwarding is realized with inter-core network linking fused cores’ back-ends.
2.2 Other Cooperative Core Architectures

2.2.1 Core Fusion

We show conceptual diagram of an 8-core Core Fusion [8, 9] chip in Figure 2.2. It fuses out-of-order cores and is able to construct up to 8-issue core by fusing four 2-issue cores. In terms of data path, Core Fusion is similar to CoreSymphony. In terms of control, Core Fusion has centralized unit to manage the fused front-end’s activities, such as instruction steering and register renaming, like clustered architecture. CoreSymphony doesn’t have centralized unit and eliminates inter-core communication from front-end. Therefore, Core Fusion differs from CoreSymphony. Because the centralized unit decreases core modularity, CoreSymphony is superior to Core Fusion in terms of core modularity.

2.2.2 Anaphase

We show block diagram of Anaphase [10, 11] in Figure 2.3. It enables parallel execution by decomposing a single-threaded program into fine-grain threads at the compile time. Each core has special hardware to maintain memory coherency. It is able to minimize hardware changes and achieve high area efficiency by compiler support. However, the compiler support decreases binary compatibility. Therefore, CoreSymphony is superior to Anaphase in terms of binary compatibility.
2.2.3 Composable Lightweight Processors

We show conceptual diagram of Composable Lightweight Processors [12] in Figure 2.4. It is a cooperative core architecture that can be configured as 32 2-issue processors or a single 64-issue processor. It uses dataflow ISA [19]. It is an effective approach to use the special ISA, because it is able to reduce hardware complexity. However, it decreases binary compatibility. Therefore, CoreSymphony is superior to Composable Lightweight Processors in terms of binary compatibility.
2.2 Other Cooperative Core Architectures

2.2.4 Voltron

We show block diagram of Voltron in Figure. It is a cooperative core architecture that uses VLIW ISA. It is an effective approach to use the special ISA. However, similarly to Composable Lightweight Processors, Voltron lacks binary compatibility.
2.2.5 Fg-STP

We show block diagram of Fg-STP \[14\] in Figure 2.6. It realizes thread decomposition of Anaphase by using hardware. Hence, It is similar to CoreSymphony compared to Anaphase. However, It needs some modules that are shared by two cores for steering and retirement. These shared modules decrease core modularity. Therefore, CoreSymphony is superior to Fg-STP in terms of core modularity.

2.2.6 CoreGenesis

We show conceptual diagram of an 8-core CoreGenesis \[15\] chip in Figure 2.7. CoreGenesis improves single-thread performance by fusion of in-order cores. To achieve high performance, instruction steering in fused in-order cores is more complex than that in fused out-of-order cores \[20\]. It solves this problem by compiler support. However, the compiler support decreases binary compatibility. It differs from CoreSymphony, because CoreSymphony adopts an out-of-order core as a baseline and tries to realize fusion of cores using only hardware to keep binary compatibility. Therefore, CoreSymphony is superior to Core Genesis in terms of binary compatibility.

2.2.7 MorphCore

We show conceptual diagram of MorphCore \[16\] in Figure 2.8. It is a cooperative core architecture that is based on a large out-of-order core. It achieves both high single-thread performance and high multi-thread performance by morphing the large core into an in-order SMT
core. It is designed based on the converse concept that some small cores are fused into a large core. When MorphCore runs on out-of-order mode, it achieves high single-thread performance because there is no overhead of cooperation. When MorphCore runs on in-order mode, it achieves high power efficiency because it does not out-of-order execution.
Chapter 3

Baseline Processor

In this chapter, we show the baseline processor that is used to construct CoreSymphony.

3.1 Instruction Fetch, Instruction Decode

There are no difference in instruction fetch and instruction decode between out-of-order processor and in-order processor. Instructions of the number of fetch width are fetched and decoded in program order.

3.2 Register Rename

Out-of-order processor start to execute instructions whose all source operands are ready regardless of program order, extract ILP, and can achieve high performance. ILP is restricted by data hazard. Assuming that after instruction $I_i$ is executed instruction $I_j$ is executed, there are three types of data hazard as follows:

a true dependency (read after write, RAW)   Before $I_i$ write result to a register, $I_j$ read the register. $I_j$ read wrong result and use it.

an output dependency (write after write, WAW)   Before $I_i$ write result to a register, $I_j$ write result to the register. The result of $I_j$ is destroyed by the result of $I_i$.

an anti-dependency (write after read, WAR)   Before $I_i$ read a register, $I_j$ write result to the register. $I_i$ read wrong result and use it.

An output dependency and an anti-dependency is called false dependency against true dependency, because result of an instruction are not used by the other.

Out-of-order processor gets rid of false dependency and maintains true dependency by register renaming. Register renaming works on two modules, free list and register map table (RMT). Free list manages unused physical register number. RMT manages correspondence
between numbers of logical registers that are shown as source register in a program and numbers of physical registers that supply source operand.

Register renaming is divided into three phases:

**Destination register renaming**  Physical destination register number is obtained from free list. If free list is empty, register renaming stage is stalled and wait for release of physical register. Free list needs read ports of the number of maximum renamed instructions at a cycle. Free list also needs write ports of the number of maximum retired instructions at a cycle.

**Source register renaming**  Number of physical register that supplies latest value is obtained by reading RMT with logical source register number. On a typical ISA of two source operands per one instruction, RMT needs read ports of the number of two times of renamed instruction at a cycle.

**RMT updating**  Logical destination register number is associated with physical destination register number obtained from free list. On this time, previously associated physical register number is read. On a typical ISA of one destination per one instruction, RMT needs write ports and read ports of the number of maximum renamed instructions at a cycle.

If there is a dependency between instructions that is renamed at a cycle, we needs to consider the dependency. For example, when two instructions are renamed at a cycle, if first instruction’s logical destination register number equals second instruction’s logical source register number, second instruction’s physical source register number becomes first instruction’s physical destination register number.

Physical register number that is read on RMT update, called \( \text{ppreg} \), is recorded to re-order buffer (discussed later), is released to free list on instruction retire.

### 3.3 Instruction Schedule, Instruction Issue

Out-of-order processor stores fetched instructions and execute instructions whose all source operands are ready. We call hardware to keep instructions waiting for operands instruction window.

There are two types of instruction window, centralized instruction window and distributed instruction window.

Centralized instruction window manages all dispatched instructions in one large instruction window. Some instructions whose all source operands are ready are selected from the window
and issued to execution unit. It can utilize its entries efficiently. However, because many ports are needed for dispatch and issue, hardware becomes complex.

Distributed instruction window consists of windows of the number of issue width. One window works for dispatch and issue of one instruction. It cannot utilize its entries due to bias of issue. However, compared to centralized instruction window, hardware of one instruction window becomes simple.

In order to simplify hardware, CoreSymphony adopts distributed instruction window.

Instruction window is divided into three functions as follows:

**Wake-up logic**  Wake-up logic consists of content addressable memory (CAM). Each entry of wake-up logic contains valid bits showing whether the entry is valid or not (valid), two physical source register numbers (psrc0, psrc1), and ready bits showing whether these source operands are already obtained (rdy0, rdy1). Physical destination register numbers of issued instructions are entered into wake-up logic and compared to psrc0 and psrc1 of entries whose valid is 1. If it is equal, corresponding rdy0 and rdy1 are set to 1.

**Select logic**  One instruction whose all operands are ready is selected. If valid, rdy0, and rdy1 are all 1, the instruction is ready to issue.

**Payload RAM**  Payload RAM contains information of instructions type of instruction, physical destination register number, entry number of re-order buffer, and so on. The information is sent to PRF and execution unit with physical source register number of selected instruction.

Issued instructions fetch operands from physical register file (PRF). If dependent instruction writes its result to PRF, issued instruction receives the result by forwarding.

### 3.4 Memory Access

Dependencies between load instructions and store instructions differ by calculated address. Also, it is difficult to quash issue of store instruction. Therefore, special mechanism for issuing load/store instruction is needed. In order to realize them, we use load/store queue (LSQ).

Entries of LSQ are allocated to memory reference instructions such as load and store in fetch order. Address is calculated in execution unit, it is stored to LSQ. If it is store instruction, store data is also stored to LSQ.

Load instruction can be issued only if it does not depend on all precedent store instructions. About all old store instructions in fetch order, if there is a store instruction whose
address equals to the load instruction’s address, the load instruction wait for issue of the store
instruction because data that should be loaded is not written. Furthermore, if there is a store
instruction whose address is not calculated, the load instruction cannot be issued because there
may be dependency between them.

Store instruction can be issued if its address and data are already obtained and ROB entry
assigned to the store instruction reaches head of ROB.

3.5 Instruction Retire

Out-of-order processor realizes program order update of processor state and precise exception
by re-order buffer (ROB). ROB is the same structure as first-in-first-out (FIFO). Dispatched
instructions are allocated to entries in program order, and the entries are released in program
order. Results of instructions are determined at instruction retire from ROB.

Behavior of ROB is as follows:

**Allocation**  Entries are allocated on tail of ROB when instructions are dispatched.

**Completion**  When instruction execution is completed, it is notified to ROB and a entry of
the instruction can retire. At this time, if branch prediction miss occurs or branch target
address is obtained, they are recorded in the entry of the instruction.

**Retire**  If head entry can retire, result of the instruction is determined and the entry is re-
leased. At this time, if the instruction causes branch prediction miss, all pipelines are
flushed and instructions after the exception are quashed by pipeline flush, and processor
restart execution from true branch target address.

RMT at front-end is updated by speculative instructions. Therefore, when pipeline flush oc-
curs due to exception, RMT is updated by quashed instructions on wrong path. It is necessary
to restore the state of RMT in order to execute correctly after restart.

In order to solve this problem, we use retirement register map table (RRMT). RRMT is a
table that is updated by instructions retiring from ROB in the same manner as RMT. Because
RRMT is update by non-speculative instructions, we can restore the state of RMT by copying
the content of RRMT to RMT.
Chapter 4

Challenges

In this chapter, we show challenges for realizing CoreSymphony and briefly explain the approaches to overcome these challenges.

4.1 Instruction Supply Unit

CoreSymphony executes a single-threaded program by distributing instructions to fused cores. The challenge on instruction supply to each core is divided to two, instruction fetch and instruction steering.

The former instruction fetch has three problems as follows:

(a) **Constructing the distributed I-$$. In order to increase the effective I-$ size by fusion, it is necessary to distribute instructions over cores.

(b) **Supplying sufficient information for handling the data dependency at the back-end.**

   In order to eliminate communication from front-end, it is necessary to supply the information.

(c) **Synchronizing control flows in fused cores.** Because fused cores execute a program as a single superscalar, fused cores must follow the same control flow.

In order to realize (a) and (b), we propose the *Local Instruction Cache* (Local I-$). CoreSymphony fetches and steers an instruction trace as a unit. We call the instruction trace *Fetch Block* (FB). Each Local I-$ contains instructions that are steered to the core. Local I-$ also contains information for handling the data dependency between FBs. Regarding to (c), all cores redundantly process branch prediction and handle speculation misses.

On CoreSymphony, instruction steering is the operation of determining which instruction is

---

*1 A sequence of at most $n$ instructions and at most $m$ basic blocks starting at any point in the dynamic instruction stream.*
executed in which core. The latter instruction steering has two problems as follows:

(d) **Reducing inter-core communication overheads.** In order to achieve high performance, it is necessary to steer dependent instructions to the same core.

(e) **Balancing loads.** Steering many instructions to the same core degrades ALU use efficiency. Therefore, it is necessary to alleviate load imbalance.

Additionally, we must satisfy next two constraints on instruction steering.

(f) **Resulting in always the same steering result about the same FB.** Because Local I-$ contains steered instruction traces, steering result of the same FB must be the same.

(g) **Eliminating inter-core data dependency of instructions in a FB.** In order to realize 2-way Renaming (discussed later), it is necessary to eliminate inter-core data dependency of instructions in a FB.

In order to achieve (d), (e), (f) and (g), we propose *Leaf-node Steering*. On this algorithm, all dependent instructions in the same FB are steered to the same core.

### 4.2 Operand Supply Unit

CoreSymphony has RMT to rename register and PRF to supply operand. The problems related to RMT and PRF are as follows:

(a) **Supporting PRF expansion by fusion.** It is desirable to increase or expand the number of PRF entries by fusion. For the PRF expansion, even if the number of fused cores increases, we must handle data dependency correctly.

(b) **Implementing RMT efficiently.** The number of ports of RMT for wide-issue superscalar is large. Therefore, on CoreSymphony, it is necessary to reduce the number of ports of RMT.

(c) **Constructing distributed PRF.** With distributed PRF, it is necessary to increase the number of entries of PRF by increasing the number of fused cores.

Regarding to (a), CoreSymphony manages the data dependency with two types of RMTs, RMT to handle the data dependency in a core and RMT to handle the inter-core data dependency. We are able to easily implement these RMTs and are able to solve (b). We call this renaming mechanism using two types of RMTs 2-way Renaming.

Regarding to (c), CoreSymphony has filtering mechanism for distributed PRF. On the write back stage, an execution result is broadcasted with inter-core network. Each core sees whether
4.3 Handling Speculation Miss

The problems related to handling speculation miss are as follows:

(a) **Constructing distributed ROB.** ROB is an important module to realize precise exceptions on out-of-order processors. A ROB entry is allocated to each in-flight instruction. On CoreSymphony, the number of in-flight instructions increases by fusion. Hence, ROB requires many entries. ROB also requires many ports, because issue width increases by fusion. Therefore, it is desirable to distribute instructions to all ROBs in fused cores.

(b) **Managing in-order state effectively.** All fused cores need in-order state for recovery from speculation miss. However, it is unrealistic that all fused cores redundantly have in-order state, because hardware complexity increases.

Regarding to (a), CoreSymphony has two types of ROBs, ROB to allocate instructions executed in each core itself and ROB to share compressed necessary information for precise control. The former ROB stores instructions steered to the core itself. Therefore, we are able to increase the number of effective ROB entries by fusion. However, this makes it more difficult to take control precisely because information, such as speculation misses, and so on, is distributed over all fused cores. Therefore, CoreSymphony shares necessary information for precise control on all fused cores. Such information is managed in a special ROB, and all fused cores have this ROB redundantly.

Regarding to (b), we prepare *logical register file* (LRF) in each core and record in-order state in LRF. LRF stores execution results of committed instructions. On the commit stage, each core reads execution results that is calculated on the core from PRF, and copies these values to its own LRF. In this way, each LRF has incomplete in-order state that is constructed by instructions executed on the core. When recovering from speculation miss, each core constructs in-order state by inter-core communication. This method is effective because this inter-core communication overhead is small.

---

*2 State constructed by last assignment to each logical register, regardless of issued or completed.
*3 State constructed by last assignment of non-speculative instructions to each logical register.
4.4 Other Challenges and Solutions

4.4.1 Load/Store Unit

We attempt to increase the number of effective entries of D-$ and LSQ by fusion. Regarding to this challenge, there is a bank separated approach for D-$ and LSQ [18, 8]. Commonly, the bank to dispatch load/store instruction is determined after execution. Bank prediction [22] has been proposed to predict this bank. However, this requires a recovery mechanism from bank mispredictions.

On CoreSymphony, in order to solve this problem, we adopt Unordered Late-Binding Load/Store Queue [23] (ULB-LSQ). ULB-LSQ is able to allocate instructions after calculation of memory address. Therefore, it doesn’t require bank prediction.

CoreSymphony regards D-$s and LSQs in fused cores as banks separated by memory address. After calculation of the memory address, CoreSymphony dispatches load/store instruction to the ULB-LSQ that the memory address indicates. We prepare a special inter-core network to dispatch instruction to ULB-LSQ in a remote core.

4.4.2 Timing of Commit

CoreSymphony doesn’t synchronize the timing of commit over all fused cores, because inter-core communication latency exists. This causes a problem.

Branch prediction results may be different, and each core may follow different control flow. Branch predictor is updated when instructions commit. Therefore, due to a gap in the timing of updating branch predictor, branch history tables may vary in fused cores.

CoreSymphony solves this problem by waiting to update in each core. We suppose that the maximum number of in-flight FBs is $N$, and also suppose that the timing when the branch instruction in $FB_t$ update the branch predictor is when $FB_{t+N}$ is fetched. By this way, branch predictor is the same in all fused cores when a FB is fetched. Therefore, all fused cores follow the same control flow.
Chapter 5
Proposal and Efficient Implementation

We aim at implementing CoreSymphony on Field Programmable Gate Array (FPGA). FPGA is built from many logic cells (called slices in Xilinx FPGA), and one logic cell consists of some look-up tables (LUTs) and flip-flops (FFs). In order to save use of LUTs and FFs, it is necessary to design CoreSymphony efficiently and implement it.

In this chapter, we describe efficient design and implementation of CoreSymphony.

5.1 Local Instruction Cache

5.1.1 Basic Idea

Cooperative Instruction Fetch Mechanism

CoreSymphony fetches and steers an instruction trace called FB as a unit. The number of instructions in FB is limited to four times of the number of cores. For example, on 4-core fusion, FB contains at most 16 instructions. In addition, FB is limited to the condition that FB doesn’t contain more than three basic blocks. Local I-$ is an instruction trace cache that contains steered instructions to the core in FB. Local I-$ also contains control information for handling the data dependency between FBs. One line in Local I-$ basically corresponds to a single FB.

In Figure 5.1, we describe the cooperative instruction fetch mechanism using Local I-. Figure 5.1 shows an example of fetching an FB containing eight instructions, $(I_0, I_1, ..., I_7)\textsuperscript{1}$, on 2-core fusion. Conventional I-$ supplies two instructions at a cycle.

In fetching a FB separately in a fused core, it may go through two phases: (a) trace con-

\footnote{I$_n$ means an instruction, and $(I_0, I_1, ..., I_n)$ means an FB.}
construction phase and (b) cooperative fetch phase.

Figure 5.1(a) shows the trace construction phase, which is executed only if Local I-$ misses. After fetching all instructions in the FB from the conventional I-$, these instructions are decoded and steered. Steered instructions and FB information (info) are then written back to the indexed entry of Local I-$ in each core.

Figure 5.1(b) shows the cooperative fetch phase, which is executed if Local I-$ hits. In this case, each core fetches only instructions that are steered to the core. Therefore, the front-end’s throughput on N-core fusion is N times greater than that on 1-core execution.

Local I-$ contains only instructions that are steered to the core. Therefore, the number of effective size of the total Local I-$ increases by fusion.

**Branch Predictor**

On CoreSymphony, all fused cores predict branches using the same branch history. This doesn’t increase the number of entries of branch predictor by fusion. However, this makes it easy to synchronize control flows of fused cores.

We adopt Tree-Based Multiple Branch Predictor [24] (TMP) as a branch predictor. This is a branch predictor suitable for trace cache. TMP is applied to CoreSymphony easily, because Local I-$ is a trace cache.

TMP uses one entry for prediction of one instruction trace. A FB is an instruction trace on CoreSymphony. As explained in Section 5.1, a FB is able to contain more instructions by fusion. This decreases the number of necessary static FBs for program execution by fusion. Therefore, it decreases the number of necessary TMP entries, and improves branch prediction accuracy by fusion.

### 5.1.2 Implementation

An entry of Local I-$ has following fields:

- **steered instruction (32bit x 16)** Instructions that are steered to the core.
- **fetch block pc (32bit)** Starting address of the first basic block. This is used for calculation of instructions’ PC in first basic block.
- **destination vector (32bit)** Vector that indicates instructions’ logical destination registers in the FB. Length of destination vector is determined by the number of logical registers. If \( n \)th bit of vector is 1, it shows that the FB contains at least an instruction whose logical

\(^{2}\) FB information contains control information for trace cache, information for handling the data dependency between FBs and so on.
5.1 Local Instruction Cache

destination register is $n$. Destination vector is used for 2-way Renaming.

**next addr (32bit)** Starting address of the second basic block. This is used for calculation of instructions’ PC in second basic block.

**next pc (32bit)** Starting address of FB that should be fetched next time.

**branch num (2bit)** The number of branch instructions. This is used for update of TMP’s global history register.

**branch pattern (2bit)** Branch Pattern of two branch instructions that are contained in the FB. This is used for Local I-$ hit/miss decision.

**TKN branch slot (4bit $\times$ 2)** Slot numbers of two branch instructions that are predicted to ‘taken’.

**slot number (4bit $\times$ 16)** Slot numbers of steered instructions.

**broadcast flag (1bit $\times$ 16)** Broadcast flags indicates whether steered instructions broadcast results.

Originally, steered instruction field is four words. If more than five instructions are steered to a core, remaining instructions are stored to next line [7]. However, if next line is already used, it needs complex cache line invalidation mechanism. Necessary invalidation are shown in Figure 5.2 and Figure 5.3.

In Figure 5.1, at first, each core stores four instructions to Local I-$. Then, in previous line, core0 stores five instructions, and core1 stores three instructions. On core0, next line is automatically invalidated because remaining instructions are stored to next line. However, next line of core1 should be invalidated even if where there are no remaining instructions. This is because, when FB shown in green is fetched, core0’s Local I-$ misses, and core1’s Local Instruction Cache.
I-$ hits. Properly, both cores of Local I-$s must miss.

As a different case, in Figure 5.3, at first, core0 stores five instructions, and core1 stores three instructions. Afterwards, when a FB is stores in next line, previous line should be invalidated, because the FB stored in previous line is broken. Even if core1’s Local I-$ contains correct line, both lines should be invalidated for control consistency over all fused cores.

These cases may happen at the same time. To avoid it, we need to judge whether next line and previous line should be invalidated by reading these lines.

On this implementation, for simplicity, we set line size of Local I-$ to 16 words. By doing this, one line of Local I-$ can hold one FB if the number of all instructions of a FB are steered to one core. However, hardware amount of Local I-$ increases, and Local I-$ becomes

Fig. 5.2 Case 1 of Local I-$ line filling (needs to invalidate next lines).
5.2 Leaf-node Steering

5.2.1 Basic idea

Leaf-node Steering allows an instruction to be steered to some cores. By this way, we are able to satisfy two conflicting requirements.

We present an example and explain this steering algorithm. Figure 5.4 shows three steered results obtained by two existing algorithms (Modulo-2, Dependency-based) and one proposal algorithm (Leaf-node). These algorithms are described as follows:

Fig. 5.3 Case 2 of Local I-$ line filling (needs to invalidate previous lines).

inefficient. More efficient implementation of Local I-$ is one of the future work.
**Modulo-2 Steering.** A sequence of instructions is separated at every two instructions. These instruction blocks are steered to each core in a round-robin manner.

**Dependency-based Steering.** All dependent instructions are steered to the same core. Independent instructions are steered to the minimal loaded core.

**Leaf-node Steering.** Instructions are steered based on the leaf-node. Leaf-node is the leaf of the dataflow graph (DFG) in the same FB. In Figure 5.4, leaf-nodes are I₃, I₄, and I₆. First, leaf-nodes are steered to each core in a round-robin manner. Then, all ancestors of Iᵢ are steered to the core where Iᵢ is already steered. We explain an example in Figure 5.4(c). The ancestors, I₀ and I₁, are steered to Core0, where I₃ is already steered. Both I₀ and I₁ are also steered to Core1, because these are also ancestors of I₄. Therefore, I₀ and I₁ are copied and executed redundantly on Core0 and Core1.

Modulo-2 Steering achieves good load balancing. However, as shown in Figure 5.4, the inter-core data dependency increases.

Dependency-based Steering is able to reduce the inter-core data dependency. However, this doesn’t achieve good load balancing.

In Leaf-node Steering, inter-core communication doesn’t occur in the same FB, and this steering is expected to achieve moderate load balancing. However, redundant steering increases the total executed instructions in all fused cores.

In Chapter 7, we compare three algorithms and evaluate the instruction redundancy on Leaf-node Steering.

### 5.2.2 Implementation

Leaf-node Steering is divided into two phases: (a) ancestor-node matrix/leaf-node vector construction phase and (b) steering result determination phase. On (a), ancestor-node matrix and leaf-node vector is constructed by analyzing DFG. On (b), instructions are steered to each core based on the matrix and the vector.

**Ancestor-node Matrix/Leaf-node Vector Construction Phase**

There are ancestor-node matrix (ANC), leaf-node vector (L), and destination table (DT) in steering logic. In the following, Iᵢ stands for iᵗʰ instruction in FB. At first, we define ancestor-node matrix, leaf-node vector, and destination table.

---

*a³ Nodes on the path from dataflow graph’s root to its leaf-node. In Figure 5.4, the ancestors of I₃ are I₁ and I₀.*
5.2 Leaf-node Steering

Ancestor-node matrix is $16 \times 16$ bit matrix\(^4\). In the following, $\text{ANC}[i][j]$ stands for the $i$th row and $j$th col element of ANC. $\text{ANC}[i][j]$ is defined as follows. $\text{ancestor}(I_i)$ is a set of ancestors of $I_i$.

$$\text{ANC}[i][j] = \begin{cases} 1 & \text{if } I_j \in \text{ancestor}(I_i) \\ 0 & \text{others} \end{cases}$$

$i$th row of ANC stands for an ancestor instruction set of $I_i$. We call the row of ANC ancestor vector ($\text{ANC}[i]$). If $I_i$ depends on $I_j$ and $I_k$, $\text{ANC}[i]$ is calculated as follows. $e_i$ stands for fundamental vector\(^5\). $|$ is logical OR operation.

$$\text{ANC}[i] = e_i | \text{ANC}[j] | \text{ANC}[k]$$

Leaf-node vector is a 16-bit vector showing whether instructions in FB are leaf-node. $L[i]$ stands for $i$th bit of leaf-node vector and shows whether $I_i$ is leaf-node.

Destination table is used to detect dependency between instructions. The number of entries of DT equals the number of logical registers. Each entry of DT contains valid bit. In the following, $D[i]$ stands for $i$th entry of DT. When $I_i$ write its result to logical register $R_j$, $DT[j]$ is set to 1. If an instruction read $R_j$, its dependent instruction is obtained by reading $DT[j]$. However, dependencies between two instructions fetch at the same time should be handled carefully. If destination register number of first instruction equals source register number of second instruction, there is dependency between these instructions.

We explain this phase about FB shown in Figure 5.4. Two instructions enter the steering stage in each cycle.

\(^4\) 16 is maximum length of FB.
\(^5\) Vector whose $i$th bit is 1 and other bits are all 0.
Initialization \textbf{L} and DT is initialized. All bits of \textbf{L} is set to 0 and all valid bits of DT is set to 0.

1-cycle \textit{I}_0 and \textit{I}_1 enter the steering stage. By reading DT and comparing destination register number and source register numbers, dependency is analyzed. \textit{I}_0 has no dependent instruction (\textit{ANC}[0] = \text{e}_0). \textit{I}_1 depends on \textit{I}_0 (\textit{ANC}[1] = \text{e}_1 \mid \text{ANC}[0]). Because the result of \textit{I}_0 is used by \textit{I}_1, \textit{I}_0 is not leaf-node (\textbf{L}[0] = 0). DT entries associated with \textit{I}_0 and \textit{I}_1 are set to 0 and 1 individually.

2-cycle \textit{I}_2 and \textit{I}_3 enter the steering stage. By reading DT and comparing register numbers, dependency is analyzed. \textit{I}_2 depends on \textit{I}_1 (\textit{ANC}[2] = \text{e}_2 \mid \text{ANC}[1]). \textit{I}_3 depends on \textit{I}_1 (\textit{ANC}[3] = \text{e}_3 \mid \text{ANC}[1]). Because the result of \textit{I}_1 is used by \textit{I}_2 and \textit{I}_3, \textit{I}_1 is not leaf-node (\textbf{L}[1] = 0). DT entries associated with \textit{I}_2 and \textit{I}_3 are set to 2 and 3 individually.

3-cycle \textit{I}_4 and \textit{I}_5 enter the steering stage. By reading DT and comparing register numbers, dependency is analyzed. \textit{I}_4 depends on \textit{I}_2 (\textit{ANC}[4] = \text{e}_4 \mid \text{ANC}[2]). \textit{I}_5 has no dependent instruction (\textit{ANC}[5] = \text{e}_5). Because the result of \textit{I}_2 is used by \textit{I}_4, \textit{I}_2 is not leaf-node (\textbf{L}[2] = 0). DT entries associated with \textit{I}_4 and \textit{I}_5 are set to 4 and 5 individually.

4-cycle \textit{I}_6 enters the steering stage. By reading DT and comparing register numbers, dependency is analyzed. \textit{I}_6 depends on \textit{I}_5 (\textit{ANC}[6] = \text{e}_6 \mid \text{ANC}[5]). Because the result of \textit{I}_5 is used by \textit{I}_6, \textit{I}_5 is not leaf-node (\textbf{L}[5] = 0). DT entries associated with \textit{I}_6 are set to 6.

After the above process, this phase is finished, and ANC and \textbf{L} are calculated.

Steering Result Determination Phase

On this phase, steering result is determined based on ancestor-node matrix and leaf-node vector. At first, we calculate steered-leaf vector (SLV) that shows leaf-nodes steered to the core. Then, we determine steering vector (SV) that shows instructions steered to the core. 

\( \text{SLV}[i] = 1 \) shows that leaf-node \textit{I}_i is steered to the core. 
\( \text{SV}[i] = 1 \) shows that instruction \textit{I}_i is steered to the core.

We show SLV determination logic in Figure 5.5. SLV determination logic receives leaf-node vector \textbf{L}, the number of fused cores (CoreNum), core-id assigned to the core as inputs, and output SLV that shows leaf-nodes steered to the core. \text{add}\_\text{lim} shown in Figure 5.5 is a special adder logic where maximum calculated value is restricted by CoreNum. The following equation is true in \text{ADD}\_\text{lim}.

\[
\text{ADD}\_\text{lim}(X, Y, \text{CoreNum}) = \begin{cases} 
X + Y - \text{CoreNum} & \text{if } X + Y \geq \text{CoreNum} \\
X + Y & \text{others}
\end{cases}
\]

We show ST determination logic in Figure 5.6. ST determination logic inputs SLV into ANC and outputs some ancestor vectors about leaf-nodes steered to the core. Then, these are
operated by logical OR and ST are calculated.

On the previous example, because 3th bit, 4th bit, and 6th bit are 1, $I_3$, $I_4$, and $I_6$ are leaf-nodes. These instructions are steered in round-robin manner. Then, about $ANC[3]$, $ANC[4]$, and $ANC[6]$, instructions whose bit is 1 are steered to each core.

### 5.3 2-way Renaming

#### 5.3.1 Basic Idea

On 2-way Renaming, the data dependency in a core and the inter-core data dependency are managed using two different tables. Information that should be shared among all fused cores is only the inter-core data dependency. Therefore, we use the compressed information for the inter-core data dependency, instead of physical register number.
Here, we use Local Tag (Ltag), the operand tag that shows the data dependency in a core and Global Tag (Gtag), the operand tag that shows the inter-core data dependency. We also use Local RMT (LRMT), the table that manages Ltag and Global RMT (GRMT), the table that manages Gtag. The sets of Ltag and Gtag are as follows.

\[
\begin{align*}
\text{Ltag} & : \{P_m : m = 0, 1, ..., NP - 1\} \\
\text{Gtag} & : \{G_t : t = 0, 1, ..., NF\}
\end{align*}
\]

NP is the total number of PRF entries. NF is the maximum number of FBID\textsuperscript{6}. Each operand has the tag of Gtag if the operand is not obtained at the steered core; otherwise each operand has the tag of Ltag. Ltag is physical register number, and Gtag is FBID of FB that generates the operand.

Gtag doesn’t identify the instruction that supplies the operand. This is sufficient information for handling the inter-core data dependency. This is because the number of instructions that supply operands to a remote core is one per each logical register at most with Leaf-node Steering.

\*FBID is an identifier that is assigned to an in-flight FB in a cyclic manner.
5.3 2-way Renaming

LRMT is equivalent to the conventional RMT whose each entry has physical register number. The number of ports of LRMT is almost the same as RMT of 2-way out-of-order core. Additionally, we extend each entry to \{valid bit, physical register number, \textit{FBID} of producer, broadcast flag of producer\}. The \textit{FBID} is the number of FB that generates the result.

GRMT is constructed as a $N \times M$ bit matrix. $N$ is the number of logical registers, and $M$ is the maximum number of in-flight FBs. The bit of GRMT in the $i$th row and $j$th column shows whether $FB_j$ contains an instruction whose destination register is $R_i$ or not.

Register renaming with GRMT is, similarly to using a conventional RMT, divided to two steps: \(a\) Destination registration; and \(b\) Source renaming. Figure 5.7 shows the behaviors of the two steps.

Each line of Local I-$\$ contains information called \textit{destination vector}. Destination vector is a vector that is the same in length of the number of logical registers. If the bit of destination vector in the $n$th is 1, the FB contains an instruction whose destination register is $R_n$.

On \(a\), destination vector associated with the FB7 indicating at least $R_{30}$ and $R_{31}$ are destinations is written to the column of GRMT. GRMTs are the same in all fused cores, because destination vectors in all fused cores are the same. On \(b\), the row of GRMT is read using logical register number ($R_1$), similarly to LRMT. The $x$th row of GRMT shows $FBID$ of the FB that contains the instruction whose destination is $R_x$. By younger logic shown in Figure 5.7, the youngest $FBID$ is obtained. In this case, $FB_6$ is younger than $FB_0$. As a result, source renaming is completed by tagging as Gtag = $G_6$.

---

### 5.3.2 Implementation

Fig. 5.7  Cooperative renaming with GRMT.
By this way, the number of ports of GRMT is one for (a) and four for (b) for renaming. However, in order to realize distributed PRF (discussed later), we add three read ports (total of seven ports) in GRMT.

On 2-way Renaming, two types of tags, Ltag and Gtag, are obtained. However, only either one is used as a valid tag. Comparing \( FBID \) obtained by renaming with LRMT to \( FBID \) obtained by renaming with GRMT, we determine which tag is used. If \( FBID \) of LRMT is younger, we adopt Ltag. If \( FBID \) of GRMT is younger, we adopt Gtag. Where \( FBID \) of LRMT equals \( FBID \) of GRMT, if broadcast flag obtained by renaming with LRMT is 1, we adopt Ltag; otherwise, we adopt Gtag.

5.4 Physical Register File Distribution

5.4.1 Basic Idea

On CoreSymphony, execution results and tags are broadcasted to all fused cores on the write back stage. However, it is undesirable to store all results from a remote core to PRF, because of the number of ports and entries on PRF.

CoreSymphony distributes execution results to all PRFs in all fused cores. The execution results broadcasted from a remote core aren’t always needed. CoreSymphony detects the case, and filters the write back. We describe the two on which execution results are filtered in the following.

5.4.2 Implementation

Filtering Broadcast to Remote Core

Commonly, broadcasts increase the hardware complexity. Therefore, we restrict the number of broadcasts. We define the number of broadcasts per cycle from a core as \( Broadcast \text{ Width} \) (BW). The reduction of BW minimize the hardware complexity, but it causes performance degradation.

We filter the broadcast in the following two cases:

1. The instruction doesn’t produce any operand (e.g., store instruction, branch instruction).
2. The execution result produced by the instruction is overwritten by other instruction in the same FB. Because Leaf-node Steering doesn’t cause inter-core data forwarding in the same FB, the overwritten execution result isn’t used by a remote core.

These are previously analyzed and the information is stored to Local I-$ as broadcast flag. The scheduler selects instructions by checking the flag. On CoreSymphony, let \( BW = 1 \). We
evaluate the effect of $BW$ in Section 7.

**Filtering Write back to Physical Register File**

Even if $BW = 1$, the number of broadcasted operands is up to three on 4-core fusion. In order to reduce the number of ports of PRF, we filter the write back to PRF.

The execution results from a remote core are stored to PRF only in the following cases.

1. The execution result is needed by any instruction in instruction window.
2. The execution result is in the architecture state.

CoreSymphony detects these cases and filters the write back to PRF. This reduces the number of ports on PRF. This reduces the number of entries on PRF, because physical register isn’t assigned to if not needed. In case of remote write back, physical register allocation is processed on the write back stage.

We detect (1) by checking whether an instruction in instruction window depends on the broadcasted Gtag or not. To do this, we add $3BW$ search ports to instruction window.

We detect (2) by reading the row of GRMT using logical destination register number of the broadcasted operand, and obtaining $FBID$ that provides the architecture state for the destination register. If the $FBID$ equals to the $FBID$ of the broadcasted instruction, the instruction is in the architecture state. To do this, we add $BW \times (NC - 1)$ read ports to GRMT. On CoreSymphony, let $NC = 4$. Therefore, we add three read ports to GRMT.

We define the number of write ports of PRF for remote instructions as **Remote write back Width ($RW$)**. When an operand from a remote core is broadcasted, the operand is enqueued. When the operand is dequeued and written back, a PRF entry is allocated and its number is registered to LRMT in $RW$ inst/cycle. To do this, LRMT needs $RW$ write ports and $RW$ read ports. On CoreSymphony, let $RW = 1$. We evaluate the effect of $RW$ in Section 7.

**Deadlock Caused by Remote Write back**

Allocation of an entry for a remote operand may cause deadlock. The deadlock occurs when the remote instruction $I_x$ is the oldest one in all in-flight instructions and PRF has no entry to write back the operand.

CoreSymphony solves this deadlock by a simple method. We reserve some PRF entries for remote instructions. We call it **Number of Reserved Physical register ($NRP$)**. If the remaining number of PRF entries is less than $NRP$, the front-end is stalled. No matter how many entries PRF has for $NRP$, deadlock may occur. On this case, we flush in all fused cores the instructions after the remote instruction that isn’t able to write the result, and rerun the program from the instruction.
5.5 Re-order Buffer Distribution

5.5.1 Basic Idea

On CoreSymphony, a core has two types of ROBs, ROB to allocate instructions executed in local core and ROB to share necessary information for precise control. We call the former Local ROB (LROB) and the latter Global ROB (GROB). Only instructions executed in local core are allocated to LROB. The size of LROB is equal to the size of 2-way out-of-order core’s ROB. A GROB entry is allocated for each dispatched FB. The number of GROB entries is equal to the maximum number of in-flight FBs, typically eight. The contents of GROB are the same on all fused cores.

5.5.2 Implementation

Flow of Instruction Retire

Figure 5.8 shows the block diagram of distributed ROB. We explain the detailed behavior below.

(a) Allocate When a FB is dispatched in a core, only instructions steered to the core is allocated to LROB and a GROB entry is allocated for the FB. Because dispatch width of a core is 2 inst/cycle, LROB has two ports for allocation. Because we dispatch one FB in a cycle at a maximum, GROB has one port for allocation.

(b) Local Complete Completions of executed instructions are notified only in the core where these instructions are steered. LROB is updated in the same manner as a conventional ROB. Issue width of a core is 2 inst/cycle, and LROB has two ports for update.

(c) Broadcast Complete If instructions broadcasts, completions of the instructions are notified to LROB after write back to remote core’s PRF. If remote write back fails due to shortage of remote PRF entry, the information is recorded on completion of the instructions. Afterwards, it is handled similar to speculation miss. For this, LROB has three ports for update.

(d) Local Retire We then broadcast compressed information for precise control to all GROBs in all fused cores. The information involve FBID of completed instruction, position of completed instruction in FB, result of branch instruction, speculation hit/miss and so on. To do this, GROB has four ports for update, because the maximum number of fused cores is four. We update GROBs in all fused cores using the same
information. Therefore, the contents of all GROBs are the same in all fused cores. The instructions that are broadcasted to all GROBs are allocated to Local Commit Buffer (LCB).

(e) **Global Retire** GROB entry collects information, such as completion flags of instructions in the FB, result of branch instruction in the FB, speculation hit/miss and so on. After all instructions in the FB are completed, instructions in LCB and RCB (discussed later) are woken up using the information. If speculation miss occurs, precise PC is sent to the front-end. We set ready flags of instructions in precise control flow.

(f) **Commit** Instructions that has ready flag are committed in 2 inst/cycle from LCB and in RW inst/cycle from RCB.

By ROB distribution, we expand the number of entries of ROB by fusion. This increases in-flight instructions and improves performance. We evaluate the effect of ROB distribution in Section 7.

**Physical Register Deallocation**

As discussed in Section 5.4, the execution result of a remote core may be stored to PRF. This and distributed ROB causes a problem about physical register deallocation.

On a conventional out-of-order processor, on register renaming of an instruction, we read the physical register number where the destination register is already mapped (ppreg) from RMT, and record the physical register number that is assigned to the instruction as the destination (preg) into RMT. ppreg is stored to ROB entry that is allocated for the instruction. When the instruction is committed, to deallocate physical register, if the instruction is in precise control flow, the ppreg is brought back to free list.

On CoreSymphony, LROB has no entry for instructions that are steered to other cores. When CoreSymphony writes back execution result of remote instruction to PRF, we aren’t able to bring back its ppreg to free list. Therefore, CoreSymphony needs special mechanism for bringing back the ppreg to free list.

For physical register deallocation for remote instructions, we use Remote Commit Buffer (RCB) in Figure 5.8. RCB is constructed in a similar way of LCB. An entry is allocated to RCB when a remote instruction writes back the result to PRF, and ppreg is stored to the RCB entry (Figure 5.8(g)). When the instruction is committed, the ppreg is brought back to free list.

This mechanism may cause deadlock. CoreSymphony solves this problem in the same manner as the deadlock problem in Section 5.4.
### 5.6 In-order State Distribution

#### 5.6.1 Basic Idea

We need the in-order state for recovery from speculation miss. CoreSymphony has logical register file that keeps a part of the in-order state. LRF has as many entries as logical register file on ISA.

LRF is updated when instructions commit. On CoreSymphony, the number of commits in all fused cores increases by fusion and is up to eight on 4-core fusion. Therefore, LRF may need eight write ports in order to keep in-order state in all fused cores redundantly.

In order to manage in-order state effectively, we distribute the in-order state in all fused cores. The basic idea of in-order state distribution is that LRF in a core has incomplete in-order state constructed by only the instructions executed in the core, and when speculation miss occurs, we construct complete in-order state by communication between LRFs.
5.6.2 Implementation

Figure 5.9 shows (a) update of in-order state when instructions commit; and (b) construction of in-order state when speculation miss occurs.

When instructions commit, each core reads results from PRF using preg that is stored to LCB, and copies the results to LRF in the core using logical destination register number (lreg) that is also stored to LCB. At the same time, lreg is broadcasted to all fused cores and each core records the lreg in LRF Manager. Furthermore, FBID and the position of the instruction in FB (slot) are recorded only to the LRF Manager in the core. By this way, each LRF has incomplete in-order state constructed by only the instructions that is steered to the core.

When recovering from speculation miss, we construct in-order state. The values of logical registers that aren’t updated recently\(^7\) in any core are always the same in all fused cores. Therefore, it is sufficient to construct logical registers that are updated recently. The updated logical register number is recorded in LRF Manager. Furthermore, it is sufficient to broadcast only the latest values of recently updated instructions. We are able to know which core’s LRF entry has the latest value by communicating and comparing FBID and slot in LRF Manager between cores. The core that has the latest value reads the value from LRF and copies it to all LRFs in all fused cores. We execute this process at every logical register.

In order to update in-order state, LRF needs two write ports that are the same in number of retirement width of LCB. Moreover, in order to construct in-order state, LRF needs as many ports as the number of LRF entries that we desire to construct per cycle. Here, we focus on the point that it is only after speculation miss to construct in-order state. Because the pipeline is flushed after speculation miss, next instruction commit begins after a while. Therefore, even if we use the same ports for both update and construction, it would have a small effect. Thus, we stall the instruction commit during construction of in-order state, and we start instruction commit after construction. Let the number of LRF entries that is constructed per cycle be 2, and we add two read ports for construction. Therefore, we are able to realize LRF that has six read ports and two write ports.

During construction, we fetch instructions as usual. However, we don’t issue instructions whose source register isn’t constructed. The scheduler sees whether the source register of the instruction is already constructed or not, and issues the instruction. Constructed logical register numbers are sent to instructions windows and instructions waiting for construction are

\(^7\) From the latest speculation miss recovery to the present.
woken up. In order to wake up, instruction window needs two ports.

The distributed in-order state may degrade performance. We evaluate the effect in Section 7.

5.7 Other Designs and Implementations

5.7.1 Instruction Window

In order to realize 2-way Renaming, physical register file distribution, and in-order state distribution, we need to modify entry’s field and wake-up CAM.

There are three types of operand supplying on CoreSymphony.

Case of Using results calculated in the core In this case, Ltag is used. Ltag is physical register number in the core and used for register fetch.

Case of Using results calculated in the remote cores In this case, Gtag is used. Gtag is a tag concatenated FBID and logical destination register number of instruction that produce the result. Logical register number is used for wake-up. If Gtag broadcasted from remote core equals Gtag in instruction window, physical register number assigned to the remote instruction is stored to src field in the instruction window. The physical register number is used for register fetch.

Case of Using result in LRF This case occurs soon after program execution and recovery from branch prediction miss, and so on. In this case, source registers are not renamed. Logical register numbers are dispatched as tags and used for fetching operands from LRF.
5.7 Other Designs and Implementations

In order to compare the three types of tags, we modify field of entry and add comparators.

We show a block diagram of an entry of an instruction window in Figure 5.10. This mechanism is for one operand. Actually, one entry contains the two mechanisms for two operands. Each entry contains following information.

\( \text{rdy}_l, \text{rdy}_r \)  Bit showing whether the operands are ready.
\( \text{remote}_l, \text{remote}_r \)  Bit showing whether the operands are broadcasted from remote cores.
\( \text{lrf}_l, \text{lrf}_r \)  Bit showing whether the operands are supplied from LRF.
\( \text{tag}_l, \text{tag}_r (\{\text{fbid}_l, \text{src}_l\}, \{\text{fbid}_r, \text{src}_r\}) \)  Field that stores tag adopted among Gtag, Ltag, and logical register number,

Ltag is sent from local core and used for wake-up. Because issue width of a core is two, two comparators are needed for Ltag comparison.

Gtag is sent from remote cores. Because up to three instructions are written back from remote cores at a cycle, three comparators are needed for Gtag comparison. Comparison
results are used for filtering write back to PRF. One of three comparison results is selected and used for wake-up.

Logical register number is sent for constructing in-order state. Because two logical registers are constructed at a cycle, two comparators are needed for logical register number comparison.

### 5.7.2 Physical Register File, Logical Register File

PRF is a 6R3W memory, and LRF is a 6R2W memory. Bit width is 32, and the number of entries is about 32-64. It is not desirable to construct these modules with FF and LUT, because many FPGA resources are required. In this situation, it is desirable to use BRAMs. However, BRAMs has only two ports. We need to devise ways to use BRAMs.

On this implementation, we use Live Value Table (LVT). By using LVT, we can construct an $nR1W$ memory with $n \times m$ BRAMs, and decrease use of FF and LUT.

We show memory structure with LVT in Figure 5.11 and Figure 5.12.

A $nR1W$ memory can be implemented by writing the same data to $n$ BRAMs redundantly. Because all BRAMs are the same, by reading data from each BRAM, we can increase the number of read ports.

A $nRmW$ memory can be implemented with $m$ BRAMs and a LVT. LVT is implemented by FF and LUT. Bit width of an entry of LVT is $\lceil \log_2 m \rceil$, and the number of entries of LVT equals the number of entries of memories. When data is wrote to memory, we record which $nR1W$ memory has the data on an entry LVT associated with the entry. By reading the information,
output data is selected by multiplexer.

### 5.7.3 Inter-core Communication

Up to three instructions are broadcasted from remote cores at a cycle. However, only one instruction can write back result to PRF. We prepare queues for these broadcasts. We show remote write back mechanism in Figure 5.13.

- At first, write back from three remote cores are enqueued to three queues separately. Each queue can be implemented by a 1R1W FIFO.
- Then, about three instructions on heads of queues, we determine whether filtering is performed. By reading GRMT, we can know whether broadcasted results are in architecture state. By comparing Gtags in instruction window, we can know whether already
dispatched instructions consume broadcasted results. By operating logical OR, we can determine whether write back is needed.

- If write back is not needed, the instruction is removed from the queues. If some instructions need to write back, one instruction is selected and others are left in the queues.
- Selected instruction wakes up instructions in instruction window, and physical register is assigned to the instruction, and result of the instruction is written to the physical register. LRMT is updated, and physical register number that should be released is written to RCB.

5.8 Overall Structure

We show the overall structure of CoreSymphony in Figure 5.14. The shaded areas are the modules that are added or modified for CoreSymphony.

We summarize the complexity of CoreSymphony in Table 5.1. Each number in modules in Figure 5.14 is correspond to the number of Function field in Table 5.1. For comparison, we show the complexity of a typical 2-way out-of-order core. On CoreSymphony, let $BW = 1$
and $RW = 1$. According to Table 5.11, while some modules and some ports are added, the complexities of all CoreSymphony modules are equal to that of a 2-way out-of-order core.
Table 5.1 Hardware complexity of CoreSymphony.

<table>
<thead>
<tr>
<th>Module</th>
<th>Function</th>
<th>CoreSymphony</th>
<th>2-way OoO</th>
<th>8-way OoO</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-$</td>
<td>(1) Fetch</td>
<td>2 inst/cycle</td>
<td>2 inst/cycle</td>
<td>8 inst/cycle</td>
</tr>
<tr>
<td></td>
<td>(1) Fetch</td>
<td>2 inst/cycle</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Local I-$</td>
<td>(1) Rename(Src)</td>
<td>4 Read</td>
<td>4 Read</td>
<td>16 Read</td>
</tr>
<tr>
<td></td>
<td>(2) Rename(Dst, Local)</td>
<td>2 Write</td>
<td>2 Write</td>
<td>8 Write</td>
</tr>
<tr>
<td></td>
<td>(3) Rename(Dst, Remote)</td>
<td>RW Write</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(4) Release(Local)</td>
<td>2 Read</td>
<td>2 Read</td>
<td>8 Read</td>
</tr>
<tr>
<td></td>
<td>(5) Release(Remote)</td>
<td>RW Read</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LRMT</td>
<td>(1) Rename(Src)</td>
<td>4 Read</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(2) Rename(Dst)</td>
<td>1 Write</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(3) State check</td>
<td>3BW Read</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>I-window</td>
<td>(1) Dispatch</td>
<td>2 inst/cycle</td>
<td>2 inst/cycle</td>
<td>8 inst/cycle</td>
</tr>
<tr>
<td></td>
<td>(2) Wakeup(Ltag CAM)</td>
<td>2 Search</td>
<td>2 Search</td>
<td>8 Search</td>
</tr>
<tr>
<td></td>
<td>(3) Wakeup(Gtag CAM)</td>
<td>3BW Search</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(4) Wakeup(Recovery)</td>
<td>2 Search</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(5) Issue</td>
<td>2 inst/cycle</td>
<td>2 inst/cycle</td>
<td>8 inst/cycle</td>
</tr>
<tr>
<td>LROB</td>
<td>(1) Allocate</td>
<td>2 entry/cycle</td>
<td>2 entry/cycle</td>
<td>8 entry/cycle</td>
</tr>
<tr>
<td></td>
<td>(2) Update(Local)</td>
<td>2 Write</td>
<td>2 Write</td>
<td>8 Write</td>
</tr>
<tr>
<td></td>
<td>(3) Update(Remote)</td>
<td>3 Write</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(4) Release</td>
<td>2 entry/cycle</td>
<td>2 entry/cycle</td>
<td>8 entry/cycle</td>
</tr>
<tr>
<td>GROB</td>
<td>(1) Allocate</td>
<td>1 entry/cycle</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(2) Update(Local)</td>
<td>1 Write</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(3) Update(Remote)</td>
<td>3 Write</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(4) Release</td>
<td>1 entry/cycle</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LCB</td>
<td>(1) Allocate</td>
<td>2 entry/cycle</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(2) Wakeup(CAM)</td>
<td>1 Search</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(3) Release</td>
<td>2 entry/cycle</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RCB</td>
<td>(1) Allocate</td>
<td>RW entry/cycle</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(2) Wakeup(CAM)</td>
<td>1 Search</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(3) Release</td>
<td>RW entry/cycle</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PRF</td>
<td>(1) Opfetch</td>
<td>4 Read</td>
<td>4 Read</td>
<td>16 Read</td>
</tr>
<tr>
<td></td>
<td>(2) Writeback(Local)</td>
<td>2 Write</td>
<td>2 Write</td>
<td>8 Write</td>
</tr>
<tr>
<td></td>
<td>(3) Writeback(Remote)</td>
<td>RW Write</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(4) Commit</td>
<td>2 Read</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LRF</td>
<td>(1) Opfetch</td>
<td>4 Read</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(2) Commit or Construct</td>
<td>2 Write</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(3) Construct</td>
<td>2 Read</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LSQ</td>
<td>Allocate</td>
<td>-</td>
<td>2 inst/cycle</td>
<td>8 inst/cycle</td>
</tr>
<tr>
<td></td>
<td>(1) Dispatch(Local)</td>
<td>1 inst/cycle</td>
<td>1 inst/cycle</td>
<td>4 inst/cycle</td>
</tr>
<tr>
<td></td>
<td>(2) Dispatch(Remote)</td>
<td>1 inst/cycle</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td>1 inst/cycle</td>
<td>1 inst/cycle</td>
<td>4 inst/cycle</td>
</tr>
<tr>
<td>D-$</td>
<td>(1) Load/Store</td>
<td>1 inst/cycle</td>
<td>1 inst/cycle</td>
<td>4 inst/cycle</td>
</tr>
</tbody>
</table>
Chapter 6

Verification

In this chapter, we verify the implementation of CoreSymphony by Verilog simulator and FPGA.

6.1 Verification on Simulation

6.1.1 Verification Method

We simulate CoreSymphony by using Icarus Verilog. Then, we compare results of the simulation to results obtained by software-based simulator and verify implementation of CoreSymphony.

Verification targets are the following two points:

- **PC trace**: Trace of program counters of executed instructions.
- **RSLT trace**: Trace of logical destination register numbers and written results.

By PC trace, we verify whether control is correct. By RSLT trace, we verify whether program is properly executed.

Verification flow is shown as follow:

1. Generating answer traces that show the process of correct execution by using software-based simulator.
2. Generating log files by using Icarus Verilog.
3. Converting log files to traces as we can compare them to answer traces.
4. Taking difference between both traces.
6.1.2 Generating Answer Traces by Software-based Simulator

For generating answer traces that show the process of correct execution, we use a cycle-level software-based simulator, SimMips [26]. Assuming that test program is Listing 6.1, about execution from the start to the end of the program, we generate traces shown in Listing 6.2 and Listing 6.3.

Listing. 6.1 Program for test on simulation.

```assembly
.text
.set noat
.globl main
.ent main
main :
   li $1, 1
   li $2, -1
   add $3, $1, $2
   srl $4, $2, 1
   add $5, $4, $1
   add $6, $6, $1
   addi $7, $2, 1
   addi $8, $4, 1
   addu $9, $2, $1
   addu $10, $4, $1
   addu $11, $11, $1
   addiu $12, $2, 1
   addiu $13, $4, 1
   addiu $14, $14, 1
   j 0
   nop
.end main
```

Listing. 6.2 Answer trace of program counter.

```
00000008
0000000c
00000010
00000014
00000018
0000001c
00000020
00000024
00000028
0000002c
00000030
00000034
00000038
0000003c
```

Listing. 6.3 Answer trace of logical destination register and result.

```
01 0000001
02 ffffffff
```
### 6.1 Verification on Simulation

We generate log files by using Icarus Verilog. We show a log file of 4-core fusion in Listing 6.4. One line shows an instruction retire. From the beginning of each line, the time of commit, core-id of the core where the instruction is steered, *FBID* of the FB that contains the instruction, slot number in the FB that contains the instruction, PC of the instructions, logical destination register number of the instruction, result of the instruction are shown.

#### Listing 6.4 Execution Log File.

```plaintext
1 VCD info: dumpfile test_csfusion.vcd opened for output.
2 23 branch to zero. stop : core0
3 24 branch to zero. stop : core1
4 25 branch to zero. stop : core2
5 26 branch to zero. stop : core3
6 27 ## 43420.000000[ns] commit [00000000000] exception [00000000000] : core=0 : fbid =00000000000 : slot=00 : pc=00000008 : ldst(01) => 00000001
7 28 ## 43420.000000[ns] commit [00000000000] exception [00000000000] : core=0 : fbid =00000000000 : slot=00 : pc=0000000c : ldst(02) => ffffffff
8 29 ## 43420.000000[ns] commit [00000000000] exception [00000000000] : core=1 : fbid =00000000000 : slot=00 : pc=00000008 : ldst(01) => 00000001
9 30 ## 43420.000000[ns] commit [00000000000] exception [00000000000] : core=1 : fbid =00000000000 : slot=01 : pc=0000000c : ldst(02) => ffffffff
10 31 ## 43420.000000[ns] commit [00000000000] exception [00000000000] : core=2 : fbid =00000000000 : slot=00 : pc=00000008 : ldst(01) => 00000001
12 33 ## 43420.000000[ns] commit [00000000000] exception [00000000000] : core=3 : fbid =00000000000 : slot=00 : pc=00000008 : ldst(01) => 00000001
13 34 ## 43420.000000[ns] commit [00000000000] exception [00000000000] : core=3 : fbid =00000000000 : slot=01 : pc=0000000c : ldst(02) => ffffffff
14 35 ## 43420.000000[ns] commit [00000000000] exception [00000000000] : core=0 : fbid =00000000000 : slot=02 : pc=00000010 : ldst(03) => 00000000
15 36 ## 43420.000000[ns] commit [00000000000] exception [00000000000] : core=0 : fbid =00000000000 : slot=03 : pc=00000014 : ldst(04) => 7fffffff
19 ## 43440.000000[ns] commit [00000000000] exception [00000000000] : core=2 : fbid =00000000000 : slot=05 : pc=0000001c : ldst(06) => 00000001
```
## 6.1.4 Converting Log File to PC/RSLT trace

On CoreSymphony, processor’s state is updated by instructions retiring from LCB. Therefore, about instruction sequence retiring from LCB, we output PCs, logical destination register numbers, and results. However, because some instructions are copied by Leaf-node Steering, the same instructions retire from some LCBs. To avoid this, we should get rid of copied instructions from log files for comparing them to answer traces.

In other case, due to bias in the steering result, difference may occur in execution speed over fused cores. Therefore, we should sort the instruction retire sequence in program order.

In Listing 6.4, copied instructions by Leaf-node Steering retire from some cores. For example, the instruction of PC=00000008 is steered to all four cores. In this case, because all copied instructions write correct data into correct logical register, we should get rid of three copied instructions leaving the one instruction. If some copied instructions produce different results, in order to debug it, we do not get rid of them.

Furthermore, in Listing 6.4, instructions do not always retire in program order over fused cores. For example, the instruction of PC=00000030 executed in core3 retire earlier than the instruction of PC=00000024 executed in core1 do. In order to compare log file to answer trace, we should sort the instruction sequence.

For example, we can extract traces from log file as shown in Listing 6.5. By `sort` command, we sort instruction retire sequence in program order. By `-u` option of `sort` command, we get rid of copied instructions that produce the same result.

Listing. 6.5 Shell script to convert log file to traces.

```bash
#!/bin/bash
CMDNAME=`basename $0`
if [ $# -ne 1 ]; then
  echo "Usage: $CMDNAME logfile" 1>&2
```
6.2 Verification on FPGA

We verify CoreSymphony on an evaluation board, VC707, manufactured by Xilinx. Target FPGA on VC707 is Virtex-7. We get CoreSymphony to work on core configuration of 1-core, 2-core, 3-core, and 4-core individually. We decide that the frequency is all 20MHz.

Test program for verification is shown in Listing 6.6. This is a night rider program that turn on LEDs on the board in rotation. For turning on LED, we use memory mapped I/O. On this verification, we decide that LED is mapped to address 0.

As a result of verification, CoreSymphony work correctly on all core configurations.

Listing 6.6 Program for test on FPGA.

```
.text
.globl main
.ent main
main :
  li $3, 1
  li $4, 2
  li $5, 4
  li $6, 8
  li $7, 16
  li $8, 32
  li $9, 64
  li $10, 128
  li $31, 0
$loop :
  sw $3, 0($0)
jal $wait
sw $4, 0($0)
jal $wait
sw $5, 0($0)
```

6.1.5 Taking Difference between Anser Traces and Converted Log Files

By comparing PC trace and RSLT trace extracted from log file to answer trace, we verify the implementation of CoreSymphony. If traces does not correspond, there are bugs in the implementation.
jal $wait
sw $6, 0($0)
jal $wait
sw $7, 0($0)
jal $wait
sw $8, 0($0)
jal $wait
sw $9, 0($0)
jal $wait
sw $10, 0($0)
jal $wait
sw $9, 0($0)
jal $wait
sw $8, 0($0)
jal $wait
sw $7, 0($0)
jal $wait
sw $6, 0($0)
jal $wait
sw $5, 0($0)
jal $wait
sw $4, 0($0)
jal $wait
j $loop
$wait :
li $11, 1000000
$waitloop :
nop	nop	nop
nop
bnez $11, $waitloop
jr $31
.end main
Chapter 7
Evaluation

In this chapter, we evaluate performance of CoreSymphony by software based simulator and Xilinx ISE.

7.1 Evaluation on Software-based Simulator

7.1.1 Evaluation Environment

We used SimMips and SPEC2006 benchmarks to evaluate CoreSymphony. The baseline processor configuration is shown in Table 7.1. Latencies and initiation intervals for functional units are configured in the same manner as MIPS R10000[27].

7.1.2 Performance Improvement by Fusion

Figure 7.1 shows IPC improvement obtained by fusion and branch prediction accuracy improvement obtained by fusion on each benchmark. 1-core, 2-core, 3-core and 4-core respectively mean 1-core execution, 2-core fusion, 3-core fusion and 4-core fusion.

The evaluation results show that the IPC of CoreSymphony is improved by fusion. On the harmonic mean of all benchmarks, 2-core fusion achieves 22%, 3-core fusion achieves 36% and 4-core fusion achieves 43% higher IPC than 1-core execution. On gobmk showing the highest IPC improvement, 2-core fusion achieves 34%, 3-core fusion achieves 90% and 4-core fusion achieves 97% higher IPC than 1-core execution.

As discussed in Section 5.1.1, we are able to improve branch prediction accuracy by fusion. With the exception of milc, on almost all benchmarks, branch prediction accuracy is improved by fusion. On hmean of all benchmarks, branch prediction accuracy on 4-core fusion is 1.9%

\[\text{A total of 128 entries of ULB-LSQ is required to avoid deadlock. It is possible to reduce the number of entries without performance degradation by adding a flow control mechanism, for example [28].}\]
Table 7.1  Baseline processor configuration.

<table>
<thead>
<tr>
<th>(1) Functional Units</th>
<th>2 iALU, 1 LD/ST, 1 fpALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2) Instruction Window</td>
<td>32/32 entries for INT/FP</td>
</tr>
<tr>
<td>(3) Local Re-order Buffer</td>
<td>32 entries</td>
</tr>
<tr>
<td>(4) Global Re-order Buffer</td>
<td>8 entries</td>
</tr>
<tr>
<td>(5) Local Commit Buffer</td>
<td>16 entries</td>
</tr>
<tr>
<td>(6) Remote Commit Buffer</td>
<td>32 entries</td>
</tr>
<tr>
<td>(7) Physical Register File</td>
<td>64/64 entries for INT/FP, 18/18 entries NRP</td>
</tr>
<tr>
<td>(8) Load/Store Queue</td>
<td>128 entries ULB-LSQ</td>
</tr>
<tr>
<td>(9) Memory Disambiguation</td>
<td>1K entries LWT</td>
</tr>
<tr>
<td>(10) Branch Prediction</td>
<td>6 bit GHR, 2K entries TMP</td>
</tr>
<tr>
<td>(11) Branch Target Buffer</td>
<td>1K entries, 2-way</td>
</tr>
<tr>
<td>(12) L1-D$</td>
<td>16KB, 2-way, 1 cycle</td>
</tr>
<tr>
<td>(13) L1-I$(Conventional)</td>
<td>8KB, 2-way, 1 cycle</td>
</tr>
<tr>
<td>(14) L1-I$(Local)</td>
<td>8KB, 4-way, 2 cycle</td>
</tr>
<tr>
<td>(15) Shared L2$</td>
<td>2MB, 4-way, 10 cycle</td>
</tr>
<tr>
<td>(16) Main Memory</td>
<td>100 cycle</td>
</tr>
<tr>
<td>(17) Inter-core Latency</td>
<td>1 cycle</td>
</tr>
<tr>
<td>(18) Broadcast Width(BW)</td>
<td>1/1 for INT/FP</td>
</tr>
<tr>
<td>(19) Remote write back Width(RW)</td>
<td>1/1 for INT/FP</td>
</tr>
</tbody>
</table>

Fig. 7.1  Performance improvement obtained by fusion. IPC improvement (left axis). Branch prediction accuracy improvement (right axis).

higher than one on 1-core execution.

7.1.3  Comparison of Steering Algorithms

Figure 7.2 shows the relationship between the steering algorithm and performance. The left axis shows the relative IPC based on 1-core Leaf-node Steering for three algorithms, Modulo-2 Steering, Dependency-based Steering and Leaf-node Steering. In the graph, the right axis shows the average instruction count for the same FB. This instruction count includes redun-
Comparison of three steering algorithms. Relative IPC based on 1-core, Leaf-node (left axis). Remote write back Width (RW) sensitivity. Relative IPC based on 1-core, RW 5.2

Broadcast Width (BW) sensitivity. Relative IPC based on 1-core, BW achieves 1.6% higher IPC than Dependency-based Steering.

(d) Steered instructions on Leaf-node Steering.

Based on these results, despite exceptions such as mcf and hmmer, Leaf-node Steering achieves the highest performance for most of the benchmarks. On hmean of all benchmarks, on 4-core fusion, Leaf-node Steering achieves 11% higher IPC than Modulo-2 Steering, and achieves 1.6% higher IPC than Dependency-based Steering.

As discussed in Section 5.2, Leaf-node Steering increases instruction count, because it copies instructions to some cores. The evaluation results shows that the increase in the instruction count ratio is 5.9% on 2-core fusion, 6.0% on 3-core fusion and 10% on 4-core
fusion. For realizing 2-way Renaming, the inter-core data dependency must be eliminated. It is a positive result that Leaf-node Steering satisfies this requirement and achieves the highest performance among three algorithms.

7.1.4 Broadcast Width Sensitivity

Figure 7.3 shows the effect of $BW$. We evaluate the relative IPC based on 1-core execution about two cases, proposal ($BW = 1$) and ideal ($BW = 2$). For both cases, let $RW = 1$. In the same graph, we show the instruction ratio of listed below: (a) Instructions that have no destination; (b) Instructions whose results are over written by instruction in the same FB; and (c) Instructions whose results are broadcasted to other cores. On (a) and (b), we are able to filter the broadcast.

The evaluation results show that two cases are almost the same. The ratio of filtered instructions is 44% on 2-core fusion, 48% on 3-core fusion and 51% on 4-core fusion, on average of all benchmarks. The ratio of filtered instructions increases by fusion. This is the effect of (b). The number of instructions whose results are over written in the same FB increases, because the number of instructions in FB increases by fusion. Therefore, the ratio of filtered instructions increases.

As a result, $BW = 1$ is sufficient for performance. $BW$ doesn’t depend on the number of
fused cores.

7.1.5 Remote Write back Width Sensitivity

Figure 7.4 shows the effect of RW. We evaluate the relative IPC based on 1-core execution about two cases, proposal ($RW = 1$) and ideal ($RW = 6$). For both cases, let $BW = 2$. In the same graph, we show the instruction ratio of listed below: (a) Instructions that doesn’t write back; (b) Instructions that write back for already dispatched instructions in instruction widow; (c) Instructions that write back for instructions in future; and (d) Instructions that write back for both cases.

The evaluation results show that two cases are almost the same. The ratio of filtered instructions is 57% on 2-core fusion, 69% on 3-core fusion and 72% on 4-core fusion, on average of all benchmarks. The portion of (b) decreases and the ratio of filtered instructions increase.

On milc, $RW = 6$ show the lowest performance on 4-core fusion. This is the effect of pipeline flush when there is no PRF entry. The large number of $RW$ leads to many pipeline flushes. Therefore, performance of $RW = 6$ degrades.

As a result, $RW = 1$ is sufficient for performance.

7.1.6 Re-order Buffer Distribution Overhead

Figure 7.5 shows the effect of ROB distribution. We compare the case that ROB is divided to two lightweight ROBs by ROB distribution (proposal) to the case that ROB has sufficient ports for update (ideal). We show the relative IPC based on 1-core execution, ROB distribution. ROB size of the latter case is equal to the sum of GROB, LROB, LCB and RCB shown in Table 7.1. In order to find out only the effect of ROB size, we decided that instruction window and PRF has sufficient entries. Thus, the dispatch stage stalls only when there is no ROB entry.

The evaluation results show that ROB distribution improves IPC on milc and lbm. On 4-core fusion, distribution achieves 6.5% higher IPC on milc and 15% higher IPC on lbm than non-distribution does. As discussed in Section 5.5, distributing ROB, we are able to expand ROB by fusion and increase the number of in-flight instructions. Therefore, these benchmarks are strongly affected by the number of in-flight instructions. On other benchmarks, ROB distribution degrades IPC. On sphinx3 showing the most IPC degradation, distribution achieves 15% lower IPC than non-distribution does on 4-core fusion. These benchmarks are strongly affected by the increase in pipeline stages.

On the harmonic mean of all benchmarks, the decreasing rate of distribution is 8.3% on 1-core, 11% on 2-core fusion, 7.1% on 3-core fusion and 4.6% on 4-core fusion. However,
ROB distribution realizes the reduction of ROB’s ports and improves performance on some benchmarks. Therefore, this IPC degradation should be considered within an allowance.

### 7.1.7 In-order State Distribution Overhead

Figure 7.6 shows the effect of in-order state distribution. We compare the case that LRF has two write ports (proposal) to the case that LRF has sufficient ports for update (ideal), and evaluate the relative IPC based on 1-core execution.

The evaluation results show that in-order state distribution has a small effect on performance. On milc showing the most IPC degradation, the decreasing rate of distribution is 5.3% on 4-core fusion. On the harmonic mean of all benchmarks, the decreasing rate is 1.0% on 2-core fusion, 1.6% on 3-core fusion and 1.1% on 4-core fusion. This IPC degradation is within an allowance compared to the advantage that in-order state distribution realizes the reduction of LRF ports.

### 7.2 Evaluation on Xilinx ISE

#### 7.2.1 Evaluation Environment

We use Xilinx ISE 14.3 for logic synthesis. We set target devise to VC707, which is an evaluation board manufactured by Xilinx. Target FPGA is xc7vx485t.

#### 7.2.2 Register Map Table

We show FPGA resources of RMT in Figure 7.7. For comparison, we also show FPGA resources of RMT of 2-way out-of-order and RMT of 8-way out-of-order. We show the detailed resources of RMT of CoreSymphony in Figure 7.8. In the following, assuming that the number of logical registers is 32. Also, it is necessary to consider dependencies between instructions that are renamed at the same cycle. However, for simplicity, we do not contain the logic that solves the dependencies.

According to Figure 7.7, compared to RMT of 2-way, renaming logic of CoreSymphony consumes about three times of FFs and LUTs. Compared to RMT of 8-way, use of LUT can be reduced by 40 percent. If CoreSymphony does not use 2-way Renaming, all cores need RMT of 8-way. Therefore, 2-way Renaming is a valuable approach.

According to Figure 7.8, GRMT is smaller than LRMT. Because content of GRMT is shared in all fused cores, GRMT is desired to be small. Therefore, 2-way Renaming is efficient.
7.2 Evaluation on Xilinx ISE

Table 7.2 Number of entries of GROB, LCB, and RCB against number of entries of LROB.

<table>
<thead>
<tr>
<th>Number of LROB entries</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of GROB entries</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Number of LCB entries</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Number of RCB entries</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
</tbody>
</table>

7.2.3 Physical Register File/Logical Register File

We show FPGA resources of PRF and LRF in Figure 7.9. For comparison, we also show FPGA resources of 4R8W register file. About all register files, we set the number of entries to 64.

Originally, PRF and LRF are 4R8W memories. However, PRF can be implemented with 6R3W by PRF distribution, and LRF can be implemented with 6R2W by in-order state distribution. This can reduce use of BRAMs. This also can reduce use of FFs and LUTs, because the number of ports and bit width of an entry of LVT is reduced.

7.2.4 Instruction Window

We show FPGA resources of instruction window in Figure 7.10. For comparison, we also show FPGA resources of instruction window of 2-way and instruction window of 8-way. We also change the number of entries to 32, 64, and 128.

According to Figure 7.10, instruction window of CoreSymphony needs more FFs and LUTs than that for 8-way. This is because an entry is extended for 2-way Renaming, and CAM needed for PRF distribution and in-order state distribution increases.

We can decrease use of BRAMs by PRF distribution and in-order state distribution. However, many FPGA resources are needed compared to instruction window of 2-way. It is necessary to reduce resources for more efficient implementation.

7.2.5 Re-order Buffer Distribution

We show FPGA resources of ROB in Figure 7.11. For comparison, we also show FPGA resources of ROB of 2-way and ROB of 8-way. We also change the number of entries to 32, 64, and 128. About ROB of CoreSymphony, the number of entries in Figure 7.11 is assumed to be the number of entries of LROB, and we change the number of entries of GROB, LCB, and RCB as shown in Table 7.2. FPGA resources of ROB of CoreSymphony are the sum of FPGA resources of GROB, LROB, LCB, and RCB. We show the detailed resources of ROB.
of CoreSymphony in Figure 7.12.

According to Figure 7.11, compared to ROB of 2-way, ROB of CoreSymphony consumes about four to five times of FFs and LUTs. This is because three update ports are added to LROB for broadcasted instructions, and new modules, such as GROB, LCB, and RCB, are added. Compared to ROB of 8-way, though FFs increase, we can decrease use of LUTs and BRAMs significantly. This is considered to be the effect by dividing a large and many ported ROB into some small and few ported ROBs. If we do not distribute ROB, all cores need ROB of 8-way. Though ROB of CoreSymphony consumes more resources than ROB of 2-way, ROB distribution is a valuable approach.

According to Figure 7.12, GROB is smaller than LROB. Because content of GROB is shared in all fused cores, GROB is desired to be small. Therefore, ROB distribution is efficient.

7.2.6 Total Resources

We show the comparison of FPGA resources between baseline processor and CoreSymphony in Figure 7.13. We show the detailed resources of core of CoreSymphony. Core of CoreSymphony is divided into the following six units:

- **Instruction Fetch Unit (IFU)**  Local I-$, conventional I-$, TMP, and BTB.
- **Instruction Decode Unit (IDU)**  Decoder.
- **Steering Unit (STU)**  Steering mechanism.
- **Register Renaming Unit (RRU)**  GRMT, LRMT, and free list.
- **Execution Unit (EXU)**  ALU, AGU, Mul/Div, PRF, and LRF.
- **Load/Store Unit (LSU)**  ULB-LSQ and D-$.
- **Retirement Unit (RTU)**  GROB, LROB, LCB, RCB, and LRF manager.

Others in Figure 7.13 is mainly inter-core network, including remote write back queue.

According to Figure 7.13, compared to baseline 2-issue out-of-order processor, CoreSymphony consumes 3.6 times of FFs, 5.6 times of LUTs, and 3.7 times of BRAMs. This is due to the effect of added modules such as Local I-$, STU and GRMT, and modified modules such as LRMT, instruction window and LROB. Reduction of resources is one of the future work.

Many FFs and LUTs are used in EXU. This is due to the effect of extended tag arrays and appended comparator. Instead of increase of FFs and LUTs, we can save use of BRAMs in EXU.

Though many BRAMs are used in IFU, the BRAMs are mainly used in Local I-$$. This is due to the effect that we set line size of Local I-$ to 16 words, and utilization efficiency of
Local I-$ decreases. More efficient implementation of Local I-$ is one of the future work.

### 7.2.7 Maximum Frequency

Maximum frequency of CoreSymphony is about 71MHz. Minimum period is 13.993ns, and completion of GROB is on critical path. Maximum frequency of baseline core is about 100MHz. According to this, we found that modules that are appended and modified for realizing CoreSymphony affect frequency. In order to improve performance of CoreSymphony, we need to tune these logic.
Fig. 7.7  FPGA resources of RMT. RMT of CoreSymphony contains GRMT and LRMT.

Fig. 7.8  Detailed resources of RMT of CoreSymphony.
7.2 Evaluation on Xilinx ISE

Fig. 7.9 FPGA resources of PRF and LRF.

Fig. 7.10 FPGA resources of instruction window.
Fig. 7.11  FPGA resources of ROB. ROB of CoreSymphony contains GROB, LROB, LCB, and RCB.

Fig. 7.12  Detailed resources of ROB of CoreSymphony.
7.2 Evaluation on Xilinx ISE

Fig. 7.13 FPGA resources of a core.

Fig. 7.14 Detailed resources of a core of CoreSymphony.
Chapter 8

Conclusion

In this paper, we designed and implemented efficient and realistic CoreSymphony architecture. CoreSymphony aims at balancing single-thread performance and multi-thread performance on CMPs. It can fuse some narrow-issue cores into a single wide-issue virtual core and accelerate single-thread execution. It can also split a single wide-issue virtual core into some narrow-issue cores and accelerate multi-thread execution.

In order to realize CoreSymphony, there are some challenges. We clarified these challenges and designed efficient CoreSymphony architecture to overcome them. Then, we implemented CoreSymphony by Verilog HDL and run it on FPGA.

According to evaluation results, 4-core fusion achieves 43% higher IPC on the harmonic mean and 97% higher IPC at a maximum than 1-core execution. Compared to baseline processor, CoreSymphony consumes 3.6 times of FFs, 5.6 times of LUTs, and 3.7 times of BRAMs. CoreSymphony is capable of operating at about 71MHz.

Future work is as follows:

- Reducing use of FPGA resources.
- Improving maximum frequency by refining logic.
- Considering the effective method for fusion/split during execution.

Use of FPGA resources of CoreSymphony is larger than that of baseline processor. In order to enhance the usefulness, it is necessary to reduce the use of resources. Maximum frequency of CoreSymphony is lower than that of baseline processor. In order to improve performance, we should refine logic of CoreSymphony. There is some overhead when CoreSymphony fuses/splits cores. In order to realize higher performance than heterogeneous multi-core architecture, we must consider the effective method that fuses and splits cores during execution.
Acknowledgements

I would like to express my sincere gratitude to my supervisor, Associate Professor Kenji Kise for giving me a chance to work in his laboratory. I am deeply grateful to all members of Kise Laboratory. I especially would like to thank Yoshito Sakaguchi for his valuable comments and cooperation on implementation. I would also like to thank Takayuki Matsumura for his useful ideas on architecture design.
Bibliography


Publications

International Conferences


Oral Presentations


2011.


Awards

