IMPROVING MULTITHREADING PERFORMANCE FOR CLUSTERED VLIW ARCHITECTURES

Manoj Gupta

Barcelona, 2013

Advisors
Fermín Sánchez
Josep Llosa

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor per la UPC

Department of Computer Architecture
Technical University of Catalonia

Abstract

Very Long Instruction Word (VLIW) processors are very popular in embedded and mobile computing domain. Use of VLIW processors range from Digital Signal Processors (DSPs) found in a plethora of communication and multimedia devices to Graphics Processing Units (GPUs) used in gaming and high performance computing devices. The advantage of VLIWs is their low complexity and low power design which enable high performance at a low cost. Scalability of VLIWs is limited by the scalability of register file ports. It is not viable to have a VLIW processor with a single large register file because of area and power consumption implications of the register file.

Clustered VLIW solve the register file scalability issue by partitioning the register file into multiple clusters and a set of functional units that are attached to register file of that cluster. Using a clustered approach, higher issue width can be achieved while keeping the cost of register file within reasonable limits. Several commercial VLIW processors have been designed using the clustered VLIW model.

VLIW processors can be used to run a larger set of applications. Many of these applications have a good Lnstruction Level Parallelism (ILP) which can be efficiently utilized. However, several applications, specially the ones that are control code dominated do not exibit good ILP and the processor is underutilized. Cache misses is another major source of resource underutiliztion. Multithreading is a popular technique to improve processor utilization. Interleaved MultiThreading (IMT) hides cache miss latencies by scheduling a different thread each cycle but cannot hide unused instructions slots. Simultaneous MultiThread (SMT) can also remove ILP under-utilization by issuing multiple threads to fill the empty instruction slots. However, SMT has a higher implementation cost than IMT.

The thesis presents Cluster-level Simultaneous MultiThreading (CSMT) that supports a limited form of SMT where VLIW instructions from different threads are merged at a cluster-level granularity. This lowers the hardware implementation cost to a level comparable to the cheap IMT technique. The more complex SMT combines VLIW instructions

at the individual operation-level granularity which is quite expensive especially in for a mobile solution. We refer to SMT at operation-level as OpSMT to reduce ambiguity. While previous studies restricted OpSMT on a VLIW to 2 threads, CSMT has a better scalability and upto 8 threads can be supported at a reasonable cost.

The thesis also proposes several other techniques to further improve CSMT performance. One particular approach, Cluster renaming remaps the clusters used by instructions of different threads to reduce resource conflicts. Cluster renaming is quite effective in reducing the issue-slots under-utilization and significantly improves CSMT performance. The thesis also covers several other approaches e.g. heterogeneous merging where instructions from some threads are merged at operation-level (OpSMT) and some at cluster-level (CSMT). Number of threads merged using either approach is decided by the maximum allowed complexity (gate delays or transistor count etc.). The heterogeneous merging approach allows for a performance better than CSMT but without the full overhead of OpSMT.

The next discussed approach is a split-issue based approach where a VLIW instruction can be split and issued in multiple cycles instead of having to issue as an unit. Splitting a VLIW instruction allows more flexibility in issuing the instruction and further improves the performance.

The thesis discusses another hybrid approach that combines the best of IMT with CSMT. In this hybrid approach, multiple thread contexts are maintained and only the threads that the merging hardware can support are used in instruction merging. Maintaining more threads mean that if some threads are blocked because of cache misses, those can be replaced by other threads. As a result, a full pool of threads if available for merging and resource under-utilization is reduced.

Overall, CSMT and the other discussed orthogonal approaches can achieve a performance that is close to the more complex approach of merging instructions from different threads at operation-level but at a cost comparable to the simple IMT technique.

Acknowledgements

I am greatly indebted to my supervisors Prof. Josep Llosa and Prof. Fermín Sánchez for their invaluable technical guidance and moral support. I would like to thank my other PhD colleagues Rakesh Ranjan, Indu Bhagat, Abhishek Deb, Miquel Moreto, Tanausu Ramirez and Govind SreekarShenoy for their cooperation, support and the great informal discussions. I would also like to thank the staff of Department of Computer Architecture at UPC Barcelona, in particular Trini, for their help. The work done in the thesis was supported by the Spanish Ministry of Science and Technology under contract CICYT TIN2007-60625, FI grant from AGAUR/Generalitat de Catalunya, SARC (Scalable computer ARChitecture) Project and HiPEAC European Network of Excellence and I would like to thank all of them for being enabler of my work. The thesis is dedicated to my parents who showed immense patience and provided me great moral support during the coarse of my work.

Contents

Abstract i

Acknowledgements iii

Index viii

List of Figures ix

List of Tables xiii

1 Introduction 1

1.1 VLIW Processors 4
1.2 Scalability Issues in VLIW Processors 9
1.3 Clustered VLIW Processors 10
1.4 MultiThreading 12
1.5 Contributions of the Thesis 17
1.6 Thesis Organization 19

2 Experimental Platform 21

2.1 VEX Architecture 21
2.2 VEX Toolchain 24
2.3 Simulating a MultiThreaded Processor 25
2.3.1 Baseline Architecture Configuration 25
2.3.2 Microarchitectural changes for multithreading 27
2.4 Benchmarks and Workloads 28
2.4.1 Workload execution 30
2.4.2 Thread Selection Policy 31
2.5 Summary 31

3 Cluster-level Simultaneous MultiThreading (CSMT) 33
3.1 Introduction . . . 33
3.2 Cluster-level Simultaneous MultiThreading . . . 35
3.3 CSMT Performance Analysis . . . 40
3.4 Merging Hardware Complexity Analysis . . . 44
3.5 Summary and Conclusions . . . 45

4 Cluster Renaming 47
4.1 Motivation . . . 47
4.2 Cluster Renaming . . . 49
4.2.1 Cluster Renaming Example . . . 52
4.3 Results . . . 53
4.3.1 Cluster Usage Analysis . . . 54
4.3.2 Cluster Renaming Performance Improvements . . . 55
4.3.3 CSMT Performance Comparison With IMT . . . 57
4.3.4 CSMT Performance Comparison with OpSMT . . . 59
4.4 Summary and Conclusions . . . 60

5 Heterogeneous Merging: Using CSMT and OpSMT Merging Hardware Together 61
5.1 Motivation . . . 61
5.2 Heterogeneous Thread Merging Schemes . . . 65
5.3 Evaluation . . . 67
5.3.1 Performance Analysis . . . 68
5.4 Summary and Conclusions . . . 71

6 Cluster-level Split-Issue 73
6.1 Operation-level Split-Issue . . . 73
6.1.1 Dataflow hazards . . . 75
6.1.2 Implementation details . . . 76
6.2 Cluster-level Split-Issue . . . 78
6.3 Implementing Cluster-level Split-Issue . . . 81
6.4 Issues with cluster-level split-issue . . . 83
6.4.1 Exceptions/Interrupts . . . 84
6.4.2 Register file port contention . . . 86

6.4.3 Memory Port Contention . . . 87
6.4.4 Issues with Inter-cluster Communication . . . 88
6.5 Performance Evaluation . . . 89
6.6 Summary and Conclusions . . . 93

7 Hybrid MultiThreading 95
7.1 Motivation . . . 95
7.2 Hybrid MultiThreading . . . 97
7.2.1 Interleaving Step . . . 99
7.3 Thread Selection Schemes . . . 99
7.4 Results . . . 103
7.4.1 Performance Evaluation of HMT with FNB Policy . . . 105
7.4.2 Detailed Performance Evaluation of Thread Selection Schemes . . . 106
7.5 Summary and Conclusions . . . 108

8 Putting It All Together 111
8.1 Motivation . . . 111
8.2 Results . . . 114
8.3 Summary and Conclusions . . . 118

9 Conclusions and Future Work 119
9.1 Future Work . . . 121

Bibliography 123

A CSMT Cost Analysis 129
A.1 Introduction . . . 129
A.2 Thread Merge Hardware . . . 130
A.2.1 Priority Encode . . . 131
A.2.2 Priority Decode . . . 132
A.2.3 Thread Select Bits Computation . . . 133
A.3 Serial Logic . . . 134
A.3.1 Thread Select Bit block . . . 135
A.3.2 Usage Vector Computation block . . . 135
A.4 Parallel Logic . . . 137
A.5 Cost Analysis . . . 138

A.5.1 Serial Logic 139
A.5.1.1 Delay 139
A.5.1.2 Transistor Count 139
A.5.2 Parallel Logic 140
A.5.2.1 Transistor Count 140
A.6 Results 142

B OpSMT Cost Analysis **145**
B.1 Introduction 145
B.1.1 OpSMT Thread Merge Control 146
B.1.1.1 Serial logic 148
B.1.1.2 Parallel logic 150
B.1.1.3 Routing Computation 150
B.2 Cost Analysis 151
B.2.1 Serial Logic 151
B.2.2 Parallel Logic 152
B.2.3 Routing Computation 152
B.3 Results 155

List of Figures

1.1 **A 6-stage pipeline** . . . 1
1.2 **Execution on the 6-stage pipeline processor** . . . 2
1.3 **Data dependencies** . . . 3
1.4 **Scalar code vs VLIW code** . . . 4
1.5 **Instruction execution on an in-order, Superscalar and VLIW Processor** 5
1.6 **VLIW vs Superscalar Pipeline** . . . 6
1.7 **EQ and LEQ Execution** . . . 7
1.8 **Register file bit cell** . . . 8
1.9 **Bypass Network** . . . 9
1.10 **A 2-cluster VLIW Processor pipeline** . . . 11
1.11 **Implicit and explicit copy** . . . 12
1.12 **Comparison of Multithreading Schemes** . . . 14

2.1 **Example of a VEX Instruction** . . . 22
2.2 **Branching in VEX** . . . 23
2.3 **Inter-cluster communication in VEX** . . . 24
2.4 **Simulation flow** . . . 24
2.5 **Baseline Cluster 0 configuration** . . . 26
2.6 **VEX Pipeline** . . . 26
2.7 **Multithreaded workload execution** . . . 30

3.1 **Horizontal and Vertical Waste** . . . 34
3.2 **SMT Merging** . . . 36
3.3 **Instruction Merging in OpSMT and CSMT** . . . 37
3.4 **IMT and CSMT execution on a 4-thread 4-cluster architecture using round robin priority** . . . 37

3.5 **Cluster Usage Vector** . . . 39
3.6 **CSMT conflict checking for 2 threads** . . . 39
3.7 **Pipeline with extra stage for merging hardware** . . . 40
3.8 **CSMT performance** . . . 41
3.9 **Thread merging hardware transistor count for OpSMT and CSMT** . 43
3.10 **Thread merging hardware gate delays for OpSMT and CSMT** . . . 43

4.1 **Cluster usage** . . . 48
4.2 **Individual cluster use in mcf and djpeg** . . . 49
4.3 **Shift tables for a 4-cluster architecture** . . . 50
4.4 **Cluster renaming logic for a 4-cluster architecture** . . . 51
4.5 **CSMT execution on a 4-thread 4-cluster architecture using cluster renaming.** . . . 52
4.6 **Cluster usage with a perfect memory model** . . . 54
4.7 **Cluster usage with a real memory model** . . . 55
4.8 **Speedup in CSMT by use of cluster renaming** . . . 56
4.9 **Speedup in OpSMT by use of cluster renaming** . . . 56
4.10 **CSMT vs IMT, Perfect Memory** . . . 57
4.11 **CSMT vs IMT, Real Memory** . . . 58
4.12 **CSMT performance comparison with OpSMT** . . . 59

5.1 **OpSMT Performance** . . . 62
5.2 **OpSMT performance advantage over CSMT** . . . 63
5.3 **CSMT and OpSMT thread merge control area** . . . 63
5.4 **CSMT and OpSMT thread merge control delays** . . . 64
5.5 **Mixed Merging Example** . . . 65
5.6 **Possible Merging Schemes for 4 Threads** . . . 66
5.7 **Merging Hardware Cost** . . . 67
5.8 **Merging schemes performance** . . . 69
5.9 **Performance vs transistors incurred** . . . 70
5.10 **Performance vs gate delays** . . . 71

6.1 **Instruction Merging in OpSMT and CSMT** . . . 74
6.2 **Instruction Execution with Split-Issue** . . . 74
6.3 **Issues with Dataflow** . . . 75
6.4 **Working of Split-Issue** . . . 77

6.5 **Applicability of split-issue** . . . 78
6.6 **Operation-level and cluster-level split-issue with operation-level merging** . . . 79
6.7 **Cluster-level split-issue with cluster-level merging** . . . 81
6.8 **Merging Hardware** . . . 82
6.9 **Delaying updates to architectural state by using buffers** . . . 84
6.10 **Buffer organization** . . . 85
6.11 **Register File Organizations** . . . 86
6.12 **Memory port contention because of delayed memory writes** . . . 87
6.13 **Issuing inter-cluster communication operations** . . . 88
6.14 **Cluster-level split-issue (CCSI) speedups over CSMT** . . . 90
6.15 **Speedups obtained over OpSMT with cluster-level (COSI) and operation-level split-issue (OOSI)** . . . 91
6.16 **Average Performance of all multithreading techniques** . . . 92

7.1 **Average IPC for IMT, CSMT and OpSMT** . . . 96
7.2 **HMT Pipeline** . . . 98
7.3 **Hybrid Multithreading Example** . . . 98
7.4 **Static vs Dynamic Thread Selection** . . . 100
7.5 **Thread Selection Schemes** . . . 101
7.6 **Static Thread Selection Hardware** . . . 102
7.7 **General Dynamic Thread Selection Hardware** . . . 102
7.8 **IPC of the workloads for HMT with FNB policy** . . . 104
7.9 **Average Performance of Different Thread Selection Schemes for HCSMT** . . . 106
7.10 **Average Performance of Different Thread Selection Schemes for HOpSMT** 107

8.1 **All Schemes** . . . 112
8.2 **Merging Hardware** . . . 112
8.3 **All Merging Example** . . . 113
8.4 **Perfect memory Performance** . . . 115
8.5 **Real memory Performance** . . . 115
8.6 **Comparison with HCSMT with split-issue, Perfect Memory** . . . 116
8.7 **Comparison with HCSMT with split-issue, Real Memory** . . . 117

A.1 **Example of Cluster Usage Vector 1010** . . . 129

A.2 **Thread merge hardware** 130
A.3 **Bundle select logic (BS)** 131
A.4 **Thread select logic (TS)** 132
A.5 **Serial logic implementation for TS block** 133
A.6 **Thread select bit block (TSB)** 134
A.7 **Usage vector computation block (UV)** 135
A.8 **Parallel logic implementation for TS block** 136
A.9 **Validity computations for three thread combinations** 137
A.10 **Transistor Count for CSMT** 141
A.11 **Gate Delays for CSMT** 142

B.1 **CSMT Thread merge hardware** 145
B.2 **OpSMT Thread merge hardware** 146
B.3 **OpSMT Serial Logic** 147
B.4 **OpSMT Parallel Logic** 147
B.5 **OpSMT Resource Vector** 147
B.6 **OpSMT Resource Routing** 148
B.7 **ALU/Operation Count Check** 149
B.8 **Multiply Count Check** 149
B.9 **OpSMT Area** 153
B.10 **OpSMT Gate Delays** 154

List of Tables

2.1 **Architectural configuration** 27
2.2 **Benchmarks** 29
2.3 **4-Thread Workload configurations** 29
2.4 **8-Thread Workload configurations** 29

A.1 **Thread select bits for thread combinations** 136

Chapter 1

Introduction

Over the past decades, computers have made their way in all aspects of life and technology. Computers are present in a wide spectrum of products, not just personal computers or computing servers, ranging from handheld devices and digital electronics, automobiles to avionic systems, etc. There is an ever-growing need for higher computing power in any application domain. Processors are the computational brain and the core of any computing system. Hence it is imperative that techniques for improving the performance of processors continue being investigated.

One of the most basic techniques to achieve higher performance is instruction pipelining [21]. Instruction Pipelining splits the execution of an instruction into several independent stages called pipeline stages. The output of each pipeline stage is connected to the input of next stage. Division into stages allows the processor to be clocked at a much higher clock speed that is limited only by the slowest pipeline stage. For instance, if the execution of an instruction is divided into p stages of equal delay, the processor with a clock frequency of f can now be theoretically clocked at a frequency $p \times f$. As a result, the runtime of a program with N instructions on a processor with p pipeline stages can be as low as p-times compared to an non-pipelined processor. Pipelining allows the processor to handle multiple instructions at the same time and reduces the overall exe-

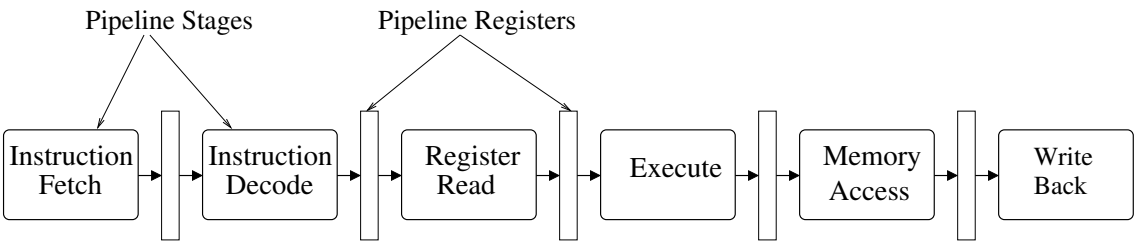


Figure 1.1: **A 6-stage pipeline**

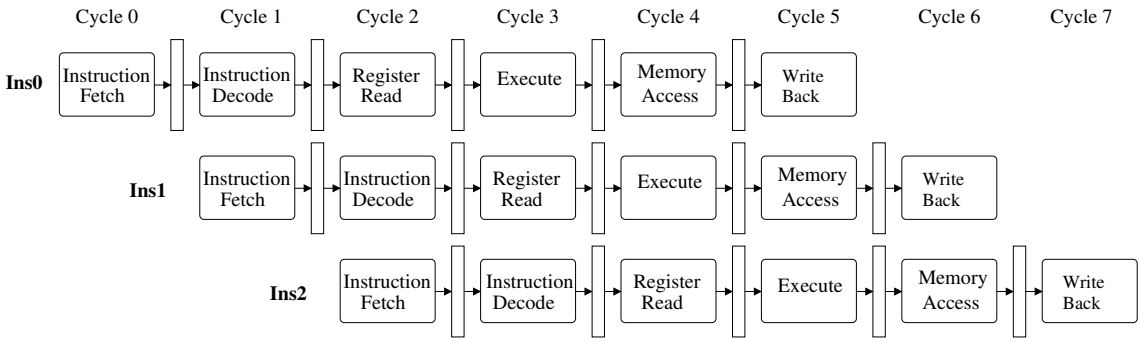


Figure 1.2: **Execution on the 6-stage pipeline processor**

cution time. Figure 1.1 shows a 6-stage pipeline viz. Instruction Fetch (IF), Instruction Decode (ID), Register Read (RR), Execute (EX), Memory Access (MA) and WriteBack (WB). The values passed from one pipeline stage to the next are placed in pipeline registers. Figure 1.2 shows instruction execution on the 6-stage pipelined processor already shown in Figure 1.1. In the execution shown in Figure 1.2, a new instruction is issued in the pipeline at every cycle. At cycle 0, instruction Ins0 is issued and enters the IF pipeline stage. At cycle 1, instruction Ins0 moves to ID pipeline stage and a new instruction Ins1 enters the IF pipeline stage, and so on. As there are six pipeline stages, up to six instructions are in-flight in the pipeline. Pipelining is a common feature in most current processors. For instance, Intel P5 microarchitecture (Pentium processor) had 5 pipeline stages (8 stages for floating point), Intel Prescott microarchitecture (Pentium 4 processor) had 31 pipeline stages, and the recently released Intel Atom Z500 processor had 16 pipeline stages. Note that while pipelining improves overall execution time of the program, the total time required to execute an individual instruction does not reduce by pipelining. In fact, individual instruction execution time worsens because:

- Dividing execution into pipeline stages of exactly equal delay is not possible.
- Pipeline registers required between the pipeline stages add to the total delay as well.

Note that for achieving peak performance, an instruction must be issued by the pipeline every cycle. However, to do so the new instruction must be independent of the instructions already in the pipeline i.e. the instruction should not depend on the results produced by the instructions in the pipeline. In the case that an independent instruction is not available, several cycles might be wasted before a new instruction can be inserted in the pipeline. Most processors employ a technique known as data forwarding to reduce the delays caused because of the data dependencies. Data forwarding requires a hardware logic known as bypass network and is discussed in greater detail later in this Chapter.

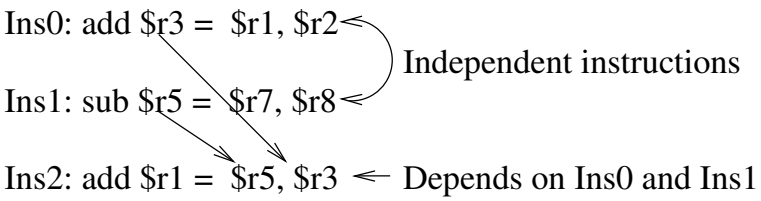
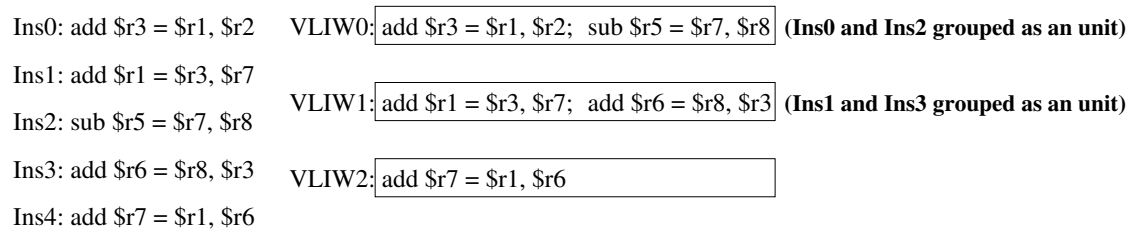


Figure 1.3: **Data dependencies**

So far, we have assumed that the processor can issue only one instruction at a given cycle. Many modern processors have multiple functional units and have the ability to execute multiple instructions concurrently. Executing multiple instructions at the same time can improve performance significantly. However, to execute multiple instructions simultaneously, no data dependencies must exist between the instructions. The code fragment shown in Figure 1.3 shows three consecutive instructions of a program. The first two instructions, Ins0 and Ins1, do not have any data dependencies. However, instruction Ins2 depends on the results generated by instructions Ins0 and Ins1. Therefore, instruction Ins2 can be executed only after both instructions Ins0 and Ins1 have finished their execution. If the processor has two functional units that can execute add and sub instructions, Ins0 and Ins1 can be executed in parallel and hence, execution can be completed in two cycles. On the other hand, a serial execution of the three instructions would have required three cycles. Thus, by executing independent instructions in parallel further improves processor performance. This phenomenon of exploiting parallelism at the instruction level for improving performance is commonly referred to as Instruction Level Parallelism (ILP). The number of instructions that the processor can issue at a given cycle is known as the issue width of the processor. The ability to extract the available ILP in an application is one of the keys to achieve higher performance in modern processors.

ILP can be extracted statically by the compiler or dynamically by the processor at runtime or using a combination of both. Superscalar processors e.g. Intel Pentium 4, IBM Power 5, etc. find independent instructions that can be issued simultaneously at runtime. To achieve high performance, superscalar processors typically employ out-of-order execution i.e. the instructions do not have to be executed in the program order but can start execution as soon as the input data is available (but commit in program order). For brevity, we would refer to superscalar out-of-order issue processors as superscalars in this chapter. In order to find independent instructions and accomplish out-of-order execution, superscalar processors require hardware structures like rename tables, issue queues, reorder buffers, etc. However, these structures have a high complexity, require large amount of



(a) Scalar code (b) VLIW code

Figure 1.4: **Scalar code vs VLIW code**

transistors, and consume high energy and power. The complexity, power consumption and area requirements of these structures further increase significantly with the issue width of the processor [43]. Superscalar processors are popular in high performance desktop and server computing. However, superscalar processors with high issue width are not well suited for embedded systems, where low cost and low power is the prime requirement.

Embedded systems are widespread in current consumer electronics domain like Mobile phones, DVD players, Navigation devices etc. Traditionally, most embedded devices had a fixed set of applications and limited customizability. However, in recent times, there is a big rise in embedded devices that are highly customizable and have a big third-party application market rivaling personal computing e.g. smartphones like iphone, android based phones, etc. The application platform on many embedded devices is approaching desktop computing with a wide set of rich applications including a fully functional web platform. Embedded systems, in particular those that are battery operated, have a low power constrain. Besides, technology convergence like integration of audio, video, imaging, security, etc. demands even higher performance while keeping the low power constrain. Very Long Instruction Word (VLIW) processors is a popular approach to extract ILP and achieve high performance at a low power and design cost suitable for embedded domain. VLIW processors are explained in detail in the following section.

1.1 VLIW Processors

VLIW processors exploit ILP by exposing the architecture details to the compiler, and ILP is extracted at compile time. The architecture details exposed to the compiler generally include the number of functional units of different types (ALU, multipliers, etc.) and their associated latencies, number of memory ports, the number of physical registers etc. In a VLIW processor, the compiler does the task of extracting the ILP in the program.

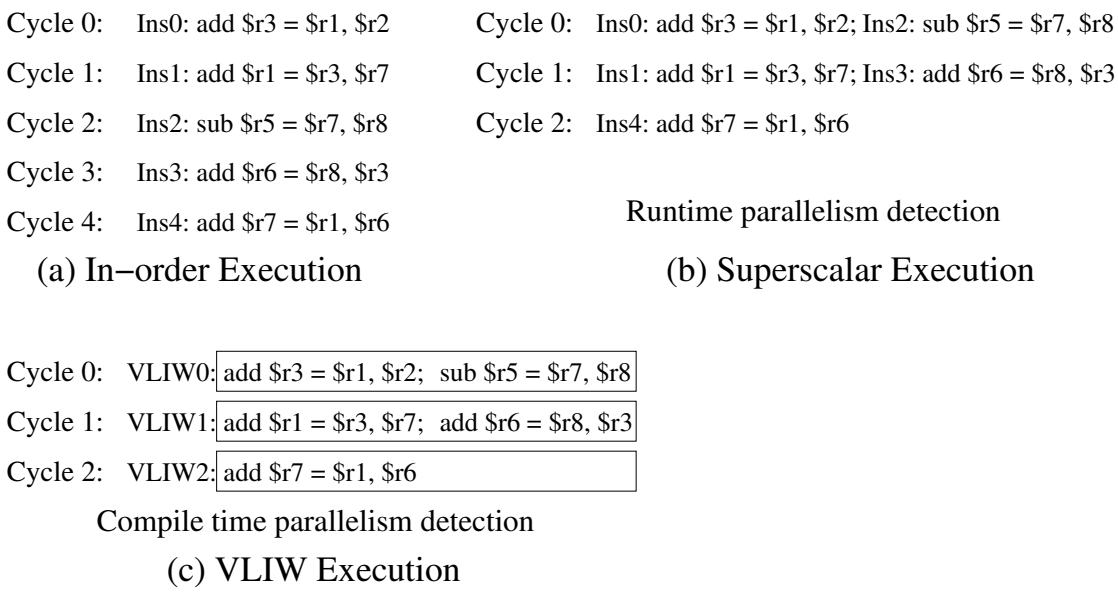


Figure 1.5: **Instruction execution on an in-order, Superscalar and VLIW Processor**

The extracted ILP is specified to the hardware as a set of independent operations by the compiler forming a VLIW instruction. Operations, in the context of VLIW architectures, are the scalar instructions e.g. addition, multiplication etc. All the operations inside a VLIW instruction are issued as a unit by the hardware. Figures 1.4(a) and (b) show five scalar instructions Ins0-Ins4 and the corresponding VLIW code respectively. For the VLIW processor, the compiler identifies the parallelism among the instructions Ins0-Ins4. The identified independent scalar instructions are then grouped together by the compiler to form the VLIW instructions. For instance, the scalar instructions, Ins0 and Ins2, are grouped together as a single VLIW instruction, VLIW0, with two operations.

Next, Figure 1.5 shows the corresponding execution of the instructions shown in Figure 1.4 on an in-order single issue processor (Figure 1.5(a)), a superscalar (Figure 1.5(b)) and a VLIW processor (Figure 1.5(c)) respectively. Assuming that each instruction requires a single cycle to execute, the in-order processor takes five cycles to execute all the instructions. A superscalar out-of-order processor executes the same scalar code. However, the superscalar processor will identify the instructions that do not have any dependencies and execute them in parallel at runtime. For instance, instructions Ins0 and Ins2 (also Ins1 and Ins3) are independent and are executed in parallel. Thus, the execution will require only three cycles on a superscalar processor. The VLIW processor executes the compiler generated VLIW code i.e. instructions VLIW0-VLIW2 (not Ins0-Ins4) in-order. All operations in a VLIW instruction are executed simultaneously. Hence, the execution

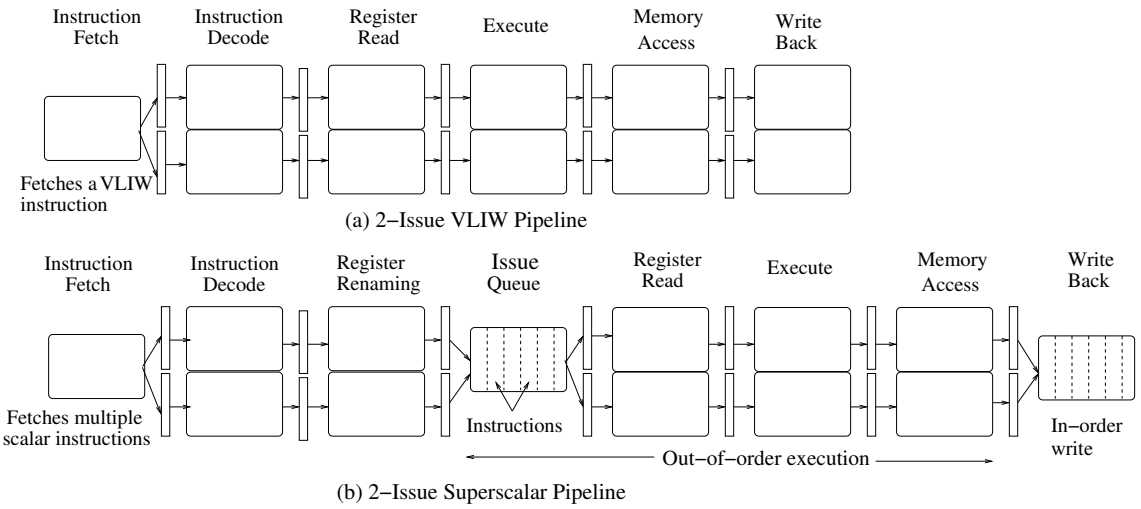
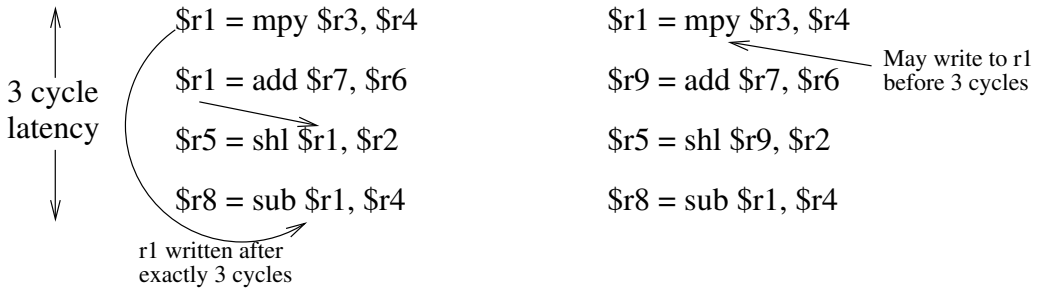


Figure 1.6: **VLIW vs Superscalar Pipeline**

of the 3 VLIW instructions also takes three cycles on the VLIW processor but without requiring any runtime ILP identification hardware.

VLIW processors make no runtime instruction scheduling decisions and execute the compiler generated instructions in-order. Figures 1.6(a) and (b) show the execution pipelines for a VLIW and a superscalar processor respectively. The pipeline for the VLIW processor is similar to the pipeline already shown in Figure 1.2. For a VLIW architecture, the only difference is the presence of multiple pipelines to handle the execution of multiple operations. Superscalar processors, on the contrary, need extra pipeline stages and hardware to support out-of-order execution and ILP detection viz. register rename tables, issue queues, reorder buffers, etc. None of these hardware structures are required in VLIW architectures. This results in a significantly lower complexity, area and power requirements for VLIW processors as compared to superscalar processors. VLIW processors have been used in general purpose computing [8, 46, 24] but the issue of binary compatibility and difficulty to extract ILP in control intensive applications have limited their applicability. However, because of the low power, low cost and hardware simplicity advantages, VLIW processors have gained widespread popularity in most embedded computing realms [12, 49, 23] where binary compatibility is not critical. For instance, VLIW processors can be found in mobile phones, digital televisions, receivers and recorders, digital cameras, multifunction printers, etc. [15]. VLIW architecture is also commonly used in Graphics Processing Units (GPUs) [2], [40].

A VLIW execution model can be defined in terms of Unit Assumed Latency (UAL) or Non-Unit Assumed Latency (NUAL) depending on the architecturally visibility of the



(a) EQ Execution: Reusing register r1 (b) LEQ Execution: Reuse not possible

Figure 1.7: **EQ and LEQ Execution**

functional unit latencies. UAL execution semantics mimic a conventional sequential program by assuming each instruction has an unit latency (the compiler may be cognizant of the actual latencies for better scheduling but is not necessary). When the actual latency is greater than one, UAL model requires additional hardware mechanisms such as pipeline interlocking [21] to preserve the data dependencies in the instruction sequence. Starcore DSPs from Freescale [52] and Tigersharc processors from Analog Devices [55] use UAL execution model.

In the NUAL execution model, operations have architecturally visible non-unit latency. Operations may have different latencies, depending on the functional unit, and the compiler has to schedule the code while respecting the latency constraints. No interlocking hardware is required in a NUAL execution model. However, a program may have to be recompiled if there is any change in the functional units latencies in the NUAL execution model. On the other hand, no recompilation is required for UAL model. Many current VLIW processors use NUAL execution model [49, 23, 59]. NUAL execution model can be further subdivided into Equals (EQ) and Less-than-or-Equals (LEQ) execution models.

In EQ model, an operation accesses its operands at the specified time and writes back the result exactly at its latency time. On the other hand, in the LEQ model, the hardware can complete an operation in the same or fewer number of cycles than assumed by the compiler i.e. if the operation latency is L, the result is available between one and L cycles after executing the operation. Hence, any operation that uses the result must be scheduled after at least L cycles even if the actual latency is less than L. Figures 1.7(a) and (b) show an example of EQ and LEQ execution respectively. In the example mpy operation has a latency of 3 cycles and the rest have unit latencies. The EQ execution model assumes that result of mpy will be written to register r1 exactly after 3 cycles and can reuse the register

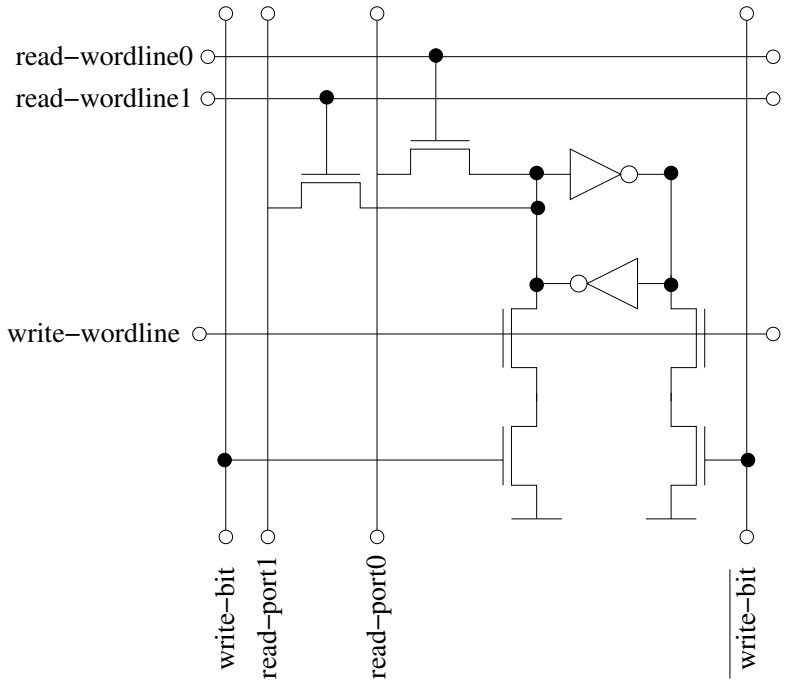
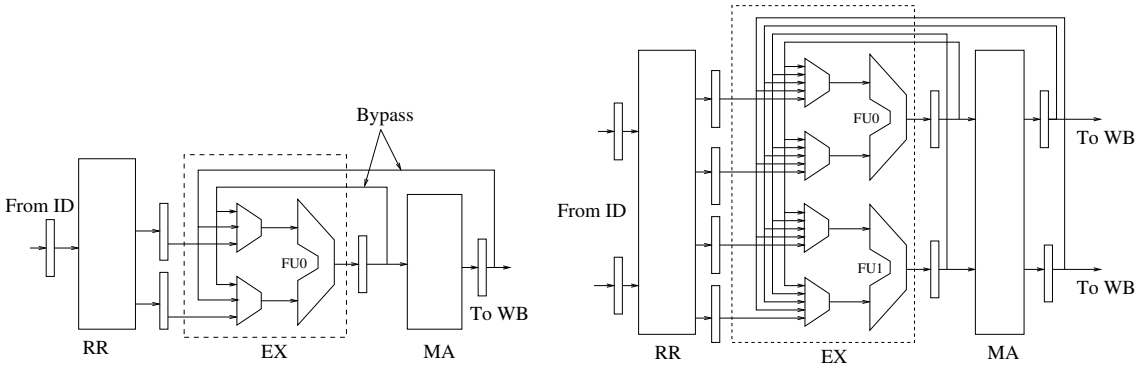


Figure 1.8: **Register file bit cell**

r1 in between. LEQ execution, on the other hand, assumes that execution may take less than 3 cycles and will not reuse register r1.

EQ model has the advantage of better register usage and better scheduling over LEQ because of the exact latency information usage. However, issues may occur when the control flow at runtime is different from the compile time assumptions. For instance, in events like exceptions, the control is transferred to the exception handling routine. This violates the assumed latency of the scheduled operations and may break the execution semantics. Hence, hardware support is required to save the status of the processor before executing exception handling code. LEQ model simplifies the implementation of runtime events like exceptions. Besides, LEQ model also offers a limited binary compatibility when the latency of some functional unit is reduced. EQ model, on the other hand, would require a recompilation of the program in case of any change in latencies. Note that recompilation is necessary for both LEQ and EQ models if other architectural assumptions like number of functional units, processor issue-width, register file specifications etc. are changed. TI C6 series VLIW DSPs from Texas Instruments [49] use EQ execution model while ST200 series VLIWs from ST [23] use LEQ execution model.



(a) Single–issue (b) Two–issue

Figure 1.9: **Bypass Network**

1.2 Scalability Issues in VLIW Processors

Many multimedia applications common in embedded domain exhibit significant amount of ILP, or at least regions with high ILP interleaved with low ILP regions. To exploit this ILP, VLIWs need to be designed with a significantly wide issue width, which is limited by the number of functional units (FUs). However, the number of FUs is limited by the scalability of the register file and the bypassing network.

A register file is an array of the processor registers. The functional units of the processor read their inputs from the register file and write the produced results to the register file. Figure 1.8 shows a standard register file bit cell with two read ports and one write port. A bit cell requires one wordline for each port (either read or write), one bitline per read port, and two bitlines for each write port. Each port requires a new routing in both horizontal and vertical direction as shown in the Figure 1.8. Register file design is dominated by this routing and the area increase is proportional to the square of the number of ports. Besides, Register file access time grows linearly and the power consumption increase quadratically with the number of ports [47]. As the number of ports is proportional to the number of FUs which are proportional to the issue width, any increase in issue width results in a significant increase in area, delay and power consumption of the register file. For instance, a processor with an issue width of 4 requires 8 read ports and 4 write ports (assuming 2-input functional units). Doubling the issue width to 8 would result in a register file that has 4-times the area, 4-times more power consumption and 2-times the access time. Since register files already consume a significant processor area and power, and lie on the critical path, this limits any significant increase in the issue width.

The data bypassing network can impact processor area and cycle time in a similar way.

Data bypassing network is required to forward the values produced from the executed operations to the dependent operations in a processor pipeline. In the pipeline shown in Figure 1.2 consisting of 6 pipeline stages, IF, ID, RR, EX. MA, and WB, the result is generated at EX stage. However, the generated result is not written to the register file until the WB stage. In order to issue an instruction immediately after the currently executing instruction which may depend on the produced result, the results must be forwarded from the pipeline stages EX and MA as an input to all the functional units. Figures 1.9(a) and 1.9(b) show the bypass network required for the pipeline for a single-issue and processor with an issue width of two respectively. The bypass network has the following inputs: the values read from register file at RR stage, produced results at EX stage, and results at the MA pipeline stage. Assuming that each functional unit has 2 inputs, the total number of bypasses required is $(2 \times I^2 \times S)$ where I is the issue width of the processor and S is the number of pipeline stages from execution to writing of the result of an instruction. The delay of the bypass network is given by $T_{bypass} = 0.5 \times R_{metal} \times C_{metal} \times L^2$ where L is the length of the result wires, and R_{metal} and C_{metal} are the resistance and parasitic capacitance of metal wires per unit length respectively [43]. An increase in the issue width results in a linear increase in the length of the result wires, and hence, causes the bypass delay to grow quadratically with issue width. The quadratic increase of the bypass network in both area and delay with the issue width further impacts the scalability of the issue width of the processor. To scale the issue width in VLIWs, many semiconductor companies have adopted clustered VLIW approach as explained in the following section.

1.3 Clustered VLIW Processors

Clustered VLIW architectures tackle the issue of scalability by introducing more than one register file and clustering the FUs according to the register files they are connected to. Clustering allows higher levels of issue width than monolithic VLIW architectures, since register file ports and bypass network are determined by the issue width in a cluster. While issue width in a cluster can be kept low, the total issue width can be easily scaled by increasing the number of clusters. Higher issue width allows to achieve higher performance levels without scaling up the clock frequency, achieving a better power budget as well. Many VLIW processors have been designed using the clustered approach [23, 49, 59]. Figure 1.10 shows a sample 2-cluster VLIW processor pipeline. In the shown pipeline, the IF pipeline stage fetches the VLIW instruction and dispatches the operations to their

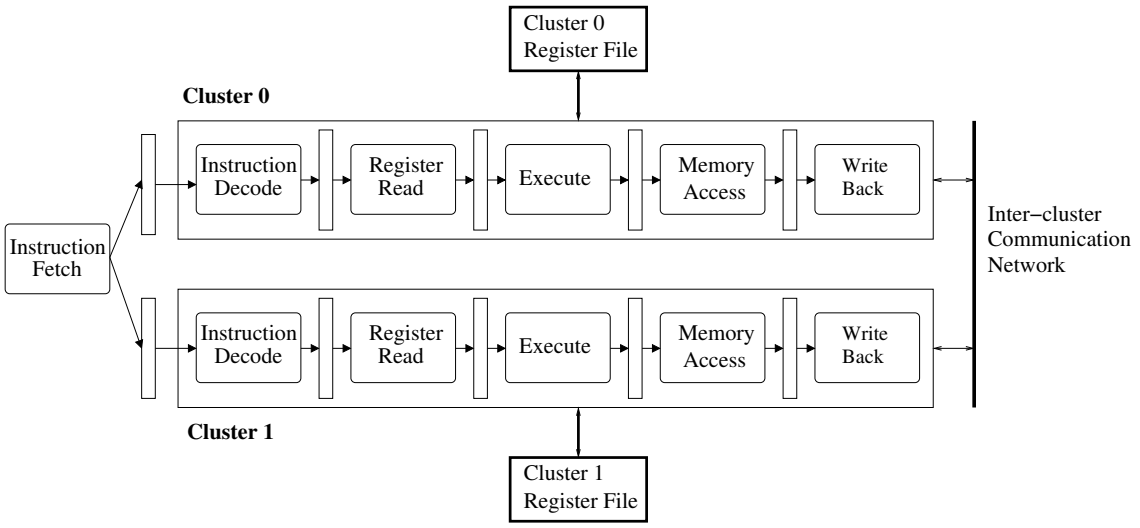


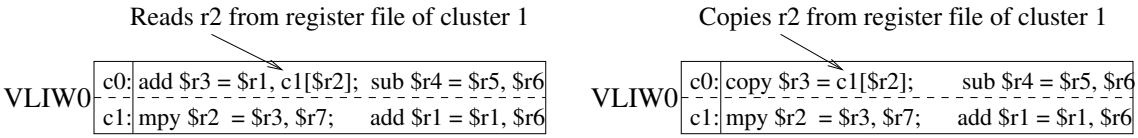
Figure 1.10: **A 2-cluster VLIW Processor pipeline**

destined clusters. The two clusters cannot access the register file of the other cluster and need to communicate through a separate intercluster communication network.

Implementation of clustering can either be architecturally invisible or can be exposed at the architectural level. When clustering is architecturally invisible, the compiler views the register file as a single monolithic unit. The hardware is responsible to create an impression of a non-clustered register file for preserving execution semantics. Architecturally invisible clustering is required if maintaining binary compatibility is important. Some VLIW processors like Sun MAJC 5200 [56] use invisible clustering.

Exposing clustering at architectural level may break binary compatibility and requires a recompilation of the programs. However, on a positive note, compiler can take advantage of the clustering information to generate more efficient code. Architecturally visible clustering can be further subdivided into implicit-copying or explicit-copying depending on the intercluster communication specification in the instruction set. Implicit copying implies that an operation may read or write registers from a remote register file i.e. an operation can operate on a register belonging to the register file of a different cluster. TI C6x series [49] and NXP Trimedia VLIW processors [59] use implicit copying for intercluster communication.

Explicit copying requires explicit copy operations in the instruction set to move data across register file of different clusters. All other operations can operate only on the registers belonging to the local register file. ST200 series VLIW processors [23] are an example of explicit copy based clustering. Figures 1.11(a) and (b) show an example



(a) Implicit copy

(b) Explicit copy

Figure 1.11: **Implicit and explicit copy**

of implicit and explicit copy. In the case of implicit copy, the operation can directly use a register from the register file of the other cluster. For explicit copy, only copy operations are permitted to read registers from the register file of other cluster. Implicit copying requires extra bits in the operations encoding to address registers from other clusters. Besides, compiler has to model the intercluster communication details as well. Explicit copying, on the other hand, only requires addition of copy operations to the instruction set that are modeled like any other operations during scheduling. However, copy operations are issued using the standard issue slots, thus competing for issue slots with other operations. If the number of copy operations is significant, it results into a larger code size which may affect instruction cache performance. A detailed analysis of the intercluster communication models for clustered VLIW processors has been done in [53] and [17].

Performance of VLIW processors is closely coupled with efficient code generation from compiler. Modulo Scheduling (also commonly referred to as Software Pipelining) [44], [33], [35] is a popular compiler based to improve performance of VLIW procesors. Modulo Scheduling improves the ILP that can be exploited within loops in a program. With emerging popularity of clustered VLIW processors, modulo scheduling for clustered VLIW processors [48], [7], [65] is a popular research area.

1.4 MultiThreading

Some applications scale well with issue width. For instance, colorspace conversion [1] used in high performance printers has an average IPC (Instructions Per Cycle) of 3.9, 6.0 and 8.9 for an issue width of 4, 8 (2 clusters of 4) and 16 (4 clusters of 4) respectively, which makes a very high issue width processor desirable. However, the ILP exposed in many applications, or in some code regions, is limited and the processor is heavily under-utilized. Also, in a production environment, high ILP applications (like image processing)

coexist with low ILP applications (like control code or the OS itself). For instance, playing a DVD requires multiple threads for decryption (low ILP), video decoding (high ILP), audio decoding (medium ILP), etc. along with the operating system threads (low ILP).

In the context of VLIW architectures, processor under-utilization can be described in terms of vertical and horizontal waste. Vertical waste are the cycles where no operations are issued at all. Horizontal waste is the under-utilization of the issue slots of the processor, i.e. number of operations issued in a cycle is less than the maximum number of operations that can be issued per cycle. Both vertical and horizontal waste arise because control and data dependencies in the program limit the number of operations that can be issued in a given cycle. Also, operations that have variable latency (for instance, memory operations) stall the processor if the actual latency is greater than the expected one, resulting in additional vertical waste. Multithreading is a popular approach for reducing the underutilization and improve processor performance. Multithreaded execution has received a lot of focus from researchers and many multithreading proposals exist in literature. An excellent survey on multithreaded processors can be found in [58].

Typically memory operations have the highest latency because of cache misses. Most initial multithreading proposals, therefore, were targeted to minimize vertical waste that arises due to memory latency cycles in a program as explained in the following lines.

Block MultiThreading (BMT) [62, 39, 3] executes instructions from a single thread until it is blocked by a long latency event (a cache miss, for instance) [10, 13]. When that happens, a fast context switch gives control to a different thread so that most of the miss latency is hidden. However, a few vertical slots are still wasted due to context switch time. BMT has been used in several commercial processors [56, 6, 38].

Interleaved MultiThreading (IMT) [50, 4] does a zero cycle context switch every cycle, so that instructions from different threads are interleaved at execution time. Interleaved multithreading allows the removal of the bypass network and interlocking hardware, since control and data dependencies between the instructions in the pipeline are eliminated when the number of interleaved threads is greater or equal than the number of pipeline stages. However, doing so hinders single thread performance when only one thread is present. In order to hide cache misses, many threads are required. Moreover, it still does nothing to remove horizontal waste. A trivial modification consists of marking as blocked any thread that produces a cache miss and issuing instructions only from non-blocked threads. This modification achieves the best from both block and interleaved multithreading. In this paper, we use this variant of IMT for our evaluations. IMT has

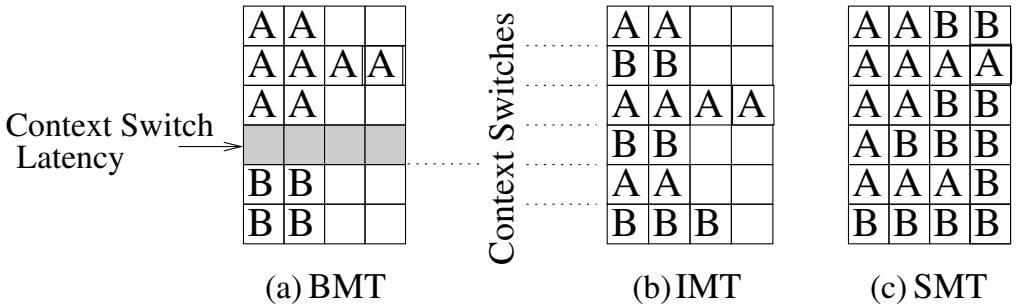


Figure 1.12: **Comparison of Multithreading Schemes**

also been used in several commercial designs [50, 54, 27, 4].

Both, block and interleaved multithreading can reduce vertical waste by issuing instructions from different threads, but not horizontal waste.

Simultaneous MultiThreading (SMT) [57] reduces both the horizontal and vertical waste inside a processor. It issues multiple instructions from multiple threads each cycle. In contrast, BMT and IMT issue instructions from only one thread at a time. In a SMT processor, issue-slots are filled by operations of different threads, converting Thread Level Parallelism (TLP) into ILP. Also, latencies occurring in execution of single threads are hidden by issuing operations from other threads. As a SMT processor simultaneously exploits coarse and fine grain ILP, the resource usage is much more efficient than BMT or IMT. It relies on dynamic issue of instructions from different threads, and fits naturally with out-of-order superscalar processors. Hence, a SMT capability can be added to a conventional superscalar with little effort. SMT has found its use in several commercial superscalar processors [28, 31].

A comparison of the different multithreading approaches viz. BMT, IMT and SMT is shown in figures 5.6(a), 5.6(b) and 5.6(c) respectively for two threads A and B on a 4-issue VLIW processor. In the figures, each box in a row represents an issue slot. A box with a label indicates that the slot is in use by that particular thread and an empty box represents an unused issue slot. BMT and IMT can reduce vertical waste by issuing instructions from different threads, but not horizontal waste. In BMT, a few vertical slots are still wasted due to context switch time. Since IMT also reduces short stalls, IMT performance, in general, is a good upper bound for BMT as well. In contrast to BMT and IMT, SMT also removes horizontal waste in the processor. Next, we discuss several multithreading proposals targeted towards VLIW processors specifically.

Kaxiras et al [29] propose a SMT VLIW extension for StarCore (SC140) VLIW DSP, wherein the issue logic selects VLIW instruction packets from ready threads as many as

it can accommodate, and assigns them to functional units. They show that, for media applications, implementing SMT is a better option than Chip MultiProcessing.

Iyer et al [26] also implement SMT on a NUAL TI DSP processor. The focus of the study, however, is binary compatibility. They propose split issue to extend a previously proposed dynamic scheduling technique [45], and then show that split issue can be used to implement rigorous changes to processor hardware (changes which break program semantics in a VLIW, like changing number of functional units, changing operation latencies, implementing SMT, etc.) while maintaining binary compatibility. An analysis of the hardware required to implement SMT is lacking, however.

Balanced MultiThreading is a related multithreading technique proposed for high end out-of-order superscalar processors. Balanced multithreading combines the SMT ability with Block MultiThreading. To do so, extra thread contexts are stored in a special storage location on-chip. If a running thread encounters a L2 cache miss, it is swapped with one of the extra contexts. Use of balanced multithreading saves the register file space requirement for the extra thread contexts but several cycles are required for context swapping.

Subset Static Interleaved Multithreading (SSIMT) is a technique similar to Balanced MultiThreading. SSIMT combines Block MultiThreading with IMT for embedded VLIW Processors. SSIMT maintains several background thread contexts on-chip. The background threads are swapped with a running thread if that thread encounters a cache miss at a low swapping penalty.

Another class of multithreading proposals are addressed to increase the performance of a sequential program [51, 37]. These rely on extracting and spawning multiple threads from the same program (a thread here means any contiguous instruction stream, not just OS threads), which are executed in parallel, thereby improving the performance of the sequential program. The threads can be computation slices of the program or simply helper threads which do some tasks like prefetching or improving branch prediction. Threads are obtained at runtime with or without compiler support. Since our focus is on VLIW processors, we restrict to following proposals.

Processor coupling [30] statically schedules multiple threads and, at runtime, interleaves them into execution clusters, consisting of a set of FUs and a common register file. Threads are generated by the compiler through explicit fork and forall operations, communicate through registers and memory and are non speculative. **M-Machine** [14] uses processor coupling as the execution model. In M-Machine either complete VLIW

instructions are issued from a single thread or short instructions are issued from multiple compiler generated threads by executing each thread in a different cluster. In the latter case, threads have been compiled to use only a single cluster.

XIMD [63] also statically schedules multiple threads, but does not perform a run-time interleaving. It has multiple functional units and a large global register file. Each functional unit has a dedicated instruction sequencer to fetch instructions. A program is partitioned into several threads by the compiler or by a specialized partitioning tool. The XIMD compiler takes each thread and schedules it separately. Threads are then statically merged to increase static code density or to optimize the execution time. No speculative threads are allowed in XIMD.

Weld [42] architecture is aimed at increasing single program performance using compiler generated (speculative) threads in a multithreaded VLIW architecture. Weld architecture has the limited capability to issue each operation in a VLIW instruction separately. According to dependencies among issue packets, the compiler adds and sets a separability bit for each operation in the opcode. The hardware looks up this bit to decide whether it can issue an operation separately or not. The Weld architecture selects VLIW instructions from ready threads and issues them to available functional units. The design of the general Weld architecture involves extensive changes to the ISA, compiler support for effective generation of multiple threads, and additional hardware resources such as buffers for speculative load and store instructions, a thread synchronization hardware, and a complex operation welder. Operation welder dynamically fills issue slots from multiple threads to enable multithreading. The welder is implemented as a crossbar and because of the complex hardware, scalability is limited beyond two threads for cost/performance reasons [41].

Barretta et al [5] discuss a multithreaded extension to multi-cluster VLIW. The processor has two modes: single threaded and multithreaded. In single threaded mode, VLIW instructions are issued from a single thread that has been compiled to make use of all the clusters. In multithreaded mode, short VLIW instructions are issued from multiple threads (part of the same program) by executing each thread in a different cluster. In the latter mode, threads have been compiled to use a single cluster. Switching between modes is also compiler (or programmer) controlled. This approach does nothing to avoid vertical waste due to cache misses and cannot exploit TLP between different applications.

1.5 Contributions of the Thesis

We note that very few multithreading proposals exist targeting clustered VLIW processors. Besides, the emphasis of those [5], however, is to improve the performance of a single program, using multithreading in a very limited way (each thread executes in a different cluster rather than using full system resources). For a VLIW processor, Block MultiThreading (BMT) and Interleaved MultiThreading (IMT) can reduce vertical waste by issuing instructions from different threads, but not horizontal waste. Simultaneous MultiThreading (SMT) removes horizontal waste as well. However, implementing SMT on VLIWs require complex structures making it unsuitable due to cost/area constraints. Also, proposals which study SMT on VLIW with an emphasis on hardware implementation [41] limit scalability of SMT to 2 threads.

There is a considerable performance difference between IMT and SMT because of the ability of SMT to remove horizontal waste but at a higher hardware cost. Our first approach, named **Cluster-level Simultaneous MultiThreading** (CSMT) and discussed in Chapter 3, exploits Thread Level Parallelism (TLP) at the cluster level achieving a performance in between IMT and SMT. CSMT issues simultaneously VLIW instructions from multiple threads only if they use different clusters. In other words, CSMT uses a coarse grain merging at cluster level whereas SMT does a fine grain merging at the operation level. For clarity and to avoid confusion, we refer to SMT (merging at operation-level) as OpSMT henceforth. CSMT is transparent to the compiler and can be used either with compiler generated threads or with threads belonging to different applications. An analysis of hardware required to implement CSMT shows that the hardware overhead of CSMT is small and is comparable to low complexity multithreading schemes like IMT. Besides, CSMT is more scalable than OpSMT in terms of both gate delays and area overhead with increasing number of threads.

We also propose a technique named Cluster Renaming, discussed in details in Chapter 4, to further improve multithreading performance. Cluster renaming reduces cluster conflicts when several threads require the same cluster by mapping logical clusters, belonging to different threads, to different physical clusters. Cluster renaming has a low implementation overhead and is equally applicable to other multithreading techniques like OpSMT as well. Using CSMT and cluster renaming together, a significant amount of horizontal waste is removed and the obtained performance is close to OpSMT performance.

The cost of the merging hardware dictates the number of threads that can be merged

at a given cycle. However, the cost of merging hardware for CSMT is negligible when compared to OpSMT. Hence, for an OpSMT architecture, if extra threads are supported using CSMT, the additional cost incurred by the merging hardware is little. Our another proposal, explained in Chapter 5, is a heterogeneous merging approach that combines OpSMT and CSMT merging hardware. This approach supports more hardware threads than that can be supported using OpSMT merging hardware alone. For instance, while OpSMT can support only two threads, four threads can be supported by the heterogeneous merging approach where the first two threads are merged using OpSMT, and the result and the rest of threads are merged using CSMT approach. The final cost of the merging hardware for four threads is practically the same as the 2-thread OpSMT merging hardware cost. Hence, this approach allows higher performance without increasing merging hardware cost.

Operations in a VLIW instruction execute in a lock-step mode and hence, a VLIW instruction has to be issued in its entirety to honor execution semantics. The restriction to issue a VLIW instruction in entirety limits the number of instructions that can be issued simultaneously from different threads. For instance, on a 2-thead 3-issue VLIW architecture with 1 memory unit and 2 ALU units, if at a given cycle, the instructions of both threads require 1 memory unit and 1 ALU unit, then VLIW instruction from only one thread can be issued at that cycle. The remaining ALU unit cannot be used even though the other thread has ALU operations. Split-issue removes this restriction and allows a VLIW instruction to be issued in parts. Using split-issue improves resource usage efficiency and multithreading performance as it creates more opportunities for a VLIW instruction to be merged with VLIW instructions from other threads. A previously proposed technique, operation-level split-issue [26] does a split at the operation-level. However, doing a split at operation-level requires complex hardware for preserving execution semantics. Our proposal, cluster-level split-issue discussed in chpater 6, improves both OpSMT and CSMT performance at a very low hardware overhead while achieving performance close to operation-level split-issue.

Both CSMT and OpSMT require a merging hardware to do resource constraints checking and combining instructions from different threads. The cost of the merging hardware increases significantly with an increase in number of threads and limits the number of threads can be merged at a given cycle. We propose Hybrid Multithreading (HMT) discussed in detail in Chapter 7, that allows supporting a high number of threads without affecting merging hardware cost. HMT tries to combine instructions from only a subset

of hardware threads every cycle. A different subset is used every cycle permitting all threads to make progress. HMT allows supporting more hardware threads and improves multithreading performance without impacting merging hardware cost.

1.6 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 discusses the basic infrastructure that includes VEX VLIW architecture and the compiler toolchain, simulation platform etc. used for evaluating the ideas proposed in the thesis. Chapter 3 focuses on CSMT, and its performance and hardware cost evaluation vs OpSMT. Detailed evaluation of OpSMT and CSMT hardware is discussed in the appendices. Cluster Renaming and its performance impact on CSMT and OpSMT is discussed in Chapter 4. Chapter 5 discusses the heterogeneous merging schemes where both CSMT and SMT are used simultaneously to improve multithreading performance. Cluster-level Split-Issue and its comparison with previous techniques is discussed in Chapter 6. Chapter 7 discusses Hybrid MultiThreading, a technique to support more hardware threads than the number of threads supported by merging hardware. Chapter 8 discusses the performance improvements achieved when all heterogeneous merging, hybrid multithreading and split-issue are combined. Finally, Chapter 9 concludes the thesis.

Chapter 2
Experimental Platform

In this chapter, we present the experimental infrastructure used to evaluate the proposals of the thesis. We describe the VEX VLIW architecture, the VEX toolchain, and the used benchmarks and workloads. We also discuss the experimental methodology used for evaluating multithreaded applications and the microarchitectural changes and requirements in the baseline VEX architecture for multithreading.

2.1 VEX Architecture

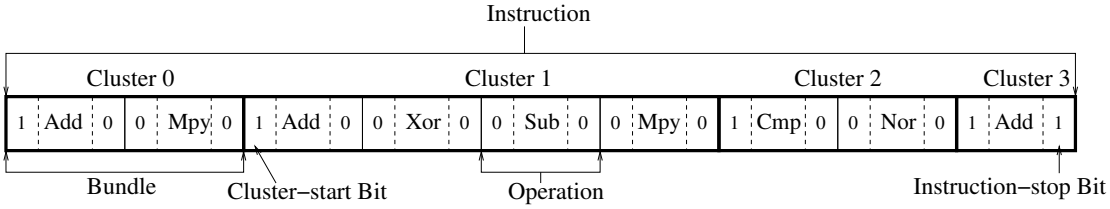
VEX clustered architecture [61] forms the basis of the experimental work done in this thesis. VEX architecture is modeled upon the commercial 32-bit Lx/ST200 VLIW family [12] jointly developed by Hewlett-Packard (HP) and STMicroelectronics (STM). The ST200 VLIW processor family is targeted towards embedded media processing and STM has shipped in excess of 70 million of these VLIW processors.

VEX uses the following terminology for the VLIW instructions: an operation is the basic execution unit, the collection of operations scheduled to execute in the same cluster form a bundle and the collection of bundles scheduled to execute together is called an instruction[1]. The VLIW instructions are issued in-order. All the operations in a given VLIW instruction execute together in a lockstep mode at the same cycle. Also, within an instruction all operands are read before the results are written. In other words, if an operation reads a register r and another operation writes to the same register r within the same VLIW instruction, the read uses the old value of the register r and not the newer value generated by the writing operation. As a result, certain actions like swapping the values of a pair of registers are possible to perform in a single instruction. The architecture

[1]Using the same terminology as the Lx architecture. In IA-64 terminology, an IA-64 instruction refers to a Lx operation and an IA-64 bundle refers to a Lx instruction

```
c0: add $r1= $r1,  $r4  ; mpy $r4= $r2,  4  ;
c1: add $r5= $r2,  $r7  ; xor $r9= $r1,  $r8  ;  sub $r11= $r15, $r7  ; mpy $r10= $r9,  $r6  ;
c2: cmp $r4= $r10, $r5 ;nor $r6= $r11, $r7  ;
c3: add $r9= $r11,  $r2  ;
```

(a) A VEX Instruction



(b) Instruction Encoding

Figure 2.1: **Example of a VEX Instruction**

also supports partial predication in the form of conditional select operations in order to eliminate some conditional branches. All the functional units are assumed to be fully pipelined. Clusters in VEX are architecturally visible and requires the compiler to take clustering and inter-cluster communication into account.

VEX uses a variable length instruction encoding to avoid wasting memory. To do so, the following two bits are reserved in every operation for sequencing and cluster encoding.

- **Instruction-stop bit:** indicates the end of a VLIW instruction.
- **Bundle-start bit:** indicates the beginning of a new bundle i.e. the current operation and the following operations are scheduled to execute in the next cluster forming a new bundle.

When the instruction-stop bit is set to '1', the current instruction is finished and the following operations belong to the next VLIW instruction. When the bundle-start bit is set to '1', the current operation and the following ones are scheduled in the next cluster starting a new bundle. For a 4-cluster implementation, an instruction can have a maximum of 4 bundle-start bits set to '1', and only the last operation of the instruction has the instruction-stop bit set to '1'. The numbering of the clusters starts from 0. In a 4-issue cluster, a bundle may contain 1 to 4 operations. An empty cluster can be encoded as an unit operation bundle with nop as the only operation. Note that encoding an empty cluster is required only if the instruction uses a higher cluster number. For instance, if an instruction uses cluster 0 and cluster 2, cluster 1 needs to be encoded but cluster 3 does

```
1. c0: cmpne $br1 = $r1,  $r8  ;   <-  Sets branch register
2. c0: br $br1,  L1  ;   <--------  Branch to L1 if br1 is set
```

Figure 2.2: **Branching in VEX**

not have to be encoded as the last operation in cluster 2 will have the instruction-stop bit set. Figure 2.1(a) shows a sample VEX instruction that uses 4-clusters and Figure 2.1(b) shows the corresponding encoding of the operations in the instruction. Note that in VEX instruction encoding, operations do not have to be encoded in any fixed order within a bundle. For instance, operations `add` and `mpy` that are scheduled in cluster 0 in Figure 2.1, can be encoded in any order i.e. `add` can be encoded either before or after `mpy`. The hardware is responsible for dispatching operations to the appropriate functional units.

VEX follows the Non-Unit Assumed Latencies (NUAL) and Less-than-or-EQual (LEQ) VLIW execution model i.e. the functional unit latencies are exposed to the compiler and the hardware can complete an operation in the same or fewer cycles. The compiler is responsible for generating the correctly scheduled code eliminating the requirement for any interlocking hardware. The only exception to LEQ model is the memory accesses that may take longer than the assumed latency for reasons like cache misses. Whenever such a scenario arise, the execution is stalled until the architectural assumptions hold true.

VEX has two register files at each cluster, a general purpose register file and a branch register file. All operations operate on integer values and can access registers from only local register files i.e. an operation scheduled in cluster C can read and modify the registers that belong to the branch and general purpose register files of cluster C only. The only exception is the branch operation, which can read branch registers from the branch register file of the other clusters. Conditional branches in VEX are two phased: the first operation does the comparison and sets the branch register ahead of the actual branch, and the second is the actual control flow changing branch operation. Figure 2.2 shows an example of the branching in VEX where the first instruction `cmpne` sets the branch register. The later branch instruction `br` jumps to the specified location (L1) if the branch register $br1 is set.

The inter-cluster communication in VEX is handled using explicit copy operations. The copies are implemented as a pair of `send` and `recv` operations that execute simultaneously in the source and destination clusters. `Send` operation specifies the destination cluster and a source register in the local register file and sends the value of the register to the specified cluster over the inter-cluster communication network. The corresponding

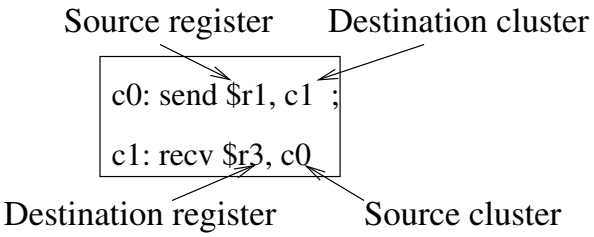


Figure 2.3: **Inter-cluster communication in VEX**

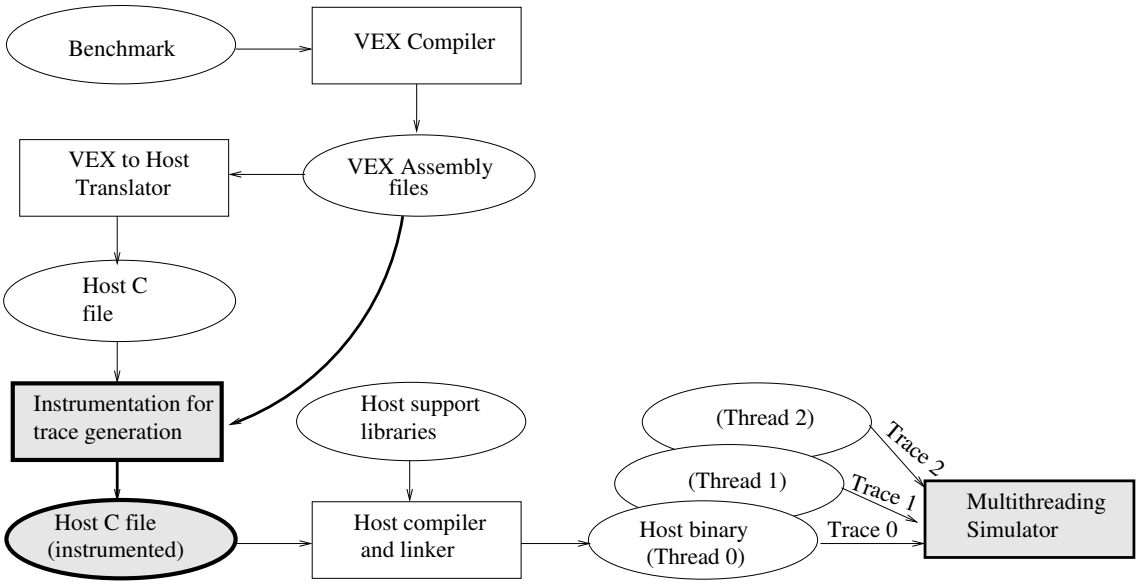


Figure 2.4: **Simulation flow**

`recv` operation retrieves the data from the specified source cluster and stores it in the specified destination register in the local cluster. Both `send` and `recv` have to be scheduled in the same VLIW instruction but in different bundles. Figure 2.3 shows a VLIW instruction that copies register $r1$ of cluster 0 to register $r3$ of cluster 1.

2.2 VEX Toolchain

VEX toolchain consists of the VEX C compiler and a set of extra tools for performance evaluation, simulation and debugging. VEX allows changing various architectural parameters like issue width, number of clusters, composition of the clusters, etc. The VEX C compiler [61] is a derivative of the HP/ST ST200 C compiler, which itself is a derivative of the Multiflow compiler [36] that uses *Trace Scheduling* [16] as global scheduling algorithm and *Bottom Up Greedy* [11] as cluster assignment algorithm.

The VEX development toolchain uses a compiled simulation technique to run the programs developed for the VEX architecture on the host system by translating the VEX code to the binary of the host computer. This translation is done by first converting the generated VEX assembly code to C. The generated C files are then compiled with the host platform C compiler (e.g. gcc) and linked with the support libraries to generate a host binary. The compiler can be instructed to add profiling code to generate timing and cache profile statistics when the generated host binary is run. The statistics, however, is useful for a single-thread configuration only. Also, the toolchain is not capable of simulating a multithreaded processor. The following section discusses the methodology and the baseline architecture configuration used for evaluating multithreading performance.

2.3 Simulating a MultiThreaded Processor

For a cycle accurate simulation with support for multithreading, we have developed a trace based simulator for the VEX microarchitecture. For generating trace information, we instrument the host C files generated by the translator to emit the program trace during the host binary execution. The trace includes the VLIW instructions that are executed and the memory addresses accessed for instruction cache and data cache. The trace information is fed to our simulator to simulate a multithreaded VLIW processor. Figure 2.4 shows the steps involved for obtaining the simulation results. The shaded parts in the figure are the extra steps that we have added for our experimental framework.

Next, we discuss the baseline architecture configuration used to evaluate the proposals made in this thesis.

2.3.1 Baseline Architecture Configuration

The evaluations in this thesis have been done in a 16-issue, 4-cluster architecture configuration (i.e. 4-issue per cluster). For the evaluations, we assume a homogeneous cluster configuration where each cluster has 4 ALUs, 2 multipliers and 1 load/store unit. The only exception to the homogeneity is the branch unit which is present only in the first cluster i.e. cluster 0. Figure 2.5 shows the structure of the cluster 0 (other clusters are identical except for the branch unit). Memory, inter-cluster communication (`send`/`recv`) and multiply operations have a latency of two cycles, and the rest of the operations have unit latency. We also assume a 6-stage execution pipeline as shown in Figure 2.6. No branch predictor hardware is used and fall-through path is the statically predicted path. The in-

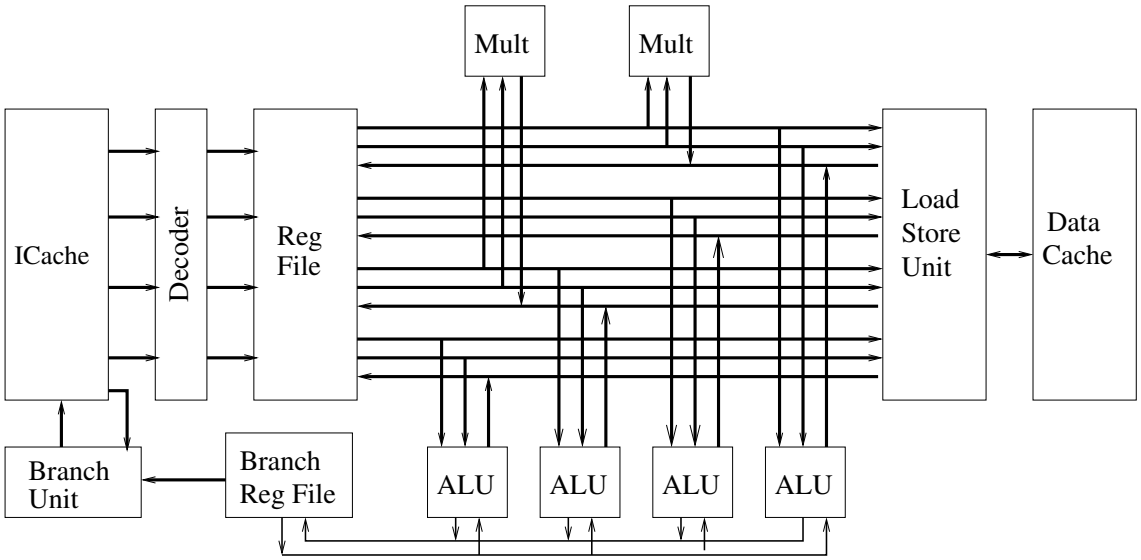


Figure 2.5: **Baseline Cluster 0 configuration**

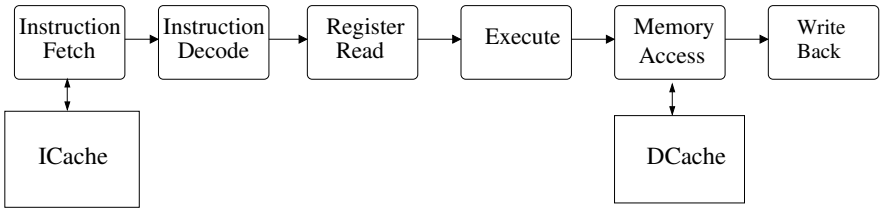


Figure 2.6: **VEX Pipeline**

correct instructions issued following a taken branch are squashed by the processor. All instructions read the values of registers at the Register Read stage except the branch instruction that can read branch registers at the Decode stage. As a result, there is a 2-cycle delay from compare to branch (Execute to Decode stage) but the taken branch penalty is only 1 cycle. A fully connected point to point communication network between clusters has been assumed. A single-level i.e. only level-1 64 KB 4-way set-associative cache with a 64-byte line-size is assumed for both ICache and DCache. The cache miss latency is assumed to be 20 cycles assuming a conservative 50 ns DRAM access time for a target processor frequency of 400 MHz. and 15/15/20 (row access time/column access time/time to transfer 8x8 bytes) DDR400 timing. The architectural configuration details are listed in Table 2.1.

Next section describes the microarchitectural changes to the baseline architecture for multithreading.

Table 2.1: **Architectural configuration**

Parameter	*Value*
ALU FU Latency	1 Cycle
Memory FU Latency	2 Cycles
Intercluster Communication Latency	2 Cycles
Multiply FU Latency	2 Cycles
ICache Configuration	4-way 64KB, 64 byte linesize
DCache Configuration	4-way 64KB, 64 byte linesize
Cache Miss Latency	50 ns (20 cycles@400 MHz.)

2.3.2 Microarchitectural changes for multithreading

Running a program in a multithreading environment may have a different behavior than the one expected in a non-multithreaded mode. This section describes the issues that arise because of multithreading and the solutions for these issues, namely cache misses, exceptions and the functional units latency variation.

- **Cache misses:** During multithreaded execution, instructions from multiple threads are in-flight at the same time. In the non-multithreaded configuration, if a thread encounters an event like cache miss, this results into stalling the processor. But, in the multithreaded mode, stalling the execution stops progress of all other threads. To solve this issue, each pipeline stage is tagged with individual thread identifiers. This tagging is used to selectively flush in-flight instructions of a particular thread on a cache miss so that the other threads can continue executing.

- **Exception detection and recovery:** A thread may generate an exception during the execution that requires detecting the exception generating thread, and flush the in-flight instructions of that thread. No extra hardware is required for exception detection because of the in-order pipeline and thread tagging done at each pipeline stage. So, if an exception is detected at any time, it is straightforward to know which thread caused the exception. When an exception occurs, the in-flight instructions of the excepting thread are flushed and exception handler is invoked.

- **Different latencies than compiler assumptions:** Our VLIW processor configuration executes the compiler generated code in-order and does not have interlocking hardware. Because of multithreading, instructions from a given thread may not be issued at every cycle. Hence, the actual latencies encountered in the execution for a program may appear to be shorter than those assumed by the compiler because of

the delay encountered in issuing next instruction. This fits the LEQ model of execution where actual latency of any FU can be shorter than the compiler assumption. Since VEX uses the LEQ execution model, no changes in the baseline architecture are required.

2.4 Benchmarks and Workloads

This section discusses the benchmarks and workloads used for evaluating multithreading performance. We have used a set of MediaBench [34] and SpecInt 2000 [22] applications. We have also included production color space conversion [1] and imaging pipeline [61] benchmarks used in high performance printers, inverse discrete cosine transform (used in various codecs) [25], and H.264 encoder [64] that are used heavily in most media applications. Note that only integer applications have been used as benchmarks since the baseline architecture, that is based on the commercial ST200 [23] family, does not have floating point operations (floating point is supported by emulation). The lack of floating point is not a big issue as the ST200 family VLIW processors have been used for many multimedia applications coded in fixed point arithmetic, which is a popular approach in the embedded domain.

All the benchmarks have been compiled with profile guided optimizations at -O4 optimization level (Trace scheduling with heavy loop unrolling). Some benchmarks have also been hand optimized to achieve higher performance. For instance, for the colorspace benchmark, the compiler pragmas ivdep and heavy unrolling is specified for the kernel. Ivdep pragma allows the compiler to reorder the memory operations in the loop that along with loop unrolling results in to a much better performance for the benchmark. A complete list of the benchmarks is shown in Table 2.2. In the table, columns IPC_r and IPC_p show the average IPC of the first 200 million VLIW instructions for each benchmark for a real memory model and a perfect memory model without cache misses respectively. Benchmarks are classified by their IPC_p in three categories: high IPC (colorspace, imgpipe, idct and x264), medium IPC (g721encode, g721decode, cjpeg and djpeg) and low IPC (mcf, bzip2, blowfish and gsmencode). This classification is shown in column $ILP\ Degree$ as L (low IPC), M (medium IPC) and H (high IPC).

For evaluating multithreading performance, we have grouped the benchmarks to form several workloads. The workload configurations are listed in Tables 2.3 and 2.4. Table 2.3 shows 4-thread workloads and Table 2.3 contains 8-thread workloads. In order to select

Table 2.2: **Benchmarks**

Benchmarks	*ILP Degree*	*Description*	IPC_r	IPC_p
mcf	L	Minimum Cost Flow	0.96	1.34
bzip2	L	Bzip2 Compression	0.81	0.83
blowfish	L	Encryption	1.11	1.47
gsmencode	L	GSM Encoder	1.07	1.07
g721encode	M	G721 Encoder	1.75	1.76
g721decode	M	G721 Decoder	1.75	1.76
cjpeg	M	Jpeg Encoder	1.12	1.66
djpeg	M	Jpeg Decoder	1.76	1.77
imgpipe	H	Imaging pipeline	3.81	4.05
x264	H	H.264 encoder	3.89	4.04
idct	H	Inverse Discrete Cosine Transform	4.79	5.27
colorspace	H	Colorspace Conversion	5.47	8.88

Table 2.3: **4-Thread Workload configurations**

ILP Comb	*Thread 0*	*Thread 1*	*Thread 2*	*Thread 3*
LLLL	mcf	bzip2	blowfish	gsmencode
LMMH	bzip2	cjpeg	djpeg	imgpipe
MMMM	g721encode	g721decode	cjpeg	djpeg
LLMM	gsmencode	blowfish	g721encode	djpeg
LLMH	mcf	blowfish	cjpeg	x264
LLHH	mcf	blowfish	x264	idct
LMHH	gsmencode	g721encode	imgpipe	colorspace
MMHH	djpeg	g721decode	idct	colorspace
HHHH	x264	idct	imgpipe	colorspace

Table 2.4: **8-Thread Workload configurations**

ILP Comb	*Thread 0*	*Thread 1*	*Thread 2*	*Thread 3*	*Thread 4*	*Thread 5*	*Thread 6*	*Thread 7*
llllllll	181.mcf	256.bzip2	blowfish	gsmencode	181.mcf	256.bzip2	blowfish	gsmencode
llllmmmm	181.mcf	256.bzip2	blowfish	gsmencode	cjpeg	g721encode	g721decode	djpeg
llllmmhh	181.mcf	256.bzip2	blowfish	gsmencode	g721decode	djpeg	colorspace	imgpipe
llllhhhh	181.mcf	256.bzip2	blowfish	gsmencode	colorspace	imgpipe	idct	x264
llmmhhhh	181.mcf	256.bzip2	g721encode	cjpeg	x264	idct	colorspace	imgpipe
llmmmmhh	blowfish	gsmencode	cjpeg	g721encode	g721decode	djpeg	idct	x264
mmmmhhhh	cjpeg	g721encode	g721decode	djpeg	idct	x264	colorspace	imgpipe
hhhhhhhh	x264	idct	colorspace	imgpipe	x264	idct	colorspace	imgpipe

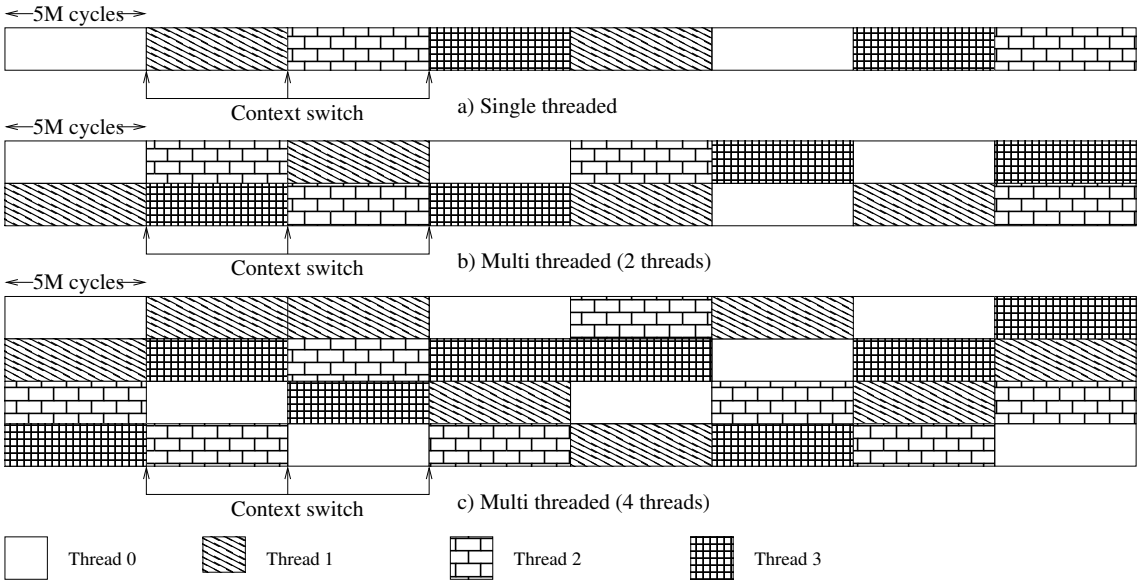


Figure 2.7: **Multithreaded workload execution**

appropriate thread configurations, benchmarks with different ILP degrees have been combined, attempting to cover representative combinations. Column labeled as $ILP\ Comb$ indicates these IPC combinations. For example, configuration LLHH in Table 2.3 has two benchmarks with low IPC and two benchmarks with high IPC, configuration LLMM has two benchmarks with low IPC and two benchmarks with medium IPC and configuration LMHH has one benchmark with low IPC, one benchmark with medium IPC and two benchmarks with high IPC.

2.4.1 Workload execution

The experiments have been carried out by arranging the workloads in a multitasking environment. The number of threads supported by the processor is exposed to the Operating System and the task scheduler schedules as many threads to run as the number of virtual CPUs, with a timeslice of 5 million cycles. After the expiry of the timeslice, a context switch takes place and the running threads are replaced by other threads from the workload. The delay of a context switch is assumed to be negligible. For a single-thread processor, the threads run in serial order with a single thread running in the whole timeslice. For a 2-thread processor, 2 threads are scheduled to run together in the same timeslice, and for a 4-thread processor, 4 threads share the timeslice. To improve fairness and to alleviate any bias, replacement threads are picked at random from the workload after the context switch. The workloads are executed till one thread completes executing 200 mil-

lion VLIW instructions. If any thread finishes execution before this time, it is restarted. Figure 2.7 shows the execution of a 4-thread workload on single-thread, 2-thread and a 4-thread processor respectively. Note that measuring accurate multithreading performance is still an undecided problem because of variance in the performance and the fact that different benchmarks complete execution at different times. A recent work, FAME [60], uses a similar methodology to the one used in the thesis where the benchmarks are executed repeatedly till a stable IPC is obtained. In our case, the IPC values for the workloads are very stable and no special measures to counter performance variance are required.

2.4.2 Thread Selection Policy

This section describes the selection policy for threads in a workload when instructions from all the threads cannot be issued together because of processor resource constraints.

A processor has only a fixed set of resources. Because of the limited number of resources like functional units, issue width etc., it is not always possible to issue instructions from all the threads simultaneously. Thus, only a limited number of threads can be selected at a given cycle without creating resource conflicts. We use a priority based selection policy to produce a conflict free thread selection that depends on the individual priority of the threads. In the priority based thread selection, first, the instruction from the highest priority thread is selected. Then, the instruction from the next highest priority thread is selected but only if it does not conflict with the already selected instruction, and so on.

We assume a round robin priority scheme where a different priority is assigned to each thread at each cycle. Round robin priority policy guarantees fairness and avoids starvation. Nevertheless, for meeting the demands of applications that have real time or Quality of Service (QoS) requirements, it is possible to have different priority schemes or fixed priority levels for threads which can be exposed to and controlled by the Operating System. For instance, pinning a thread to the highest priority level ensures that the thread runs at its full speed as instructions from other threads are selected only when they do not conflict with the highest priority thread.

2.5 Summary

This section has presented the baseline VLIW architectural configuration that is used for the evaluations done in the thesis. We have described the VEX VLIW architecture, the

underlying toolchain and the changes required in VEX toolchain for a cycle accurate simulation of the multithreaded architecture. We have also discussed the benchmarks and the workloads that are used for performance evaluations, and how the workloads are run on the multithreaded simulator.

Chapter 3

Cluster-level Simultaneous MultiThreading (CSMT)

In this chapter, we discuss our first proposal, Cluster-level Simultaneous MultiThreading (CSMT) targeted towards clustered VLIW processors. Because of cache misses and insufficient Instruction Level Parallelism in many applications, VLIW processors suffer high resource under-utilization. Multithreading techniques like Block MultiThreading (BMT) and Interleaved MultiThreading (IMT) reduce memory wait cycles because of cache misses but do not improve function unit utilization. Simultaneous MultiThreading (SMT) at operation-level reduces function units under-utilization but requires expensive hardware. CSMT is a restricted form of SMT that can reduce a significant amount of functional units under utilization. CSMT hardware cost is comparable to low cost multithreading schemes like Interleaved MultiThreading while achieving performance close to the more complex multithreading techniques.

3.1 Introduction

Clustered VLIW processors are an attractive choice in embedded domain as they allow to exploit Instruction-Level Parallelism (ILP) at low hardware cost. Clustered VLIW processors offer multiple functional units and multiple issue capabilities. However, many applications are not able to use all the processor resources which results in an under-utilization of these resources. Resource under-utilization can be divided into two categories [57].

1 **Vertical waste:** The cycles where no instruction is issued at all.

2 **Horizontal waste:** is the under-utilization of the issue width of the processor i.e. less operations are issued at a given cycle than the processor issue width.

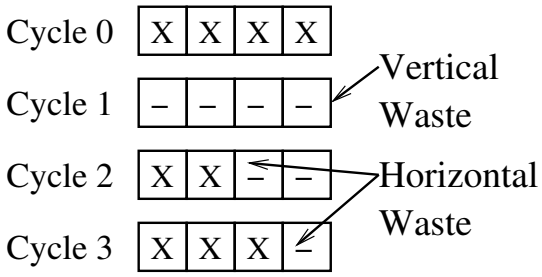


Figure 3.1: **Horizontal and Vertical Waste**

Figure 3.1 shows an example of the horizontal and vertical waste. The figure shows a 4-issue VLIW processor and boxes marked with 'X' represent that the issue slot is used. All four issue slots are used in cycle 0, but no issue slots are used at all at cycle 1 resulting in vertical waste. At cycles 2 and 3, only 2 and 3 issue slots are used respectively giving rise to horizontal waste. Both horizontal and vertical waste arise because control and data dependencies in a program limit the number of instructions that can be issued at a given cycle. Besides, runtime events like cache misses result in a significant amount of vertical waste.

Multithreading is a popular approach to improve resource utilization and mitigate horizontal and vertical waste in modern processors. Multithreading approaches like Block MultiThreading (BMT) [62] and Interleaved Multithreading (IMT) [50], as already explained in Chapter 1, target the vertical waste cycles in the processor. BMT context switches the running thread with another thread if a high latency event is encountered by the running thread. IMT maintains several thread contexts and does a context switch every cycle issuing instructions from a different thread. Both BMT and IMT reduce the vertical waste but do not reduce horizontal waste. Simultaneous MultiThreading (SMT) [57] reduces horizontal waste as well. While BMT and IMT issue instructions from only one thread at a given cycle, SMT can issue instructions from different threads at the same cycle improving utilization of the processor FUs. SMT takes advantage of both Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP) to improve processor performance.

SMT fits naturally with out-of-order superscalar processors because of the already present dynamic issuing capability [57]. However, implementing SMT on VLIW processors require a thread merging hardware [20] to combine VLIW instructions from different threads into a single issue packet. An example of SMT issue on a VLIW processor is shown in Figure 3.2. Figure 3.2(a) shows two VLIW instructions belonging to two

different threads. The first instruction has two operations in the issue slots IS0 and IS2, and the second instruction has 2 operations in the issue slots IS0 and IS1. As shown in Figure 3.2, the merging hardware does resource constraints checking and also reroutes the operations `shl` and `mov` from Thread 1 to form the final issue packet. As a VLIW instruction consists of multiple operations, we refer to SMT on VLIW processors to as OpSMT henceforth. The cost of the merging hardware in OpSMT is non trivial and increases significantly with the number of threads limiting the scalability of OpSMT beyond 2 threads [41]. The implementation complexity and overhead of the thread merging hardware can be attributed to:

1. The merging hardware must honor the processor resource constraints (number of FUs, issue width etc.). Hence, a resource conflict detection hardware is required to check whether instructions from different threads can indeed be issued simultaneously.

2. Functional units in a VLIW processor are tied to the issue slots. For instance, multiply capability may be present only at issue slots 1 and 2. To avoid resource conflicts, the operations of the instructions may have to be rerouted to different issue slots. This necessitates an instruction rerouting logic to reroute and pack the operations on a VLIW instruction from different threads into a single VLIW issue packet.

The cost of thread merging hardware in terms of area and power consumption limits the scalability of OpSMT for VLIW Processors for embedded systems. Other multithreading techniques like BMT and IMT have a low implementation cost and complexity suitable for an embedded environment as they do not require any instruction merging capability. However, both BMT and IMT have lower performance as compared to OpSMT because of their inability to reduce horizontal waste. Our proposal, Cluster-level Simultaneous MultiThreading (CSMT), has a low implementation complexity comparable to IMT, and provides a limited capability to remove horizontal waste. The following Section explains CSMT in details.

3.2 Cluster-level Simultaneous MultiThreading

Cluster-level Simultaneous MultiThreading (CSMT) [18], [19] is a restricted form of SMT where the resource granularity is at the cluster level instead of individual operations. Hence, in CSMT, resource conflicts are resolved at cluster level and CSMT issues simultaneously VLIW instructions from different threads only if the threads use different clusters. Merging VLIW instructions at cluster-level is significantly cheaper than at

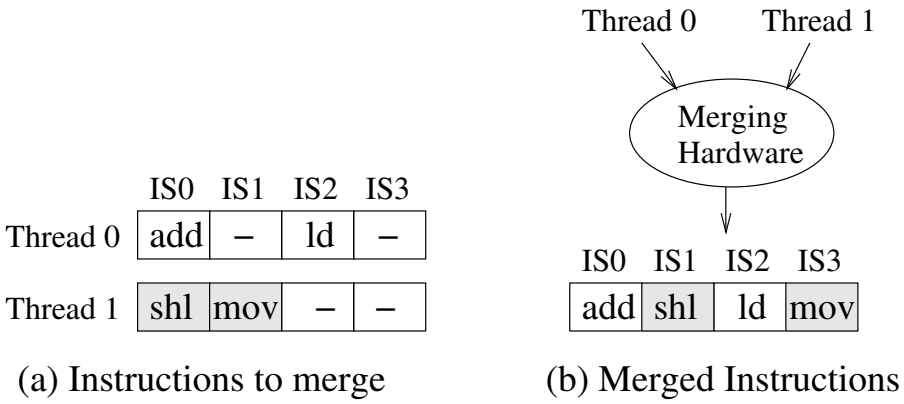


Figure 3.2: **SMT Merging**

operation-level as done in OpSMT. In fact, CSMT has an implementation complexity closer to IMT while achieving much higher performance than IMT.

To illustrate how instructions from different threads are merged in CSMT, Figure 3.3(a) displays 3 pairs of instructions belonging to different threads for a 4-cluster 2-issue per cluster (8-issue) architecture. In the figure, operations in the white background belong to Thread 0 and operations with a gray background belong to Thread 1. Note that when two instructions are merged, they have to be merged in their entirety i.e. it is not possible to choose only some operations from a VLIW instruction because doing so breaks VLIW execution semantics. Note that if both CSMT and OpSMT can merge a pair of instructions, the final merged instruction is identical for both OpSMT and CSMT. However as CSMT merging is stricter than OpSMT, if a pair of instructions can be merged by CSMT, it can always be merged by OpSMT but not vice-versa. The final instructions obtained by merging are shown in Figure 3.3(b). Neither CSMT nor OpSMT can merge Pair I because of conflicts at clusters 0, 1 and 3, both at operation-level and cluster-level. Pair II can be merged by OpSMT since there are no conflicts at operation-level. CSMT, however, cannot merge this pair, since both instructions in the pair use clusters 0, 2 and 3. As CSMT checks resource conflicts at the cluster-level, there is a conflict at these clusters. Pair III, however, can be merged by CSMT (and OpSMT as well) as the first instruction uses only clusters 1 and 2, which are not used by the other instruction. Note that in CSMT some FUs of each cluster can be idle despite the cluster being in use, since CSMT considers a cluster is in use when some operation is assigned to the respective bundle, and the bundle may use only a few FUs if there is not enough available ILP (the bundle may contain nops).

To further illustrate how CSMT works, a sample multithreaded execution for a 4-

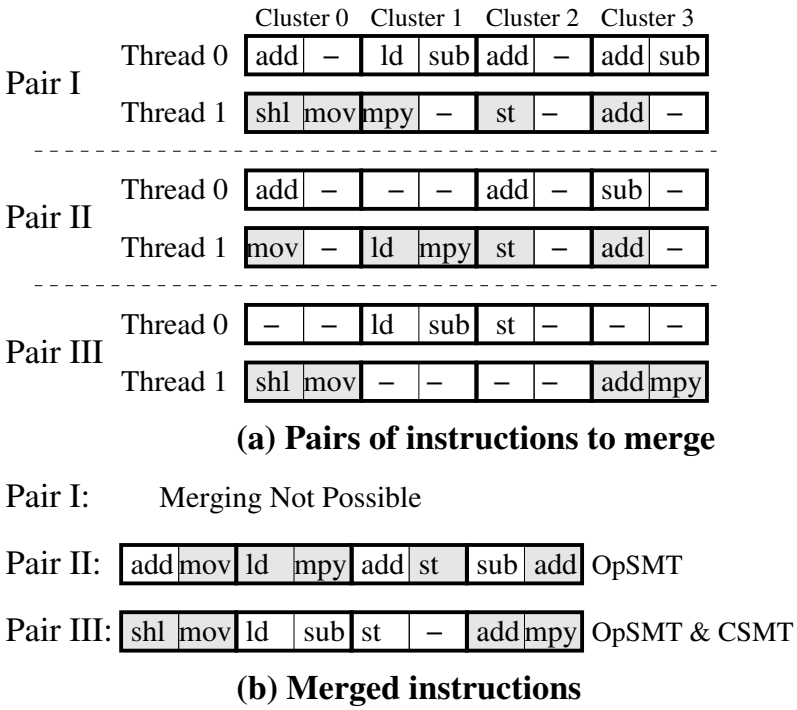
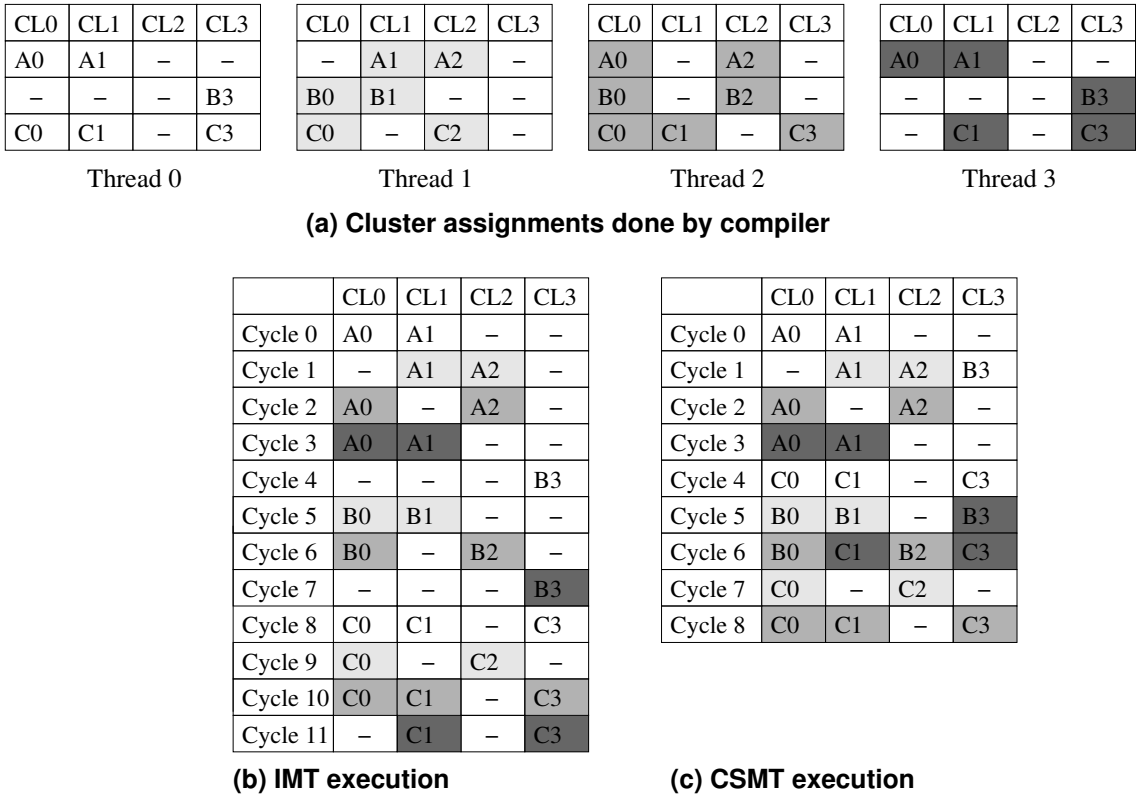


Figure 3.3: **Instruction Merging in OpSMT and CSMT**



	CL0	CL1	CL2	CL3
Cycle 0	A0	A1	–	–
Cycle 1	–	A1	A2	–
Cycle 2	A0	–	A2	–
Cycle 3	A0	A1	–	–
Cycle 4	–	–	–	B3
Cycle 5	B0	B1	–	–
Cycle 6	B0	–	B2	–
Cycle 7	–	–	–	B3
Cycle 8	C0	C1	–	C3
Cycle 9	C0	–	C2	–
Cycle 10	C0	C1	–	C3
Cycle 11	–	C1	–	C3

	CL0	CL1	CL2	CL3
Cycle 0	A0	A1	–	–
Cycle 1	–	A1	A2	B3
Cycle 2	A0	–	A2	–
Cycle 3	A0	A1	–	–
Cycle 4	C0	C1	–	C3
Cycle 5	B0	B1	–	B3
Cycle 6	B0	C1	B2	C3
Cycle 7	C0	–	C2	–
Cycle 8	C0	C1	–	C3

Figure 3.4: **IMT and CSMT execution on a 4-thread 4-cluster architecture using round robin priority**

thread 4-cluster architecture is shown in Figure 3.4. Figure 3.4(a) represents 4 different threads and the cluster assignment done by the compiler for each bundle (group of operations assigned to a cluster). Letters A to D represent a bundle scheduled in clusters 0 to 3 (CL0-CL3) and the subscript indicates the execution cycle in a single-thread environment. So, A_2 means the group of operations belonging to bundle A that are assigned to cluster CL0 and scheduled at cycle 2 by the compiler. For instance, for the example shown in Figure 3.3(a), bundles A contain operations of Cluster 0, bundles B contains operations of Cluster 1 etc. For a N-issue per cluster architecture, a bundle may have up to N operations.

Figure 3.4(b) shows the execution of the 4 threads on an IMT architecture assuming no stalls. At cycle 0, instructions from Thread 0 are issued. At cycle 1, instruction from Thread 1 are issued and so on. In total, the execution would require 12 cycles on an IMT architecture. Figure 3.4(c) shows the execution of the 4 threads on a CSMT architecture. For CSMT execution, the priority of thread assignment changes at each cycle following a round robin policy. Thread 0 has initially the highest priority. Thus, cycle 0 starts by assigning bundles A_0 and B_0 from Thread 0 to clusters CL0 and CL1. Thread 1 cannot be scheduled because of collision at cluster CL1, and Threads 2 and 3 cannot be scheduled either due to collision at cluster CL0. At cycle 1, the highest priority is assigned to instruction belonging to Thread 1 following the round robin scheme. Therefore, bundles B_0 and C_0 from Thread 1 are assigned to clusters CL1 and CL2. Bundles from Thread 2, 3 cannot be scheduled due to collision at clusters CL2 and CL1 respectively, but bundle D_1 from Thread 0 can be scheduled as there are no collisions. The highest priority is assigned to Thread 2 in the next cycle, and so on. While an IMT machine takes 12 cycles to execute the 12 instructions, CSMT, however, requires only 9 cycles, as shown in Figure 3.4(c).

CSMT improves performance over IMT because of its ability to remove horizontal waste. However, the merging of instructions is more restrictive in CSMT than OpSMT as described earlier. This restriction, however, allows a very cheap and low complexity hardware implementation for CSMT as compared to OpSMT.

To compute resource conflicts amongst instructions from different threads, OpSMT thread merging hardware requires functional unit usage information for each operation of the instructions. Hence, a predecoding hardware is required to compute functional unit usage of each operation. CSMT thread merging hardware does not need functional unit usage information and therefore does not require any predecoding. To compute resource

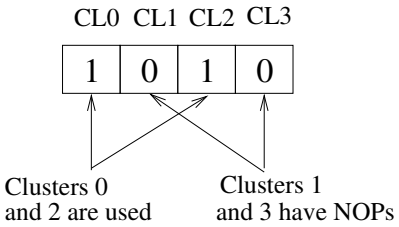


Figure 3.5: **Cluster Usage Vector**

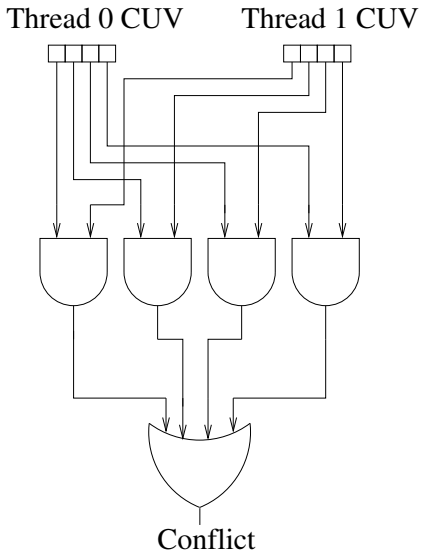


Figure 3.6: **CSMT conflict checking for 2 threads**

conflicts amongst instructions, CSMT thread merging hardware just needs a cluster usage vector (CUV) per thread. A CUV is a C bit-wide array, where C is the number of clusters. Every set bit in CUV indicates the presence of operations in that cluster. The entries of CUV are sorted in the increasing order of the clusters, with MSB denoting cluster 0 and LSB cluster $C-1$. E.g. CUV 1010 means that clusters 0 and 2 are used but clusters 1 and 3 have nops, as shown in Figure 3.5. Two threads have a resource conflict if, for any CUV position, both threads have the corresponding bit set. Figure 3.6 shows the CSMT conflict checking logic using CUVs for 2 threads for a 4-cluster architecture. Computing CUV of a thread is simple and the computation can be integrated within the instruction fetch pipeline stage. Using CUVs to compute resource conflicts makes the CSMT merging hardware cheap, simple and scalable.

OpSMT thread merging hardware also requires a complex operation repacking logic to reroute operations from different threads to different issue slots. For instance, in Figure 3.2, OpSMT merging hardware reroutes operation `shl` from issue slot IS0 to IS2. On the

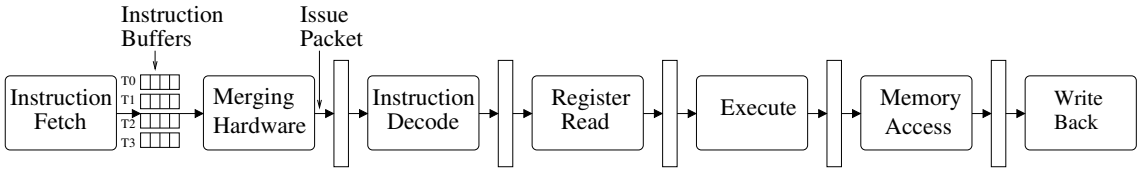


Figure 3.7: **Pipeline with extra stage for merging hardware**

other hand, CSMT merging hardware does not need to do any rerouting as a bundle in CSMT has operations from only one thread at a given time. These factors allows CSMT to have a very cheap implementation with a hardware complexity comparable to IMT rather than OpSMT.

Following sections compare the performance and implementation complexity results of CSMT and OpSMT.

3.3 CSMT Performance Analysis

This section presents the performance evaluation of CSMT and comparison with other multithreading schemes namely IMT and OpSMT. For the performance comparison across different multithreading schemes, we evaluate multithreading performance of single-thread, 2-Thread and 4-Thread configurations for IMT, CSMT and OpSMT. The performance is measured as the IPC throughput of the multithreaded workloads. The architectural parameters are the same for IMT, OpSMT and CSMT. However, to account for the delay of thread merge hardware, an extra pipeline stage is assumed for both CSMT and OpSMT. Figure 3.7 shows the pipeline with an extra pipeline stage for the thread merging hardware. The addition of this extra pipeline stage results into the increase in the taken branch penalty for CSMT and OpSMT to 2 cycles instead of 1 for IMT. Note that CSMT thread merge hardware may fit with the decode stage, which makes the presented CSMT results pessimistic. Also note that despite OpSMT thread merge hardware has a much higher delay than CSMT, we still assume that OpSMT hardware also fits in one pipeline stage. Hence, the results for OpSMT are optimistic. We use two memory models for the performance evaluation, a perfect memory model without any cache misses and a real memory model that takes cache misses into account. The details of the two memory models are available in Chapter 2.

Figure 3.8 presents performance results in terms of IPC for the different multithreading configurations for the 4-thread workloads discussed in Chapter 2. In the figure, Base-

line is the original single-thread base architecture, IMT2, CSMT2 and OpSMT2 are 2-thread processor configurations, and IMT4, CSMT4 and OpSMT4 denote 4-thread processor configurations for IMT, CSMT and OpSMT respectively. To evaluate the impact of an extra pipeline stage on single-thread performance, we also evaluate configuration Extrapipe that represents the single-thread base architecture with an extra pipeline stage. For each bar, the filled portion shows the results for a real memory configuration and the white portion on top represents the extra performance achieved in a perfect memory model without cache misses.

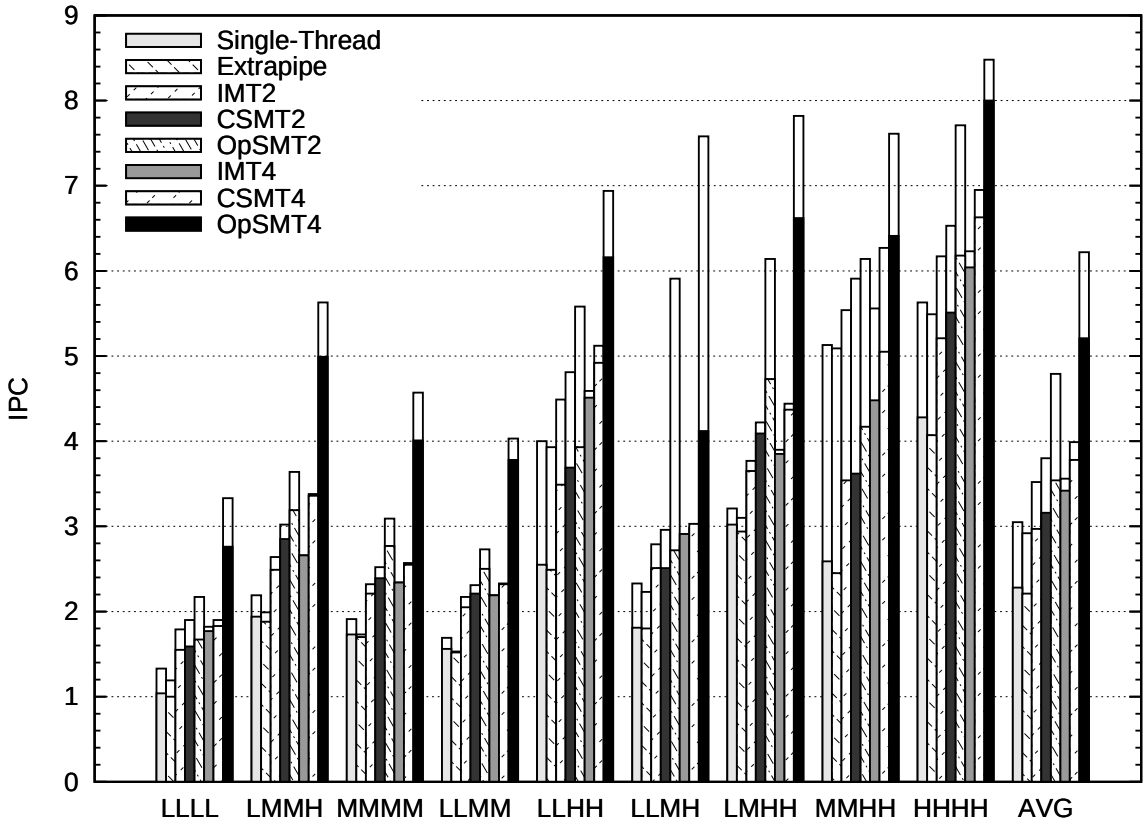


Figure 3.8: **CSMT performance**

With a perfect memory model, the first thing to notice is that for the single thread configurations (single thread and extrapipe), extrapipe has a a small degradation performance because of the effect of the extra pipeline stage. Also note that IMT does better than the baseline single-thread configurations. This is because, although there is no vertical waste due to memory stalls, some issue cycles are still lost due to taken branches and def-to-use latency of operations like loads, multiplies, and comparisons. Our IMT implementation also hides this vertical waste by issuing instructions from an alternate thread. On average, CSMT has an improvement of 7.9% and 12.1% over IMT for the 2-thread

and a 4-thread configurations respectively. Since a OpSMT processor can efficiently mix operations of two threads, it achieves higher performance than CSMT, on average 20.7% and 35.8% for the 2-thread and a 4-thread configurations respectively. Also note that IMT with a four-thread configuration (IMT4) achieves almost the same performance as a two-thread IMT configuration (IMT2). This happens because in a perfect memory model, little opportunities to further remove vertical waste exist. However, both CSMT and OpSMT performance improves significantly when number of threads increase to 4 because of their ability to hide horizontal waste as well. On average, OpSMT improves by 47% and CSMT performance improves by 19.6% when moving from 2-thread (OpSMT2 and CSMT2) to a 4-thread configuration (OpSMT4 and CSMT4).

When a real memory model is considered, the performance of single-thread configurations (single thread and extra-pipe) degrade significantly, while IMT, CSMT and OpSMT suffer relatively only a minor impact. This shows the ability of these approaches to hide vertical waste. On an average, CSMT achieves 6.6% and 10.7% higher performance than IMT. Also CSMT performance is with 10.6% and 27.3% of OpSMT performance for the 2-thread and the 4-thread configurations respectively.

Notice that for single-thread configurations, there is a small performance degradation when an extra pipeline stage is assumed (single-thread vs extrapipe). This degradation will appear in CSMT architectures when executing a single thread. However, that will happen only if the extra pipeline stage is actually required to meet cycle time constraints. If the extra pipeline stage is not required, the CSMT performance for multithreaded configurations will be better than the results presented in this thesis (hence, CSMT results presented here are ”pessimistic”).

Despite the fact that CSMT has better performance than IMT, the performance improvement was lower than our expectations. An analysis of the lower performance improvements pointed to a cluster usage bias in the programs. Chapter 4 discusses the usage bias in greater details and proposes solutions for reducing it to achieve much higher performance with CSMT. OpSMT achieves higher performance than CSMT at the cost of a more expensive merging hardware. The following section discusses the merging hardware cost of CSMT and OpSMT.

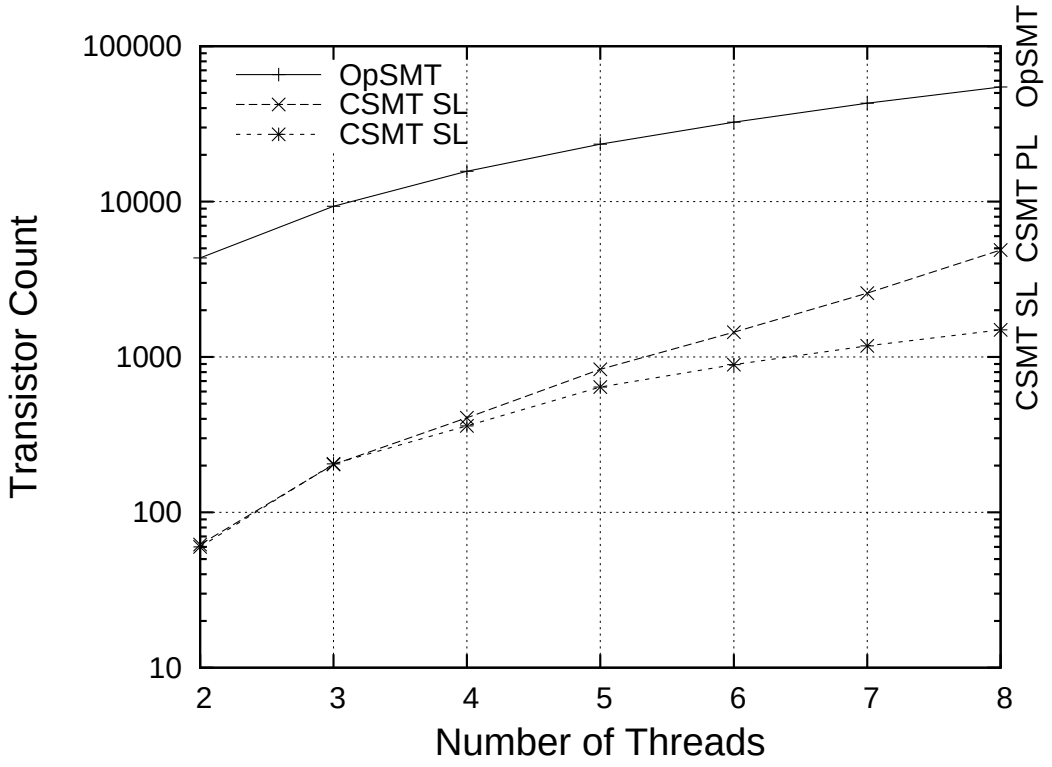


Figure 3.9: **Thread merging hardware transistor count for OpSMT and CSMT**

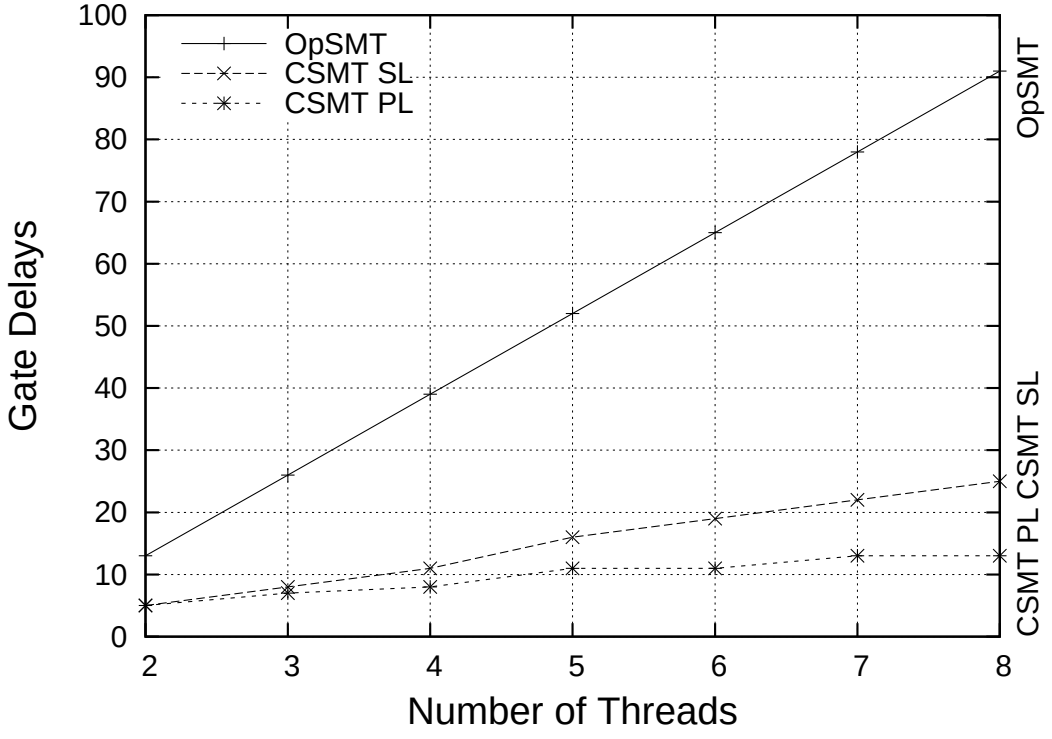


Figure 3.10: **Thread merging hardware gate delays for OpSMT and CSMT**

3.4 Merging Hardware Complexity Analysis

This section presents the results for complexity comparison of the thread merging hardware for CSMT and OpSMT in terms of transistor count and gate delays. The implementation details of the CSMT and OpSMT thread merging hardware and the methodology used for computing gate delays and transistor count is available in the Appendix.

Two implementations of the thread merging hardware are considered for CSMT viz. serial and parallel. The serial implementation is a cascading logic that checks a different thread at each level. The parallel implementation, on the other hand, checks in parallel all possible thread selections. The parallel implementation has a lower delay and is functionally equivalent to the serial implementation. However, parallel implementation has a higher hardware overhead, which grows exponentially with the number of threads. For OpSMT thread merging hardware, an implementation similar to the CSMT Serial approach is considered. The parallel approach is not feasible for OpSMT because the cost of checking, in parallel, all possible thread selections is prohibitively expensive in terms of area.

Figures 3.9 and 3.10 show the cost of the thread merging hardware with varying number of threads for a 4-cluster 4-issue per cluster architecture for both CSMT and OpSMT. Figure 3.9 shows on a log scale the cost in terms of number of transistors required, and Figure 3.10 shows the cost in terms of gate delays. In the figures, labels 'CSMT PL' and 'CSMT SL' refer to the parallel and serial implementations of CSMT thread merging hardware, while label 'OpSMT' refers to the OpSMT thread merging hardware.

As shown in the Figure 3.9, the cost of the thread merging hardware for OpSMT increases significantly with the number of threads and constrains its scalability. OpSMT requires much higher number of transistors than either of the two CSMT implementations. For instance, OpSMT requires close to 20,000 transistors for four threads while both CSMT implementations require an insignificant (around 300) number of transistors. Due to its high cost, OpSMT is expensive to implement for a large number of threads. In fact, previous studies [41] that have done an evaluation of the hardware required for OpSMT on VLIW limit the number to only 2. When the Serial and Parallel approaches are compared for CSMT, the parallel approach does require more transistors but the difference is small. Besides, the number of transistors required by the parallel approach for 8 threads is less than the transistor requirement of 2-Thread OpSMT.

When gate delays is used as the complexity comparison metric (Figure 3.10), it is

evident that Serial approach has higher delay than the Parallel approach for CSMT. The difference between the two approaches is small when the number of threads is low but increases significantly with the number of threads. Hence, for higher number of threads, Parallel approach is better. If we compare the gate delays of OpSMT with CSMT, it is clear that OpSMT has much higher gate delays than CSMT. The high gate delays for OpSMT is another reason for poor scalability of OpSMT. In comparison, CSMT gate delays are lower and hence, more threads can be supported. Higher delays in OpSMT thread merging hardware can be tolerated by using extra pipeline stages. However, adding extra pipeline stages degrades the single-thread performance, which may not be acceptable.

Assuming that only a single extra pipeline stage is acceptable (single-thread performance degradation is small) and a limit of 10,000 transistors, a processor with up to eight threads and four clusters is feasible with the CSMT parallel logic based design, as the transistor count is not yet a problem and delays are small enough to fit into a single pipeline stage. For the 4-thread 4-cluster architecture used in our evaluations both, the delay and the transistor count overhead, are very small for CSMT. Hence, our claim that for the configurations evaluated, CSMT has almost the same cost as IMT.

3.5 Summary and Conclusions

This chapter has presented Cluster-level Simultaneous MultiThreading (CSMT) for clustered VLIW Processors. CSMT targets both horizontal and vertical waste in a processor at a low implementation complexity. In CSMT, instructions from different threads are issued simultaneously only if they use different clusters. The results show that the CSMT implementation cost is close to low complexity multithreading schemes like Interleaved MultiThreading (IMT) while achieving higher performance than IMT. However, the performance improvement was lower than expected because of a cluster usage bias in most of the applications. The following chapter discusses this bias in greater details and proposes a simple solution to mitigate it.

Chapter 4
Cluster Renaming

This Chapter presents Cluster Renaming, a technique to improve multithreading performance of Cluster-level and Operation-level Simultaneous MultiThreading (CSMT and OpSMT). The performance of CSMT, presented in the previous chapter, improved significantly on IMT performance. But the performance improvement was lower than expected. An analysis of the benchmarks showed that the clusters used by an application are biased towards the first clusters. This bias results into a high contention for the first clusters and hence, the performance improvement is low. Cluster Renaming reduces this bias by allocating different clusters than assigned by the compiler. Doing so reduces resource conflicts and results in better multithreading performance for both CSMT and OpSMT.

4.1 Motivation

As explained in Chapter 3, CSMT always performed better than IMT. The improvement, however, was lower than what we expected. As shown in Figure 3.8, CSMT had an improvement of 6.6% on a 2-thread machine and 10.7% on a 4-thread machine over IMT, which though not insignificant, was not up to our expectations. We found out that the low performance improvement is because of a cluster usage imbalance amongst the threads. There is a strong bias in favor of the first clusters (in particular cluster 0) since the compiler always starts assigning the operations with cluster 0. As a result, most threads compete for use of first clusters, while the other clusters are used less frequently. This results in high resource conflicts at first clusters limiting the opportunities for merging instructions of different threads. The rest of the Section presents a detailed cluster usage analysis of the benchmarks.

Figure 4.1 shows the percentage of time a given number of clusters are used in a 4-cluster architecture for three benchmarks viz. colorspace [1], mcf [22] and djpeg [34].

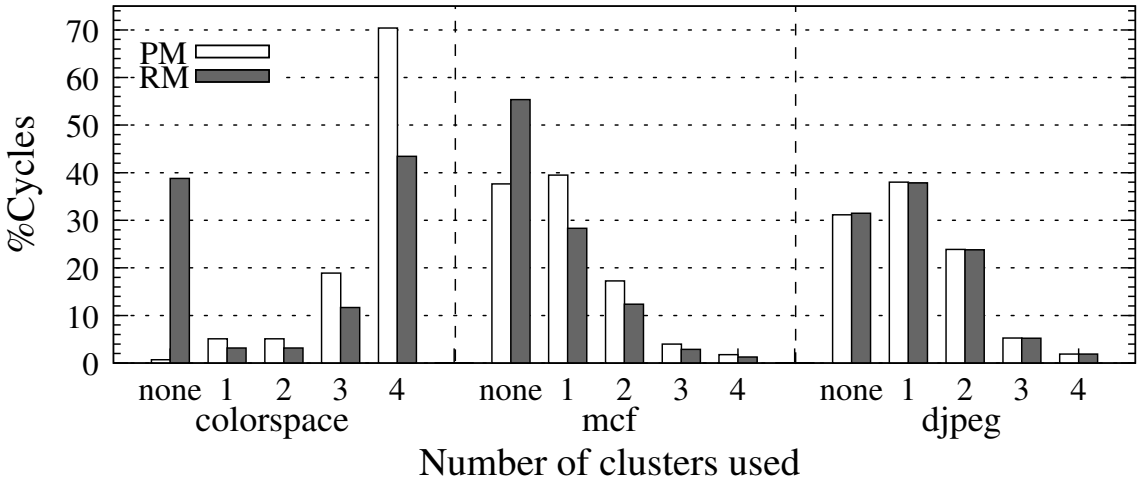


Figure 4.1: **Cluster usage**

The figure presents data for the perfect memory model (PM) with no cache misses and the more realistic memory model (RM) explained previously in Chapter 2. Colorspace benchmark has a high IPC, close to 6 with a 8-issue width and close to 9 with a 16-issue width. Due to this high IPC, with the perfect memory model all the clusters are simultaneously in use most of the time. However, when the real memory model is considered, a significant amount of time is wasted in handling the cache misses (close to 40% of the execution). In addition, many applications exhibit much lower ILP. For instance, the average IPC in SPEC CPU Integer 2000 [22] or Mediabench [34] benchmarks range, in general, between 1 and 2. Thus, when repeating the previous experiment using SPEC and Mediabench benchmarks, we obtain significantly different results for the cluster usage, as mcf and djpeg examples show in Figure 4.1. A considerable amount of time is spent in cycles where no cluster is used. However, when any cluster is used, the cluster usage is very unbalanced and most of the time only a single cluster is used.

Figure 4.2 shows the percentage of cycles in which each cluster is in use (whenever any cluster is used) for mcf and djpeg benchmarks. As can be seen, there is a heavy cluster load imbalance in both programs. Cluster 0 is the most heavily used and there is little use of other clusters. This is reasonable, since the compiler tries to schedule as many operations as possible in a single cluster to avoid communication overhead. Only a small number of clusters thus are simultaneously used most of the time, since there is not always enough ILP available during the program's execution.

The use of CSMT improves the cluster utilization but the threads compete for the same clusters most of the time, rather than using other clusters which are heavily under-utilized.

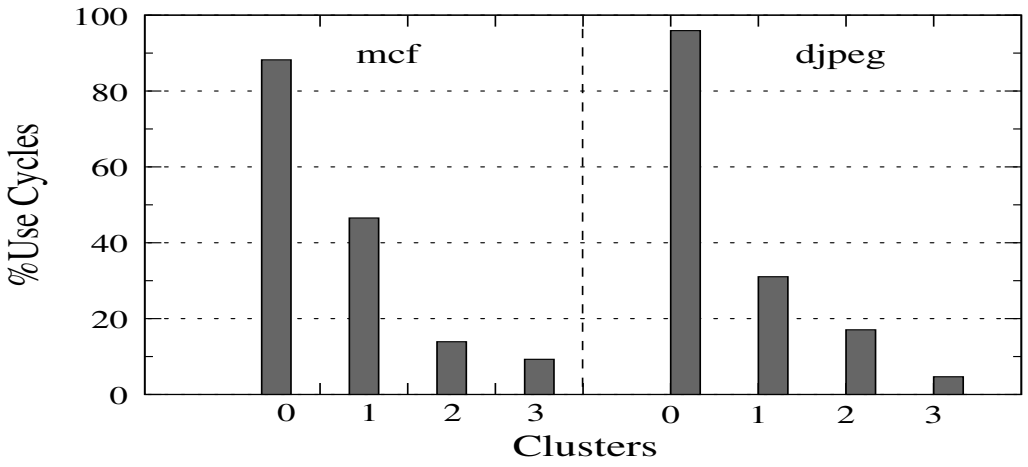


Figure 4.2: **Individual cluster use in mcf and djpeg**

In particular, there is a very heavy contention for cluster 0, as can be easily derived from Figure 4.2. This explains the lower than expected performance improvement obtained by CSMT, as discussed in the previous chapter. To solve the problem of cluster usage imbalance, we propose a technique named Cluster Renaming. Cluster renaming remaps the compiler assigned Logical Clusters of different threads to different Physical Clusters at runtime so that cluster conflicts between threads are reduced. Cluster renaming is explained in details in the following Section.

4.2 Cluster Renaming

Cluster renaming is a technique to reduce resource conflicts amongst the running threads on a clustered VLIW architecture. Cluster renaming distributes the logical clusters assigned by the compiler to different physical clusters at runtime. This mapping is fixed for each logical CPU. While a thread is executing on a logical CPU, the physical cluster assignment for the thread is according to the fixed mapping of that logical CPU. We propose a simple but effective renaming function to remap compiler assigned clusters by assigning a cluster shift value to each thread. The shift value is based on the number of clusters and the number of simultaneous threads supported by the processor as shown in Equation (4.1). Figure 4.3 shows the shift tables for a 2-thread 4-cluster and a 4-thread 4-cluster architecture. For the 2-thread 4-cluster architecture, Thread 0 is shifted by 0 and Thread 1 is shifted by 2. On the 4-thread 4-cluster architecture, Thread 0 is shifted by 0, Thread 1 by 1, Thread 2 by 2 and Thread 3 by 3. Figure 4.4 shows the cluster renaming mechanism for a 2-thread , 4 thread and an 8-thread processor with 4 clusters. In the figure, LCi

Thread Num	*Shift*
0	0
1	2

(a) **Two Threads**

Thread Num	*Shift*
0	0
1	1
2	2
3	3

(b) **Four Threads**

Figure 4.3: **Shift tables for a 4-cluster architecture**

means logical cluster i assigned by the compiler and PCi means the physical cluster i. So, while logical cluster 0 on Thread 0 still maps to physical cluster 0, logical cluster 0 on Thread 1 may map to physical cluster 1, and so on. Note that if the number of threads is more than the number of clusters, more than 1 thread will have the same shift value. For instance, on an 8-thread 4-cluster processor, Threads 0 and 4, threads 1 and 5 etc. share the same logical to physical cluster mapping. Note that this renaming consists simply in rotating the original clusters by a fixed value. Cluster renaming is visible only to the processor and transparent to the compiler. This avoids any special compilation to achieve extra performance on a multithreaded platform. Note that an effect similar to renaming could have been produced by the compiler. Doing so, however, requires the compiler to know all the applications that will run simultaneously in a multithreaded environment.

$$Shift = Thread\_Num \times \begin{cases} \left\lfloor \frac{Number\ of\ Clusters}{Number\ of\ Threads} \right\rfloor, & (No.\ of\ Clusters \geq No.\ of\ Threads) \\ 1, & (No.\ of\ Clusters < No.\ of\ Threads) \end{cases} \tag{4.1}$$

The shift value is fixed for each thread and can be easily hardcoded in the hardware for a given number of threads and clusters. Thus, a very cheap cluster rename logic is possible by rerouting the wires from the instruction buffer of the threads to a physical cluster.

Besides renaming the clusters, the operands for any operation where an explicit cluster number is used also need to be renamed. In VEX architecture, only the inter-cluster communication operations send/recv use cluster numbers in their operands. The renaming of the operands is very simple and only requires adding the shift value to the cluster number. Hence, renaming for these operations can be done at any pipeline stage before execution of the operation, since the pipeline is tagged with a thread identifier. In fact, the operands can be renamed at the decode stage itself, where the logic for identifying the operations is already available. Cluster renaming requires that all clusters are homogeneous. This is

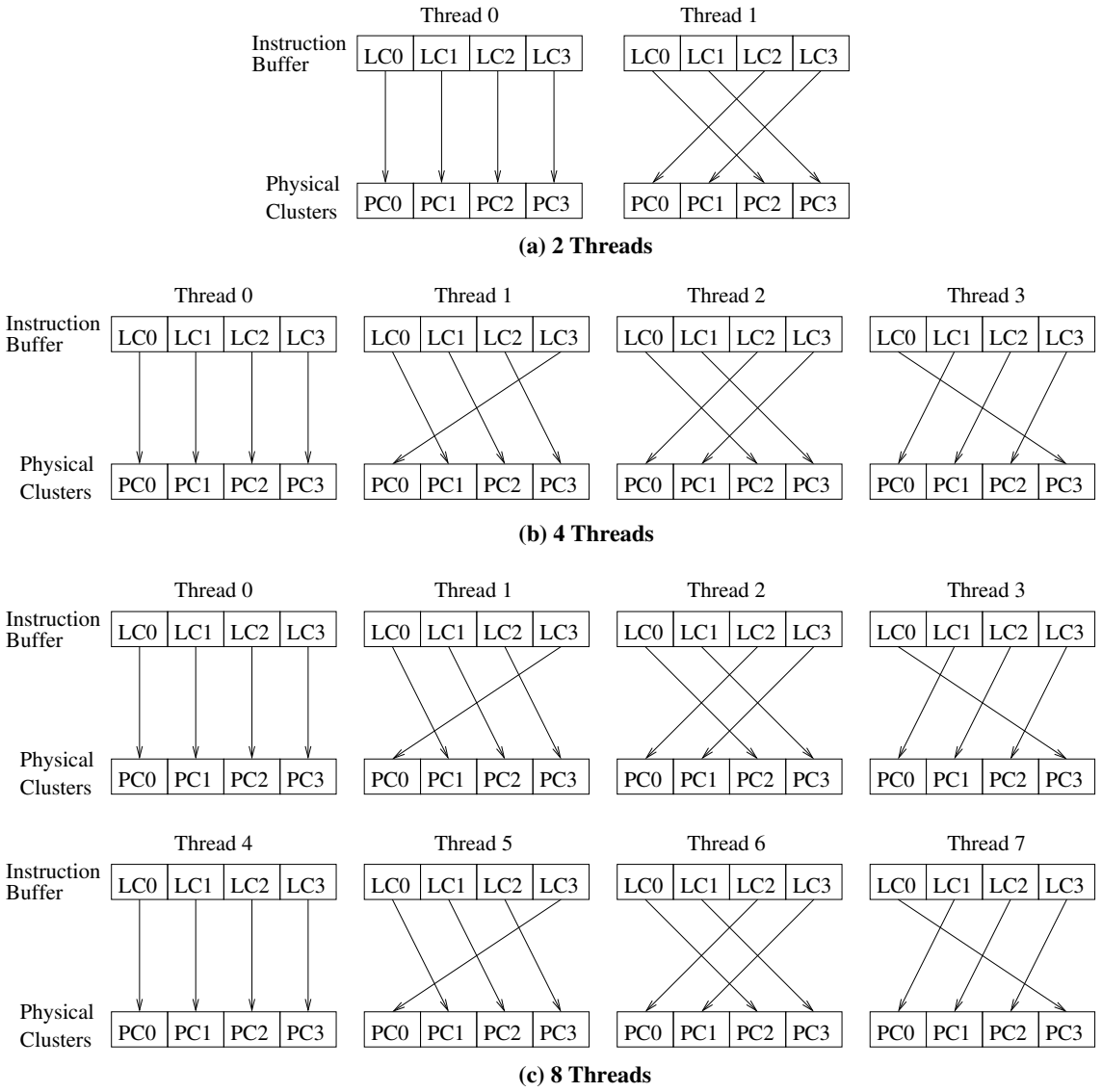


Figure 4.4: **Cluster renaming logic for a 4-cluster architecture**

usually true for most of the existing clustered VLIW processors and, in particular, for Lx. The only exception in Lx is the branch unit, which has to be replicated in all the clusters to allow cluster renaming. The cost of replicating the branch unit is very small due to its simplicity.

The following section presents an example of Cluster Renaming on a multithreaded system.

CL0	CL1	CL2	CL3
A0	A1	–	–
–	–	–	B3
C0	C1	–	C3

Thread 0

CL0	CL1	CL2	CL3
–	A1	A2	–
B0	B1	–	–
C0	–	C2	–

Thread 1

CL0	CL1	CL2	CL3
A0	–	A2	–
B0	–	B2	–
C0	C1	–	C3

Thread 2

CL0	CL1	CL2	CL3
A0	A1	–	–
–	–	–	B3
–	C1	–	C3

Thread 3

(a) Cluster assignments done by compiler

PC0	PC1	PC2	PC3
A0	A1	–	–
–	–	–	B3
C0	C1	–	C3

Thread 0

PC0	PC1	PC2	PC3
–	–	A1	A2
–	B0	B1	–
–	C0	–	C2

Thread 1

PC0	PC1	PC2	PC3
A2	–	A0	–
B2	–	B0	–
–	C3	C0	C1

Thread 2

PC0	PC1	PC2	PC3
A1	–	–	A0
–	–	B3	–
C1	–	C3	–

Thread 3

(b) Physical cluster assignments after cluster renaming

	CL0	CL1	CL2	CL3
Cycle 0	A0	A1	–	–
Cycle 1	–	A1	A2	B3
Cycle 2	A0	–	A2	–
Cycle 3	A0	A1	–	–
Cycle 4	C0	C1	–	C3
Cycle 5	B0	B1	–	B3
Cycle 6	B0	C1	B2	C3
Cycle 7	C0	–	C2	–
Cycle 8	C0	C1	–	C3

(c) CSMT execution before cluster renaming

	PC0	PC1	PC2	PC3
Cycle 0	A0	A1	A1	A2
Cycle 1	A1	B0	B1	A0
Cycle 2	A2	–	A0	B3
Cycle 3	C0	C1	B3	C3
Cycle 4	B2	C0	B0	C2
Cycle 5	–	C3	C0	C1
Cycle 6	C1	–	C3	–

(d) CSMT execution after cluster renaming without cache miss

	PC0	PC1	PC2	PC3
Cycle 0	A0	A1	A1	A2
Cycle 1	A1	B0	B1	A0
Cycle 2	A2	–	A0	B3
Cycle 3	C0	C1	B3	C3
Cycle 4	C1	C0	C3	C2
Cycle 5	B2	–	B0	–
Cycle 6	–	C3	C0	C1

(e) CSMT execution after cluster renaming and considering cache miss

Figure 4.5: **CSMT execution on a 4-thread 4-cluster architecture using cluster renaming.**

4.2.1 Cluster Renaming Example

To illustrate the effect of cluster renaming on performance, we present the same example for CSMT as in Figure 3.4. Figure 4.5(a) shows the original logical cluster assignment (LC0-LC3) done by the compiler for instructions of each thread and Figure 4.5(c) shows the CSMT execution when no cluster renaming is used. Figure 4.5(b) shows the physical mapping of clusters (PC0-PC3) done by the cluster renaming logic for each thread at execution time. The logical and physical assignment of clusters is the same for Thread 0, but is different for the remainder threads. For Thread 1, logical cluster 0 (CL0) is mapped to physical cluster 1 (PC1), CL1 to PC2, CL2 to PC3 and CL3 to PC0. In the same way, CL0 from Thread 2 is mapped to PC2 and so on, and CL0 from Thread 3 is mapped to PC3 and so on.

Figure 4.5(d) shows the execution of the 4 threads on a CSMT architecture with cluster renaming. Thread 0 has initially the highest priority. Thus, cycle 0 starts by assigning bundles A0 and A1 from Thread 0 to physical clusters PC0 and PC1. Bundles A1 and A2 from Thread 1, which are mapped at physical clusters PC2 and PC3, are also assigned as these physical clusters are free. More threads cannot be scheduled, as no free clusters are available.

At cycle 1, the highest priority is assigned to operations belonging to Thread 1 following a round robin scheme. So, bundles B0 and B1 from Thread 1 are assigned. Bundles from Thread 2 cannot be assigned due to conflict at PC2. Bundles from Thread 3, however, can be assigned to the free clusters. Bundles from Thread 0 are not scheduled since no free clusters are available. The highest priority is assigned in next cycle to Thread 2, and so on.

In a perfect memory machine without cache misses, the sequential execution of the four threads in a perfect memory model requires 12 cycles. CSMT takes 9 cycles without cluster renaming (Figure 4.5(c)), but with cluster renaming, CSMT execution time is further reduced to only 7 cycles, as shown in Figure 4.5(d).

Now, lets assume that Thread 2 has a cache miss in its first execution cycle at bundle A2, and a cache miss penalty of 2 cycles in this architecture. Due to the cache miss, the execution takes 14 cycles on a single-thread machine. The execution time of the four threads by using Interleaved MultiThreading is still 12 cycles, since the cache miss latency is hidden by the interleaving. With CSMT, when the cache miss of A2 is detected in Thread 2 at cycle 2, the thread is marked as blocked and no instruction from this thread is issued during next 2 cycles. Instructions from other threads are executed in these 2 cycles. The execution with CSMT still takes 7 cycles while absorbing the cache miss, as shown in Figure 4.5(e). As shown in this example, the use of cluster renaming improves CSMT performance from 9 cycles to 7 cycles, and further increases performance benefit over IMT.

The following section presents the detailed impact of cluster renaming on the workloads evaluated.

4.3 Results

This section presents the impact of cluster renaming for the IMT, CSMT and OpSMT multithreading techniques. First, we analyze the cluster usage analysis before and after

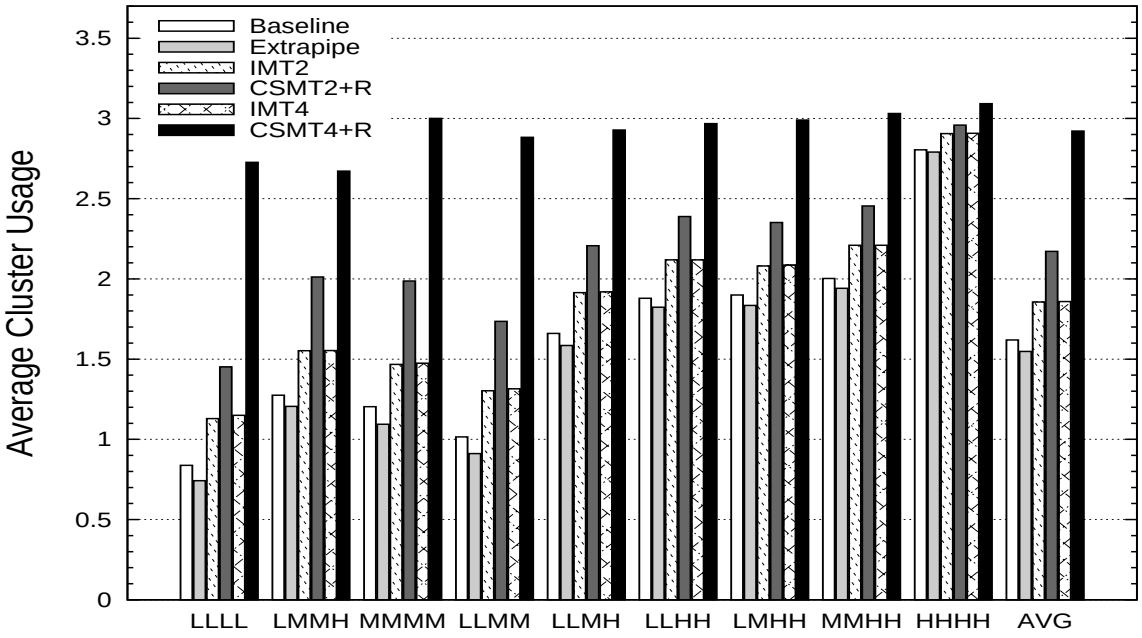


Figure 4.6: **Cluster usage with a perfect memory model**

cluster renaming for the workloads used in our experiments.

4.3.1 Cluster Usage Analysis

Cluster usage is the average number of clusters used per cycle by the workload during execution. It can range from 0 to the maximum value of 4. Figures 4.6 and 4.7 represent the cluster usage for the workloads used in our experiments. In the figures, Baseline is the original single-thread base architecture, IMT2 and CSMT2 are 2-thread processor configurations, and IMT4 and CSMT4 denote a 4-thread processor configuration for IMT and CSMT respectively. Use of cluster renaming is indicated by an explicit +R. For instance, CSMT4+R means a 4-thread CSMT architecture with cluster renaming. Also, IMT configurations do not include the extra pipeline stage while CSMT configurations are evaluated with the extra pipeline stage.

For the perfect memory model, the cluster usage improvement of CSMT over IMT on a 2-thread processor is only moderately better (19.4%) because of the limited opportunities to merge threads. However, when a 4-thread processor is used, the cluster usage improves quite significantly (62.4%), with an average cluster usage close to 3. Note that in a high ILP workload like HHHH, the improvement in cluster usage over IMT is the least. This is because very few opportunities to merge instructions exist, since both approaches already use a high number of clusters every cycle. However, in low ILP workloads like

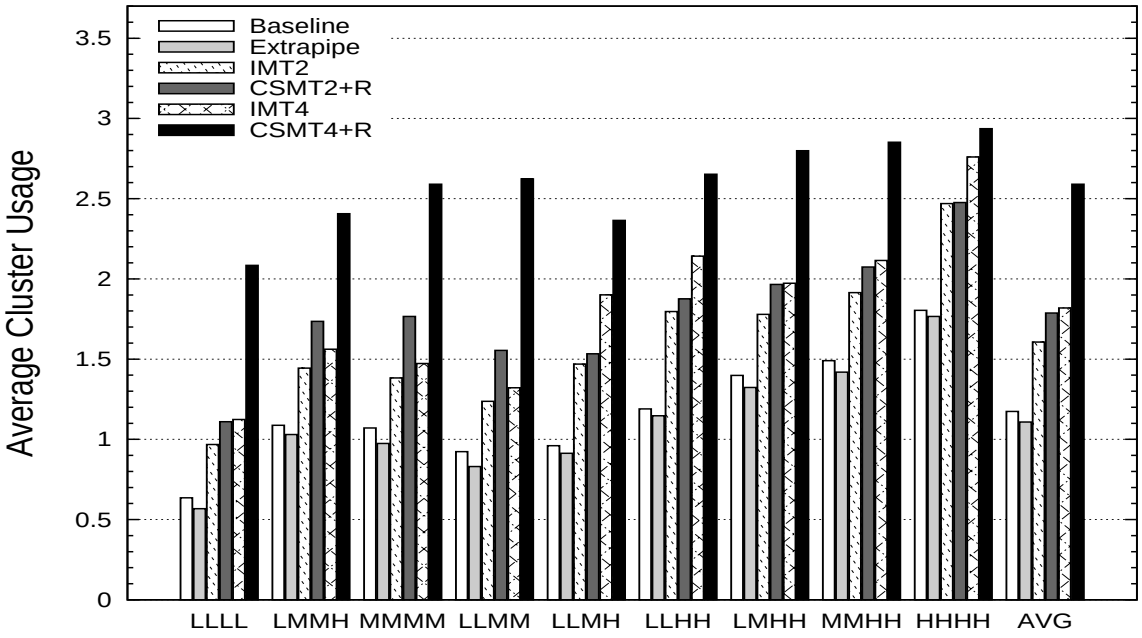


Figure 4.7: **Cluster usage with a real memory model**

LLLL, where many opportunities exist to merge instructions, CSMT has a significantly higher cluster usage (28.3% for 2-threads and 136.8% for 4-threads). With a real memory model, the improvement of CSMT over IMT in cluster usage is not so significant for a 2-thread processor, though CSMT still has a better cluster usage (13.6% higher). This is because the inter-thread parallelism most of the time is used to hide cache misses and IMT already does a good job at hiding cache misses. However, with a 4-thread processor, CSMT has a significantly higher cluster usage than IMT (47.6% on average), with a cluster usage improvement of 12.1% for even the least opportunity workload HHHH.

The following sections present a detailed performance impact of Cluster Renaming on the workloads evaluated.

4.3.2 Cluster Renaming Performance Improvements

To evaluate the benefit of cluster renaming, we evaluated performance of both CSMT and OpSMT techniques, both with and without cluster renaming, for a 2-thread and a 4-thread configuration. Figures 4.8 and 4.9 shows the speedup obtained over the performance without cluster renaming for CSMT and OpSMT for the real memory model. For CSMT, cluster renaming allows, on an average, a performance improvement of 8.3% with a 2-thread processor and 26.6% with a 4-thread processor. For particular cases like LLMM, a speedup of 86.2% is obtained. For an OpSMT processor, cluster renaming does not

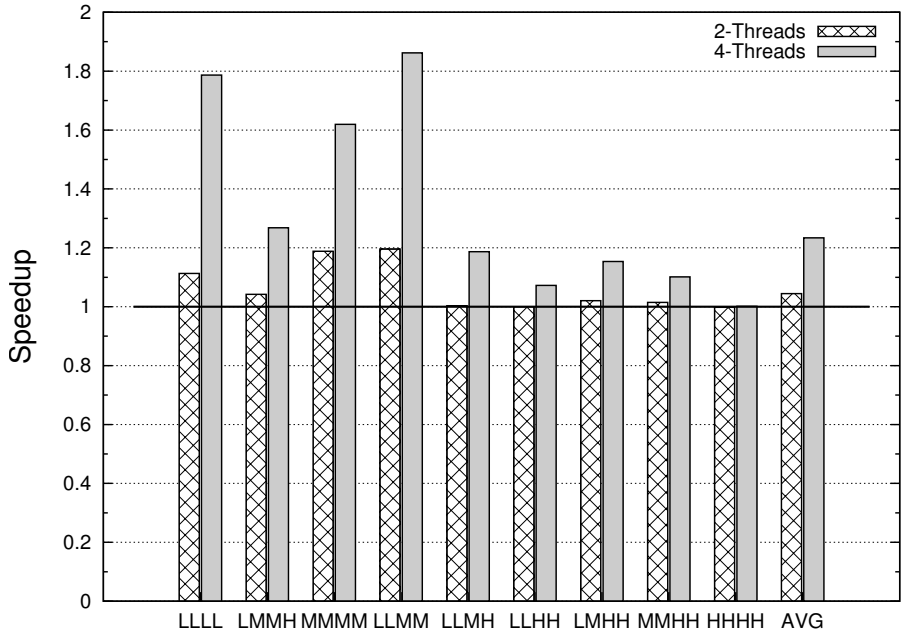


Figure 4.8: **Speedup in CSMT by use of cluster renaming**

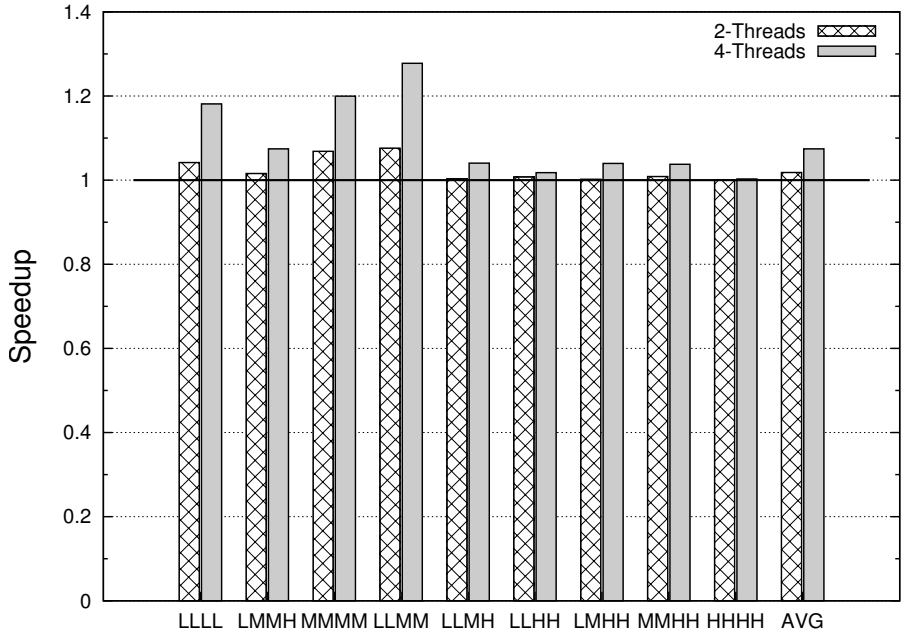


Figure 4.9: **Speedup in OpSMT by use of cluster renaming**

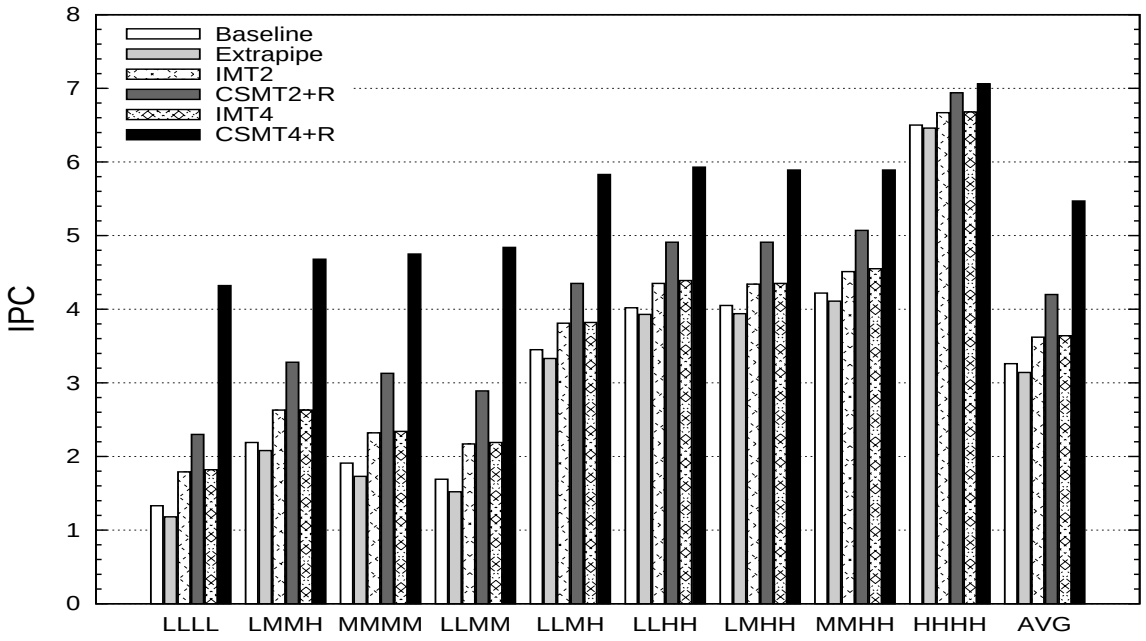


Figure 4.10: **CSMT vs IMT, Perfect Memory**

add much to the performance with 2 threads (4%), since a OpSMT processor can already efficiently mix operations of two threads. Also, a lot of parallelism gets used for hiding cache misses. With a 4-thread OpSMT processor, merging all threads in the same cluster is more difficult (there is an already high demand for resources in first clusters and having more threads worsens the situation). In this case, cluster renaming significantly adds to the performance (9.7% on average and 27.8% for the particular case of LLMM) by lowering the contention on heavily used clusters.

4.3.3 CSMT Performance Comparison With IMT

Figures 4.10 and 4.11 show the IPC obtained for the perfect memory model with no cache misses and the real memory model respectively for the single-thread configurations (single-thread and extrapipe), and 2-thread and 4-thread configurations for IMT and CSMT. With a perfect memory model (Figure 4.10), the first thing to notice is that IMT does slightly better than the baseline single-thread processor. This is because, despite the fact that there is no vertical waste due to memory stalls, a few issue cycles are still lost due to taken branches and def-to-use latency of operations like loads, multiplies and comparisons. Our IMT implementation also hides this vertical waste by issuing instructions from an alternate thread. Since CSMT hides horizontal waste as well, it clearly outperforms IMT, specially with a a 4-thread processor configuration. In this case, CSMT

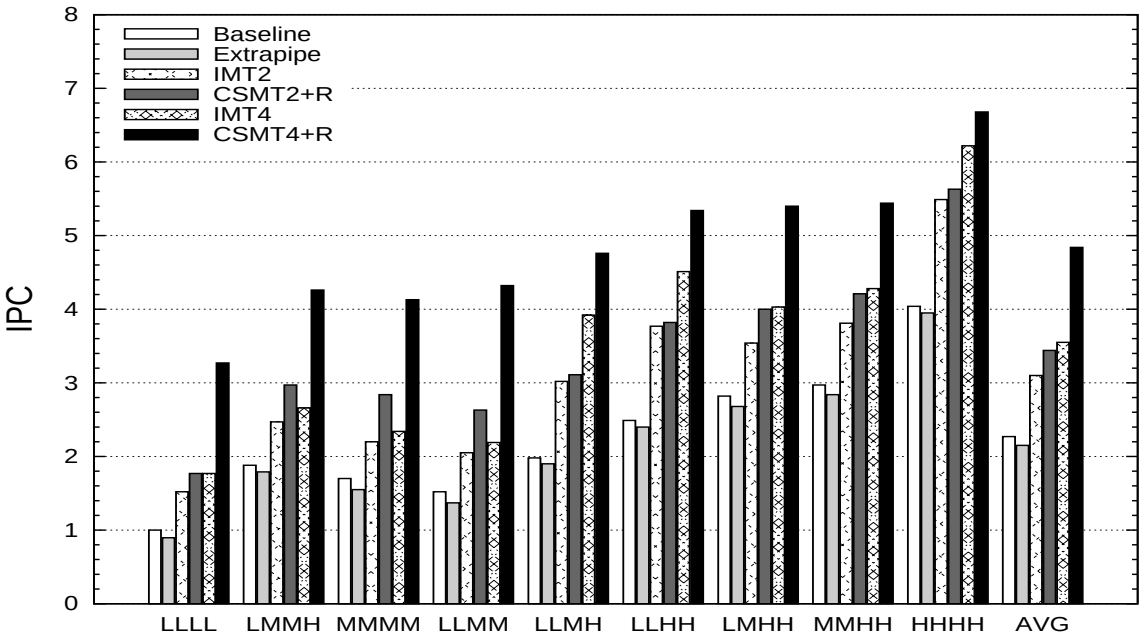


Figure 4.11: **CSMT vs IMT, Real Memory**

(CSMT4+R) has an average speedup of 77.6% over the baseline single-thread and 52% over a 4-thread IMT configuration. Notice that for low ILP workloads like LLLL, the speedup is as high as 225% over the baseline single-thread and 84% over the 4-thread IMT configuration. Also notice that IMT with a 4-thread configuration achieves almost the same performance as a 2-thread IMT configuration because little opportunities to remove vertical waste exist. CSMT with a 4-thread configuration, however, has a significant performance improvement (33.2%) over a 2-thread configuration (CSMT2+R), since there are more opportunities to reduce the horizontal waste. Even with the workload HHHH, CSMT has a visible improvement in IPC with a 4-thread configuration (7.6% over a 2-thread CSMT and 12% over a 4-thread IMT). Moreover, even a 2-thread CSMT configuration outperforms 4-thread IMT by 14.3%. This shows the ability of CSMT to remove a significant part of horizontal waste.

When real memory model is considered (Figure 4.11), the performance of single-thread configurations (single-thread and extrapipe) degrade significantly, while IMT and CSMT suffer only a minor impact. This fact shows the ability of both approaches to hide vertical waste. CSMT, however, still outperforms IMT by a significant margin. On average, a 2-thread CSMT configuration (CSMT2+R) performs almost as well as a 4-thread IMT (IMT4 is only 0.3% better), while a 4-thread CSMT configuration outperforms both by a significant margin. For instance, a 4-thread CSMT configuration (CSMT4+R) has an

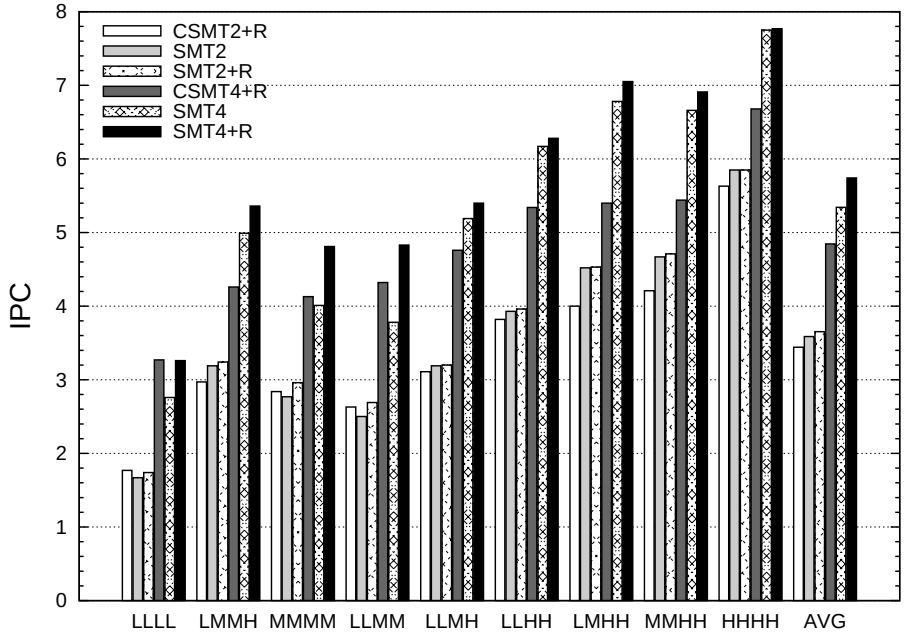


Figure 4.12: **CSMT performance comparison with OpSMT**

average speedup of 110% over the baseline single-thread, 39.6% over a 2-thread CSMT (CSMT2+R) and 40% over a 4-thread IMT configuration (IMT4). In particular cases, speedup with a 4-thread CSMT configuration can be as high as 214% over baseline single-thread (LLLL) and 97.3% over a 4-thread IMT configuration (LLMM). Even for the workload HHHH, a 2-thread CSMT configuration has improvement of 10.2% over a 2-thread IMT configuration, and a 4-thread CSMT configuration has a noticeable improvement of 11.6% over a 4-thread IMT configuration.

4.3.4 CSMT Performance Comparison with OpSMT

Finally, we compare CSMT (with cluster renaming) performance to Operation-level SMT (OpSMT) (both with and without cluster renaming), as shown in Figure 4.12 for both 2-thread and 4-thread configurations for the real memory model. In the figure, label OpSMT2 means a 2-thread OpSMT configuration and OpSMT4 means a 4-thread OpSMT configuration. A +R appended to the labels indicates use of cluster renaming. Since OpSMT merges instructions at operation level, performance obtained by OpSMT (after cluster renaming) is a good upper bound for CSMT performance. First important thing to note is that in some cases (e.g. LLMM), CSMT (with cluster renaming) outperforms OpSMT (without cluster renaming). This is simply because, with cluster renaming, CSMT is able to merge operations from the threads better than OpSMT due to a reduced

number of conflicts, which also highlights the positive impact that cluster renaming has on performance. On average, CSMT performance is within 3% of OpSMT performance for 2-thread and 8% for a 4-thread configuration. Even when cluster renaming is used with OpSMT, CSMT is still within 7% of OpSMT performance for a 2-thread configuration, and within 19% for a 4-thread configuration. Despite the fact that OpSMT performance is higher than that of CSMT, CSMT still has a quite competitive performance with lower hardware complexity. Thus, CSMT acts as a bridge between the two extreme multithreading schemes, IMT and OpSMT, keeping low hardware complexity but maintaining high performance levels.

The results reinforce the fact that there is a cluster usage imbalance in most of the programs, and exploiting this fact using a very cheap technique of cluster renaming, as proposed in this chapter, significantly improves processor performance. Note that renaming has a lower impact on OpSMT performance than CSMT. This happens because CSMT is more sensitive to the cluster usage imbalance than OpSMT. In OpSMT, instructions may be merged even when the instructions use same clusters which is not possible with CSMT. Hence renaming is more critical for CSMT performance than OpSMT. Also note that the effect of renaming is more pronounced with higher number of threads. This happens because, with lower number of threads, there are not many opportunities to merge instructions anyway as most of the parallelism gets used to hide cache misses. When more threads are available, there is a higher contention for the use of clusters (or resources in the cluster). Cluster renaming eases the contention by distributing the heavily used clusters.

4.4 Summary and Conclusions

This Chapter presented Cluster Renaming, a technique that reduces cluster conflicts between threads by remapping compiler allocated clusters to different physical clusters. Compiler assigned clusters are biased towards usage of first clusters. This results in high resource conflicts at first clusters. Cluster renaming distributes the heavily used first clusters to different physical clusters to improve load balance and to reduce resource conflicts. Cluster renaming is simple to implement and significantly improves performance of both cluster-level and operation-level Simultaneous MultiThreading techniques (CSMT and OpSMT).

Chapter 5

Heterogeneous Merging: Using CSMT and OpSMT Merging Hardware Together

Both CSMT and OpSMT require a merging hardware to combine instructions from different threads into a single packet. The cost of this merging hardware is significantly higher for OpSMT in comparison to CSMT. As a result, the number of hardware threads that can be supported using OpSMT is lower than CSMT. Besides the lower hardware overhead than OpSMT, CSMT also achieves performance reasonably close to OpSMT on average. However, in several cases, there is a significant performance difference between OpSMT and CSMT. This chapter presents a heterogeneous merging technique that maintains the hardware budget while achieving high performance by using CSMT and OpSMT simultaneously. The following section presents the motivation for heterogeneous Merging.

5.1 Motivation

Supporting a large number of threads on a OpSMT processor is desirable because of the performance gains that are achieved. To illustrate this, Figure 5.1 shows the average IPC[1] obtained by OpSMT for the workloads evaluated in this thesis (discussed in chapter 2) for a single-thread, 2-thread and a 4-thread processor. OpSMT performance improves significantly with the number of threads and even the 4-Thread OpSMT processor has a performance advantage of 61% over a 2-Thread OpSMT Processor. However, the hardware complexity of OpSMT hardware increases significantly with the number of threads.

[1] All evaluations presented in this chapter make use of cluster renaming

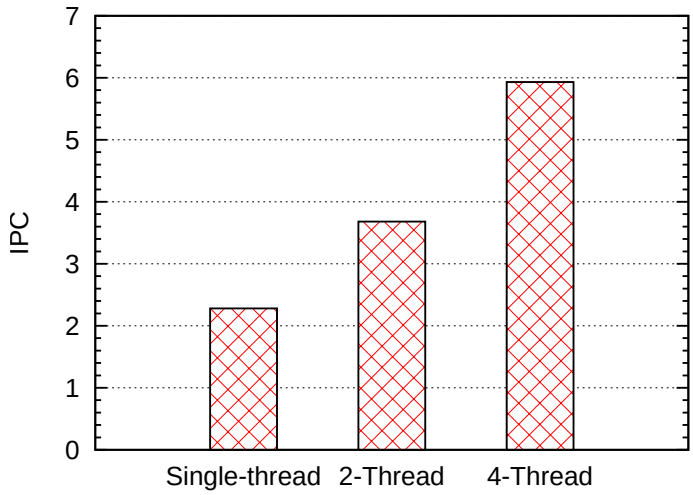


Figure 5.1: **OpSMT Performance**

OpSMT merging hardware requires a conflict detection logic at operation-level and a rerouting logic to route operations to free issue slots. The complexity of these limit the number of threads that can be supported. In fact, previous studies that have done an evaluation of the hardware required for OpSMT on VLIW [41] also limit the number of threads that can be realistically supported to only 2.

Compared to OpSMT, CSMT scales better with number of threads and 4 threads can be easily supported. However, CSMT has a lower performance than OpSMT. Figure 5.2 shows the difference in performance of OpSMT with CSMT for a 4-Thread processor. On an average, OpSMT performance is 27% higher than CSMT. For particular cases like LLHH, the performance difference is as high as 58%. SHOW 2Thread and 4Thread SMT serial logic.

Figures 5.3 and 5.4 show the cost of the thread merge control with varying number of threads for a 4-cluster 4-issue per cluster architecture for both CSMT and OpSMT. Figure 5.3 shows the cost in terms of number of transistors required on a logscale, and Figure 5.4 shows the cost in terms of gate delays. Two implementations of thread merge control for CSMT [20] are considered in this thesis viz. serial and parallel. The serial implementation is a cascading logic checking a different thread at each level. The parallel implementation, on the other hand, checks, in parallel, all the possible thread selections amongst the thread selections and selects one conforming to the selection policy. The parallel implementation has lower delay than the serial one but has a much higher hardware overhead, which grows exponentially with number of threads.

For OpSMT thread merge control, an implementation similar to the serial approach is

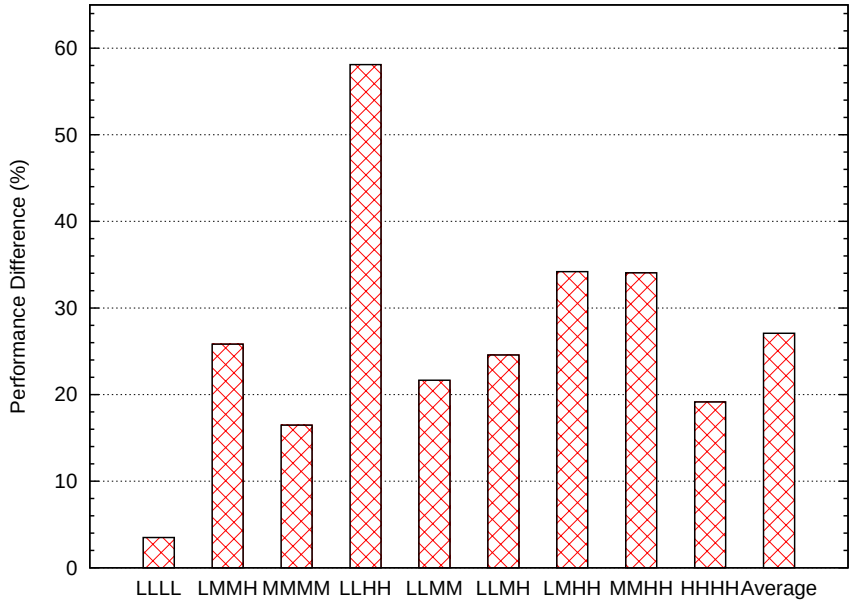


Figure 5.2: **OpSMT performance advantage over CSMT**

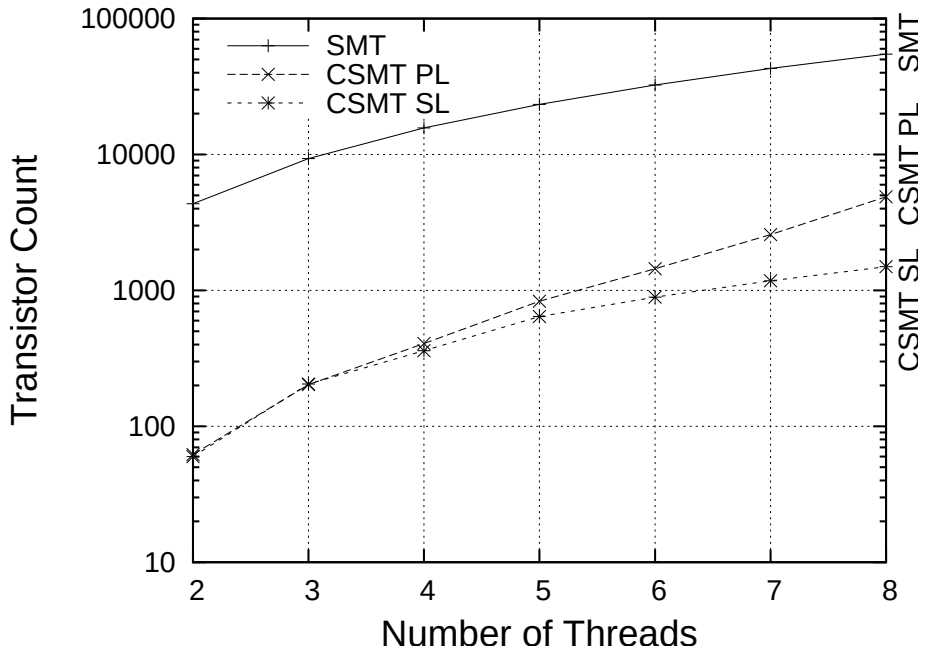


Figure 5.3: **CSMT and OpSMT thread merge control area**

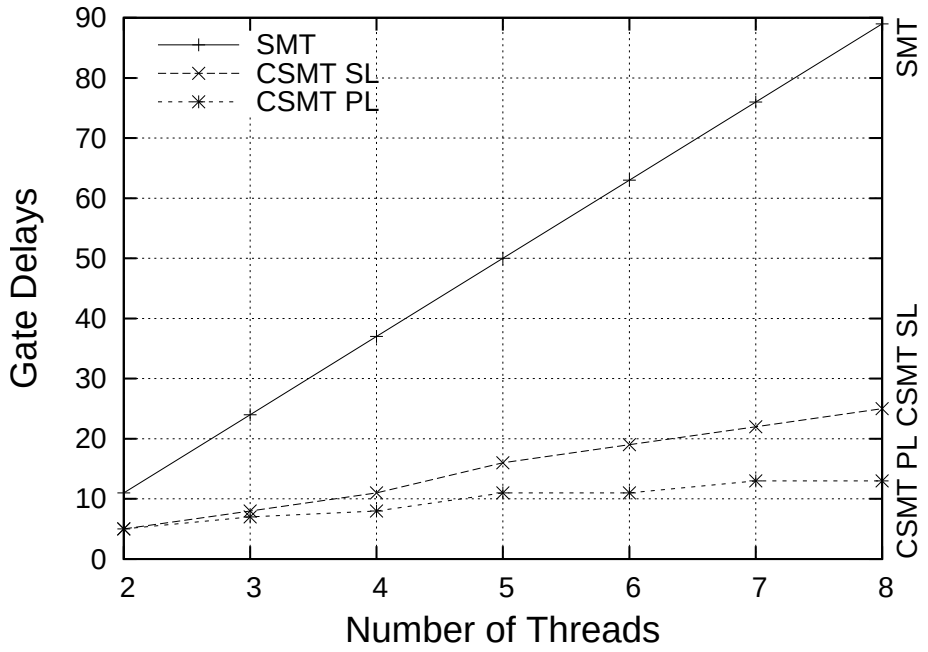
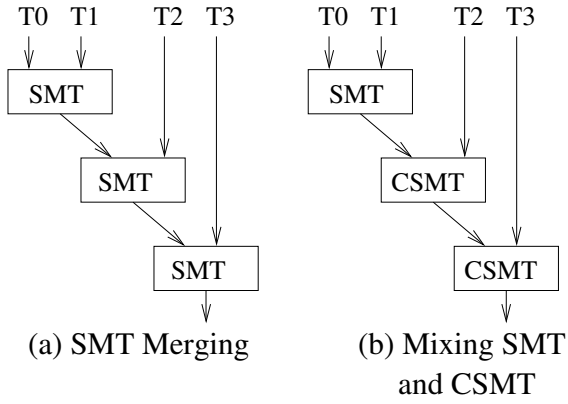


Figure 5.4: **CSMT and OpSMT thread merge control delays**

considered. The parallel approach is not feasible for OpSMT because the cost of checking, in parallel, all possible thread selections is prohibitively expensive in terms of area. In the figures, labels 'CSMT PL' and 'CSMT SL' refer to the parallel and serial implementations of CSMT thread merge control, while label 'OpSMT SL' refers to the OpSMT serial thread merge control. The implementation and the computation details of the gate delays and transistor count for OpSMT thread merge control are available in Appendix B. As shown in the figures, the complexity of the thread merge control for OpSMT increases significantly with the number of threads, both in terms of transistors required and gate delays, and constrain its scalability.

We note that the complexity of CSMT merge control is much smaller compared to OpSMT merge control. To improve performance while keeping the cost of merging hardware low, we intend to use a heterogeneous combination of both OpSMT and CSMT merge control to support more threads. For instance, it is possible to merge the first two threads using OpSMT and the result be merged with another thread using CSMT merging. While with OpSMT, it is expensive to support even an extra thread beyond 2, supporting the extra threads using CSMT has little impact on the total cost of the merge logic.

Figure 5.5 presents an example of the heterogeneous merging. Figure 5.5(a) shows the standard merging scheme used for OpSMT. In the scheme, the first two threads (T0 and T1) are merged, the result is merged with the next Thread T2 and so on, using operation-



(a) SMT Merging (b) Mixing SMT and CSMT

Figure 5.5: **Mixed Merging Example**

level merging hardware in all three steps. CSMT merge control has a much lower cost than OpSMT. This provides an opportunity to support more threads at reasonable cost and achieve higher performance at low cost by combining both CSMT and OpSMT approaches for the merge control. Figure 5.5(b) shows an example of the heterogeneous merging scheme using both OpSMT and CSMT. In the example, the first two threads (T0 and T1) are merged at operation-level (OpSMT) but the result and the next two threads (T2 and T3) are merged at cluster-level (CSMT). Since the cost of CSMT merging is much lower than OpSMT, the addition of CSMT has little impact on the overall cost of the merging hardware. Thus, the cost is comparable to a 2-Thread OpSMT merging even though 4 threads are simultaneously supported and merged. In addition, when more than two threads are merged, OpSMT requires a serial merging implementation, while CSMT can do the merge in parallel. Next section discusses the heterogeneous merging in details.

5.2 Heterogeneous Thread Merging Schemes

This section explores various heterogeneous merging schemes to support more than 2 threads at low cost. We limit our evaluations in this chapter to a 4-Thread architecture.

Figures 5.6((a)-5.6((o) show several possible schemes in which OpSMT and CSMT merge control are combined for a 4-Thread architecture. In all the schemes shown, the first digit refers to the number of levels of cascade, and each following letter indicates whether the merging at each level is CSMT ('C') or OpSMT ('S'). For instance, scheme 3SCC implies that there are 3 levels of cascade, with OpSMT merging at the first level and CSMT merging at the second and the last level.

Note that if more than two threads are merged by CSMT, two different implementa-

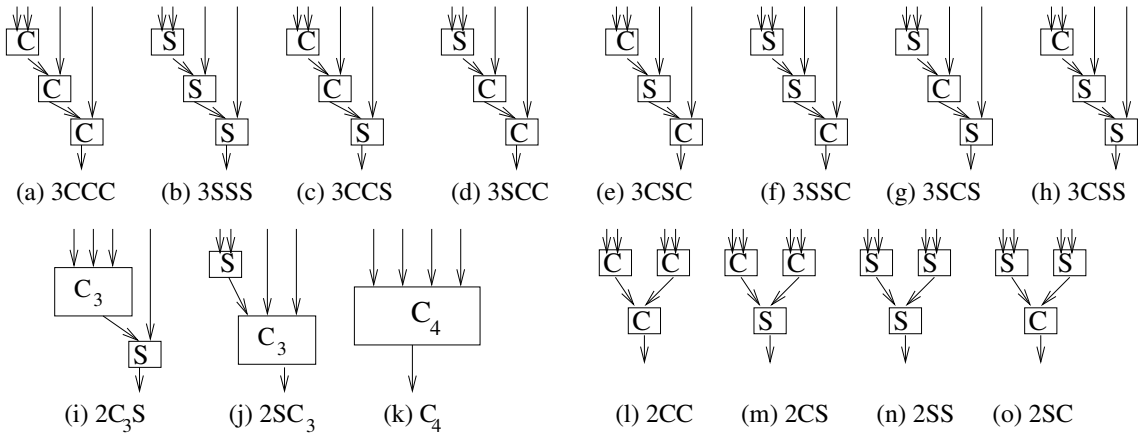


Figure 5.6: **Possible Merging Schemes for 4 Threads**

tions, serial and parallel, are possible. Figures 5.6(a), 5.6(c) and 5.6(d) show the serial implementations. The parallel implementations of these schemes are shown in figures 5.6(i)-5.6(k). Thus, merging 4 threads by using CSMT serial implementation corresponds to Figure 5.6(a) while merging 4 threads using CSMT parallel implementation is shown in Figure 5.6(k). Use of the CSMT parallel implementations is indicated by a subscript showing the number of threads being merged in parallel. For instance, C_4 refers to merging 4 threads using the parallel implementation of CSMT. The parallel implementation is also possible for OpSMT. However, as explained previously, the cost of checking multiple threads in parallel is prohibitively expensive in terms of area. For this reason, parallel implementations for OpSMT for more than 2 threads are not considered.

The merging schemes explored in figures 5.6(a) to 5.6(k) use a cascade to merge the threads i.e. initially, the first two threads are merged, then the result is merged with the next thread and so on. We also explore a different scheme for merging the threads, which is analogous to a balanced tree structure and lowers the delay of the merging hardware when compared to the serial approaches (Figures 5.6(a)-(h)). In these schemes, the 4 threads are divided into groups of 2 threads each. Threads in each group are first merged independently of the other group. The mergings obtained for the two groups are then merged producing the final execution packet. Figure 5.6(l) shows an example of this approach where threads (T0,T1) and (T2,T3) are merged with CSMT and the two resulting merges are merged again using CSMT. This approach results into a lower delay than the serial ones because of the reduced merge levels. These schemes are shown in figures 5.6(l)-5.6(o). Note that the first merging level of these schemes use same merging block (CSMT or OpSMT) to ensure that delay of both divisons is the same.

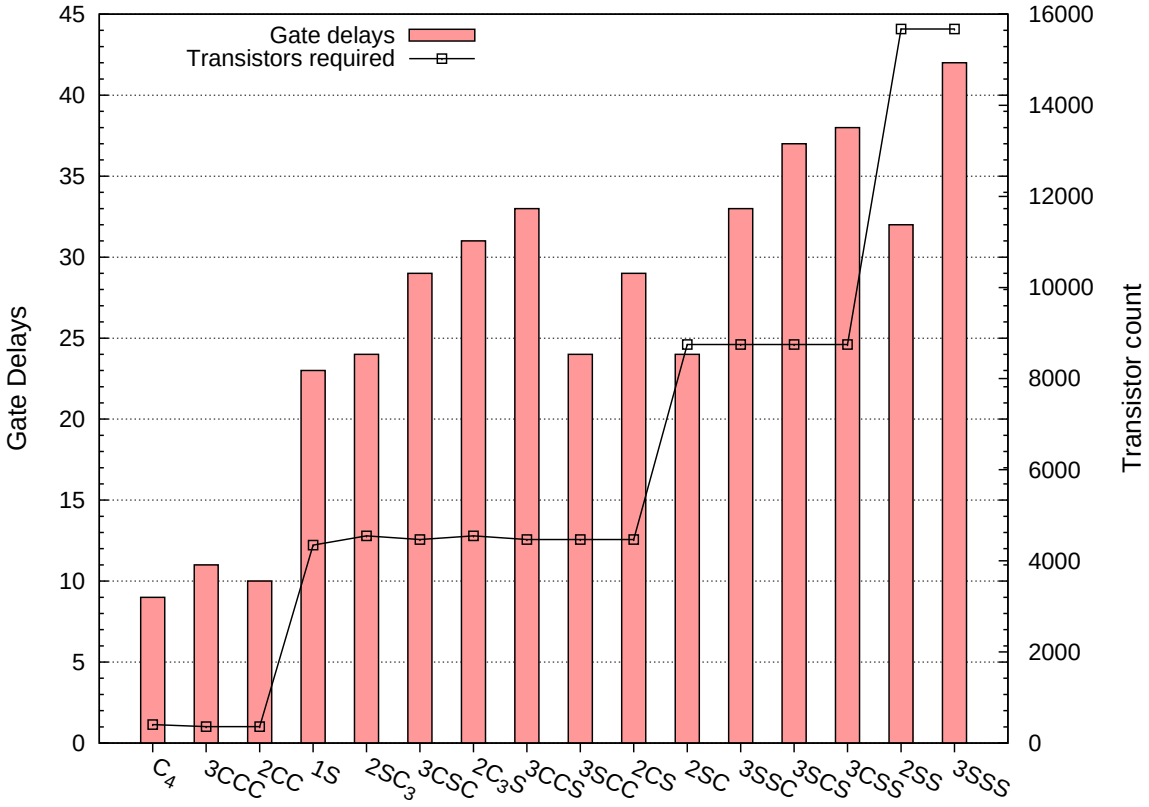


Figure 5.7: **Merging Hardware Cost**

The schemes 2SS and 2CC use either OpSMT or CSMT merging only and can be considered as alternative implementations for a 4-Thread CSMT or OpSMT processor. Note that while these approaches lower the delay of the merging hardware, there might be a negative impact on performance. This is because merging threads T2 and T3 creates an instruction larger than the T2 or T3 individual instructions. It may happen that this larger instruction can not be merged with the merging produced by threads T0 and T1 even though threads T2 or T3 could be individually merged. The following section presents the cost evaluation of all the merging schemes.

5.3 Evaluation

This section presents the cost analysis of the merging hardware for all the schemes considered in Section 5.2 in terms of gate delays and transistor count. Figure 5.7 shows the gate delays and the number of transistors required for each scheme. In the figure, the bars show the gate delays and the line shows the transistors required for each scheme. For the sake of comparison, the figure also includes the cost of the merging hardware of a 2-

Thread OpSMT configuration (1S). In general, the number of transistors required by any scheme is dominated by the number of OpSMT merge control blocks used by the scheme. Schemes that use only CSMT merging (C_4, 2CC and 3CCC) have the lowest area overall. Amongst the schemes that use OpSMT, schemes with only 1 OpSMT merge control block (1S, $2SC_3$, 3CSC, 3CCS, 3SCC) require the least number of transistors, while schemes with 3 OpSMT merge control blocks (2SS and 3SSS) are the most expensive with highest number of transistors required. As expected, there is little difference in the transistor requirements of a 2-Thread OpSMT (1S) and the schemes that use only 1 OpSMT merge control block because the cost of CSMT merge control is an insignificant addition to the total cost.

Schemes that use only CSMT (C_4, 2CC and 3CCC) also have the lowest gate delays, highlighting the advantages of CSMT over OpSMT. Schemes that use OpSMT have higher delays. Note that the gate delays of the schemes $2SC_3$, 3SCC and 2SC are very close to the gate delays for a 2-Thread OpSMT (1S). Schemes $2SC_3$ and 3SCC also have a transistor requirement similar to 1S, making these schemes particularly attractive for implementation. Other schemes have much higher gate delays. Also note that the schemes 3CSC and 3CCS have higher gate delays than 3SCC or $2SC_3$ even though all these schemes use a single OpSMT merge control block. This is because in 3SCC and $2SC_3$, using OpSMT merge control earlier during merging allows the computation of routing of the operations in a VLIW instruction to be done in parallel with CSMT merging. Parallel computation of the routing also results into the lowest delay for scheme 3SSC compared to similar schemes 3SCS and 3CSS.

The performance evaluation of the merging schemes and a cost-vs-performance analysis is discussed in the following section.

5.3.1 Performance Analysis

This section presents the performance results obtained for the merging schemes evaluated in this chapter. Figure 5.8 shows the performance obtained for all the schemes evaluated and also a 2-Thread OpSMT (1S) configuration. Schemes 3CCC and C_4 are two different implementations of a 4-Thread CSMT configuration, while scheme 3SSS is the 4-Thread OpSMT configuration and achieves the peak performance. We found that several groups of schemes (3CCC,C_4), (3SCC,3CSC,3CCS,$2SC_3$,$2C_3S$) and (3CSS,3SCS,3SSC) obtained very similar performance than the other schemes in the same group (less than 1% difference in performance for all workloads). Besides, several schemes like (3SCC,$2SC_3$)

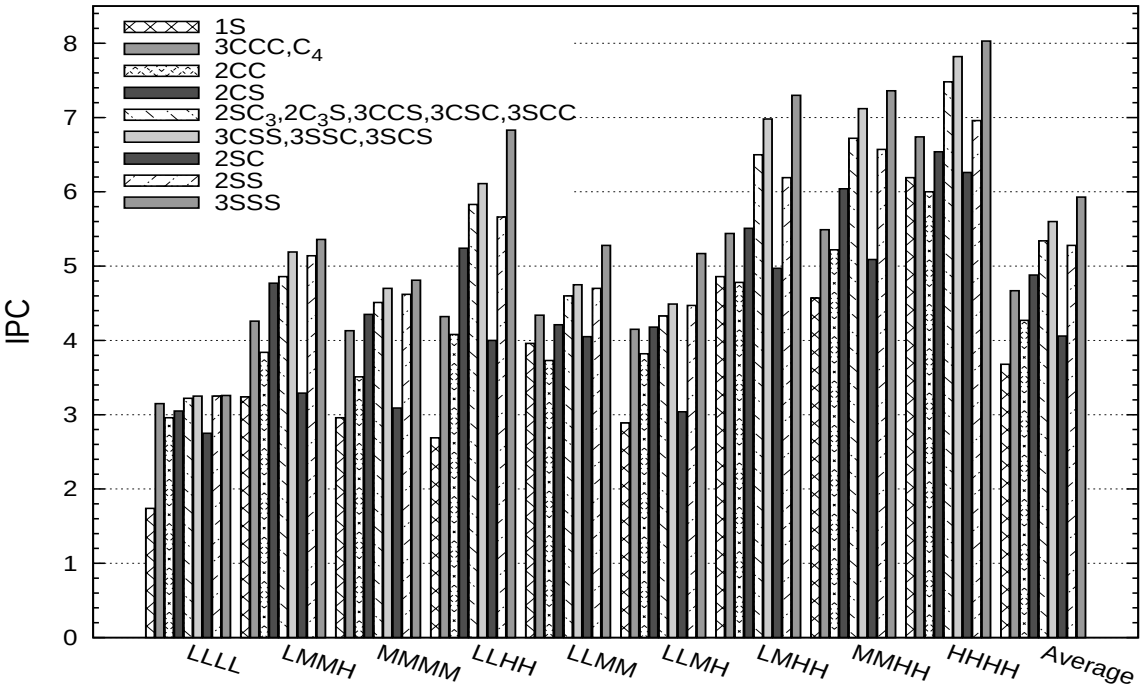


Figure 5.8: **Merging schemes performance**

and (3CCC,C_4) etc. are identical in terms of performance. Hence, these schemes have been grouped together in the figures.

Amongst all the schemes (except 1S and 3SSS, that represent the minimum and the maximum performance), the group of schemes (3CSS,3SCS,3SSC) performs the best with a performance within 5.6% of the peak 3SSS performance. On the other hand, the scheme 2SC performs the worst and achieves a performance even lower than 3CCC (and only marginally better than 1S) despite having a much higher cost. This is because using OpSMT first to merge threads first result into 2 instructions that are beefier than the original instructions of the individual threads. As a reult, CSMT may not be able merge these beefier instructions even though the individual instructions of the threads could have been merged. This results into a significant restriction on merging.

Scheme 2CC also has a lower performance than 3CCC albeit a little better than 2SC. Next, we discuss the the schemes where only 1 OpSMT merge control block is used in the merging hardware, i.e. the schemes 3SCC, 3CCS, 2CS, $2C_3S$, $2SC_3$ and 3CSC. In general, scheme 2CS performs the worst among all the schemes with a single OpSMT merge control block and achieves a performance only marginally higher than 3CCC. Other schemes with a single OpSMT merge control block, (3SCC,3CSC,3CCS,$2SC_3$,$2C_3S$), have a performance lower than the schemes (3CSS,3SCS,3SSC). However, they still

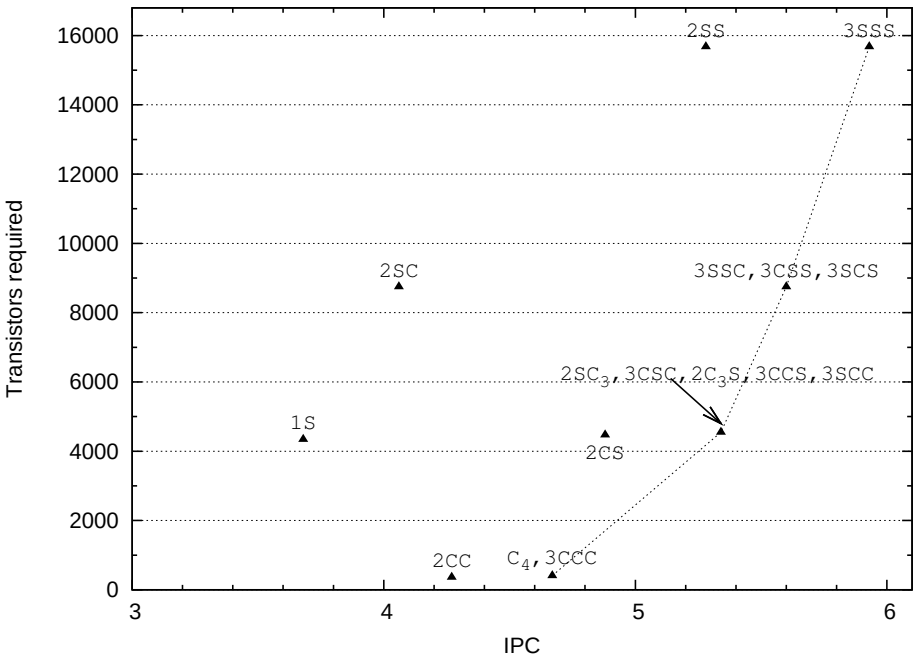


Figure 5.9: **Performance vs transistors incurred**

significantly outperform a 4-Thread CSMT (14%) and a 2-Thread OpSMT processor (45%). Also, the performance is only 11% lower than the 4-Thread OpSMT processor (3SSS) on an average. Scheme 2SS also has a performance comparable to schemes (3SCC,3CSC,3CCS,$2SC_3$,$2C_3S$).

We now present an analysis of the performance of the merging schemes while taking their cost into account as well. Figures 5.9 and 5.10 show the performance of the merging schemes in combination with the required transistors and the gate delays respectively. In the Figure 5.9, schemes 3SCC,3CSC,3CCS,$2SC_3$,$2C_3S$ and schemes 3CSS,3SCS,3SSC have been grouped together as they have similar results for performance and transistors incurred.

As shown in figures 5.9 and 5.10, schemes that use only CSMT merging (3CCC, C_4 and 2CC) are the cheapest in terms of both delay and transistors incurred. These schemes are the only choice if the design is quite constrained and even the cost of a 2-Thread OpSMT cannot be supported. However, scheme 2CC is not an attractive choice because of its lower performance compared to schemes 3CCC and C_4, that have higher performance but similar cost. If the cost of a 2-Thread OpSMT can be afforded, then schemes $2SC_3$ and 3SCC are attractive as they have a higher performance than a 2-Thread OpSMT but at a similar cost both in terms of area and gate delays. These schemes even outperform the more complex schemes 2SS, 2SC and 2CS.

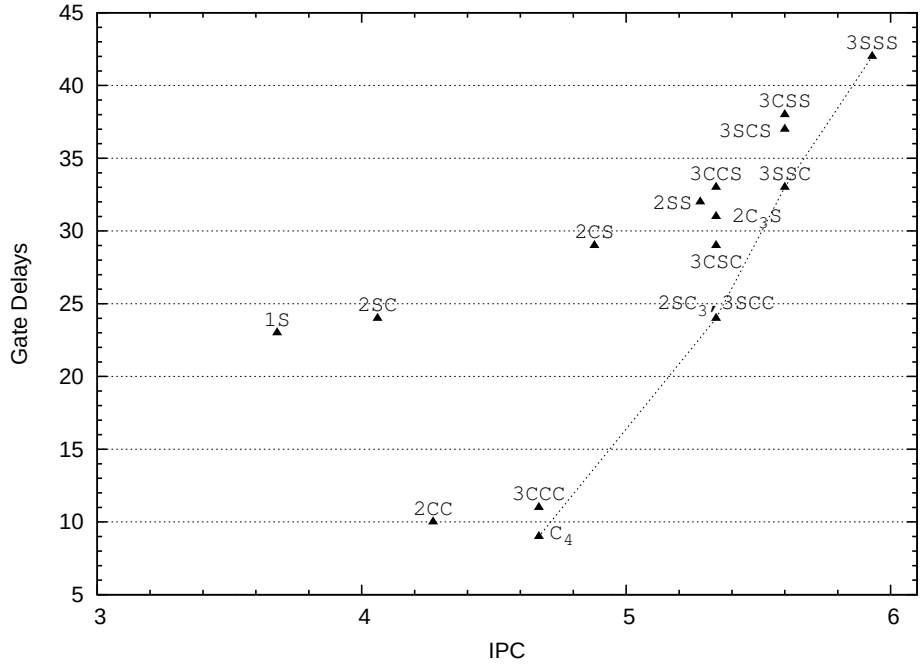


Figure 5.10: **Performance vs gate delays**

Only the schemes 3SSC, $3SC_3$ and 3SCS have a higher performance than $2SC_3$. Among these schemes, 3SSC is the best choice because of its lower delay compared to others with similar performance and transistor requirement. However, the extra performance between 3SSC and $3SC_3$ comes at the cost of supporting 2 OpSMT merge control blocks. Besides, the performance difference is not significant enough to justify the additional cost.

In conclusion, for all the schemes evaluated, scheme $2SC_3$ presents a good trade off between performance and cost because of its low cost (comparable to a 2-Thread OpSMT) and a performance close to a 4-Thread OpSMT.

5.4 Summary and Conclusions

Operation-level Simultaneous MultiThreading (OpSMT) is a popular approach for improving processor performance. However, OpSMT is expensive to scale beyond 2 threads because of the increased cost of the merging hardware. Other schemes like Cluster-level Simultaneous MultiThreading (CSMT) has a lower merging hardware cost and can support higher number of threads. However, CSMT performance is lower than OpSMT.

This chapter explored several merging hardware designs which use a combination of both CSMT and OpSMT merging. The use of CSMT in the merging hardware allows

supporting more threads without significantly affecting the cost of the merging hardware. The experimental results prove that the performance of the combined merging is quite reasonable and achieves a performance in-between CSMT and OpSMT. One of the schemes, $2SC_3$ in particular, which uses OpSMT to merge first 2 threads and then merges the remaining threads using CSMT, is quite attractive. $2SC_3$ merges 4 threads, has a cost close to the 2-Thread SMT merging but has a much higher performance. On an average, $2SC_3$ achieves 14% higher performance than a 4-Thread CSMT processor, 45% higher performance than a 2-Thread OpSMT processor and is within 11% of a 4-Thread OpSMT processor performance.

Chapter 6
Cluster-level Split-Issue

Multithreading techniques like Cluster-level Simultaneous MultiThreading (CSMT) and Operation-level Simultaneous MultiThreading (OpSMT) issue VLIW instructions from multiple threads simultaneously for improving processor performance. If two VLIW instructions have a resource conflict, only one of them can be issued. This is because all operations in a VLIW instruction must be issued in a lock step mode i.e. all operations must be issued at the same cycle to preserve the semantics of the program. This limits the number of instructions that can be issued at a given cycle and restricts the performance improvement potential of both OpSMT and CSMT.

Split-issue at operation-level [45, 26] is a technique that removes the lock step issue restriction and allows issuing operations of same VLIW instruction separately at different cycles. However, the implementation requires complex dynamic scheduling hardware. The hardware requirements of split-issue at operation-level makes it impractical for embedded VLIW processors. In this chapter, we propose a new split-issue approach named cluster-level split-issue. Cluster-level split-issue allows splitting a VLIW instruction only at a cluster boundary and does not allow splitting individual operations in a cluster. Cluster-level split-issue is simple to implement and has a low hardware cost. Besides, the performance achieved by cluster-level split-issue is close to the more complex operation-level split-issue.

6.1 Operation-level Split-Issue

Simultaneous MultiThreading (SMT) increases the throughput of the processor by issuing instructions from multiple threads simultaneously. To issue multiple instructions, no resource conflicts should exist among the instructions. Each instruction in a VLIW processor comprises of several operations. Therefore, while merging two instructions, resource

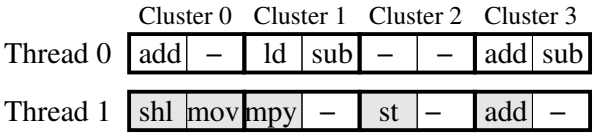


Figure 6.1: **Instruction Merging in OpSMT and CSMT**

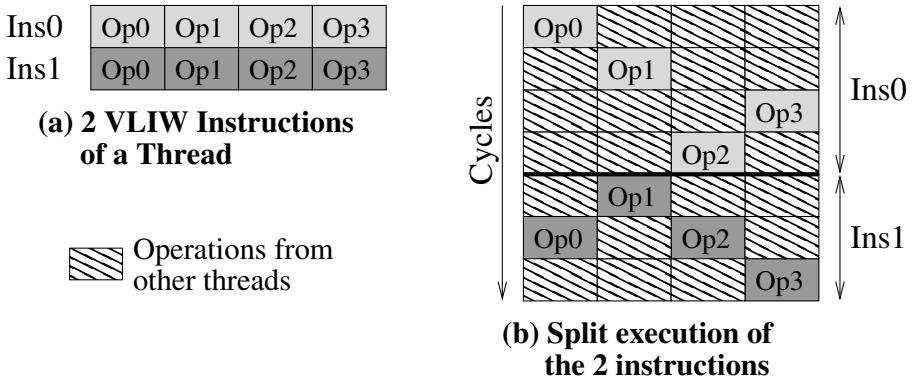


Figure 6.2: **Instruction Execution with Split-Issue**

conflicts among instructions are quite frequent (e.g. because of insufficient issue-slots to issue both instructions completely or because both instructions require use of a particular resource which is not available in sufficient number). Since the full instruction is the issue unit, two VLIW instructions may not be issued simultaneously even if one of them does not utilize all the issue slots. For instance, for the two VLIW instructions shown in Figure 6.1, both instructions have resource collisions at clusters 0, 1 and 3. As a result, only one of them can be issued at a time. If the first instruction is issued, two issue slots are wasted as selecting only two operations from the other instruction is not possible. Hence, the limitation to issue the VLIW instruction as a unit restricts the opportunities to reduce the horizontal waste in the processor.

Split-issue [45, 26] allows issuing of the operations of a VLIW instruction in parts instead of as a complete unit. Split-issue was originally proposed as a mechanism to maintain binary compatibility in VLIW processors [45]. However, it can also be used to improve processor throughput in SMT [26] as parts of an instruction of one thread can be executed along with an instruction of another thread. Figure 6.2 shows an example of split execution for 2 instructions, Ins0 and Ins1, of a thread on a multithreaded 4-issue VLIW processor. Both instructions have 4 operations each, Op0-Op3, as shown in Figure 6.2(a). Without split-issue, all operations of an instruction would have to be issued at the same cycle. Using split-issue, however, removes this constrain and the operations can be

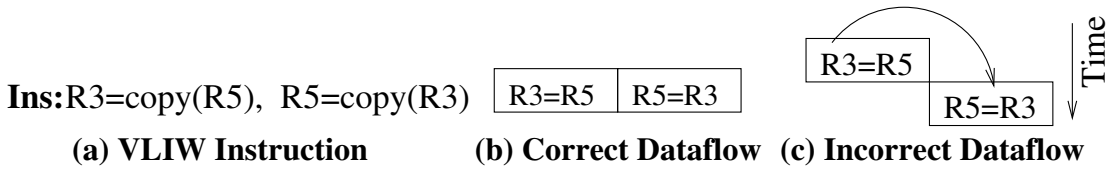


Figure 6.3: **Issues with Dataflow**

flexibly issued in any order while maintaining program semantics. Figure 6.2(b) shows a sample split execution of the operations of the 2 instructions. Operations of the first instruction, Ins0, are issued in 4 separate cycles and operations of the second instruction, Ins1, are issued in 3 cycles. The remaining issue-slots are filled by operations from other threads. Note that operations from instruction Ins1 are not issued until all operations of the instruction Ins0 have been issued, i.e. the in-order[1] relationship between the VLIW instructions is still maintained. The implementation complexity of split-issue is because of the dataflow hazards that arise because of a potential violation of complier assumptions. The following section discusses the dataflow hazards in split-issue and working of split-issue in greater details.

6.1.1 Dataflow hazards

In general, the ISA (Instruction Set Architecture) and architecture are tightly coupled in VLIW processors and the actual latencies of the operations are exposed to the compiler or the programmer. The dependency checking and latency-cognizant instruction scheduling is done entirely at compile time. Instructions are scheduled with the assumption that all operations belonging to an instruction are issued simultaneously. No runtime dependency checking or interlocking hardware is required. The compiler's data dependency assumptions can be violated if the operations of a VLIW instruction are not issued in a lock step. To illustrate this fact, Figure 6.3(a) shows a VLIW instruction consisting of 2 operations. The instruction does a single cycle swap of the registers R3 and R5 (of the same cluster) without using extra registers and it is a legal VLIW instruction. The two operations read initial values of the source registers if the operations are not split, as shown in Figure 6.3(b). Now, let us assume that the 2nd operation is issued at a later cycle. The delayed

[1]In theory, operations from instruction Ins1 may be executed with the operations of Ins0 as long as the data dependencies are taken care of. However, this changes the VLIW processor close to a full fledged out-of-order processor, with a VLIW ISA requiring all the complex and power hungry hardware. This complexity is not desirable for VLIW processors, whose prime attraction is their low complexity & low power.

issue results in an incorrect dataflow as represented in Figure 6.3(c), since the 2nd operation will read an incorrect value of register R3. Hence, measures are required to avoid breaking the compiler dataflow assumptions when instructions are split-issued.

Following section discusses the implementation of split-issue in a VLIW processor while maintaining the correct dataflow.

6.1.2 Implementation details

For split-issuing the VLIW instructions, the operations of the VLIW instructions are first divided into 2 phases at runtime. Phase I performs the operation and writes the result to a delay buffer. Phase II copies the value from the delay buffer to the original destination register. After the division of the operations of a given VLIW instruction into the two phases, an amalgamated instruction corresponding to the given VLIW instruction is formed. For a given VLIW instruction, phase I of all the operations are part of the corresponding amalgamated instruction. Unlike phase I of an operation which always belong to the corresponding amalgamated instruction, phase II of an operation can belong to the corresponding amalgamated instruction itself or a later one, depending upon the FU latency. For a FU with a latency of N cycles, the phase II for that operation is part of the amalgamated instruction corresponding to the (N-1)th VLIW instruction later. Thus, an amalgamated instruction for a given VLIW instruction contains the phase I of all the operations of the given VLIW instruction, phase II of the operations of the corresponding VLIW instruction with unit latency, and phase II of the operations belonging to earlier VLIW instructions because of non-unit latencies. Once an amalgamated instruction is obtained, phase I of the operations belonging to the amalgamated instruction can be issued dynamically. All phase II in the amalgamated instruction are issued simultaneously with the last phase I operation of the instruction. Issue of phase II is integrated with the delay buffers implementation and does not consume issue-slots [26]. Note that phase I of the operations from the next amalgamated instruction are issued only when all phase I of the operations of the current amalgamated instruction have been issued. This enforces that the execution is still in-order wrt the VLIW instructions.

Figure 6.4 shows an example illustrating how split-issue works. The example shows two VLIW instructions of a thread, Ins0 and Ins1, having two operations each, as shown in Figure 6.4(a). In the example, we assume that `add` and `sub` operations have a latency of 1 cycle and, `mpy` and `ld` operations have a latency of 2 cycles. First, each operation of both instructions is divided into two phases. Figure 6.4(b) shows the breakup of the

Ins0 R5=add(R1,R2), R4=mpy(R1,R5)
Ins1 R3=sub(R5,R1), R6=ld(R2,R7)

(a) Two VLIW instructions

Operations	Phase I	Phase II
R5=add(R1,R2)	B1=add(R1,R2)	R5=B1
R4=mpy(R1,R5)	B2=mpy(R1,R5)	R4=B2
R3=sub(R5,R1)	B3=sub(R5,R1)	R3=B3
R6=ld(R2,R7)	B4=ld(R2,R7)	R6=B4

(b) Breakup of operations into Phases I and II

VLIW Ins.	Corresponding Amalgamated Instruction	
	Phase I	Phase II
Ins0	B1=add(R1,R2), B2=mpy(R1,R5)	R5=B1
Ins1	B3=sub(R5,R1), B4=ld(R2,R7)	R4=B2, R3=B3
Ins2	...	R6=B4

(c) Formation of amalgamated instructions

Cycle	Ins.	Phase I Operations	Phase II Operations
1	**Ins0**	B1=add(R1,R2)	–
2	**Ins0**	B2=mpy(R1,R5)	R5=B1
3	**Ins1**	B3=sub(R5,R1)	–
4	**Ins1**	B4=ld(R2,R7)	R4=B2, R3=B3
5	**Ins2**	...	R6=B4

(d) Cyclewise execution of each operation

Figure 6.4: **Working of Split-Issue**

operations of the instructions into their corresponding phases I and II. After the division of the operations into the two phases, amalgamated instructions are formed for Ins0 and Ins1. Figure 6.4(c) shows the corresponding amalgamated instructions obtained for both Ins0 and Ins1. The phase II of the `add` operation is in the amalgamated instruction corresponding to Ins0 itself because of its single-cycle latency. However, phase II of the `mpy` operation belongs to the amalgamated instruction corresponding to Ins1 because of the 2-cycle latency. Thus, the amalgamated instruction corresponding to Ins0 contains the phase I of both `add` and `mpy` operations but only phase II of the `add` operation. The corresponding amalgamated instruction for Ins1 contains the phase I of both `sub` and `ld` operations, the phase II of the `sub` operation, and the phase II of the `mpy` operation of Ins0. After the amalgamated instructions are formed, the phase I of the operations in the amalgamated instruction can be dynamically issued. Dynamic issuing allows a flexible issuing of operations and do not require all operations to be issued simultaneously. Figure 6.4(d) shows the issuing of operations when exactly one operation of an instruction is issued at a given cycle instead of issuing all operations simultaneously.

Split-issue allows a flexible and dynamic issuing of operations of a VLIW instruction creating more opportunities for removing horizontal waste. However, this flexibility comes at the cost of the extra hardware required to honor execution semantics. As the operations are dynamically issued, the issue logic has a complexity similar to the issue logic in superscalars. For instance, an issue queue logic of 32 entries is required for supporting split-issue on a 4-thread 8-issue VLIW processor. Also, a logic similar to register renaming is required for assigning the delay buffers. Both issue queue and register renaming are amongst the most complex and power hungry structures on superscalar processors [43].

Split−type	Merging policy	
	operation−level (OpSMT)	cluster−level (CSMT)
operation−level	OOSI	−
cluster−level	COSI	CCSI

Figure 6.5: **Applicability of split-issue**

Addition of these structures takes away the key advantages of low power and low cost of VLIWs making them impractical[2] for use in embedded domain.

We propose a restricted variant of split-issue for clustered VLIW processors which we refer to as cluster-level split-issue. Cluster-level split-issue has a low hardware implementation cost. In comparison to earlier proposals, where the operations of an instruction can be issued in any order, cluster-level split-issue allows instructions to be split only at a cluster-level boundary, i.e. splitting of operations belonging to the same bundle is not allowed. For the sake of clarity and to avoid confusion, we refer to the split-issue proposal detailed in Section 6.1.1 as operation-level split-issue from now onwards. The following section discusses cluster-level split-issue in detail.

6.2 Cluster-level Split-Issue

In a clustered VLIW processor, no dependencies exist between different bundles, as they read from and write to different register files. Hence, the bundles of an instruction can be independently issued without breaking execution semantics. Cluster-level split-issue is more restrictive than operation-level split-issue and does not allow the operations in a given bundle to be issued separately. As a result of this restriction, dynamic scheduling requirement of operations is avoided. Further, operations do not have to be split into different phases, which is a primary requirement for operation-level split-issue. Cluster-level split-issue still does a dynamic issuing of the bundles. The dynamic issuing of bundles is achieved with only small changes in the merging hardware and does not increase the hardware complexity (explained in detail in Section 6.3).

CSMT merges instructions at a cluster-level granularity, while OpSMT merges the instructions at an operation-level granularity. Enhancing instruction merging with split-

[2]Even the original split-issue proposal [45] is also not in favor of split-issue because of complex hardware requirements for split-issue

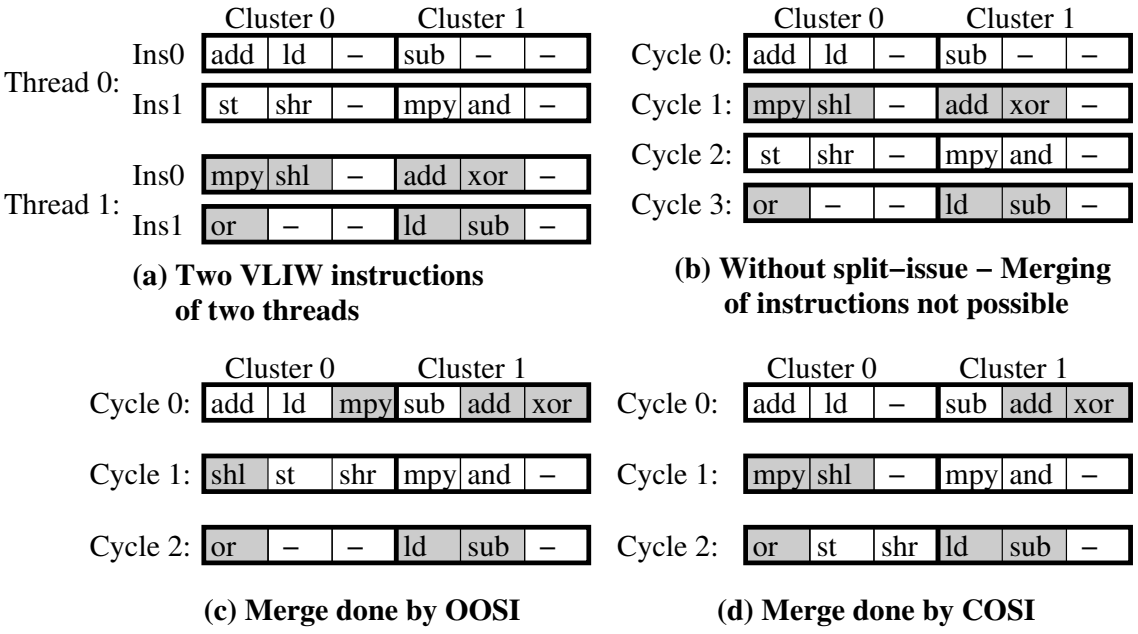


Figure 6.6: **Operation-level and cluster-level split-issue with operation-level merging**

issue techniques leads to the following configurations, as shown in Figure 6.5.

1 **OOSI:** Operation-level split and operation-level merging (The split-issue technique described in Section 6.1.2).

2 **COSI:** Cluster-level split but operation-level merging.

3 **CCSI:** Cluster-level split and cluster-level merging.

Cluster-level split-issue can be used with both operation-level and cluster-level merging. However, operation-level split-issue makes sense only with operation-level merging.

To illustrate the difference between operation-level and cluster-level split-issue when operation-level instruction merging is used (OOSI and COSI), Figure 6.6 shows an example with the merging done by both techniques in a 2-cluster architecture. We assume that number of issue slots is the only critical resource in this example. Also, it is assumed that priorities for merging change each cycle between the threads in a round-robin way. Thus Thread 0 has the higher priority in the first cycle, but in the second cycle, Thread 1 has the higher priority, and so on. Figure 6.6(a) shows 2 instructions each for 2 threads on a 2-cluster 6-issue architecture (3-issue per cluster). Instruction Ins0 from Thread 0 uses two issue slots in cluster 0 and one in cluster 1. While, instruction Ins0 from Thread 1 uses two issue slots in both cluster 0 and cluster 1. Without split-issue, both instructions

cannot be merged because of insufficient number of issue slots in cluster 0. Hence, only instruction Ins0 of Thread 0 is issued at the first cycle. In the second cycle, Thread 1 has the higher priority. Again instruction Ins0 of Thread 1 cannot be merged with instruction Ins1 of Thread 0. This forces Ins0 of Thread 1 to be issued alone and so on. 4 cycles are required to execute the 4 VLIW instructions without split-issue as shown in Figure 6.6(b), since merging of the instructions of the two threads is not possible at any cycle.

Using split-issue, however, reduces the number of execution cycles from 4 to 3. Figure 6.6(c) shows the execution of the instructions when operation-level split-issue (OOSI) is used. At cycle 0, `mpy` operation from cluster 0 of Thread 1 can be issued with the operations of cluster 0 of Thread 0. Similarly, at cluster 1, operations `add` and `xor` of instruction Ins0 of Thread 1 are issued at cluster 1 along with `sub` operation of instruction Ins0 of Thread 1. At cycle 1, operations `st` and `shr` of instruction Ins1 of Thread 0 can be issued with the remaining operations of instruction Ins0 of Thread 1. The remaining operations of Thread 1 (i.e. Ins1) are issued at the third cycle.

The execution of the instructions of two threads when cluster-level split-issue (COSI) is used also takes 3 cycles as shown in Figure 6.6(d). With COSI, `mpy` and `shl` operations of instruction Ins0 of Thread 1 cannot be issued at different cycles. Hence, no operations are selected from cluster 0 of Thread 1 at cycle 0. However, operations `add` and `xor` belonging to cluster 1 of Thread 1 are issued with `sub` operation of cluster 1 of Thread 0 because no splitting of operations inside the bundle is required. At cycle 1, Thread 1 has the higher priority. Thus, the remaining operations of instruction Ins0 of Thread 1 (`mpy` and `shl` of cluster 0) are issued. Operations of cluster 1 from instruction Ins1 of Thread 0 are also issued at cycle 1. At cycle 2, pending operations of instruction Ins1 of Thread 0 are issued and are merged with instruction Ins1 of Thread 1.

Note that even though execution using either OOSI or COSI takes the same number of cycles in the example, OOSI is more efficient than COSI. For instance, COSI issues operations from both Thread 0 and Thread 1 at cycle 2. OOSI, however, issue operations only from Thread 1. The operations of Thread 1 can possibly be merged with the next instruction (Ins2) of Thread 0 further improving OOSI performance.

Next we present an example of cluster-level split-issue when instructions are merged at cluster-level. Figure 6.7(a) shows 2 VLIW instructions each for 2 threads. Instruction Ins0 of Thread 0 uses only cluster 0 but instruction Ins0 belonging to Thread 1 uses both clusters. Hence, the two instructions cannot be simultaneously issued as both of them use cluster 0. The same priority rotation policy used in the previous example is assumed.

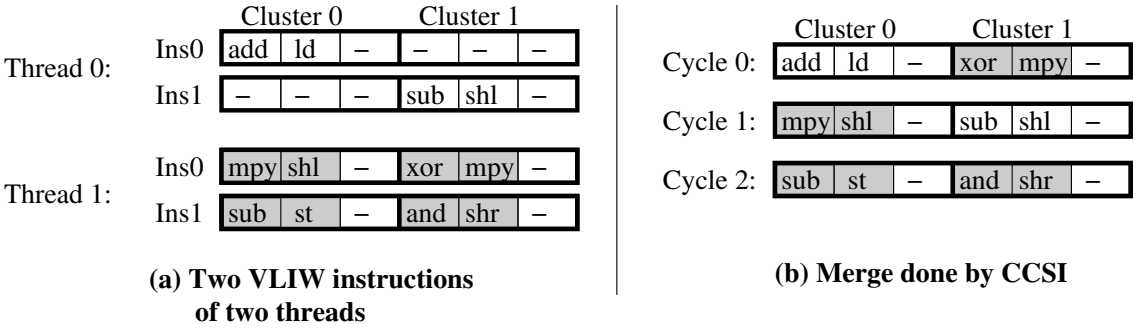


Figure 6.7: **Cluster-level split-issue with cluster-level merging**

Without split-issue, normal execution would require 4 cycles as no merging is possible at any cycle. Using CCSI, however, reduces the execution time to 3 cycles by creating more opportunities for merging. Figure 6.7(b) shows the execution of the two threads when CCSI is used. At cycle 0, operations belonging to cluster 1 of Thread 1 can be issued with Thread 0, as only cluster 0 is used by instruction Ins0 of Thread 0. In the second cycle, operations from cluster 0 of instruction Ins0 of Thread 1 are issued. Operations from cluster 1 of instruction Ins1 of Thread 0 are also issued as cluster 1 is no longer used by Thread 1. Finally, at cycle 2, instruction Ins1 of Thread 1 is issued.

As illustrated in the examples, shown in figures 6.6 and 6.7, using split-issue further improve processor performance in a multithreaded environment. The extra performance is achieved because more opportunities for merging instructions are created. The following section discusses the implmentation of cluster-level split-issue.

6.3 Implementing Cluster-level Split-Issue

This section describes the implementation of the original OpSMT/CSMT merging hardware and the changes required in it to support cluster-level split-issue. First, Figure 6.8(a) shows the original merging hardware for a 4-cluster 3-thread architecture. In the figure, T0-T2 represent the 3 threads, C*i* represents the operations of the bundle assigned to cluster *i* for a given thread, CL is the collision detection logic and ML is the merge logic. CL checks if there is a resource conflict between the two bundles given as input. ML merges the bundles of two instructions corresponding to the same cluster. The output of ML is controlled by a signal which dictates whether the output is a merged bundle of the inputs or the first input is passed as the output (i.e. no merging is possible). For merging two instructions, the merging hardware has to check for resource conflicts at all the clusters.

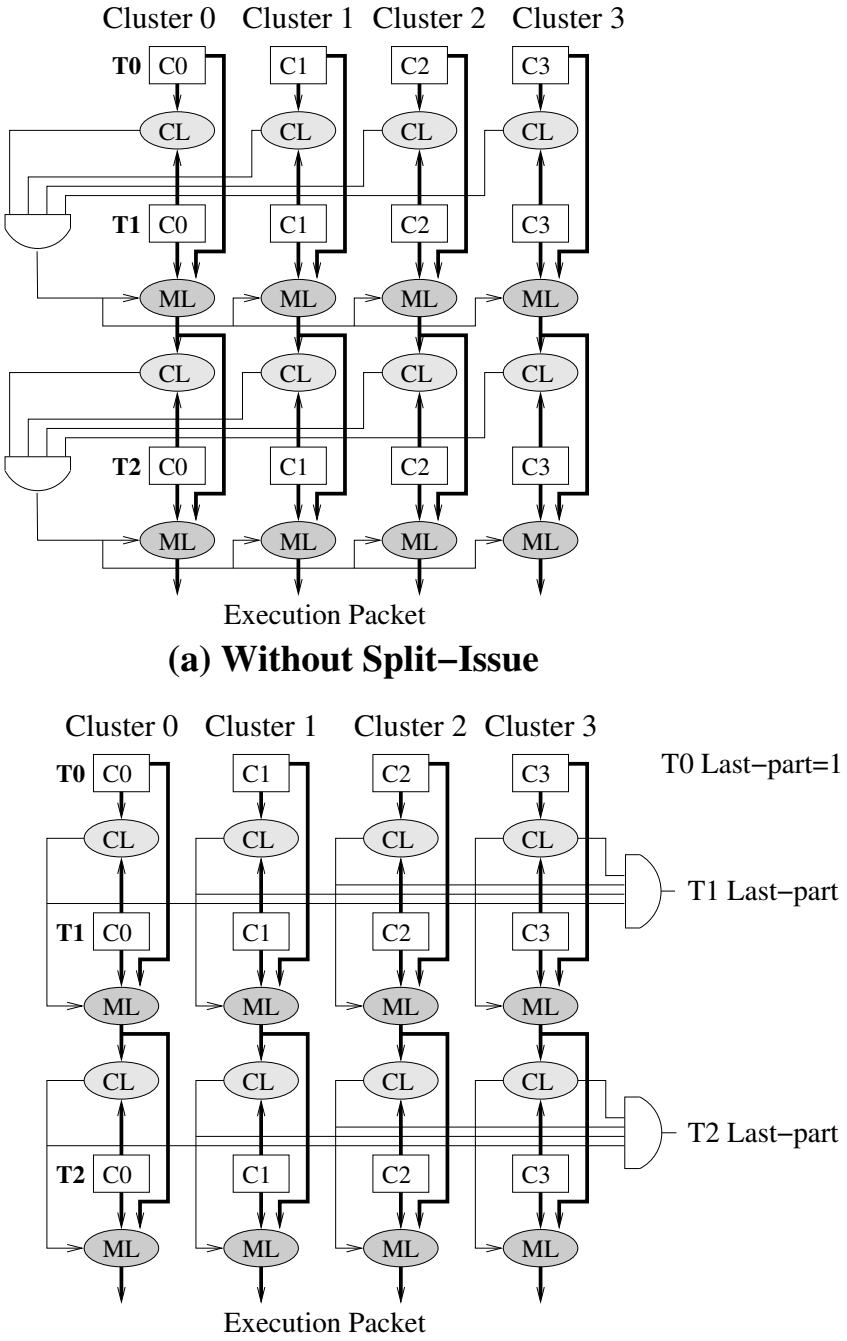


Figure 6.8: **Merging Hardware**

Only when there are no resource conflicts at all the clusters (enforced by the AND gates), two instructions are merged. The internal implementation of CL and ML varies depending on the approach used for merging (operation-level or cluster-level). We assume that Thread T0 has the highest priority for merging, Thread T1 has the next level of priority and Thread T2 has the lowest priority (computation of priority is independent of the merging hardware implementation). Thus, first threads T0 and T1 are tried for merging. Then, the output of this merge (T0 merged with T1 if merging was possible or only T0 if T0 and T1 could not be merged) is tried to be merged with Thread T2. The output of the final merge forms the execution packet.

The merging hardware required for supporting cluster-level split-issue requires only minor modifications in the original OpSMT/CSMT merging hardware as shown in Figure 6.8(b). Without cluster-level split-issue, two instructions can be merged only when there are no conflicts at any of the clusters. On the other hand, when cluster-level split-issue is supported, the resource conflict status of other clusters is not required to merge the bundles corresponding to a given cluster. Hence, the merging process is completely independent for each cluster and, in fact, may result in a lower delay for the merging hardware.

The merging hardware with cluster-level split-issue support also generates a last-part signal for each thread. Last-part signal indicates whether the instruction of a given thread has been merged in its entirety or not. The generation of this signal is not on the critical path of the merging hardware and does not affect the delay of the merging hardware. This signal is labeled as T*i* Last-part for Thread T*i* as shown in Figure 6.8(b). Last-part signal is required at a later pipeline stage (explained in the following section). Note that Thread T0 is always selected in its entirety because it is the highest priority thread. Hence, T0 Last-part signal is always set to 1.

Even though cluster-level split-issue does not need to do dynamic scheduling of operations, which avoids most dataflow issues, Split-issuing still has several runtime issues. These issues are discussed in detail in the following section.

6.4 Issues with cluster-level split-issue

This section discusses several implementation issues that arise when implementing cluster-level split-issue on VEX clustered architecture and solutions for those issues.

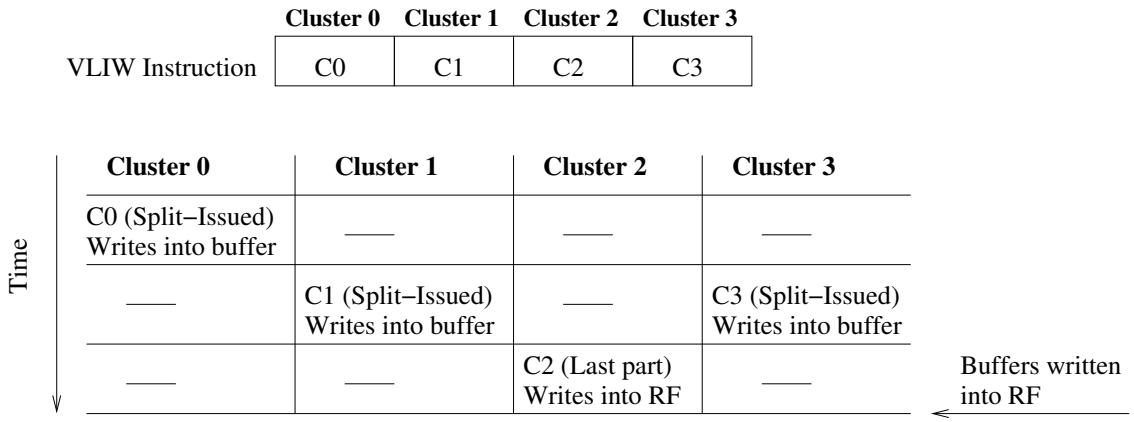
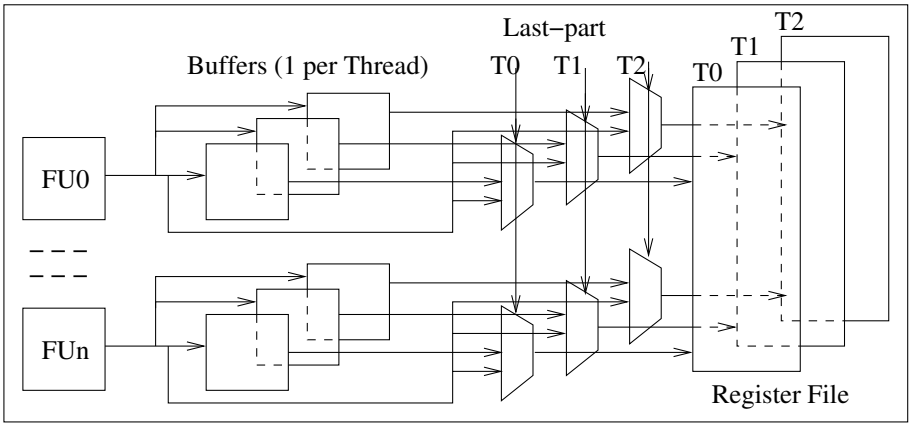


Figure 6.9: **Delaying updates to architectural state by using buffers**

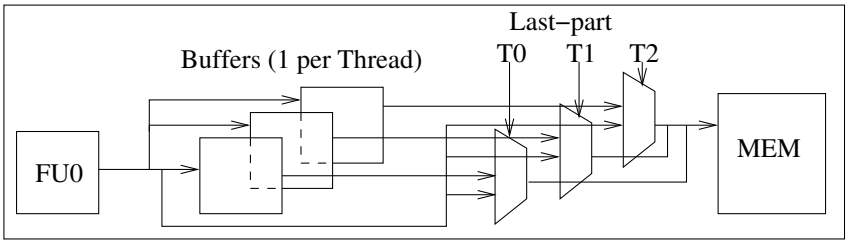
6.4.1 Exceptions/Interrupts

Using split-issue may result into an inconsistent architectural state which has a direct impact on supporting precise exceptions/interrupts. Restoration to a consistent state is mandatory before an exception/interrupt is taken. However, if split-issue is used, restoration to a consistent state may not be possible. For instance, let us assume that an instruction is issued in two parts. The first part executes without raising any exception and updates the architectural state by writing into register file or memory. The second part, however, raises an exception. To take the exception, the architectural state should be rolled back to the state of the processor just before the execution of the excepting VLIW instruction. However, the rollback is not possible because of the updates already made by the first part. Hence, the split-issued operations should not be allowed to update the architectural state of the processor to allow restoration to a consistent state. For doing so, buffers are required to hold values that the split-issued operations write into register file and memory.

Figure 6.9 illustrates the usage of buffers on a 4-cluster architecture. In the figure, bundles C0-C3 refers to the groups of operations scheduled to execute in clusters 0-3 respectively. If the operation executed by a FU is split-issued, then the result is written to the buffer. If the operation is not split-issued (i.e. the operation belongs to the last part of instruction), the result is directly written to register file (or memory). The results from the buffers are written to register file (or memory) when the last part of the instruction is executed. Bundle C0 (i.e. all operations belonging to cluster 0) is issued first, bundles C1 and C3 are issued next and finally bundle C2 is issued. Bundles C0, C1 and C3 are split-issued and do not write the results into their register files but write into buffers.



(a) Buffer organization for register file



(b) Buffer organization for memory

Figure 6.10: **Buffer organization**

However, bundle C2, being the last part of the instruction, writes directly into its RF. The content of the buffers holding the results of bundles C0, C1 and C3 are also written to their corresponding register files at the same time as C2.

Since the execution is always in-order between VLIW instructions, results from only one VLIW instruction per thread may have to be stored in buffers at any time. Thus, the storage requirement for the buffers to hold the results that are going to be written into RF is the issue-width of the processor for each thread. For instance, N buffers per thread for RF are required for a N-issue processor. Figure 6.10(a) shows the organization of the buffers used for register file to hold the values temporarily till the last part of an instruction is executed. The results (from FU or from buffers) are written to RF only when the Last-part[3] signal is set.

An extra set of buffers is required to hold the memory writes, which is equal to the number of memory functional units multiplied by the number of threads. Figure 6.10(b) shows the organization of buffers for a single memory unit. The buffers used to store the split-issued results both for memory and RF neither require to be multiported nor any data

[3] Already computed at issue stage by merging hardware, Figure 6.8(b))

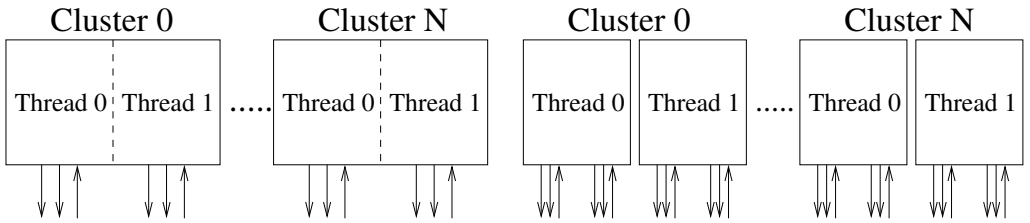


Figure 6.11: **Register File Organizations**

forwarding is required. Hence, little hardware overhead is incurred because of the buffers.

Delaying the updates to the architectural state, however, creates another issue because of a contention for register file and memory ports. These issues are discussed in the forthcoming sections.

6.4.2 Register file port contention

The last part of the instructions from multiple threads may be executed at the same time. For instance, the instruction shown in Figure 6.9 updates the values from buffers to register file and memory only when its last part is executed at cluster 2. There is a possibility that another thread might have its last part executing at the same time. Hence, the operations being executed may contain the last parts of the instructions of several threads. Now, results of the operations of several threads need to be written to the register file at the same time. To allow all these writes to register file, W register file write ports per thread must be available at each cluster for a W-issue per cluster architecture.

Two register file organizations can be used for a multithreaded VLIW architecture, namely, shared and partitioned. Figure 6.11 shows the shared and partitioned organization of the register files for a 2-Thread N-cluster architecture. The shared organization has a single register file with twice the registers while the partitioned organization has an individual register file for each thread. Both organizations have similar power, area and delay characteristics. A detailed discussion of the two designs is beyond the scope of this thesis.

A shared register file organization cannot be used with split-issue because the sharing of the ports limits the number of simultaneous writes. More write ports could be added to the shared register file, but doing so has a non-trivial hardware cost. On the other hand, the partitioned register file organization already provides the requisite register file ports. Hence, a partitioned register file organization is assumed for our experiments.

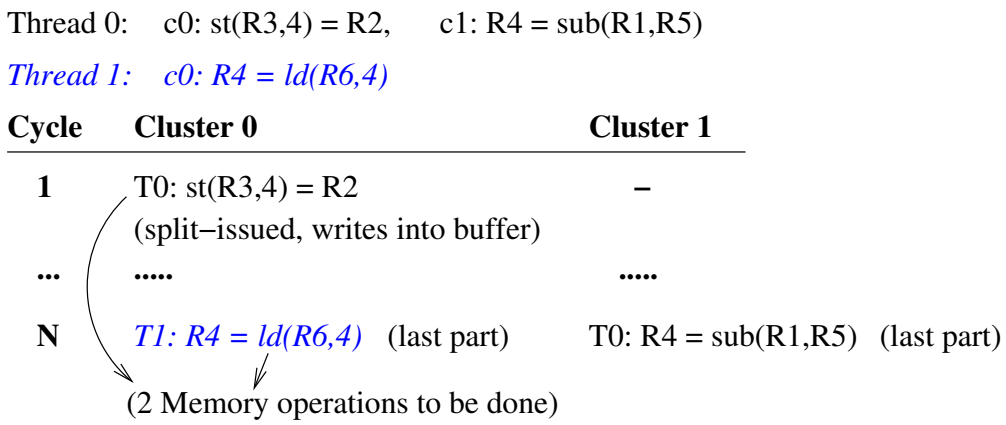


Figure 6.12: **Memory port contention because of delayed memory writes**

6.4.3 Memory Port Contention

Similar to the contention for register file write ports, a contention for memory ports also arises because of delayed updates to architectural state. If an instruction has a memory write in the split-issued part, the memory write writes into a buffer. The contents of the buffer are written to memory when the last part of the instruction is executed. The last part of the instruction can be issued with the last part of another instruction (belonging to a different thread). If the other instruction also has a memory operation in the same cluster as the split-issued part of the former instruction, it may happen that more memory operations have to performed than the number of memory ports available at a cluster at that cycle.

Figure 6.12 shows an example illustrating this issue on a 2-cluster machine with 1 memory port per cluster. The figure shows 2 instructions, each belonging to a different thread. Thread 0 has 2 operations, a memory write in cluster 0 and an alu operation in cluster 1. Thread 1 has only 1 memory read in cluster 0. At cycle 1, the memory write operation of Thread 0 is split-issued. Since the memory write is split-issued, it does not write into memory but into the buffer. The data in the buffer is committed to memory only when the last part is executed. At cycle N, the memory operation of Thread 1 and the last part of Thread 0 (i.e. alu operation) are issued. Now, two memory operations have to be performed at cluster 0 (the pending memory write of Thread 0 and the memory read of Thread 1) when the two instructions reach the execution pipeline stage. However, both memory operations cannot be performed at the same cycle, as there is only 1 memory port per cluster.

To solve this issue, if such a collision for a memory port is detected, the pipeline is

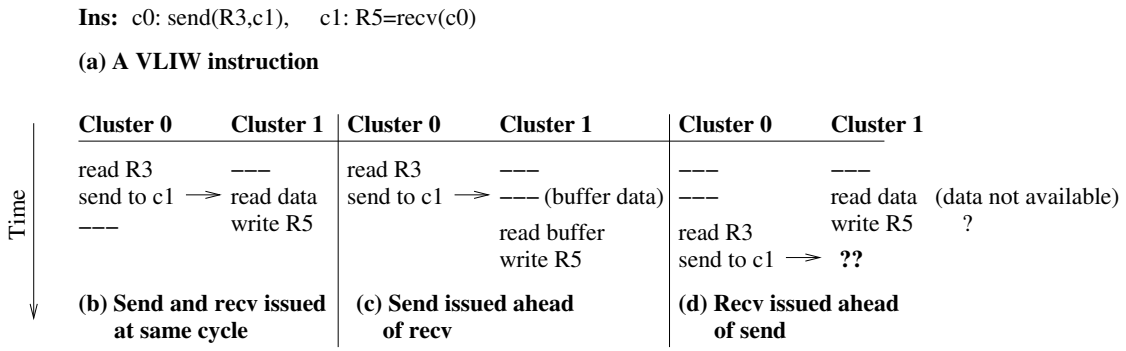


Figure 6.13: **Issuing inter-cluster communication operations**

stalled till all the memory operations have been performed.

6.4.4 Issues with Inter-cluster Communication

The experimental platform used in this thesis, VEX, uses inter-cluster `send` and `recv` operations to transfer data across clusters. Send operation reads a register from its corresponding register file and sends its value to another cluster over the inter-cluster communication network. `Recv` operation reads the data sent from the inter-cluster communication network and writes it into the corresponding register file. According to VEX semantics, `send` and `recv` operations should be simultaneously issued. However, as a consequence of split-issue, they might get issued in different cycles, which can result in an incorrect transfer of data.

Figure 6.13 shows an example highlighting this issue. For simplicity, the VLIW instruction shown in Figure 6.13(a) has only two operations, `send` operation in cluster 0 and `recv` operation in cluster 1. The `send` operation reads register R3 from register file of cluster 0 and sends its value to cluster 1. The `recv` operation in cluster 1 writes the received data to register R5 of cluster 1. Figure 6.13(b) shows the normal execution of the two operations. If `send` is issued earlier than `recv`, the data arrives at cluster 1 before `recv` is executed. A simple solution for this issue is to buffer the data till `recv` is executed, as shown in 6.13(c). However, if `recv` is issued ahead of `send`, as shown in Figure 6.13(d), the data is not available when `recv` executes, resulting in an incorrect transfer. Hence, while `send` can be ahead of `recv`, `recv` cannot be issued ahead of the corresponding `send`.

Note that `recv` performs 2 functions: Read the data from interconnection network and, write the data to the destination register. None of these functions require using a

particular FU or a complex hardware. A solution to the early issue of `recv` can be the following: If `recv` detects that data is not available when reading from inter-cluster communication network, it simply saves the destination register number to a buffer. When the data arrives later, the data is written to the corresponding register. Note that a write port in the register file should be available for writing the value to the register file. This requires usage of the partitioned register file organization to guarantee the availability of a write port to the register file.

6.5 Performance Evaluation

In this section, we present the experimental results obtained by implementing split-issue both at cluster-level and operation-level. Two different architectural configurations have been evaluated in this section to analyze the impact of inter-cluster communication operations on split-issue.

1 **No split communication:** In this configuration, VLIW instructions with inter-cluster communication operations are not split at all. Disallowing split of instructions with inter-cluster communication operations guarantees that compiler assumptions are never violated. Thus, no measures are required to maintain correctness.

2 **Always split**: This configuration allows splitting of instructions with inter-cluster communication operations. Splitting these instructions can violate the compiler assumptions because recv and send operations may execute at different cycles. In particular, the greater concern is with `recv` executing ahead of `send`. As a result, this configuration requires extra hardware to avoid breaking execution semantics as discussed previously in Section 6.4.4.

First, we discuss the performance results obtained when instructions can be merged only at cluster-level (CSMT). Only cluster-level split-issue is applicable in this case. Figure 6.14 shows the speedups (measured in terms of IPC) obtained by CCSI (cluster-level merging with cluster-level split-issue) over a 2-Thread and a 4-Thread CSMT (cluster-level merging but no split-issue) configuration respectively. In the figure, label 'AS' denotes the 'Always split' architectural configuration and label 'NS' represents the 'No split communication' architectural configuration. When splitting of instructions with inter-cluster communication operations is not permitted (NS), CCSI achieves, on an average, a speedup of 6.1% over a 2-Thread CSMT architecture and 3.5% over a 4-Thread CSMT

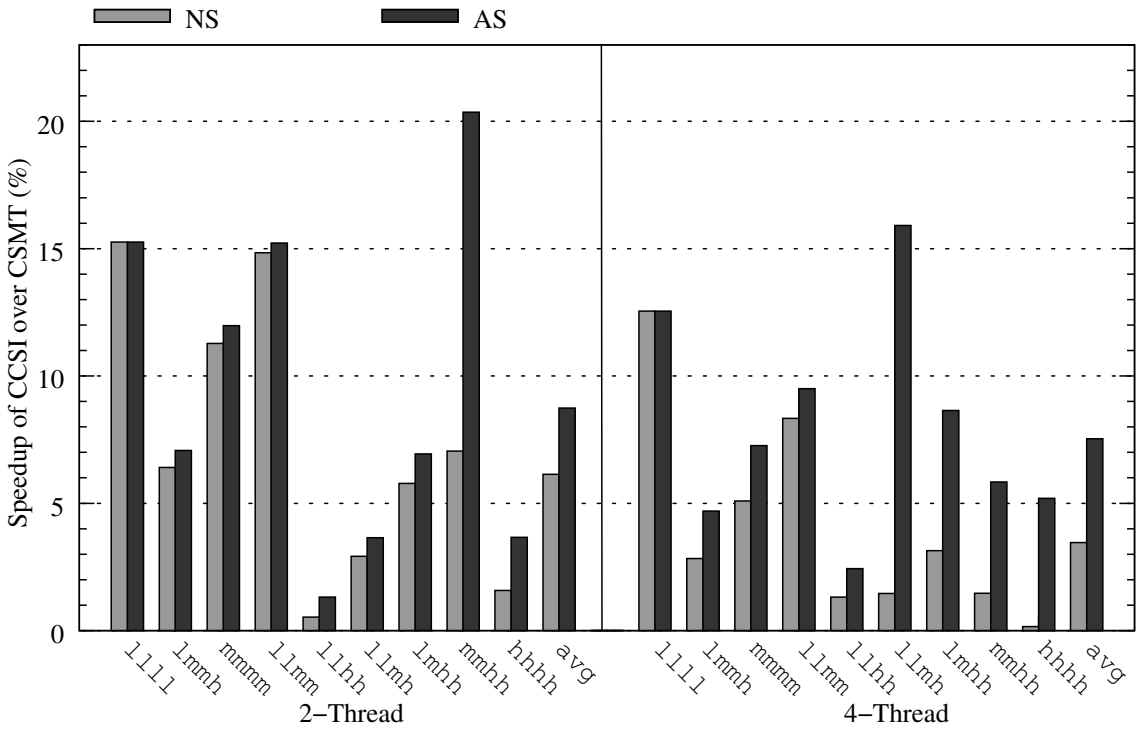


Figure 6.14: **Cluster-level split-issue (CCSI) speedups over CSMT**

architecture. For particular cases like llll, speedups as high as 15.1% are obtained for 2 threads. When splitting of inter-cluster communication operations is allowed (AS), an average speedup of 8.7% over a 2-Thread and 7.5% over a 4-Thread CSMT configuration is achieved on an average. For particular cases like mmhh, a speedup of 20.3% is obtained over a 2-Thread CSMT machine.

Next, we discuss the performance results obtained when instructions are merged at operation-level (OpSMT). In this case, both operation-level split-issue (OOSI) and cluster-level split-issue (COSI) techniques are applicable. Figure 6.15 shows the speedups obtained by OOSI and COSI over a 2-Thread and a 4-Thread OpSMT (operation-level merging but no split-issue) machine. In the figure, label 'AS' denotes the 'Always split' architectural configuration and label 'NS' represents the 'No split communication' architectural configuration. When splitting of instructions with inter-cluster communication operations is not permitted (NS), COSI achieves, on an average, a speedup of 7.5% and 6.4% over a 2-Thread and a 4-Thread OpSMT machine respectively. Using OOSI achieves higher performance improvements, on an average 8.2% over a 2-Thread and 7.9% over a 4-Thread OpSMT machine.

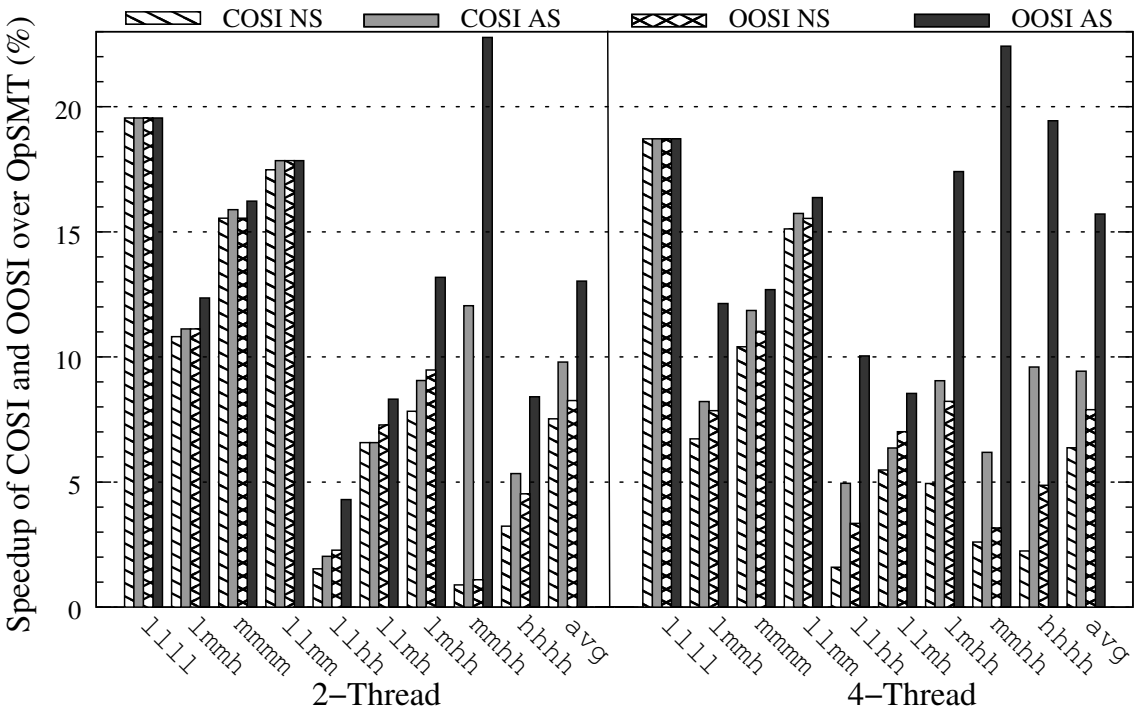


Figure 6.15: **Speedups obtained over OpSMT with cluster-level (COSI) and operation-level split-issue (OOSI)**

When splitting of instructions with inter-cluster communication operations is permitted (AS), COSI speedups increase to 9.8% over a 2-Thread OpSMT machine and 9.4% over a 4-Thread OpSMT machine. Speedup as high as 19.5% is achieved for particular cases like llll for 2-Thread OpSMT machine and 18.7% for a 4-Thread OpSMT machine. For OOSI, performance improvements increase to 13% and 15.7% on an average. For particular cases like mmhh, OOSI achieves speedups as high as 22.7% over a 2-Thread OpSMT machine and 22.4% over a 4-Thread OpSMT machine.

Note that when splitting of instructions with inter-cluster communication operations is not allowed (NS), performance improvement obtained for workloads containing benchmarks with high IPC is much lower than the one obtained when splitting of instructions with inter-cluster communication operations is allowed (AS). This holds true for any split-issue scheme used (CCSI, COSI and OOSI). For instance, for workload `mmhh`, using CCSI results into a performance gain of 7.4% on a 2-Thread CSMT machine for 'No split communication' model. For the 'Always split' model, the speedup increases almost three-fold to 20.3%. The large difference arises because high IPC benchmarks use inter-cluster communication operations more frequently than the low and medium IPC benchmarks. Constrain on splitting instructions with inter-cluster communication operations leads to

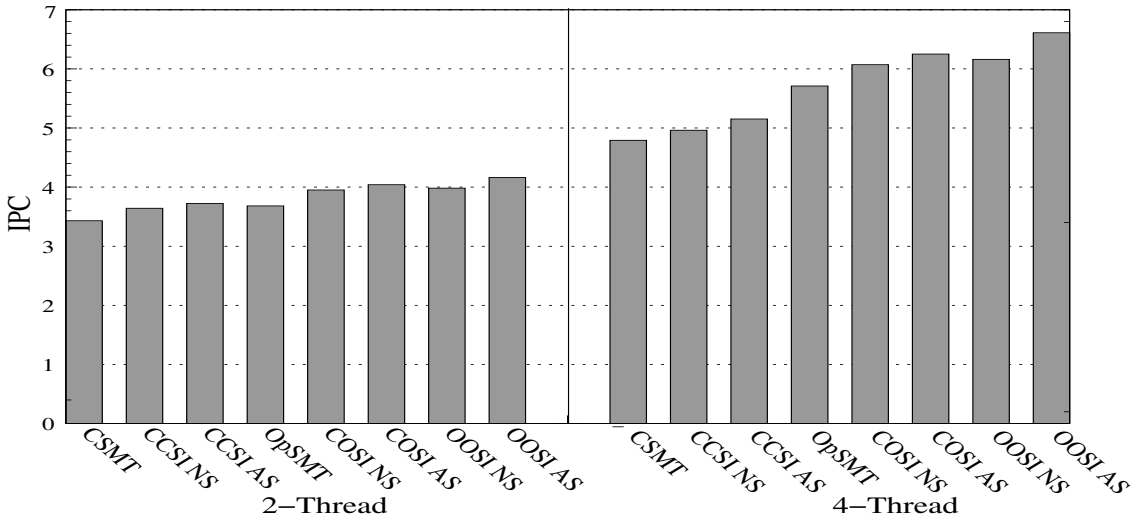


Figure 6.16: **Average Performance of all multithreading techniques**

an infrequent use of split-issue making it harder to fill the empty issue-slots. Hence, a significant difference in performance is observed.

Finally, we present an absolute performance comparison of all the multithreading techniques for all architectural configurations evaluated in this chapter. For ease of comparison, Figure 6.16 shows the average performance of all the techniques for a 2-Thread and a 4-Thread machine. The first thing to note is that by use of split-issue, cluster-level merging (CCSI AS) has practically the same performance (in fact, slightly better) as operation-level merging (OpSMT) for 2-Thread machine. Even on a 4-Thread processor, the performance difference between cluster-level merging (CSMT) and operation-level merging (OpSMT) decreases from 27% to only 13% when split-issue is used (CCSI AS). Since cluster-level merging is much cheaper to implement than operation-level merging, this makes cluster-level merging even more attractive.

Next, we focus on the performance difference between operation-level (OOSI) and cluster-level split-issue (COSI) when instructions are merged at operation-level. In general, COSI performance is always lower than OOSI. However, the performance difference is small. When the splitting of instructions with intercluster communication operations is not permitted, operation-level split-issue (OOSI NS) has an average performance advantage of only 0.7% over cluster-level split-issue (COSI NS) for 2-Thread, and 1.4% for a 4-Thread configuration. When the splitting of intercluster communication operations is permitted, the performance advantage of operation-level split-issue (OOSI AS) over cluster-level split-issue (COSI AS) is only a little higher, on an average 2.7% for a

2-Thread configuration and 5.7% for a 4-Thread configuration.

6.6 Summary and Conclusions

Operation-level Simultaneous MultiThreading (OpSMT) and Cluster-level Simultaneous MultiThreading techniques reduce horizontal and vertical waste in a VLIW processor by simultaneously issuing instructions from multiple threads. However, the restriction to issue a VLIW instruction in its entirety limits the opportunities to reduce horizontal waste.

Operation-level split-issue, removes the constrain of having to issue an instruction in entirety and allows a flexible issue of operations of a VLIW instruction. Operation-level split-issue does a dynamic scheduling of operations of VLIW instruction and increases OpSMT performance. However, implementing dynamic scheduling requires complex hardware which is not practical for area and power sensitive embedded clustered VLIW processors. Cluster-level split-issue, the technique proposed in this chapter, splits an instruction only at a cluster-level boundary. This eliminates the need for dynamic scheduling and has a much cheaper hardware implementation. Besides, the hardware changes required to implement cluster-level split-issue are quite small in nature and can be easily incorporated.

Experimental results show that cluster-level split-issue significantly improves performance. When instructions are merged at cluster-level, a performance improvement 8.7% is obtained over a 2-thread and 7.5% over a 4-thread processor (operation-level split-issue is not applicable for cluster-level merging). With instruction merging at operation-level (traditional OpSMT), a performance improvement of 9.8% is obtained over a 2-thread and 9.4% over a 4-thread OpSMT processor. In particular cases, performance improvements as high as 18.7% are achieved. Further, cluster-level split-issue, despite being more restrictive than operation-level split-issue, achieves similar performance. On an average, cluster-level split-issue performance is within 2.7% for a 2-thread processor and 5.7% for a 4-thread processor when compared to the performance obtained by using operation-level split-issue.

In conclusion, using cluster-level split-issue improves the multithreading performance of clustered VLIW processors at a low hardware overhead. Cluster-level split-issue achieves performance close to previously proposed split-issue technique, operation-level split-issue, but at a much lower complexity. Implementing operation-level split-issue requires dynamic scheduling and hence, it requires complex structures. On the other hand, the

changes required by cluster-level split-issue are much smaller in nature and does not increase the complexity of the processor significantly. Hence, cluster-level split-issue is a better tradeoff than operation-level split-issue for clustered VLIW processors.

Chapter 7
Hybrid MultiThreading

Supporting a large number of threads can improve a VLIW processor performance significantly. However, both operation-level and cluster-level Simultaneous MultiThreading (OpSMT and CSMT) suffer from scalability issues of merging hardware. While OpSMT is scalable only till 2 threads, scalability sweet spot of CSMT is 4 threads. As a result, a larger number of threads cannot be supported. In this chapter, we discuss our proposal named Hybrid MultiThreading (HMT) that can support large number of threads with a merging hardware that can merge fewer threads. HMT can be used with both CSMT and OpSMT merging hardware. For supporting higher number of threads than that can be merged, HMT selects a subset of threads to be merged every cycle. A new subset of threads is selected every cycle. While number of threads in the subset can be kept according to the merging hardware limitation, the restriction on overall number of threads supported is removed. Using HMT improves performance significantly without impacting the merging hardware cost.

7.1 Motivation

Supporting a larger number of threads is a possible way for improving performance. However, some multithreaded schemes do not benefit from an increase beyond a small number of threads. To illustrate this, Figure 7.1 shows the average IPC obtained by Interleaved MultiThreading (IMT), Cluster-level Simultaneous MultiThreading (CSMT) and Operation-level Simultaneous MultiThreading (OpSMT) for the 8-threaded workloads evaluated in this chapter (The workloads have been described earlier in Chapter 2) on a single-thread, 2-thread, 4-thread and an 8-thread machine. In the figure, the filled portion of the bars is the IPC obtained for the real memory model (described earlier in Chapter 2) and the extra white bar on top represents the additional IPC obtained for a perfect memory

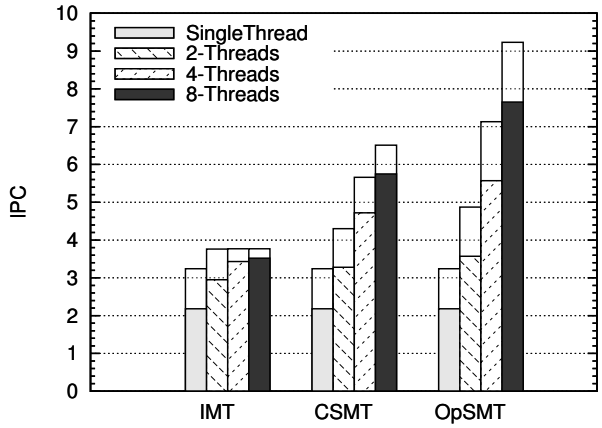


Figure 7.1: **Average IPC for IMT, CSMT and OpSMT**

model assuming no cache misses. For an ease of comparison, the single-thread IPC bar is shown for all IMT, CSMT and OpSMT.

The first thing to notice is that even with a perfect memory model, IMT performs better than the single-thread processor. This is because, despite the fact that there is no vertical waste due to memory stalls, a few issue cycles are still lost due to taken branches and def-to-use latency of operations like loads, multiplies and comparisons. IMT also hides this vertical waste by issuing instructions from an alternate thread. However, little improvement in performance is achieved in moving from a 2-thread machine to a 4-thread or an 8-thread machine. With a real memory model, a moderate performance difference still exists between a 2-thread and a 4-thread machine, but an 8-thread machine provides only marginal performance improvement over a 4-thread machine. This happens because IMT can only remove vertical waste. The amount of vertical waste keeps on reducing with an increase in number of threads. Thus, IMT gets little benefit by supporting more than 4 threads. In contrast to IMT, CSMT and OpSMT also remove horizontal waste, and thus, both CSMT and OpSMT performance keep on improving significantly up to 8 threads. In moving from a 4-thread machine to an 8-thread machine, CSMT performance improves by 15% while OpSMT performance improves by 29% for the perfect memory model, while for the real memory system, there is a performance improvement of 22% for CSMT and 37% for OpSMT.

It is desirable to support a large number of threads on a OpSMT or CSMT processor because of the performance gains that are achieved. However, the hardware complexity increases with the number of threads and limits the number of threads that can be supported simultaneously. In general, most of the hardware complexity in a VLIW processor

can be attributed to the extra register sets required to support multiple threads and the merging hardware required to merge instructions from different threads. Most of the multithreading schemes including IMT, OpSMT, CSMT, etc. require a register set per thread. The cost of extra register sets is not trivial. Nevertheless, systems like Cray MTA/Tera [4] had 128 register sets per processor to support 128 threads using IMT with a VLIW ISA. Compared to extra register sets, which has the same cost for all multithreading schemes, the cost of merging hardware varies significantly depending on the multithreading scheme used.

IMT selects a different thread at each cycle. Hence, IMT thread selection can be simply implemented as a counter which is incremented at each cycle. In comparison to IMT, CSMT can issue instructions from multiple threads at a time. Hence, CSMT also requires to check for resource collisions at cluster-level amongst the threads. This results in a more complex merging hardware than IMT but till 4 threads the cost of merging hardware is very low. OpSMT checks resource collisions at operation-level and can select operations from multiple threads at the same cluster, in contrast to CSMT. To fit operations from multiple threads in the same cluster, OpSMT merging hardware also needs to route the operations of the instructions. Because of this added complexity, OpSMT is not practical to use beyond 2 threads. Our proposal, Hybrid MultiThreading (HMT), can support larger number of threads then the number of threads supported by the merging hardware. HMT is explained in detail in the following section.

7.2 Hybrid MultiThreading

Hybrid MultiThreading (HMT) combines the advantages of interleaving threads and executing simultaneously instructions from different threads. HMT merges instructions from only a subset of threads instead of merging all the threads. Every cycle, a new subset is selected in a manner analogous to IMT. If one or more threads produce a cache miss (or even a short latency delay that arises because of data dependencies), selecting a subset of threads helps in hiding the latency efficiently. Moreover, the merging hardware can still operate efficiently even if a few threads are blocked. The number of threads that can be merged at a time is dictated by the merging hardware cost. However, HMT approach supports a larger number of threads without incurring a large merging hardware cost. For instance, 4 threads can be supported with a merging hardware that can merge only 2 threads at a time. Conceptually, HMT is similar to the M:N Thread Scheduling [9],

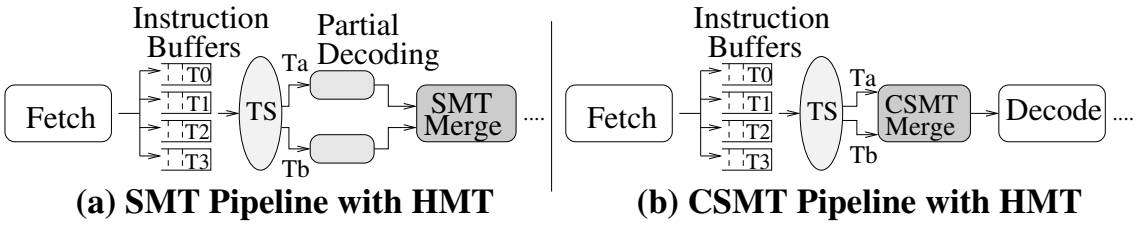


Figure 7.2: **HMT Pipeline**

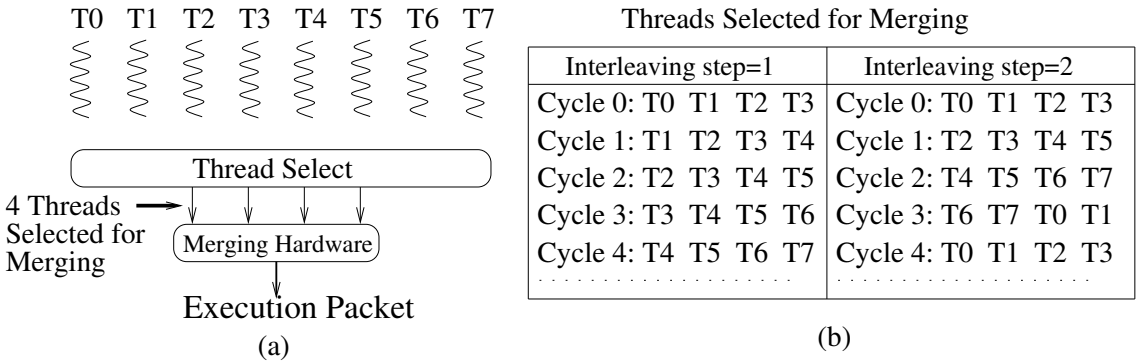


Figure 7.3: **Hybrid Multithreading Example**

a commonly used Operating System thread scheduling implementation.

HMT is independent of the implementation of the merging hardware. Either OpSMT or CSMT can be used to merge instructions from different threads. Figures 7.2(a) and (b) show the first pipeline stages for both OpSMT and CSMT processors with HMT respectively, where 4 threads are supported but only 2 are merged at a time by selecting a subset of 2 threads each cycle. In the figure, TS represents the Thread Select which selects a subset of threads for further execution. For a OpSMT processor, using a subset of threads also means that fewer threads need to be partially decoded (OpSMT requires partially decoded operation to recognize the FU used).

Figure 7.3 shows an example of HMT. The HMT processor shown in the figure 7.3(a) supports eight threads, T0-T7, but the merging hardware has the ability to merge instructions from only 4 threads at a time. Thus, up to 4 threads are selected every cycle by the Thread Select. The merging hardware takes the selected threads and produces an execution packet by merging their instructions. Note that the selection of a thread does not guarantee its inclusion in the execution packet. The selection only implies that the thread's instruction will be considered for merging. The actual inclusion in the execution packet depends on the merging scheme used by the merging hardware and whether the thread is blocked or not. Every cycle, a new subset of threads is selected which is decided

by the interleaving step. Interleaving step is the number of threads skipped to select the new set of threads. Following section discusses interleaving step in detail.

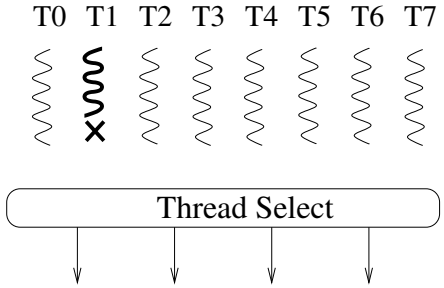
7.2.1 Interleaving Step

The first step in HMT is to select the subset of the threads that should be considered for merging in next cycle. The primary factor that influences the selection of threads is the value of interleaving step used. IMT, for instance, uses always an interleaving step of 1 and selects the next thread every cycle. In HMT, using different values for interleaving step creates interesting thread selections which may obtain different performance. For example, for the 8-thread HMT machine with a 4-thread merging hardware shown in figure 7.3(a), with an interleaving step of 1, the 4 threads selected at a given cycle contain 3 threads that were also selected in the previous cycle. With an interleaving step of 2, the selected threads contain only 2 threads that were also selected in previous cycle. Figure 7.3(b) shows the thread selections obtained by using different values of interleaving step of 1 and 2. Initially, threads T0-T3 are selected at cycle 0 for all examples. With an interleaving step of 1 (column 1 of Figure 7.3(b)), the first thread T0 is skipped at cycle 1, and the next 4 threads T1-T4 are selected, and so on. With an interleaving step of 2, first two threads T0 and T1 are skipped at cycle 1, and the next 4 threads T2-T5 are selected. At cycle 2, threads T2 and T3 are skipped and threads T4-T7 are selected, and so on.

Thread Select (TS) initially considers the selections obtained by the interleaving. However, some of the threads in a given selection may be blocked, lowering the opportunities for merging. Following section discusses several thread selection schemes that can be employed by the TS to replace a blocked thread.

7.3 Thread Selection Schemes

Thread selection in HMT can be divided into two categories: Static and Dynamic. In a static thread selection, TS does not take any runtime decisions and completely relies on interleaving for selecting the threads. In Dynamic thread selection, the subset of threads obtained using interleaving is initially considered, but if some of the threads in the subset are blocked (because of a cache miss, for instance), they can be replaced by other threads. Dynamic thread selection is more complex than a static one, but allows a better use of the resources. In particular, an IMT implementation where the blocked threads are skipped



(a)

	Static Selection	Dynamic Selection
Cycle 0	T0 ~~T1~~ T2 T3	T0 T2 T3 T4
Cycle 1	~~T1~~ T2 T3 T4	T2 T3 T4 T5
Cycle 2	T2 T3 T4 T5	T2 T3 T4 T5
Cycle 3	T3 T4 T5 T6	T3 T4 T5 T6

(b)

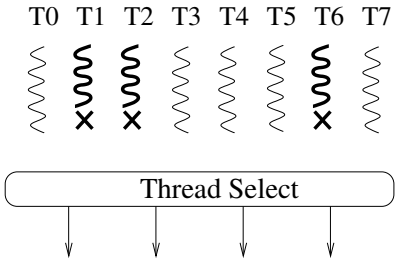
Figure 7.4: **Static vs Dynamic Thread Selection**

also belongs to dynamic thread selection category, as instructions from a non-blocked thread are issued if the given thread is blocked.

Figure 7.4 shows a comparison of a static vs dynamic thread selection for a machine with 8 threads (T0-T7) and a thread select that can select up to 4 threads. An interleaving step of 1 is assumed in this example. As shown in the Figure 7.4(a), Thread T1 is assumed to be blocked. Static thread selection selects Thread T1 at both cycle 0 and cycle 1 despite T1 being blocked as shown in Figure 7.4(b). Thus, a selection slot is wasted at these cycles. Dynamic thread selection, however, skips Thread T1 and instead selects the non-blocked threads T4 at cycle 0 and T5 at cycle 1. Thus, all selection slots are completely utilized. The following dynamic thread selection schemes have been considered in this chapter:

- **First Non Block** (FNB): If one of the threads in the set considered for merging is blocked, then it is substituted by the first non-blocked thread. This scheme is used in the example presented in Figure 7.4(b).

- **Equivalent** (Eq): This scheme is a consequence of cluster renaming. Recall that *cluster renaming* (discussed in Chapter 4) distributes logical clusters assigned by the compiler to different physical clusters. The use of cluster renaming creates an interesting case: If the number of threads is greater than the number of clusters, multiple threads share the same renaming value. To try to maximize the number of instructions that can be merged, the thread selection scheme should avoid the selection of threads that have the same renaming value. Using FNB scheme, however, may result in cases where multiple threads in the selected subset have the same renaming value, resulting in an increased contention for resources for some clusters. For instance, using the same example as in Figure 7.4(b), where Thread



(a)

	Selected Threads			
	Static	FNB	Eq	Eq+FNB
Cycle 0	T0 ~~T1~~ ~~T2~~ T3	T0 T3 T4 T5	T0 T5 ~~T2~~ T3	T0 T5 T4 T3
Cycle 1	~~T1~~ ~~T2~~ T3 T4	T3 T4 T5 T7	T5 ~~T2~~ T3 T4	T5 T3 T4 T7
Cycle 2	~~T2~~ T3 T4 T5	T3 T4 T5 T7	~~T2~~ T3 T4 T5	T3 T4 T5 T7
Cycle 3	T3 T4 T5 ~~T6~~	T3 T4 T5 T7	T3 T4 T5 ~~T6~~	T3 T4 T5 T7
Cycle 4	T4 T5 ~~T6~~ T7	T4 T5 T7 T0	T4 T5 ~~T6~~ T7	T4 T5 T7 T0

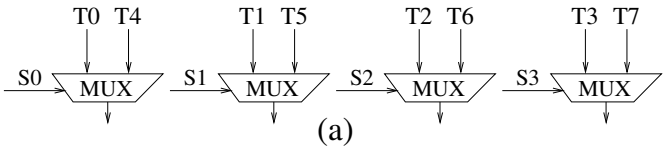
(b)

Figure 7.5: **Thread Selection Schemes**

T1 is blocked, and assuming a 4-cluster machine, FNB scheme selects at cycle 0 Thread T4 as the replacement for Thread T1. Thus, threads T0, T2, T3 and T4 are selected to be merged. Now on a 4-cluster 8-thread machine, threads T0 has same renaming value as T4, T1 as T5 and so on. Since, Thread T0 and Thread T4 have the same renaming value, both threads will compete for resources in same clusters. This limits the benefits of selecting a replacement thread. A better choice would be to pick Thread T5 as the replacement for Thread T1, as both of them have the same renaming value. Then, if T0, T2, T3 and T5 (instead of T4) are the selected threads, none of them have the same renaming value. Hence, a better merging of the instructions from these threads can be expected.

- **Equivalent + First Non Block** (Eq+FNB): This scheme is a combination of Eq and FNB schemes. If a thread is blocked, first an equivalent thread is considered as the replacement. If the equivalent thread is also blocked, then the first thread which is not blocked is used as a replacement for the blocked thread.

To illustrate the differences between these thread selection schemes, Figure 7.5(b) shows the thread selections done by Static, FNB, Eq and Eq+FNB schemes for an 8-thread 4-cluster machine with a 4-thread merging hardware. For all the schemes, an interleaving step of 1 is used. Threads T1, T2 and T6 are assumed to be blocked. At cycle 0, all the proposed schemes start with threads T0, T1, T2 and T3 as the initial subset. In static selection, since threads T1 and T2 are blocked, two selection slots are wasted. FNB skips the blocked threads and thus 4 non-blocked threads T0, T3, T4 and T5 are selected. Eq scheme selects Thread T5 instead of Thread T1, which is blocked, but cannot replace Thread T2 since its equivalent Thread T6 is also blocked. This results in an unutilized selection slot. Eq+FNB selects Thread T5 in place of Thread T1. Since both Thread T2 and its equivalent Thread T6 are blocked, Eq+FNB selects the first non-blocked Thread



(a)

	Interleaving step=1					Interleaving step=2					Interleaving step=4				
	Threads				S0 S1 S2 S3	Threads				S0 S1 S2 S3	Threads				S0 S1 S2 S3
Cycle 0	T0	T1	T2	T3	0 0 0 0	T0	T1	T2	T3	0 0 0 0	T0	T1	T2	T3	0 0 0 0
Cycle 1	T4	T1	T2	T3	1 0 0 0	T4	T5	T2	T3	1 1 0 0	T4	T5	T6	T7	1 1 1 1
Cycle 2	T4	T5	T2	T3	1 1 0 0	T4	T5	T6	T7	1 1 1 1	T0	T1	T2	T3	0 0 0 0
Cycle 3	T4	T5	T6	T3	1 1 1 0	T0	T1	T6	T7	0 0 1 1	T4	T5	T6	T7	1 1 1 1
Cycle 4	T4	T5	T6	T7	1 1 1 1	T0	T1	T2	T3	0 0 0 0	T0	T1	T2	T3	0 0 0 0

(b)

Figure 7.6: **Static Thread Selection Hardware**

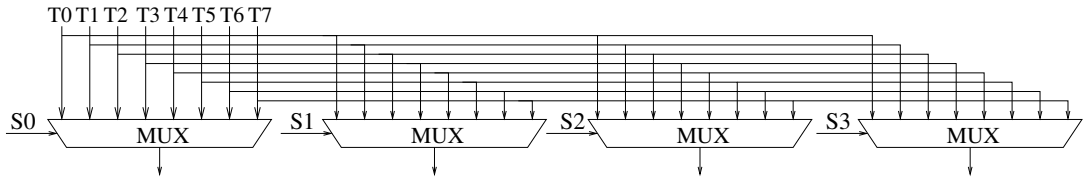


Figure 7.7: **General Dynamic Thread Selection Hardware**

T4 as the replacement. At cycle 1, using an interleaving step of 1, the thread selection for all the schemes starts from threads T1, T2, T3 and T4 and so on.

We now discuss the relative strengths and weaknesses of both static and dynamic thread selection schemes. Figure 7.6(a) shows a very simple implementation for the static thread selection scheme for an 8-thread architecture where the merging hardware can support 4 threads. In the figure, T0-T7 represent the 8 threads and S0-S3 are the select signals for the muxes. This design can be used with different values of interleaving step by using a different select logic for muxes. Figure 7.6(b) shows the corresponding select signals for the muxes with different values of interleaving step and the selected threads. The select signals can either be stored on-chip in a table or can be generated by an on-chip logic. For instance, for an interleaving step of 1, a simple 4-bit twisted ring counter can be used.

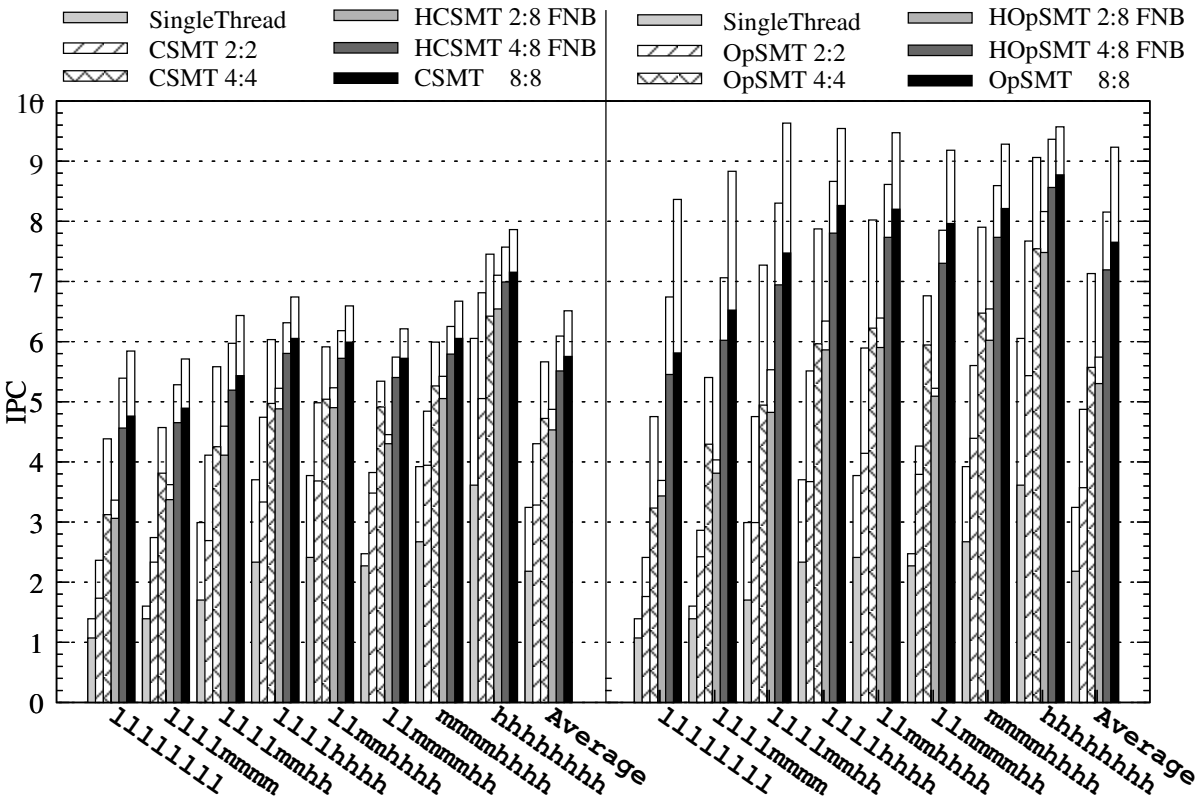
In a dynamic thread selection, any thread can be selected at any selection slot. For instance, in FNB scheme there is no restriction in the selection. Hence, for each selection slot, all threads have to be checked, as shown in Figure 7.7. However, each input line to a mux is a complete VLIW instruction. Hence, the muxes themselves consume significant area and power as a lot of wiring and routing needs to be done. The select signal gener-

ation for the muxes is also more complex than for static selection, as the blocked threads need to be skipped. This adds to the delay of the merging hardware, which already has a high delay. Since the thread select should fit in the same pipeline stage as merging hardware, implementing a dynamic thread selection scheme may not be practical.

However, the Eq thread selection scheme is an exception to the dynamic selection scheme complexity since it restricts the threads that can be used for replacement. In Eq scheme, if a thread is blocked, it can be substituted only by a fixed equivalent thread. For instance, with a 8-thread 4-cluster machine and a merging hardware of 4 threads, Thread 0 can be substituted only by Thread 4, Thread 1 by Thread 5, and so on. It is possible that more than 2 threads have the same renaming value. For instance, for a 2-cluster 8-thread machine, 4 threads have the renaming value of 0, and rest 4 have the renaming value of 1. However, with Eq scheme, we restrict to one the number of threads that are considered as replacement. This results in a design with significantly lower complexity in comparison to the general dynamic thread selection scheme. In fact, the same simple design used in static thread selection shown in Figure 7.6(a) can be used to implement Eq scheme. The generation of the select signals for the muxes is different though, as the blocked state of the threads has to be considered. Nevertheless, the complexity and delay of Eq thread selection scheme is similar to static thread selection. Note that the scheme Eq+FNB requires a selection hardware similar to the general dynamic thread selection presented at Figure 7.7.

7.4 Results

This section presents the performance results obtained by using HMT with both operation-level (OpSMT) and cluster-level (CSMT) merging approaches. This section also evaluates the influence of the different thread selection schemes on HMT performance. 8-threaded workloads described earlier in Table 2.4 in Chapter 2 are used to evaluate HMT performance. HMT performance results are shown in the figures 7.8 to 7.10. In the figures, a HMT approach with CSMT merging hardware is referred as HCSMT, while a HMT approach with OpSMT merging hardware is referred as HOpSMT. An extra label of the format $A : B$ is appended to all multithreading configurations, where A is the number of threads that can be merged by the merging hardware and B is the number of the threads supported by the processor (virtual CPUs). A and B values are always the same for a pure CSMT and OpSMT machine, but are different for a HMT machine since the number of



(a) HCSMT (b) HOpSMT

Figure 7.8: **IPC of the workloads for HMT with FNB policy**

threads merged is lower than the number of threads executed. Another label, indicating the thread selection policy, is appended to HMT configurations. For instance, label ”CSMT 4:4” indicates a pure 4-thread CSMT machine (4 threads supported and also merged at a time), and label ”HCSMT 4:8 FNB” states a HMT machine that supports 8 threads using the FNB thread selection policy with a CSMT merging hardware that can merge 4 threads at a time. In all the figures, the filled portion of the bars is the IPC obtained for the real memory system, and the extra white bar on top represents the additional IPC obtained while using the perfect memory model. In all the experiments shown in this section, an interleaving step of 1 is used. We repeated the experiments with varying degree of interleaving step. However, little difference in HMT performance was observed. Hence, we restrict the results presented to an interleaving step value of 1.

7.4.1 Performance Evaluation of HMT with FNB Policy

Figures 7.8(a) and (b) show the performance comparison between HMT with the FNB thread selection policy and a pure OpSMT/CSMT architecture. The figures also include the performance of a single-thread machine and the peak performance achievable (”CSMT 8:8” and ”OpSMT 8:8”). In pure CSMT and OpSMT machines, a significant amount of performance is lost because of cache misses. Even short latencies block threads from executing, which further results into a reduction in the number of threads that can be merged during this time. HMT mitigates this performance loss because of its ability to issue instructions from different threads every cycle. As a consequence, HMT approaches significantly improve the performance over a pure CSMT/OpSMT machine when using the same merging hardware. HMT achieves a significant improvement in performance even with a perfect memory model because of its ability to hide short latencies.

On an average, for a CSMT merging hardware, using HMT improves the performance over pure CSMT by 38% with a 2-thread merging hardware ("HCSMT 2:8 FNB" vs "CSMT 2:2"), and 17% with a 4-thread merging hardware ("HCSMT 4:8 FNB" vs "CSMT 4:4") with the real memory model. For the OpSMT merging hardware, there is a performance improvement of 48% over pure OpSMT with a 2-thread merging hardware ("HOpSMT 2:8 FNB" vs "OpSMT 2:2"), and 29% over pure OpSMT with a 4-thread merging hardware ("HOpSMT 4:8 FNB" vs "OpSMT 4:4"). Further, the performance of a 4-thread merging hardware by using HMT is quite close to the peak performance of 8-thread merging hardware in pure CSMT and OpSMT (”CSMT 8:8” and ”OpSMT 8:8”). On an average, with a CSMT merging hardware, "HCSMT 4:8 FNB" performance is within 4.3% of the the "CSMT 8:8" performance, while with a OpSMT merging hardware, "HOpSMT 4:8 FNB" performance is within 6.3% of "OpSMT 8:8" performance. Besides, HMT with a 2-thread merging hardware achieves performance close to pure 4-thread CSMT/OpSMT configurations (within 4% for CSMT merging hardware and 4.8% for OpSMT merging hardware) and even outperforms them in some cases (for instance, `hhhhhhhh` for CSMT merging hardware and `llllllll` for OpSMT merging hardware) for the real memory model. Thus, using HMT improves the performance of a pure CSMT/OpSMT machine significantly with a lower merging hardware cost.

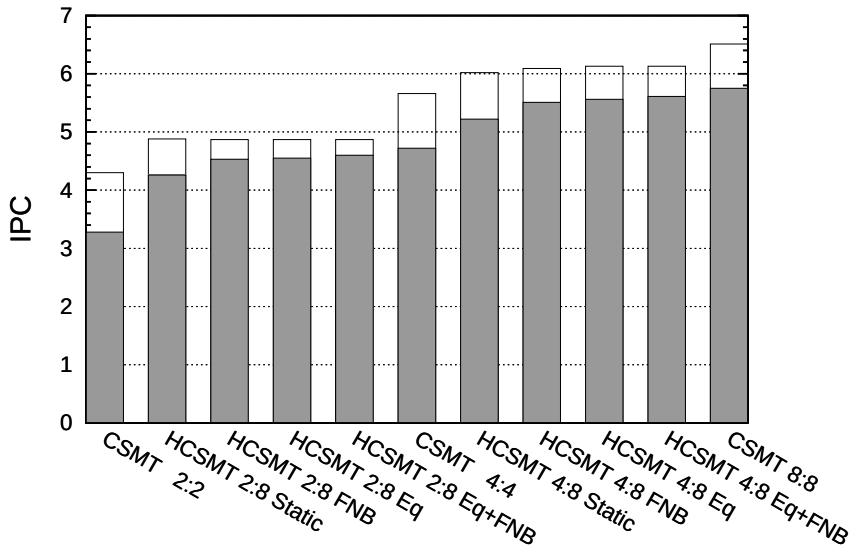


Figure 7.9: **Average Performance of Different Thread Selection Schemes for HCSMT**

7.4.2 Detailed Performance Evaluation of Thread Selection Schemes

Finally, we present an evaluation of the effect of the different thread selection schemes described earlier in Section 7.3, namely static, FNB, Eq and Eq+FNB. Figure 7.9 shows the IPC obtained by different thread selection schemes for a CSMT merging hardware for both perfect and real memory models. Figure 7.10 shows the same data for a OpSMT merging hardware. The figures also include the performance obtained by pure CSMT/OpSMT machines ("CSMT 2:2", "CSMT 4:4", "OpSMT 2:2" and "CSMT 4:4") and the peak performance achievable ("OpSMT 8:8" and "CSMT 8:8"). For clarity, only the average IPC achieved for all workloads is shown in the figures.

In general, static thread selection has the lowest performance across all the schemes, while scheme Eq+FNB performs the best. On an average, for the real memory model, the Eq+FNB scheme with a CSMT merging hardware obtains a performance improvement of 38% over pure CSMT with a 2-thread merging hardware ("HCSMT 2:8 Eq+FNB" vs "CSMT 2:2") and an 18% performance improvement with a 4-thread merging hardware ("HCSMT 4:8 Eq+FNB" vs "CSMT 4:4"). With OpSMT merging hardware, Eq+FNB has a performance improvement of 48% with a 2-thread merging hardware ("HOpSMT 2:8 Eq+FNB" vs "OpSMT 2:2") and 29% with a 4-thread merging hardware ("HOpSMT 4:8 Eq+FNB" vs "OpSMT 4:4"). Further, the performance achieved by Eq+FNB with a 4-thread merging hardware is quite close to the peak performance achievable (within 2.4% for CSMT and 6.3% for OpSMT merging hardware).

Note that even though static thread selection is the lowest performing scheme, it still

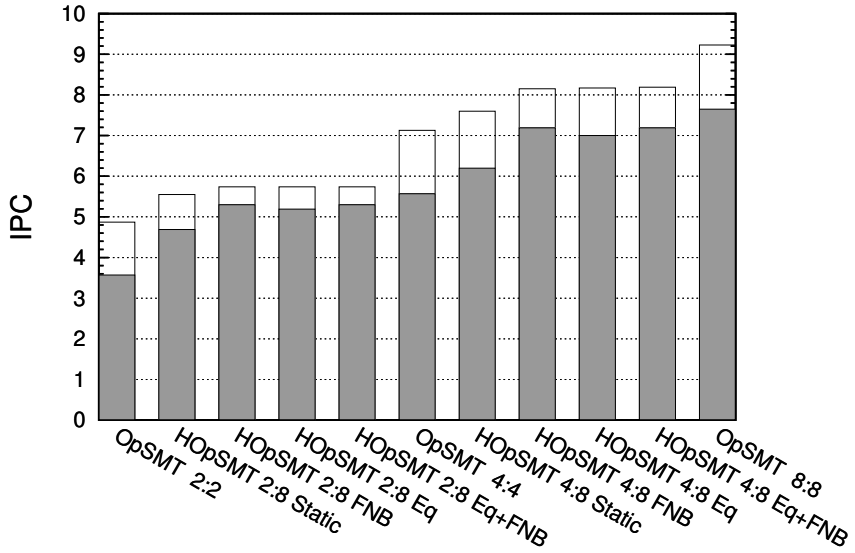


Figure 7.10: **Average Performance of Different Thread Selection Schemes for HOpSMT**

improves the performance significantly. On an average, with the real memory model, static selection has 29% higher performance than the pure CSMT machine with a 2-thread merging hardware ("HCSMT 2:8 Static" vs "CSMT 2:2"), and 11% with a 4-thread merging hardware ("HCSMT 4:8 Static" vs "CSMT 4:4). With the OpSMT merging hardware, static selection has 31% higher performance than the pure OpSMT machine with a 2-thread merging hardware ("HOpSMT 2:8 Static" vs "OpSMT 2:2"), and 11% with a 4-thread merging hardware ("HOpSMT 4:8 Static" vs "OpSMT 4:4).

Another interesting thing to note is that for a CSMT merging hardware, Eq scheme outperforms FNB scheme (though the difference in performance is not much). This happens because the replacement thread selected by Eq scheme does not share the renaming value with other threads in the selection set. Thus, more instructions are in general issued simultaneously because of less resource conflicts. Even the best performing scheme, Eq+FNB, has only a very small performance advantage over Eq (1.6% with a 2-thread merging hardware and only 0.8% with a 4-thread merging hardware). Thus, Eq scheme seems to be the most suitable thread selection scheme with a CSMT merging hardware as it has a lower hardware complexity (similar to static) with a performance similar to the most complex Eq+FNB scheme. With OpSMT merging hardware, FNB scheme outperforms Eq in the real memory model (but the performance difference is small, on an average 2.4% with a 2-thread and 4.6% with a 4-thread merging hardware). This is opposite to the behavior observed with the CSMT merging hardware, where Eq outperforms

FNB most of the time. This happens because CSMT merges instructions at cluster level, and thus it heavily depends on cluster renaming to reduce contention for clusters. On the other hand, OpSMT can merge instructions from different threads even if they use same clusters. Hence, cluster renaming is not so critical for OpSMT as for CSMT. As a result, replacing a blocked thread with a non-blocked one is more important than only checking the equivalent thread in OpSMT. With perfect memory, where all the latencies are short, Eq scheme outperforms FNB, as the replacement threads selected by Eq fare better at merging because all selected threads have different renaming values.

In conclusion, while Eq scheme is not the best performer among the dynamic thread selection schemes, its performance is very competitive with a significantly lower hardware complexity. Hence, Eq is the most suitable dynamic thread selection scheme.

7.5 Summary and Conclusions

Several multithreading schemes like Interleaved MultiThreading (IMT) and Simultaneous MultiThreading (SMT) have been proposed to reduce the resource underutilization in VLIW processors. IMT provides significant performance improvements if the number of threads supported is small because of its ability to reduce vertical waste. With a larger number of threads, less opportunities exist for removing vertical waste, resulting in only marginal performance improvements with IMT. On the other hand, SMT performance keeps on improving significantly because of its ability to also reduce horizontal waste by merging instructions from different threads. However, in SMT only a small number of threads can be supported because of the complexity of merging hardware. In contrast, IMT does not require a merging hardware and can support a larger number of threads.

In this chapter, we have presented Hybrid MultiThreading (HMT), a technique which combines the advantages of both IMT and SMT. HMT supports a larger number of threads than SMT with a given merging hardware cost. This is achieved by merging only a subset of threads at a time. Every cycle, a new subset of threads is selected. HMT is independent of the merging scheme used by the merging hardware. In particular, this chapter evaluated HMT with operation-level and cluster-level simultaneous multithreading (OpSMT and CSMT). The chapter also evaluated several thread selection schemes that are used to select the subset of threads to consider for merging every cycle namely static, equivalent (Eq), first non-block (FNB), and a combination of equivalent and first non-block (Eq+FNB).

The experimental results show that HMT significantly improves the performance over

a OpSMT/CSMT processor that does not use HMT. Even with the simplest static thread selection, there is a significant performance improvement of 31% with a 2-thread merging hardware and 11% with a 4-thread merging hardware over OpSMT. While with a more complex thread selection scheme like Eq+FNB, using HMT improves performance by 48% with a 2-thread merging hardware and 29% with a 4-thread merging hardware over OpSMT. Further, using HMT with a 4-thread merging hardware achieves a performance similar to an 8-thread merging hardware without having to incur the cost of 8-thread merging hardware. Also, the Eq scheme performs quite well even though it has a much lower complexity compared to FNB and Eq+FNB. Interestingly, Eq outperforms FNB with CSMT merging hardware but the contrary is true with OpSMT merging hardware. This arises because cluster renaming is not so critical for OpSMT merging hardware as for CSMT. Nevertheless, the performance difference is quite small, making Eq scheme the most attractive choice for the thread selection scheme, as it has a performance close to the most complex Eq+FNB scheme but a complexity similar to the simplest static thread selection scheme.

Chapter 8
Putting It All Together

In the thesis, we have presented Cluster-level Simultaneous MultiThreading (CSMT) and several techniques that further improve the performance of CSMT namely Heterogeneous Merging, Cluster-level Split-Issue and Hybrid MultiThreading. These techniques are also applicable to Operation-level Simultaneous MultiThreading (OpSMT). So far, all these techniques have been discussed in isolation. The three proposed techniques are orthogonal to each other and can be used simultaneously. Using all the techniques together result in a further performance improvement. This chapter presents an evaluation of all these techniques when put to use at the same time. The following section discusses it in more detail.

8.1 Motivation

The thesis has presented several proposals that improve performance and multithreading scalability for Cluster-level Simultaneous MultiThreading (CSMT) and Operation-level Simultaneous MultiThreading (OpSMT).

- **Heterogeneous Merging** combines both CSMT and OpSMT merging hardware to support more threads at a low merging hardware cost and achieve better performance.

- **Cluster-level Split-Issue** improves multithreading performance by issuing a clustered VLIW instruction in parts with a split at the cluster boundary. The increased flexibility in issuing the instructions improves multithreading performance for both OpSMT and CSMT.

- **Hybrid MultiThreading** supports more threads than the merging hardware limit

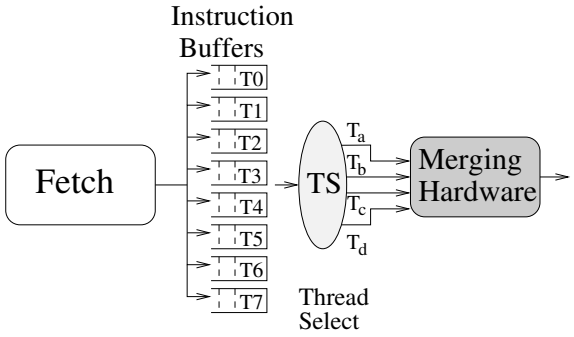


Figure 8.1: **All Schemes**

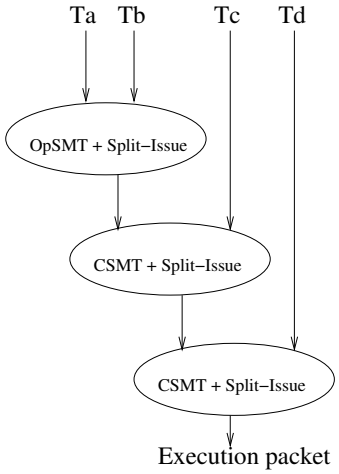


Figure 8.2: **Merging Hardware**

by selecting a subset of threads to merge every cycle thereby improving processor performance. During execution, some of the running threads can be blocked (cache misses etc.) reducing the multithreading efficiency. Supporting more threads allows to replace blocked threads with running ones and results in a better utilization of the multithreaded hardware.

All these approaches use different techniques for improving the resource utilization in a clustered VLIW processor. But the employed method in each approach is orthogonal to others in the way they tackle the resource under-utilization issue. Hence, we would like to evaluate the combined effect on the processor performance by using all these approaches at the same time.

Figure 8.1 shows an example of using all the three techniques simultaneously. In the figure 1, the processor support 8 threads but the merging hardware can support only 4 threads. Using HMT approach, Thread Select chooses instructions from 4 threads out

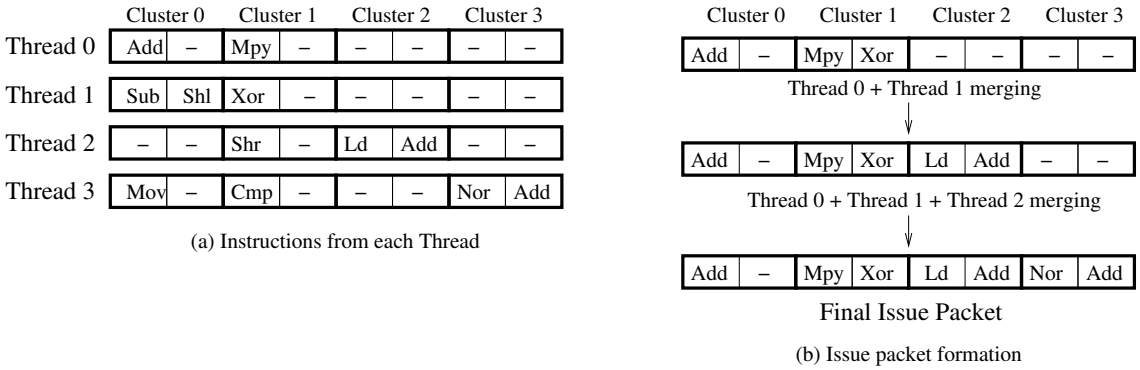


Figure 8.3: **All Merging Example**

of the 8 threads to be passed on to merging hardware. The merging hardware as shown in Figure 8.2 uses a heterogeneous merging approach where both OpSMT and CSMT merging techniques are used. The merging logic at each stage further leverages cluster-level split-issue to improve resource utilization.

All of the Hybrid MultiThreading, heterogeneous merging and split-issue approaches, themselves, have multiple possible configurations which yield different performance results. As an exhaustive evaluation of all the possible configurations of these schemes together is not feasible, we desire to obtain the peak performance achievable when all the techniques are used simultaneously as a limit study. To do so, only those configurations that in general achieve the best performance for each of the techniques are used.

- **Heterogeneous merging**: The $2SC_3$ scheme is used where first 2 threads are merged with OpSMT merging hardware and the rest are merged using CSMT merging hardware. The scheme is applicable only for a merging hardware with 4-Thread support.

- **Cluster-level split-issue**: The used scheme allows splitting of inter-cluster communication operations in Cluster-level level split-issue.

- **Hybrid MultiThreading**: EqFNB thread replacement scheme is used where first the thread with same renaming value is sought as the replacement. If that thread is blocked, then the first non-blocked thread is used as the replacement.

Figure 8.3 shows a merging example for 4-cluster 2-issue per cluster merging hardware. In the example, the first two threads are merged using OpSMT, and the rest are merged using CSMT. Cluster-level Split-Issue permits operations from non-colliding clusters to be issued. First 4 non-blocked threads T0-T3 are selected by Thread selection logic. The instructions belonging to each thread is shown in Figure 8.3(a). Figure

8.3(b) shows the issue packet generation by merging threads. First, threads T0 and T1 are merged using OpSMT. Assuming thread T0 has higher priority, operations from cluster 0 and cluster 1 of thread T0 are selected. When trying to merge Thread T1, there is a collision at cluster 0 since cluster 2 of Thread 1 has 2 operations but only 1 slot is available. Cluster 1 of thread 1 has only 1 operation, so OpSMT merging can merge this operation with 1 empty slot in cluster 1. Now, the obtained merging of Thread T0 and T1 is merged with remaining threads T2 and T3 using CSMT. Thread T2 has 1 operation in cluster 1 which cannot be merged because cluster 1 already has operations from other threads but operations of clusters 2 can be merged in the issue packet. Similarly, for thread T3 operations from clusters 0 and 1 cannot be merged but operations belonging to cluster 3 can be merged in the issue packet.

Using all the techniques together will be more complex than using each technique separately. However, the resulting complexity will be still be quite less than supporting Hybrid multithreading with a 4-Thread OpSMT merging hardware. Note that any performance gain will depend on the how well the schemes interact with each other. As all the three schemes target resource under-utilization, a fair amount of resource-underutilization is already tackled. Any further improvement depends on how much each scheme improves independently of others. This also serves as a measure of orthoganility of the schemes.

8.2 Results

This section presents the results obtained by supporting by all Hybrid MultiThreading, Cluster-level Split-Issue and Heterogeneous merging simultaneously. We also present the results when only OpSMT and CSMT are used with Hybrid MultiThreading. 8-threaded workloads described earlier in Table 2.4 in Chapter 2 are used to evaluate HMT performance. The presented results in Figures 8.4 and 8.5 evaluate the performance on both perfect-memory (PM) and real-memory (RM) configurations. We refer to the use of all the techniques together as AT in the figures. In the figures, the bars HCSMT and HOpSMT represent the performance obtained using the Hybrid MultiThreading alone with CSMT and OpSMT merging hardware.

In general, AT gets a performance which is in middle of 4-Thread HOpSMT and 4-Thread HCSMT only configurations. For perfect memory, AT improves performance by around 18% on average when compared to HCSMT configuration. While the performance

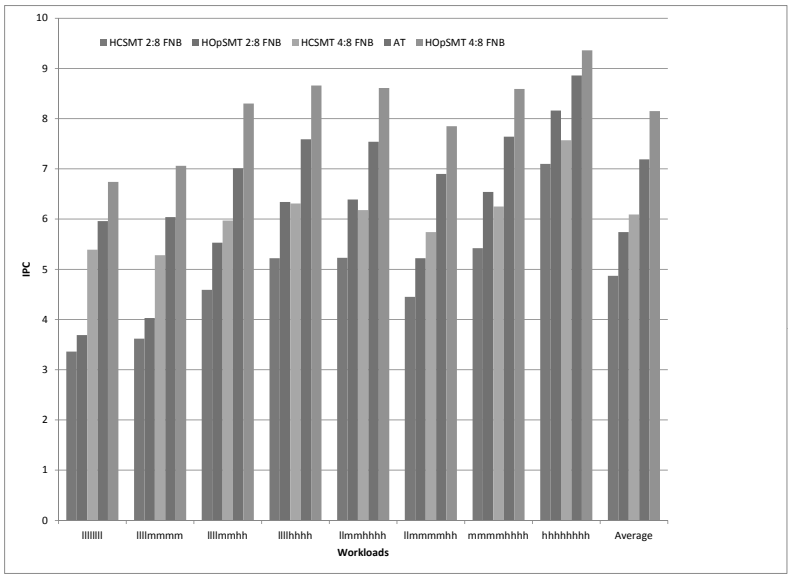


Figure 8.4: **Perfect memory Performance**

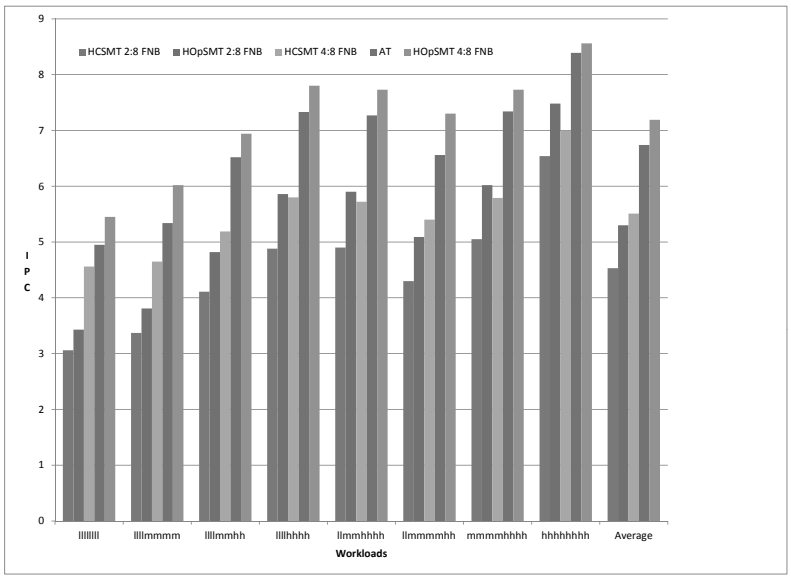


Figure 8.5: **Real memory Performance**

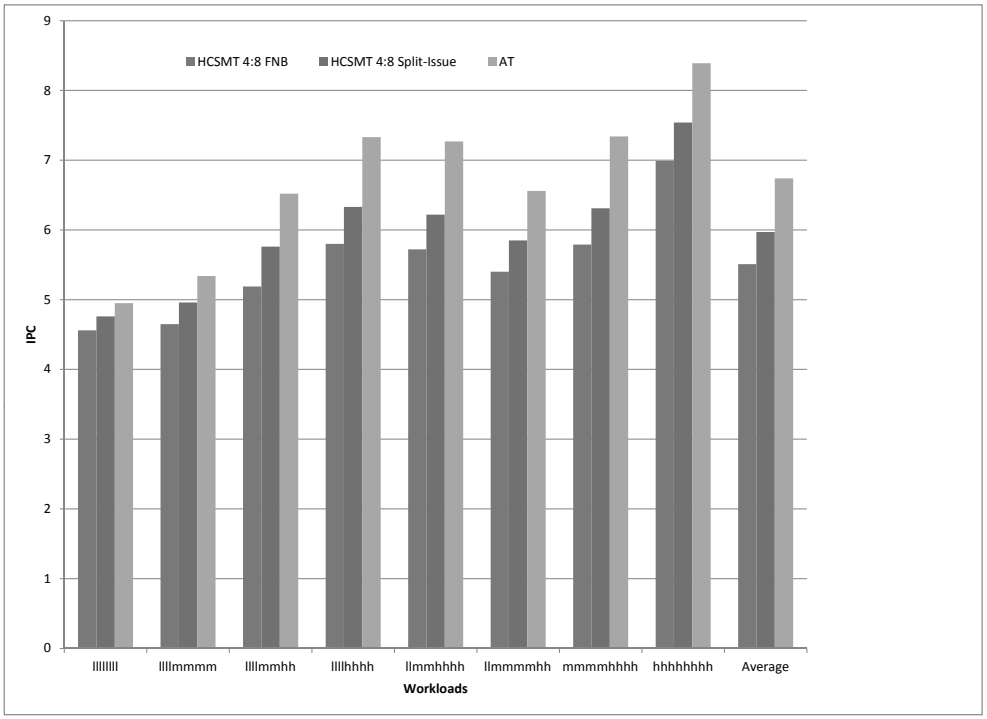


Figure 8.6: **Comparison with HCSMT with split-issue, Perfect Memory**

difference between 4-Thread HOpSMT and AT is around 13%.

When real memory is considered, the performance improvement over 4-thread HCSMT configuration increases to 22%. Also, the performance difference between AT and the more complex 4-Thread HOpSMT configuration reduces to only 6.7%. It is evident that the performance improvement using all the three techniques is quite significant and that leveraging the techniques orthogonality does indeed help in further improving multithreading performance. Improved performance makes AT a very attractive choice for implementation.

We also experimented with just enabling split-issue for the 2-Thread and 4-Thread HOpSMT and HCSMT configurations. This was done to decide whether the improvement is because of split-issue or heterogeneous merging The results presented in Figures 8.6 and 8.7 show the results obtained by enabling split-issue on HCSMT on both perfect and real memory configurations. The results show that while split-issue does improve the performance, a major chunk of improvement comes from heterogeneous merging.

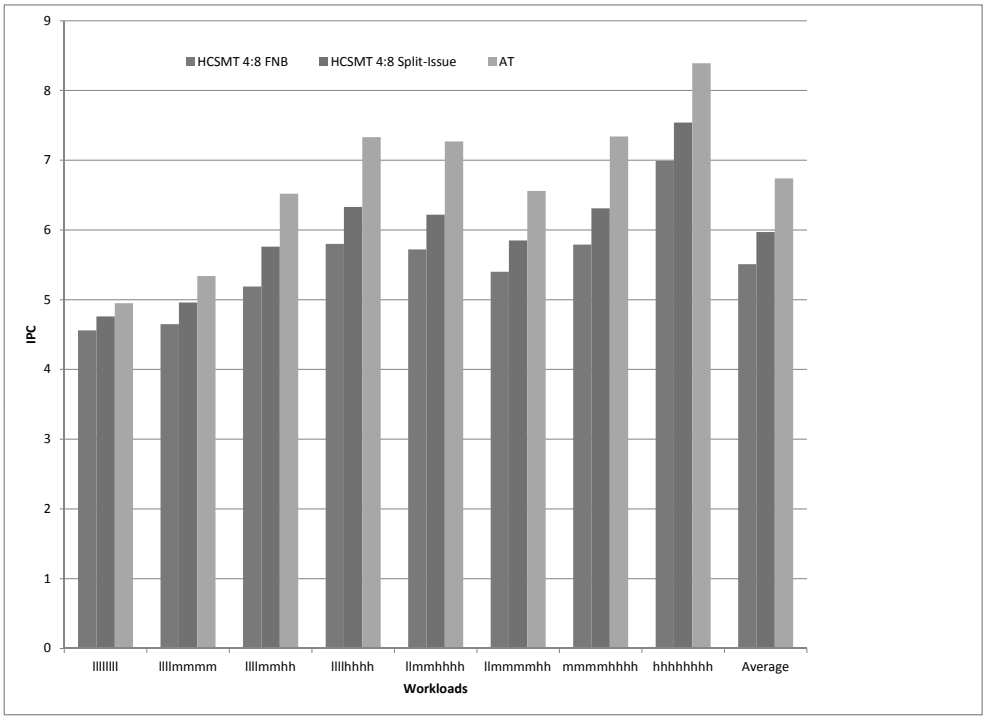


Figure 8.7: **Comparison with HCSMT with split-issue, Real Memory**

Hence, split-issue alone is not sufficient to improve the performance but both split-issue and heterogenous merging working together is important to achieve high performance.

8.3 Summary and Conclusions

This thesis has described several techniques to enhance multithreading performance on SMT clustered VLIW processors. All the described techniques, namely Hybrid Multi-Threading, Cluster-level Split-Issue and Heterogeneous merging are orthogonal to each other. Hence there is a unique opportunity to analyze the processor performance when all these techniques are applied at the same time. This chapter has presented an analysis of simultaneous application of all the described techniques.

The results indicate that using all the techniques together boosts the multithreading significantly. On average, usage of all the techniques result in an improvement of 22% over HCSMT. Besides the performance is close to the more complex HOpSMT configurations (HOpSMT is only 6.7% better). The results prove that the low overhead schemes put together can achieve performance close to more complex schemes which can save both power and chip area for a given performance target.

Chapter 9

Conclusions and Future Work

In this thesis we have presented CSMT, a new Simultaneous MultiThreading approach for clustered VLIW processors. CSMT considers the set of operations that execute simultaneously in a given cluster (named bundle) as the assignment unit. All bundles belonging to a VLIW instruction from a given thread are simultaneously issued.

An analysis of the hardware required to implement CSMT shows that it is cheap, practical and realistic to implement. Depending on the target frequency of the processor, thread merge may be implemented in the instruction decode stage if delay is small enough. A 4-thread 4-cluster machine has been evaluated in our work, since it has a neglegible area and delay penalty. However, assuming that only a single extra pipeline stage is acceptable (performance degradation is very small) and a limit of 10000 transistors, upto an 8-thread 4-cluster machine can be supported.

The analysis performed on a set of benchmarks using the Lx architecture [12] shows that in general, no cluster is used during a significant amount of time due (mostly) to cache misses. Moreover, low ILP applications use only a few clusters most of the time. The compiler assigns operations mainly to the first clusters and tries to reduce the number of used clusters in order to reduce communication overhead among different clusters. As a consequence, the assignment of clusters collides when several threads are simultaneously executed. We have presented a mechanism to avoid this problem and allow a more parallel execution of the threads by renaming, at execution time, the clusters previously assigned by the compiler. The renaming mechanism has a very low hardware complexity.

The results show that CSMT makes a better use of clusters than interleaved multithreading (IMT) with a negligible hardware cost. In terms of performance, CSMT significantly outperforms IMT. For instance, for a 4-cluster machine with 4 threads, CSMT shows an average speedup of 77% over a single threaded machine and 52% over IMT assuming no cache misses, which shows the ability of CSMT to remove horizontal waste.

When a realistic memory system is considered, the speedup of CSMT over single thread increases to 110% while speedup over IMT gets limited to 40%. This is because most of the resources wasted are due to stalls caused by cache misses, and IMT already does a good job hiding memory latency. However, CSMT still has a very significant advantage over IMT due to its ability to remove both vertical and horizontal waste. Compared to an OpSMT machine (operation-level merging), CSMT is within 7% and 19% of the SMT performance for a 2-thread and 4-thread machine respectively, thus bridging the gap between IMT and OpSMT performance.

The thesis also presented several orthogonal approaches to further improve CSMT performance.

- **Heterogeneous Merging Hardware:**
  Heterogeneous merging combines both CSMT and OpSMT merging hardware where some threads are merged using OpSMT way while rest are merged using CSMT. This keeps the merging hardware low but improves the overall performance. One of the schemes that merges 2 threads using OpSMT and rest using CSMT achieves 14% higher performance than a 2-Thread OpSMT and is within 11% of a 4-Thread OpSMT architecture.

- **Cluster-level Split-Issue:**
  If two VLIW instructions have a resource conflict, only one of them can be issued. This is because all operations in a VLIW instruction must be issued in a lock step mode i.e. all operations must be issued at the same cycle to preserve the semantics of the program. This limits the number of instructions that can be issued at a given cycle and restricts the performance improvement potential of both OpSMT and CSMT. Split-issue at operation-level removes the lock step issue restriction and allows issuing operations of same VLIW instruction separately at different cycles. However, the implementation at operation-level requires complex dynamic scheduling hardware. The proposal Cluster-level split-issue presented in this thesis allows splitting a VLIW instruction only at a cluster boundary and does not allow splitting individual operations in a cluster. Cluster-level split-issue is simple to implement and has a low hardware cost. Besides, the performance achieved by cluster-level split-issue is close to the more complex operation-level split-issue. When instructions are merged at cluster-level, a performance improvement 8.7% is obtained over a 2-thread and 7.5% over a 4-thread processor (operation-level split-issue is not ap-

plicable for cluster-level merging). With instruction merging at operation-level (traditional OpSMT), a performance improvement of 9.8% is obtained over a 2-thread and 9.4% over a 4-thread OpSMT processor.

- **Hybrid MultiThreading:**
  Interleaved MultiThreading (IMT) provides significant performance improvements if the number of threads supported is small because of its ability to reduce vertical waste. With a larger number of threads, less opportunities exist for removing vertical waste, resulting in only marginal performance improvements with IMT. On the other hand, for SMT based techniques like CSMT and OpSMT performance still keeps on improving significantly because of its ability to also reduce horizontal waste by merging instructions from different threads when number of threads are increased. However, in both CSMT and OpSMT, only a small number of threads can be supported because of the complexity of merging hardware. In contrast, IMT does not require a merging hardware and can support a larger number of threads.

  The presented approach, Hybrid MultiThreading (HMT) supports a larger number of threads than CSMT or OpSMT with a given merging hardware cost. This is achieved by merging only a subset of threads at a time. Every cycle, a new subset of threads is selected. HMT is independent of the merging scheme used by the merging hardware and be used with both OpSMT and CSMT. Using HMT with a 4-thread merging hardware achieves a performance similar to an 8-thread merging hardware without having to incur the cost of 8-thread merging hardware. Even with the simplest static thread selection, there is a performance improvement of 11% over OpSMT. While with a more complex thread selection scheme like Eq+FNB, using HMT improves performance by 29% over OpSMT.

9.1 Future Work

This thesis has focused on multiThreading for clustered VLIW processors with certain assumptions. E.g. the execution latency model is less-than-or-equal (LEQ) where the latency of an operation can be less than the specified latency. Some processors use equals model (EQ) where the latency of an operation is exactly same as specified. Also, the inter-cluster communication network in the thesis uses copy operations. Several commercial VLIW processors use different models. A wide taxanomy of the different communication

models can be found in [53]. The work reported in the thesis can be used as a base for exploring multiThreading implementations on clustered VLIW architectures with different latency model (LEQ vs EQ) and intercluster communication networks.

Bibliography

[1] Colorspace Conversion Program Used in High Performance Printers, Personal Communication.

[2] Advanced Micro Devices Inc. R600-Family Instruction Set Architecture. *http://developer.amd.com/gpu_assets/R600_Instruction_Set_Architecture.pdf*, January, 2009.

[3] A. Agarwal, J. Babb, D. Chaiken, G. D'Souza, K. L. Johnson, D. A. Kranz, J. Kubiatowicz, B.-H. Lim, G. Maa, K. Mackenzie, D. Nussbaum, M. Parkin, and D. Yeung. Sparcle: A Multithreaded VLSI Processor for Parallel Computing. In *Parallel Symbolic Computing*, 1992.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. J. Smith. The Tera computer system. In *ICS*, 1990.

[5] D. Barretta, W. Fornaciari, M. Sami, and D. Bagni. Multithreaded Extension to Multicluster VLIW Processors for Embedded Applications. In *DATE*, 2005.

[6] J. Borkenhagen, R. Eickemeyer, R. Kalla, and S. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development*, 44(6), 2000.

[7] J. Codina, J. Llosa, and A. González. A comparative study of modulo scheduling techniques. In *Proceedings of the 16th international conference on Supercomputing*, pages 97–106. ACM, 2002.

[8] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. In *ASPLOS-II*, 1987.

[9] M. Drozdowski. Scheduling multiprocessor tasksŮan overview. *European Journal of Operational Research*, 94(2), 1996.

[10] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, B.-H. Lim, M. S. Squillante, and C.-F. E. Wu. Evaluation of Multithreaded Processors and Thread-Switch Policies. In *ISHPC*, 1997.

[11] J. R. Ellis. *Bulldog: a compiler for VLSI architectures*. MIT Press, Cambridge, MA, USA, 1986.

[12] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *ISCA*, 2000.

[13] M. Farrens and A. Pleszkun. Strategies for achieving improved processor througput. In *ISCA*, pages 362–369, 1991.

[14] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee. The M-Machine multicomputer. In *MICRO*, pages 146–156, 1995.

[15] J. Fisher, P. Faraboschi, C. Young, and R. Colwell. 10 VLIW Processors: From Blue Sky to Best Buy. *IEEE Solid State Circuit Magazine*, 1(2), 2009.

[16] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Computers*, 30(7), 1981.

[17] A. Gangwar, M. Balakrishnan, and A. Kumar. Impact of intercluster communication mechanisms on ILP in clustered VLIW architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(1), 2007.

[18] M. Gupta, F. Sánchez, and J. Llosa. Cluster Level Multithreading for VLIW Processors. In *ACACES*, 2006.

[19] M. Gupta, F. Sánchez, and J. Llosa. Cluster-Level Simultaneous MultiThreading for VLIW Processors. In *ICCD*, 2007.

[20] M. Gupta, F. Sánchez, and J. Llosa. Merge Logic for Clustered Multithreaded VLIW Processors. In *EUROMICRO Conference on Digital System Design*, 2007.

[21] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[22] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, 33(7), 2000.

[23] F. Homewood and P. Faraboschi. ST200: A VLIW Architecture for Media-Oriented Applications. *Microprocessor Forum*, 2000.

[24] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 Architecture. *IEEE Micro*, 20(5), 2000.

[25] Inverse discrete cosine transform, taken from ffmpeg. In *http://www.ffmpeg.org*.

[26] B. Iyer, S. Srinivasan, and B. L. Jacob. Extended Split-Issue: Enabling Flexibility in the Hardware Implementation of NUAL VLIW DSPs. In *ISCA*, 2004.

[27] R. Jr. and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *ISCA*, pages 443–451, 1988.

[28] R. Kalla, B. Sinharoy, and J. Tendler. SMT implementation in POWER 5. *Hot Chips*, 15, 2003.

[29] S. Kaxiras, G. Narlikar, A. Berenbaum, and Z. Hu. Comparing power consumption of an SMT and a CMP DSP for mobile phone workloads. In *CASES*. ACM Press New York, NY, USA, 2001.

[30] S. Keckler and W. Dally. Processor coupling: integrating compile time and runtime scheduling for parallelism. In *ISCA*, pages 202–213, 1992.

[31] D. Koufaty and D. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2), 2003.

[32] R. Kumar, N. Jouppi, and D. Tullsen. Conjoined-Core Chip Multiprocessing. In *MICRO*, 2004.

[33] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, volume 23, 1988.

[34] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *MICRO*, 1997.

[35] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing module scheduling: a lifetime-sensitive approach. In *Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on*, pages 80–86. IEEE, 1996.

[36] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1-2), 1993.

[37] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative Multithreaded Processors. In *ICS*, pages 77–84, 1998.

[38] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2), 2005.

[39] A. Mikschl and W. Damm. MSparc: A Multithreaded Sparc. In *Euro-Par, Vol. II*, 1996.

[40] T. Olson. Mali 400 MP: A Scalable GPU for Mobile and Embedded Devices. In *High-Performance Graphics*, 2010.

[41] E. Ozer and T. Conte. High-performance and low-cost dual-thread VLIW processor using Weld architecture paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 16(12), 2005.

[42] E. Ozer, T. Conte, and S. Sharma. Weld: A Multithreading Technique Towards Latency-Tolerant VLIW Processors. In *HiPC*, pages 192–203, 2001.

[43] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *ISCA*, pages 206–218, 1997.

[44] B. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74. ACM, 1994.

[45] B. R. Rau. Dynamically Scheduled VLIW Processors. In *MICRO*, 1993.

[46] B. R. Rau, D. W. L. Yen, W. C. Yen, and R. A. Towle. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs. *IEEE Computer*, 22(1), 1989.

[47] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens. Register Organization for Media Processing. In *HPCA*, 2000.

[48] J. Sánchez and A. González. Modulo scheduling for a fully-distributed clustered vliw architecture. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 124–133. ACM, 2000.

[49] N. Seshan. High VelociTI Processing. *IEEE Signal Processing Magazine*, 15(2), March 1998.

[50] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *SPIE*, pages 241–248, 1981.

[51] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *ISCA*, pages 414–425, 1995.

[52] StarCore DSPs. http://www.freescale.com.

[53] A. Terechko, E. Thenaff, M. Garg, J. Van Eijndhoven, and H. Corporaal. Inter-cluster communication models for clustered VLIW processors. In *9th International Symposium on High Performance Computer Architecture, Anaheim, California*, pages 298–309, 2003.

[54] M. R. Thistle and B. J. Smith. A Processor Architecture for Horizon. In *SC*, pages 35–41, 1988.

[55] TigerSharc Embedded Processors. www.analog.com.

[56] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC Architecture: A Synthesis of Parallelism and Scalability. *IEEE Micro*, 20(6), 2000.

[57] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA*, 1995.

[58] T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Computing Surveys (CSUR)*, 35(1), 2003.

[59] J.-W. van de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, D. Amirtharaj, K. Kalra, P. Rodriguez, and H. van Antwerpen. The TM3270 Media-Processor. In *MICRO*, 2005.

[60] J. Vera, F. Cazorla, A. Pajuelo, O. Santana, E. Fernandez, and M. Valero. FAME: FAirly MEasuring Multithreaded Architectures. In *PACT*, 2007.

[61] VEX Toolchain. http://www.hpl.hp.com/downloads/vex/.

[62] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. In *ISCA*, 1989.

[63] A. Wolfe and J. Shen. A Variable Instruction Stream Extension to the VLIW Architecture. In *ASPLOS*, 1991.

[64] x264 - a free h264/avc encoder. In *http://www.videolan.org /developers/x264.html*.

[65] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Modulo scheduling with integrated register spilling for clustered vliw architectures. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 160–169. IEEE Computer Society, 2001.

Appendix A
CSMT Cost Analysis

Cluster-level Simultaneous MultiThreading (CSMT) forms the basis of this thesis. This chapter discusses the hardware implementation of thread merging hardware for CSMT in detail. In particular, we discuss two different implementations of the merging hardware that have quite different area and delay characteristics.

A.1 Introduction

CSMT issues instructions from multiple threads only if the threads use different clusters. The cluster usage information is in the form of a Cluster Usage Vector (CUV). A CUV is a C bit-wide array, where C is the number of clusters. Every set bit in CUV indicates the presence of operations in that cluster. The entries of CUV are ordered in the increasing order of the clusters, with MSB denoting cluster 0 and LSB cluster $C - 1$. E.g. a vector 1010, as shown in Figure A.1, means that clusters 0 and 2 are used but clusters 1 and 3 have nops. The CUV of a thread is '0000' when the thread is blocked by an event (e.g. a cache miss). The CUV of each thread is computed at the beginning of the thread merge. The CUVs of the threads are passed on to the thread merge hardware. Thread merge hardware analyzes the conflicts between the different instructions and generates an execution packet.

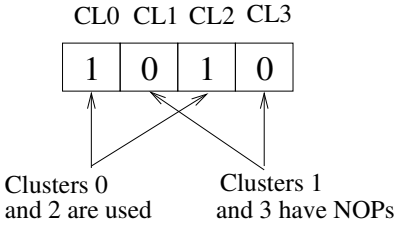


Figure A.1: **Example of Cluster Usage Vector 1010**

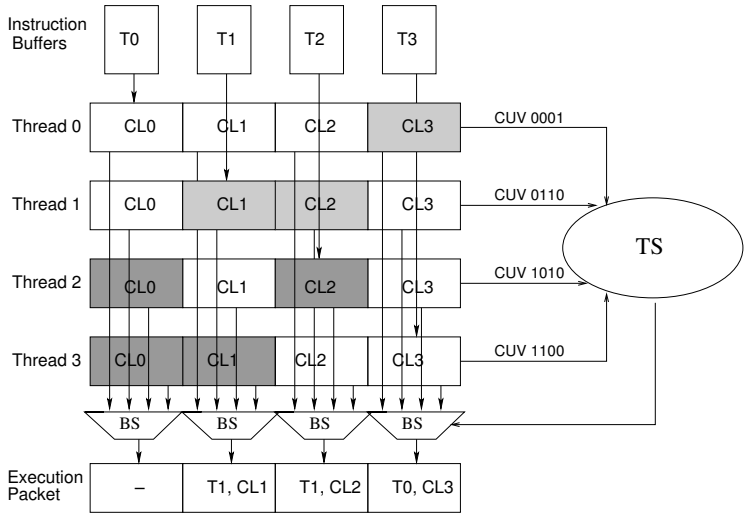


Figure A.2: **Thread merge hardware**

A.2 Thread Merge Hardware

Figure A.2 shows the thread merge hardware used to merge instructions from different threads. In the figure, Ti refers to Thread i, and CLi refers to Cluster i. This hardware consists of two parts: *Thread Select Logic* (TS) and *Bundle Select Logic* (BS). For each cluster, BS selects a bundle from bundles of different threads, and TS controls this selection.

TS receives as input the CUV for each thread and the thread numbers sorted according to the thread priorities. TS generates the appropriate signals to merge the bundles from different threads to form the execution packet. The example in the Figure A.2 depicts a scenario where Thread 1 has the highest priority, and threads 2, 3 and 0 have the next highest priorities respectively. TS detects that threads 2 and 3 cannot be scheduled, as there is a collision with the higher priority Thread 1, but Thread 0 can be scheduled as it does not have a collision. So, TS generates appropriate signals for BS to merge threads 0 and 1, and BS selects bundle CL3 from Thread 0 and bundles CL1 and CL2 from Thread 1 (light shaded) while bundles from threads 2 and 3 are not selected (dark shaded) (see Figure A.2).

TS block generates the signals to select the appropriate threads in the execution packet. BS block takes the thread selections provided by TS block and merge the corresponding bundles in the execution packet. To do so, BS also checks the cluster usage bit of each respective thread. The cluster usage bit indicates whether the cluster contains operations

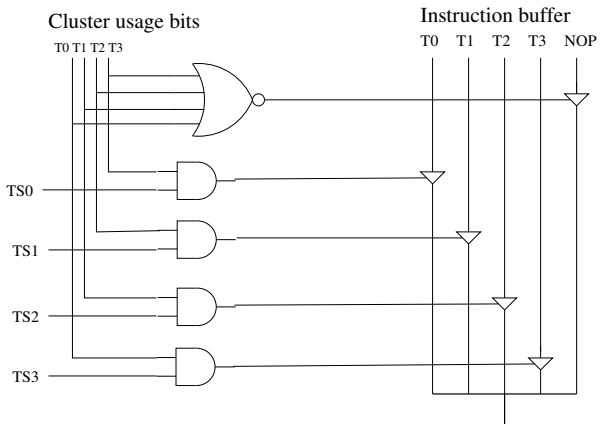


Figure A.3: **Bundle select logic (BS)**

or not. When both, thread select bit and cluster usage bit, are set for a bundle, that bundle is selected to be included in the final execution packet. When none of the threads use a given cluster, a bundle containing a nop is injected in the execution packet for that cluster. The BS for a cluster in a 4-thread machine is shown in Figure A.3. Since BS operates on a per cluster basis, C copies of BS block are required for a C-cluster architecture.

TS logic is further divided into three different blocks, named Priority Encode (PE), Thread Select Bits Computation (TSBC) and Priority Decode (PD), as shown in Figure A.4. These blocks are described in the following sections.

A.2.1 Priority Encode

Priority Encode (PE) block sorts the CUVs of the threads according to the priorities of the threads in order to provide a sorted CUV sequence to TSBC block. It is implemented by using a set of multiplexers (same as number of threads), each with the CUV of all threads as input and a thread number (computed by thread scheduling policy) used as the mux select. Each multiplexer selects, as output, the CUV of the thread identified by the thread number that controls the mux. P-0 is the thread number of the highest priority thread, and P-1, P-2 and P-3 represent respectively the thread number of next higher priority threads. So, if Thread 2 is the highest priority thread and a simple sequential priority scheme is used for scheduling the remaining threads, then P-0 = 2, P-1 = 3, P-2 = 0 and P-3 = 1. P-0 to P-3 signals are generated by the thread scheduling policy which decides the priorities of the threads.

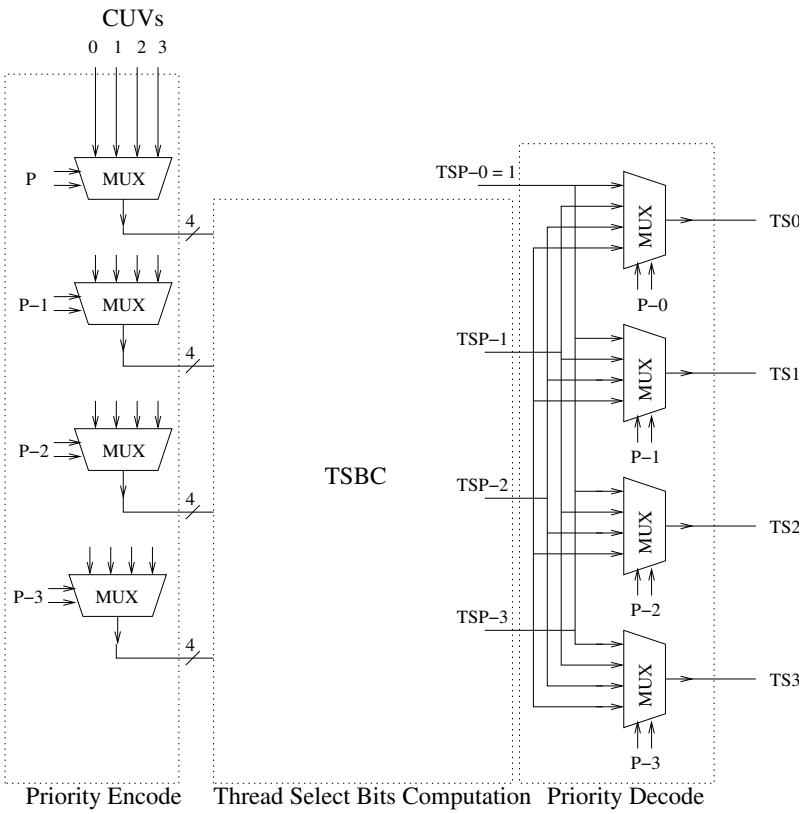


Figure A.4: **Thread select logic (TS)**

A.2.2 Priority Decode

Priority Decode (PD) block reroutes the thread select bits generated by TSBC block, which are sorted by thread priorities, to the corresponding thread select bits. So, if Thread 2 is the highest priority thread, TSP-0 is selected to appear as TS2, TSP-1 as TS3, TSP-2 as TS0 and TSP-3 as TS1. When TSi is '1', Thread i is selected to be merged in the execution packet.

PE and PD blocks are required for a proper combination of threads in the execution packet according to their priority, and are common to any dynamic priority scheme. These blocks are not required if a static priority scheme is used. Note that priority computation is decoupled from the PE and PD logic and, therefore, these blocks can be used with any scheduling policy without any change.

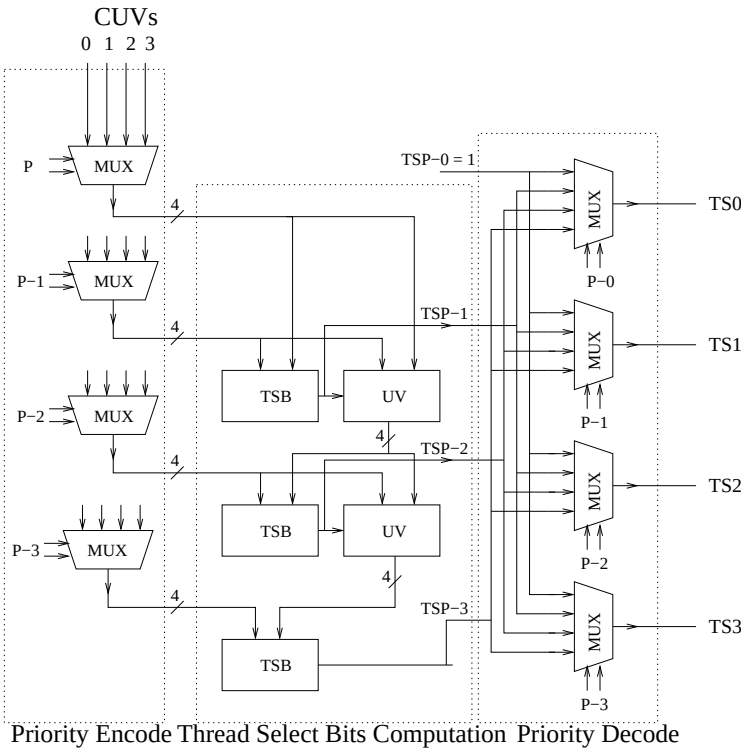


Figure A.5: **Serial logic implementation for TS block**

A.2.3 Thread Select Bits Computation

Thread Select Bits Computation (TSBC) block explores the CUV of the threads (already sorted according to the thread priority) and generates a thread select bit (TSP-0 to TSP-3) for each thread. The thread select bit indicates whether the thread is selected to be included in the execution packet or not. A thread is selected to be merged in the execution packet if no conflicts exist with previous (higher priority) threads already selected. TSP-0 is the thread select bit for highest priority thread, TSP-1 the thread select bit for the next higher priority thread, and so on. Note that the highest priority thread is always selected (TSP-0 = 1).

Next, we present two different implementations for the TSBC block:

Serial Logic is a cascading logic which are repeated according to number of threads and clusters.

Parallel Logic computes all possible combinations and selects one conforming to the thread selection policy.

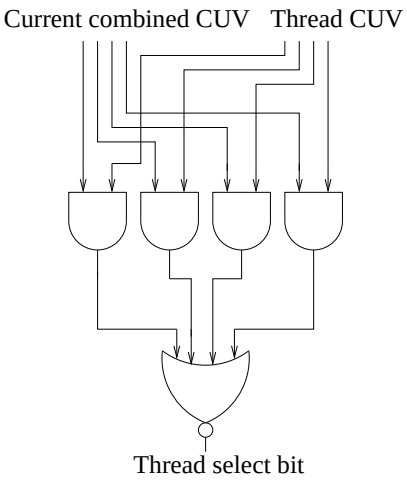


Figure A.6: **Thread select bit block (TSB)**

A.3 Serial Logic

Serial logic is a cascading logic which explores a different thread at each level and computes whether the thread can be included in the final execution packet or not. This logic is made of blocks which are repeated according to the number of threads and clusters. Figure A.5 shows an implementation for a 4-thread 4-cluster architecture.

First, the CUV of the thread with highest priority (P-0) is selected as the initial combined CUV (CUV of the execution packet). Then, at each level, the current combined CUV is compared to the CUV of next highest priority thread. The thread select bit for the explored thread is set to '1' by block TSB (Thread Select Bit logic) when no collision exists between both CUVs. Block UV (Usage Vector computation logic) uses the obtained thread select bit to compute a new combined CUV for the next level. This process is repeated till a thread select bit is obtained for each thread.

Note that the block UV is not required at the last level, since computing the final combined CUV is not required. Also, note that the thread select bit for Thread P-0 (TSP-0) is always '1', since the highest priority thread is always selected to be in the execution packet.

For N threads, $N-1$ TSB blocks and $N-2$ UV blocks are required. The two blocks TSB and UV have a low hardware complexity, as we shall inspect in following sections.

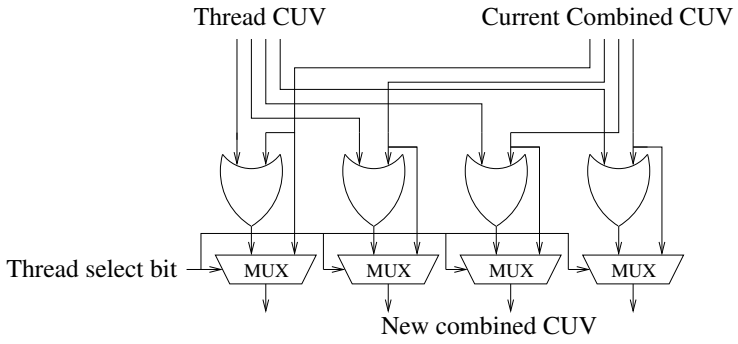


Figure A.7: **Usage vector computation block (UV)**

A.3.1 Thread Select Bit block

Thread Select Bit (TSB) block computes the thread selection bit for the current thread by computing the collision of its CUV with the current combined CUV. A collision exists when the thread uses a cluster that is already in use in the combined CUV. For instance, Thread 1 and Thread 2 in Figure A.2 have CUVs '0110' and '1010' respectively, and there is a collision at cluster 2.

Figure A.6 shows the TSB implementation for a 4-cluster architecture. Each AND gate receives two bits which indicate the usage information of a given cluster from the current combined CUV and current thread CUV. If both bits are set for any AND gate, a collision exists. The thread is selected for merging only when there is no collision for any cluster. This logic can be easily extended to check collision for any number of clusters. With C clusters, C 2-input AND gates and 1 C-input NOR gate are required.

A.3.2 Usage Vector Computation block

Usage Vector Computation (UV) block computes the combined CUV for the next level. If the thread select bit computed by TSB is set, then the combined CUV for the next level is obtained by merging current combined CUV with the CUV of the thread under consideration; otherwise, the current combined CUV is passed to the next level. As a set bit in the CUV means cluster occupation, the merging is a bit-wise OR of both CUVs. The logic requires C 2-input OR gates and C 2-input multiplexers for C clusters. Figure A.7 shows the implementation for a 4-cluster architecture.

Table A.1: **Thread select bits for thread combinations**

Thread combination	Thread select bits
	TSP-0=1, TSP-1 TSP-2 TSP-3
0 1 2 3	**1, 1 1 1**
0 1 2	**1, 1 1** 0
0 1 3	**1, 1** 0 **1**
0 1	**1, 1** 0 0
0 2 3	**1**, 0 **1 1**
0 2	**1**, 0 **1** 0
0 3	**1**, 0 0 **1**
0	**1**, 0 0 0

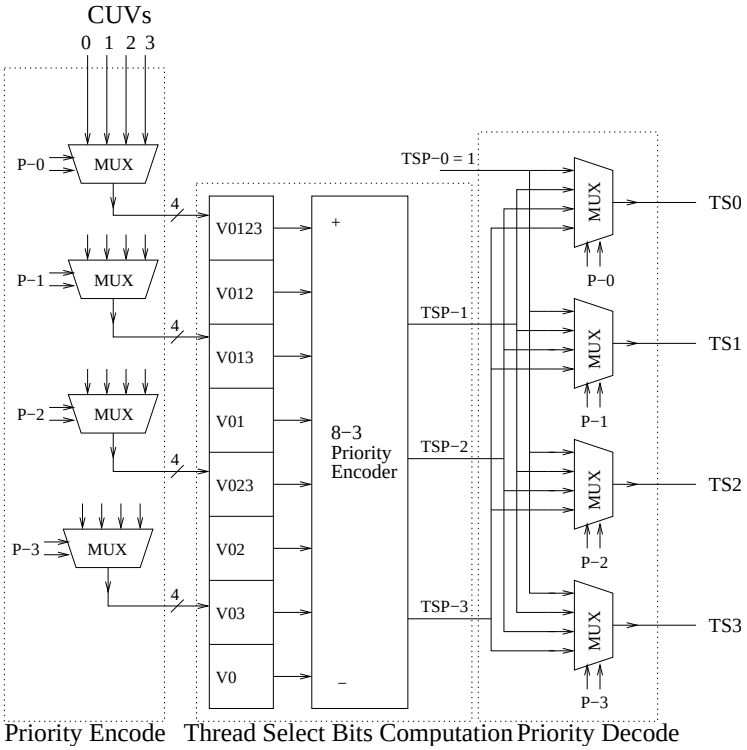


Figure A.8: **Parallel logic implementation for TS block**

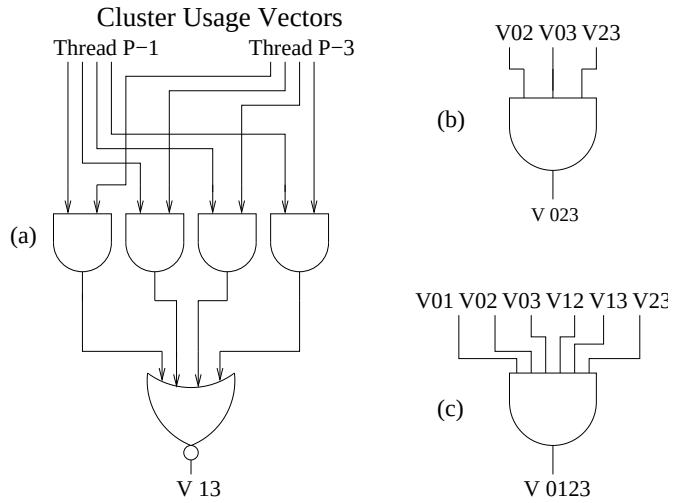


Figure A.9: **Validity computations for three thread combinations**

A.4 Parallel Logic

We propose, in this section, an alternate implementation for TSBC block. The proposed logic compute all possible thread combinations in parallel and selects one amongst the possible combinations.

Table A.1 lists all the possible thread combinations[1] that can be generated by the thread merge logic and the corresponding thread select bits for a 4-thread architecture. The thread combinations shown in Table A.1 are sorted according to the thread merge policy. For instance, the first thread combination is 0123, since the thread merge policy will merge all threads if there are no collisions among them. Next thread combination is 012 followed by 013, since Thread 2 is explored before Thread 3. Therefore, if merging threads 012 is possible, then thread merge logic does so despite merging 013 is also possible. As a last example, combination 01 appears before 023 since thread merge logic will try to merge Thread 1 before exploring threads 2 and 3. Notice that, since the highest priority thread (Thread 0) will always be merged, all thread combinations include Thread 0.

A thread combination is said to be valid if the threads involved can be merged without producing any cluster conflicts (i.e. there are no conflicts among the CUVs of the threads). Parallel logic checks in parallel the validity of all thread combinations. Amongst all the

[1]For brevity, threads P-0 (highest priority), P-1 (next highest), etc. are labeled as 0, 1 and so on respectively in the thread combinations

valid thread combinations, parallel logic selects the one conforming to the thread merge policy and generates the thread select bits for it. Note that, for all the thread combinations shown in Table A.1, their corresponding thread select bits (excluding TSP-0, which is always 1) is a binary countdown. Hence, to compute the thread select bits once the validities of all the thread combinations have been computed, a standard priority encoder is used, with validities of the thread combinations as its input. Using a priority encoder also takes care of the cases where more than one thread combination is valid. For instance, whenever thread combination 0123 is valid, thread combinations 012, 013 etc. are also valid, but only 0123 is selected. Figure A.8 shows the parallel logic implementation for a 4-thread 4-cluster machine. For a N-thread C-cluster machine, 2^{N-1} validity computations and a 2^{N-1} to $N-1$ priority encoder are required in the parallel logic based implementation.

To compute the validities for the thread combinations, first the validities of all 2-thread combinations are computed in parallel. For a thread combination with more than two threads to be valid, all permutations of 2-thread combinations of its constituting threads should be valid. For instance, a valid thread combination 0123 requires that all the 2-thread combinations 01, 02, 03, 12, 13 and 23 are valid. Similarly, for the thread combination 023 to be valid, thread combinations 02, 03, 23 must all be valid. For a N-thread machine, $\binom{N}{2}$ 2-thread validity computations are required for the thread combination which includes all threads (thread combination $012..N-1$). To illustrate the logic required to compute a validity, the hardware for validity functions for thread combinations 13, 023 and 0123 is shown in figures A.9(a), (b) and (c) respectively (assuming a 4-cluster architecture). As can be seen, validities for combinations of three or more threads are computed by using validities of 2-thread combinations. The validity bits for other thread combinations are computed in a similar way.

A.5 Cost Analysis

The total delay in the thread merge hardware, measured in terms of gate delay, is the total sum of delays in the critical path of the Priority Encode/Decode (PE, PD), Bundle Select (BSL) and Thread Select Bit Computation (TSBC) blocks. Assuming the maximum fan-in of a single gate to be 4, a n-input gate has a delay of $\lceil log_4 n \rceil$. A $k-$input multiplexer requires a $log_2 k$ to k decoder and has a delay of $\lceil log_4 k \rceil$. Assuming a C-Cluster N-Thread architecture and the maximum fan-in of a single gate to be 4, Hence, for a N-thread C-

cluster architecture, the blocks PE, PD and BSL have gate delays of $\lceil log_4 N \rceil$, $\lceil log_4 N \rceil$ and 1 respectively.

For a CMOS gate, number of transistors required is double of the number of inputs. A $k-$input multiplexer requires a $log_2 k$ to k decoder, therefore estimated transistor count for a multiplexer is $2k \times logk + k \times (input\text{-}width)$, where the first term accounts for the decoder and the second term accounts for the pass transistors per output line of decoder. Therefore, for PE and PD blocks which use N $N\text{-}input$ multiplexers, with the input-width of C and 1 respectively, the respective transistor count is:
$T(PE) = N \times (2Nlog(N) + CN)$
$T(PD) = N \times (2Nlog(N) + N)$

A.5.1 Serial Logic

For a fair comparison between the different multithreading schemes, the cost of bundle select block (BS) is excluded because a logic similar to BS is required for most multithreading schemes including IMT, CSMT and OpSMT.

A.5.1.1 Delay

The TSBC block in the serial logic design requires $(N-1)$ TSB and $(N-2)$ UV blocks arranged in a cascade. The delay for TSB block is $1 + \lceil log_4 C \rceil$, where the first term accounts for the AND gates and second for the $C-$input NOR gate. Similarly, UV block has a delay of 2 gates, but contribute only 1 gate delay to the critical path, as the bitwise OR of the combined CUV and the CUV of the thread is done in parallel with the computation of the thread select bit in TSB block.

Therefore, Serial logic has a delay of $(N-1)(2 + \lceil log_4 C \rceil) + 2\lceil log_4 N \rceil$, which is $\mathrm{O}(NlogC)$.

A.5.1.2 Transistor Count

Serial logic design requires $(N-1)$ TSB and $(N-2)$ UV blocks. Each TSB block requires C 2-input AND gates and 1 C-input NOR gate, and, UV block requires C 2-input OR gates and C 2-input multiplexers. Therefore,
$T(TSB) = 4C + 2C = 6C$
$T(UV) = C(4 + 4 + 2) = 10C$

$T(TSBC) = (N-1)T(TSB) + (N-2)T(UV)$

Hence, the total transistor count in serial logic is:
$T(SL) = 16CN - 26C + 4N^2log(N) + CN^2 + N^2 = \mathbf{O}(CN^2 + N^2log(N))$. Assuming a fixed number of threads, $T(SL) = \mathbf{O}(C)$, and for a fixed number of clusters, $T(SL) = \mathbf{O}(N^2log(N))$.

A.5.2 Parallel Logic

The delay for TSBC in parallel logic based design is the sum of delay of priority encoder and the critical delay in validity computation. The critical delay in validity computation is the delay for the thread combination $012..N-1$, i.e. the combination with highest number of threads. Computing validity of this thread combination requires $\binom{N}{2}$ inputs (all possible 2-Thread combinations). The validity computation of thread combination with 2 threads has a delay of 1 + $\lceil log_4 C \rceil$. Hence, the critical delay in the validity computation is 1 + $\lceil log_4 C \rceil + \lceil log_4(N(N-1)/2) \rceil$.

The delay for priority encoder is $log_4 2^{N-1}$ or $\lceil N/2 \rceil - 1$. The parallel logic based design, therefore, has a delay of $1 + \lceil N/2 \rceil + 2\lceil log_4 N \rceil + \lceil log_4 C \rceil + \lceil log_4(N(N-1)/2) \rceil$, which is $\mathbf{O}(N + logC)$.

A.5.2.1 Transistor Count

We also try to broadly estimate the transistors required to implement the thread merge hardware with CMOS technology assuming a C-Cluster N-thread architecture with a issue-width of W per cluster.

For parallel logic based implementation of TSBC logic, the number of transistors is the sum of transistors used in validity computations of thread groups and the priority encoder. Therefore,
$T(TSBC) = T(Validities) + T(PrEnc)$
$T(PrEnc) = 2^{N-1}(N-1)$
For a validity computation of a thread combination with k-threads ($k > 2$), $\binom{k}{2}$ 2-thread validity inputs are required. Hence the number of transistors required is $2\binom{k}{2}$ or $k(k-1)$. For a N-thread machine, there are $\binom{N-1}{0}$ thread combinations with N threads, $\binom{N-1}{1}$ with $N-1$ threads and so on. Hence, the number of transistors required for all validity

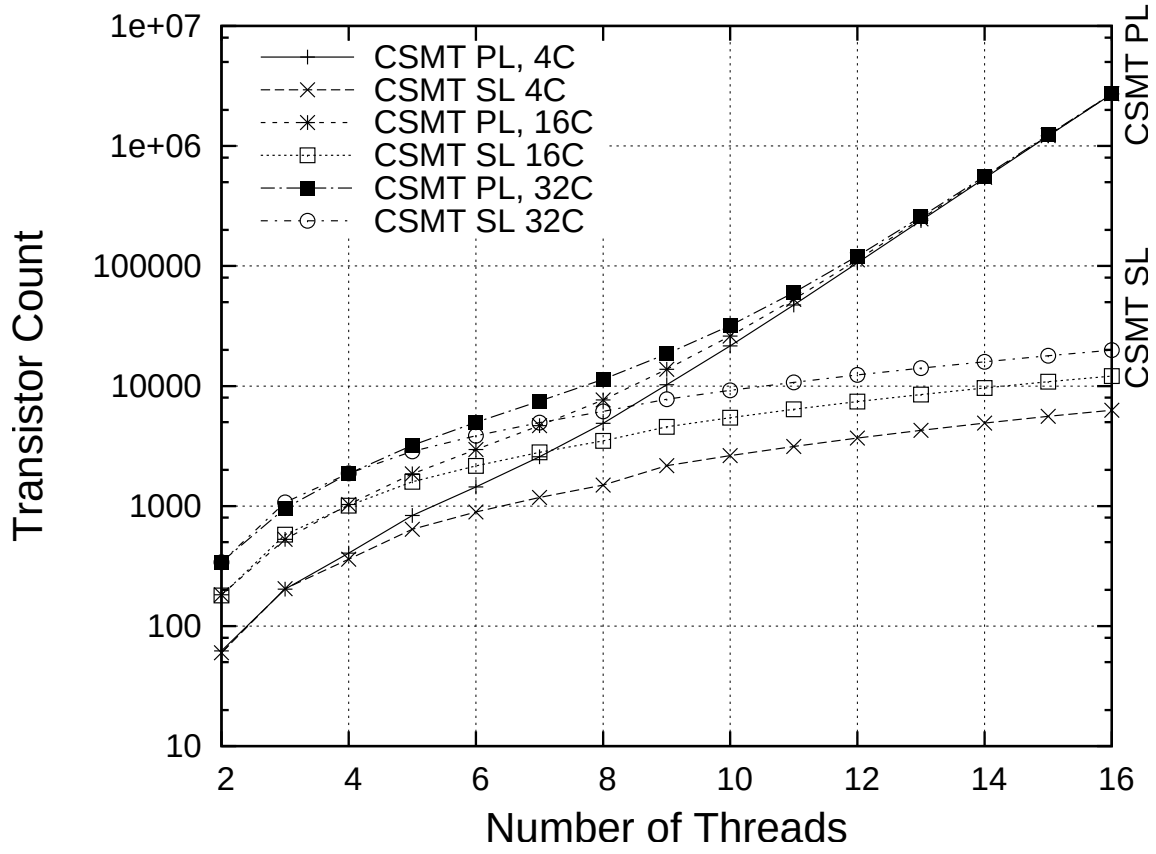


Figure A.10: **Transistor Count for CSMT**

computations for thread combinations with more than 2 threads is:
$\sum_{k=N}^{3} k(k-1)\binom{N-1}{k-1} = (N-1)(N+2)2^{N-3} - 2(N-1)$

For computing the validity of a 2-thread combination, $4C + 2C = 6C$ transistors are required. Since, there are $\binom{N}{2}$ 2-thread validity computations, $6C \times \binom{N}{2}$ or $3CN(N-1)$ transistors are required. Therefore, the number of transistors required in computing all validities is:
$(N-1)(N+2)2^{N-3} - 2(N-1) + 3CN(N-1)$

The total transistor count in parallel logic is:
$T(PL) = (N-1)(N+6)2^{N-3} + 4N^2log(N) - 3CN + 4CN^2 + N^2 - 2N + 2 = \mathrm{O}(N^2 2^N + CN^2)$. Hence, for a fixed number of threads, $T(PL) = \mathrm{O}(C)$, and for a fixed number of clusters $T(PL) = \mathrm{O}(N^2 2^N)$.

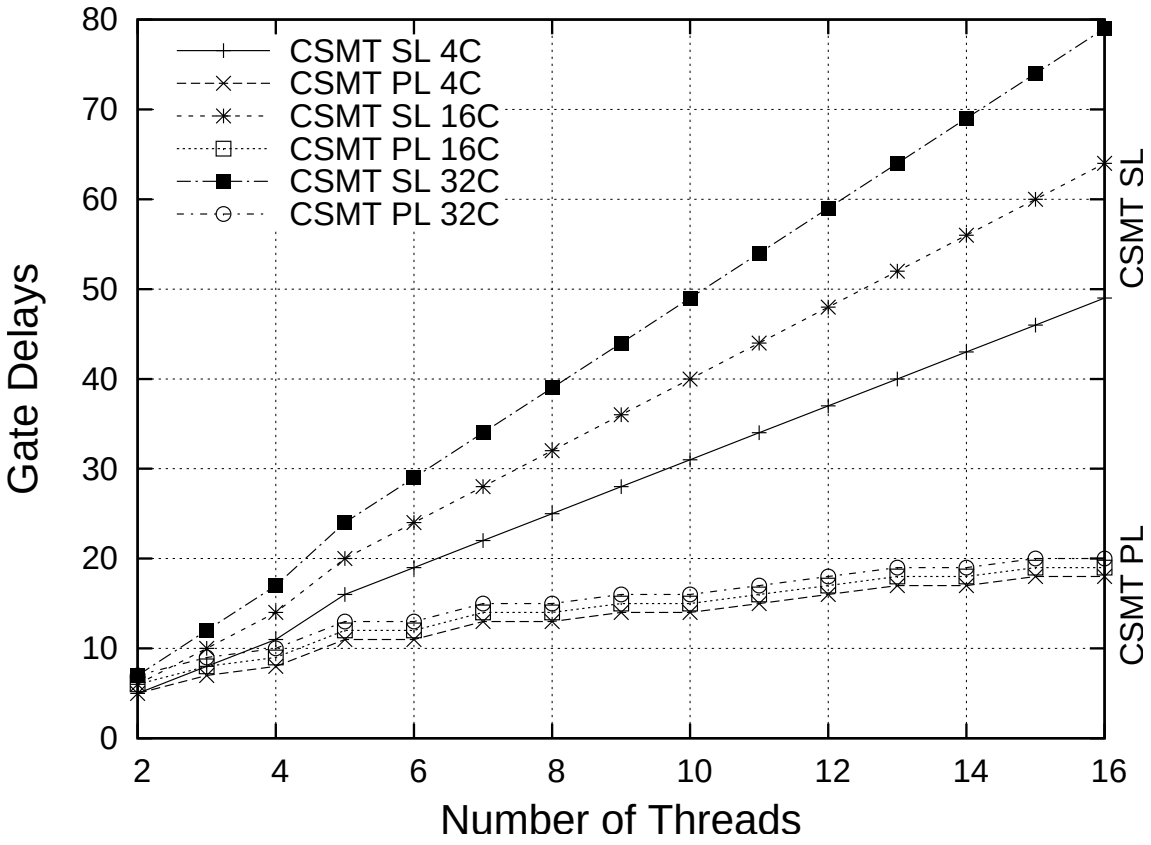


Figure A.11: **Gate Delays for CSMT**

A.6 Results

This section compares the two designs for Thread Select Logic (Serial and Parallel) discussed in this chapter based on their transistor count and delay characteristics. Figure A.10 shows on a logscale the transistor count required in the thread select logic for both, serial logic and parallel logic, with a varying number of threads for 4, 16, and 32 clusters, and assuming a per cluster issue-width of 4. In the figure, labels PL and SL refer to parallel logic and serial logic respectively and C is the number of clusters. Serial logic design has a modest increase in number of transistors when threads or clusters increase. However, transistor count for parallel logic increases exponentially, requiring millions of transistors when hardware for 16 threads is provided. Note that till 4 threads, both designs use similar number of transistors.

For parallel logic, the transistor count grows exponentially with number of threads For a large number of threads, the dominating factor in transistor count is the number of threads and the effect of number of clusters is insignificant. This effect can be seen

in Figure A.10. When the number of threads increases, transistor counts for different number of clusters start approaching similar numbers. As can be seen from the figure, the transistor count is very similar irrespective of the number of clusters after 12 threads.

Figure A.11 shows the delay for thread merge hardware for serial logic and parallel logic with a varying number of threads for 4, 16 and 32 clusters. As can be seen, the delay increases almost linearly with the number of threads for both serial and parallel logic. For small number of threads, both serial and parallel logic have similar delay.

The delay for serial logic, however, grows significantly with the number of threads. For instance, for 4 clusters, the delay for serial logic is 11 with 4 threads but increases to 25 with 8 threads. This happens because of the cascading nature of serial logic. Since as many levels as the number of threads are required, there is a substantial increase in delay. Parallel logic, however, has only a little increase in delay compared to serial design when number of threads is increased. Hence, parallel logic is much more feasible than serial logic because of the lower delay if a large number of threads have to be supported.

When both delay and transistor count is considered, none of the two designs have a clear advantage for a large number of threads over the other representing the classic area-delay trade off. Nevertheless, the area and delay requirements are very similar for both designs when the number of threads is small. Note that supporting more than 8 threads and 8 clusters is not expected to be practical in a production environment. In fact, we expect 4-Thread 4-Clusters to be a more common scenario. At 4-Thread 4-Clusters (the configuration evaluation in this thesis), both Serial and Parallel implementations have a very small area requirement and low delay. In fact, the cost of Parallel Implementation is low enough that CSMT hardware overhead can be considered to be close to low cost multithreading techniques like Interleaved MultiThreading.

Appendix B
OpSMT Cost Analysis

This chapter discusses the implementation of Thread Merging Hardware for supporting Operation-Level Simultaneous MultiThreading (OpSMT) for our target clustered VLIW processor. Similar to CSMT thread merging hardware, we discuss two implementations viz. serial and parallel for OpSMT thread merging hardware. We also discuss the area and delay characteristics of both implementations and compare them to CSMT implementations.

B.1 Introduction

CSMT merges instructions from different threads at cluster-level and therefore, at a given cluster instructions from only 1 thread is selected. CSMT thread merge control generates the signal for mux to select a given thread as explained previously in Appendix A and has a very simple implementation. Unlike CSMT, OpSMT checks resource collisions at operation-level and can select operations from multiple threads at the same cluster. Figures B.1 and B.2 show the high level thread merging hardware for both

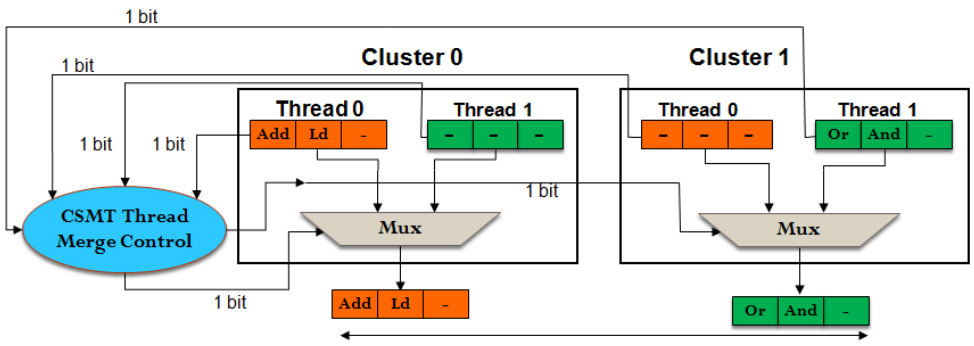


Figure B.1: **CSMT Thread merge hardware**

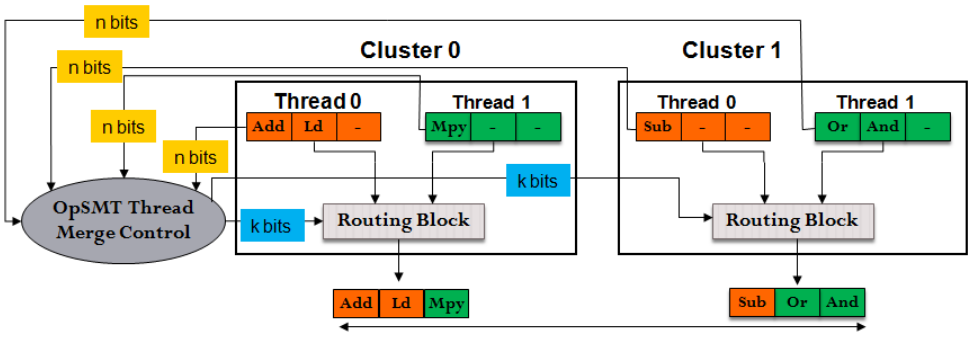


Figure B.2: **OpSMT Thread merge hardware**

CSMT and OpSMT. To fit operations from multiple threads in the same cluster, OpSMT merging hardware needs to route the operations of the instructions. For instance, in Figure B.2 operations "Or" and "And" are routed from slots 0 and 1 to slots 1 and 2 in the execution packet. Hence, OpSMT thread merge control also needs to generate appropriate signals for routing the operations. The merging block uses these routing signals to produce the final execution packet. The thread merge control is more complex for OpSMT than CSMT. However, the area required by the merging block to do the routing is similar to the area required for the multiplexers in CSMT, assuming the methodology used in [32] for interconnect area computation. This is because the area is dominated by the wires and the number of input and output wires for the merging block is same as that of the multiplexers. (Approximate Area consumption = $Num\_Input\_Wires \times Num\_Output\_Wires \times Wire\_pitch^2$) Hence the cost of the thread merge control is the only variable cost and the only factor that influences scalability of CSMT vs OpSMT. Previous studies that have done an evaluation of the hardware required for OpSMT on VLIW [19] limit the number of threads that can be supported realistically to only 2. We discuss the thread merge control implementation in the following section.

B.1.1 OpSMT Thread Merge Control

OpSMT thread merge control can be implemented using the same serial and parallel strategy as already discussed for CSMT. As a reminder, serial logic is a cascading logic where instruction from each thread is tried to be selected as each level of cascade. Parallel logic, on the other hand, tries merging instructions from all threads at the same time. Serial logic has higher delay but low area cost while parallel logic requires more area but has lower

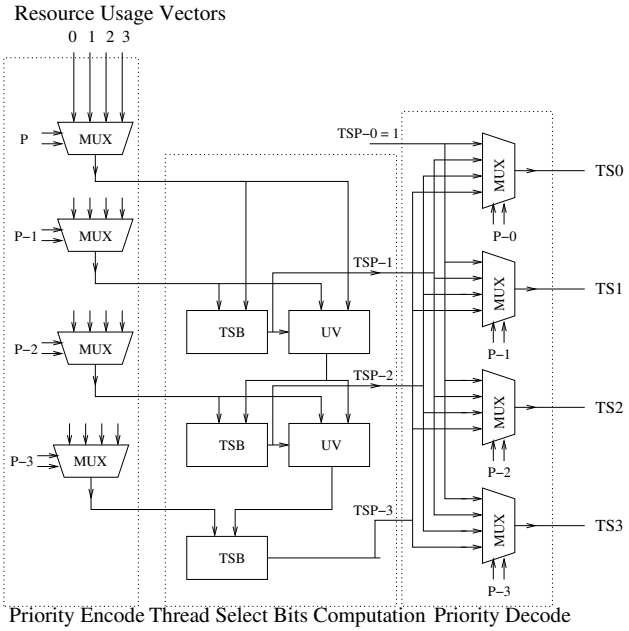


Figure B.3: **OpSMT Serial Logic**

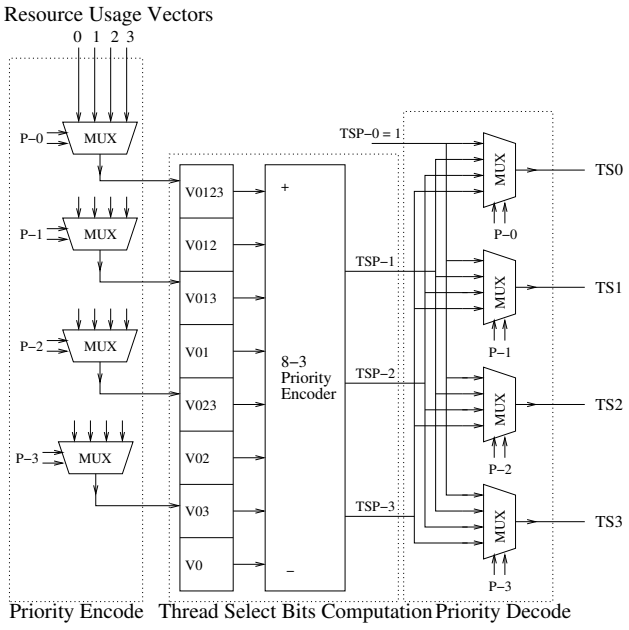


Figure B.4: **OpSMT Parallel Logic**

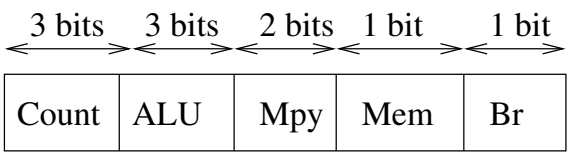


Figure B.5: **OpSMT Resource Vector**

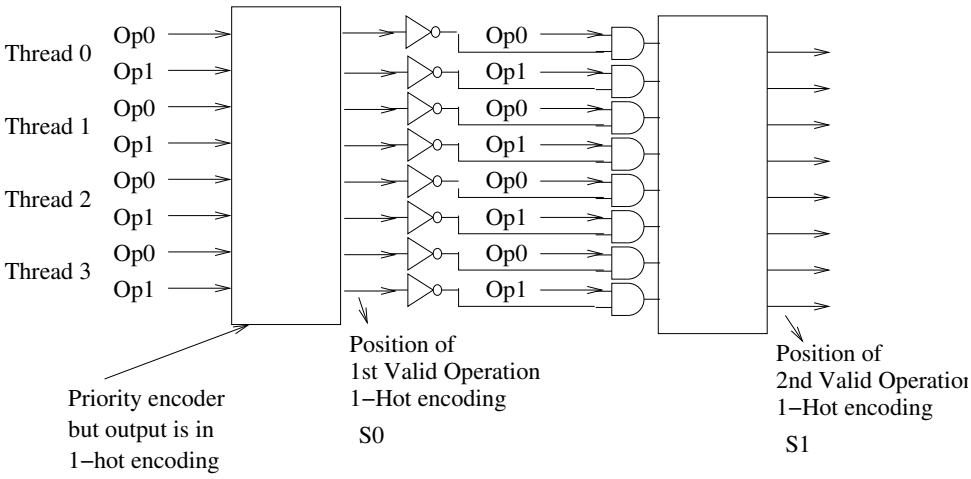


Figure B.6: **OpSMT Resource Routing**

delay. Figure B.3 shows the serial logic implementation and Figure B.4 shows the parallel logic implementation for OpSMT. The only minor difference in this block at high level is presence of a resource usage vector instead of a simple cluster usage vector. After the thread selections are obtained, the operations needs to be rerouted to the respective slots in hardware to form the execution packet. Figure B.6 shows the routing of operations at a given cluster. Note that we skip redescribing the blocks common to CSMT and OpSMT e.g. Priority Encode/Priority Decode etc.

B.1.1.1 Serial logic

Serial logic is a cascading logic that computes a thread selection bit at each level of cascade. To avoid conflicts between multiple threads at a given, none of the resources should be over subscribed at each cluster. The resources include total operation count, ALUs, Multipliers, Memory and Branch Units as shown in Figure B.5. Therefore, The hardware required for computing a collision between 2 threads at one cluster require:

Operation count check: A maximum of 4 operations can be issued at each cluster. Assuming the 3 bits to represent number of operations of each Thread to be A[2:0]:A2, A1, A0 and B[2:0]:B2, B1, B0, Then Collision condition = A + B > 4 => A2B0 + A2B1 + A2B2 + A1B1(A0 + B0) + B2A1 + B2A0. Also, the Critical Delay = 3 gates as shown in Figure B.7.

ALU Count check: Same as Operation Count.

Multipliers count check: 2 multiply operations can be issued at a given cluster. Lets assume 2 bits are used to represent number of multiply operations of each Thread to be

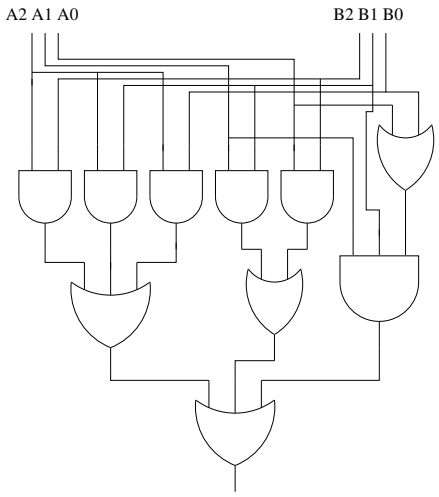


Figure B.7: **ALU/Operation Count Check**

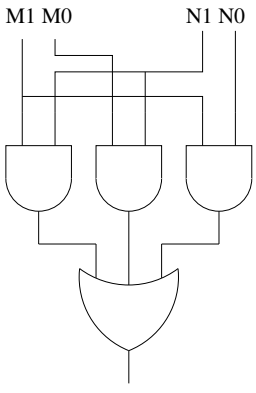


Figure B.8: **Multiply Count Check**

M1, M0 and N1, N0, Then Collision condition = M1N1 + M1N0 + M0N1. Also, Critical Delay = 2 gates as shown in Figure B.8.

Loads/Stores check: Only 1 operation can be issued at a given cycle. Therefore only 1 AND Gate is required and the Critical Delay = 1 gate.

Branch Units check: Same as Loads/Stores.

Thread Select Bit: Finally, one 5xC input AND Gate is required to compute collision between the 2 threads for a C-cluster architecture in the same way as CSMT.

Resource vector computation New resource vector computation is done for all resources e.g. Operation Count/ALUs/Multipliers etc. The resource vector computation is simply an add operation for each resource type. Overflow checks are not required since that is part of collision detection. Therefore, ALU count requires a 3-bit adder, Multiply requires a 2-bit adder etc.

B.1.1.2 Parallel logic

As described earlier in the previous chapter, Parallel logic checks all threads for resource collision simultaneously and selects a thread combination that conforms to the thread selection policy. For a N-thread C-cluster machine, 2^{N-1} validity computations are required. To compute the validities for the thread combinations, first the validities of all 2-thread combinations are computed in parallel. For a thread combination with more than two threads to be valid, all permutations of 2-thread combinations of its constituting threads should be valid. For instance, a valid thread combination 0123 requires that all the 2-thread combinations 01, 02, 03, 12, 13 and 23 are valid. Similarly, for the thread combination 023 to be valid, thread combinations 02, 03, 23 must all be valid. For a N-thread machine, $\binom{N}{2}$ 2-thread validity computations are required for the thread combination which includes all threads (thread combination $012..N-1$).

B.1.1.3 Routing Computation

Routing computation is another important logic required in OpSMT architecture. Routing is required to move operations from different threads to correct execution slots in the VLIW architecture. e.g. For a 3-issue per cluster 2-Thread VLIW architecture, if first threads have 2 operations, then operations from first threads need to go to slot 0 and slot 1. However, operations belonging to thread 1 need to execute at slot 2. This is illustrated in Figure B.2 where for cluster 0, operations Add and Ld from Thread 0 are executed at slots 0 and 1 while operation Mpy from Thread 1 is moved to execution slot 2. The

routing is computed for each issue slot in a round robin way. First routing for issue slot 0 is decided, then slot 1 and so on. Figure B.6 shows an example of routing logic for a 2-issue per cluster and 4-Thread architecture. A set Opi bit indicates if the operation is valid or not. Op ids of a thread not selected for merging are already set to 0.

B.2 Cost Analysis

This section presents a cost analysis of both serial and parallel logic in terms of both gate delays and transistor count. First we discuss the serial logic cost.

B.2.1 Serial Logic

The cost of serial logic depends on the cost of collision detection logic and usage vector computation at each level of cascade.

- **Gate Delays**:
  The critical gate delay for checking resource collision is the delay of checking ALU count i.e. 3. Also, one 5xC input AND Gate is required to compute collision between the 2 threads for a C-cluster architecture in the same way as CSMT. Since 3 gates is the critical delay for resource check, Total delay = $3 + log4(5C)$. Besides, adders are required to compute new usage vector for each resource. New usage vector computation is done in parallel with collision computation but adds another level of delay.

  Therefore For N-Threads, Total Delay = (N-1)(3 + log4(5C)) + (N-2) = (N-1)(4 + Log4(5C)) - 1 which is $O(N \times log(C))$.

- **Transistor Count**: Assuming that for a N-input gate, number of transistors required is double of number of inputs. Therefore the transistor requirement for For collision detection: ALU and Operation Count: 36 each, Mpy : 18, Ld and Br: 4 each. 5xC AND Gate: 13C. For resource vector update, the cost is the transistors incurred in adders. Transistor cost for adders: 36 for both Operation Count and ALU each, 16 for Mpy, 4 for both memory and branch units. Therefore, Approximate transistors required: C(13C(N-1) + 120(N-1) + 96(N-2)) = 13NC2 - 13C2 + 214NC - 330C which is $O(NC^2)$.

B.2.2 Parallel Logic

Parallel logic does the collision checks in parallel for all the thread combinations. The cost in terms of gate delays and transistor is discussed as following .

- **Gate Delays**:
  The critical gate delay in parallel logic is the gate delay for the thread combination $012..(N-1)$ i.e. the case where all threads are selected. The case itself requires computing collision checks and resource usage vector computation for all sub combinations i.e 01, 012, 23 etc. Using a divide and conquer approach, the validity can be computed if combinations $01..(N-1)/2$ and $(N+1)/2..(N-1)$ are valid and the resource usage vectors of these combinations do not have any collision either. i.e. thread combination 0123 is valid if combinations, 01 and 32 are valid and the combinations 01 and 23 do not have resource conflicts themselves. This way, the validities are computed in a binary tree fashion with a height of log(N). Therefore, total delay = $delay(2Threads) * log2(N)$ = $(4 + log(5C)) * log2(N)$ which is $O(log(C)log(N))$.

- **Transistor Count**:
  Since the validities of all possible thread combinations in parallel a large area overhead is incurred because of large number of possible combinations (2^{N-1} possible combinations for N-threads). Besides, even for computing validity of a single combination using the divide and conquer approach discussed above, computing validities of sub combinations is required. e.g. computing validity of combination with K threads requires checking $2^K - 1$ cases. Therefore, total number of combinations to be checked (in a 2-Thread split) = $\sum_{k=2}^{N}(2^k - 1)\binom{N}{k}$ which grows exponentially with number of threads. Overall transistor count therefore = $cost(2Threads) \times Num\_Combinations$ = $(13C + 120 + 96) \times Num\_Combinations$. For 4 threads, $Num\_Combinations = 65$, so transistor count for a 4-cluster architecture = approx 17000 transistors.

B.2.3 Routing Computation

Routing computation requires a set of priority encoders per issue slot for each cluster. If there are C-clusters and the issue width is I operations per cluster, then $C \times I$ priority encoders are required.

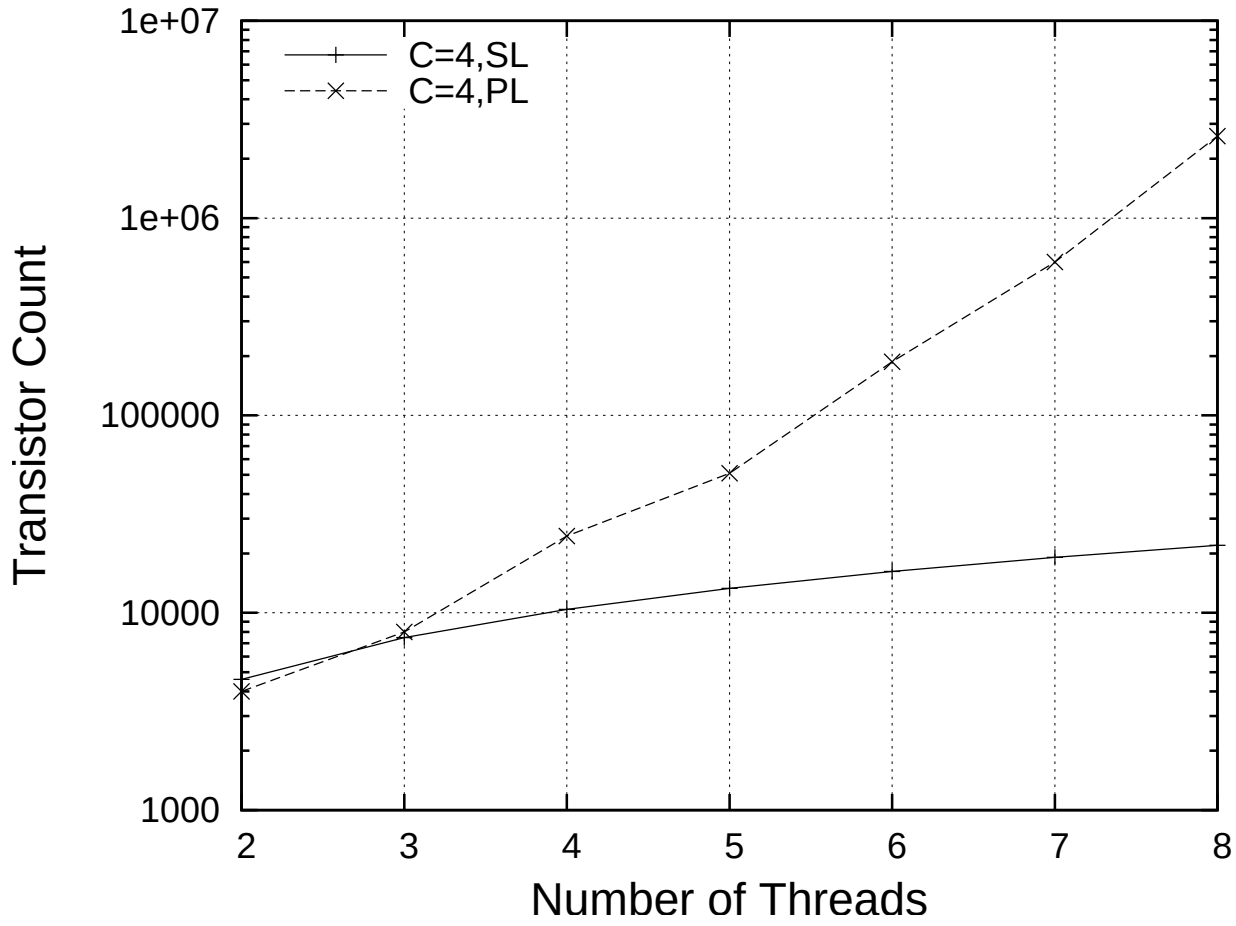


Figure B.9: **OpSMT Area**

- **Gate Delays:**
  The delay of the priority encoder = $log_4(NI)$, where I is issue width per cluster. For I issue slots, delay = $I \times log_4(NI) + 2(I-1) + 1$. For a 4-Thread 4-issue per cluster architecture, delay = 15 approximately.

- **Transistor Count:**
  Transistor requirement for Priority encoder: $4NI(NI+1)/3$ approximately. Hence, approximate transistors required for Routing computation = $4C(I(NI(NI+1)/3))+ 8C(I-1)NI$. For a 4-Thread 4-Cluster architecture, just the routing computation requires approximately 6000 transistors.

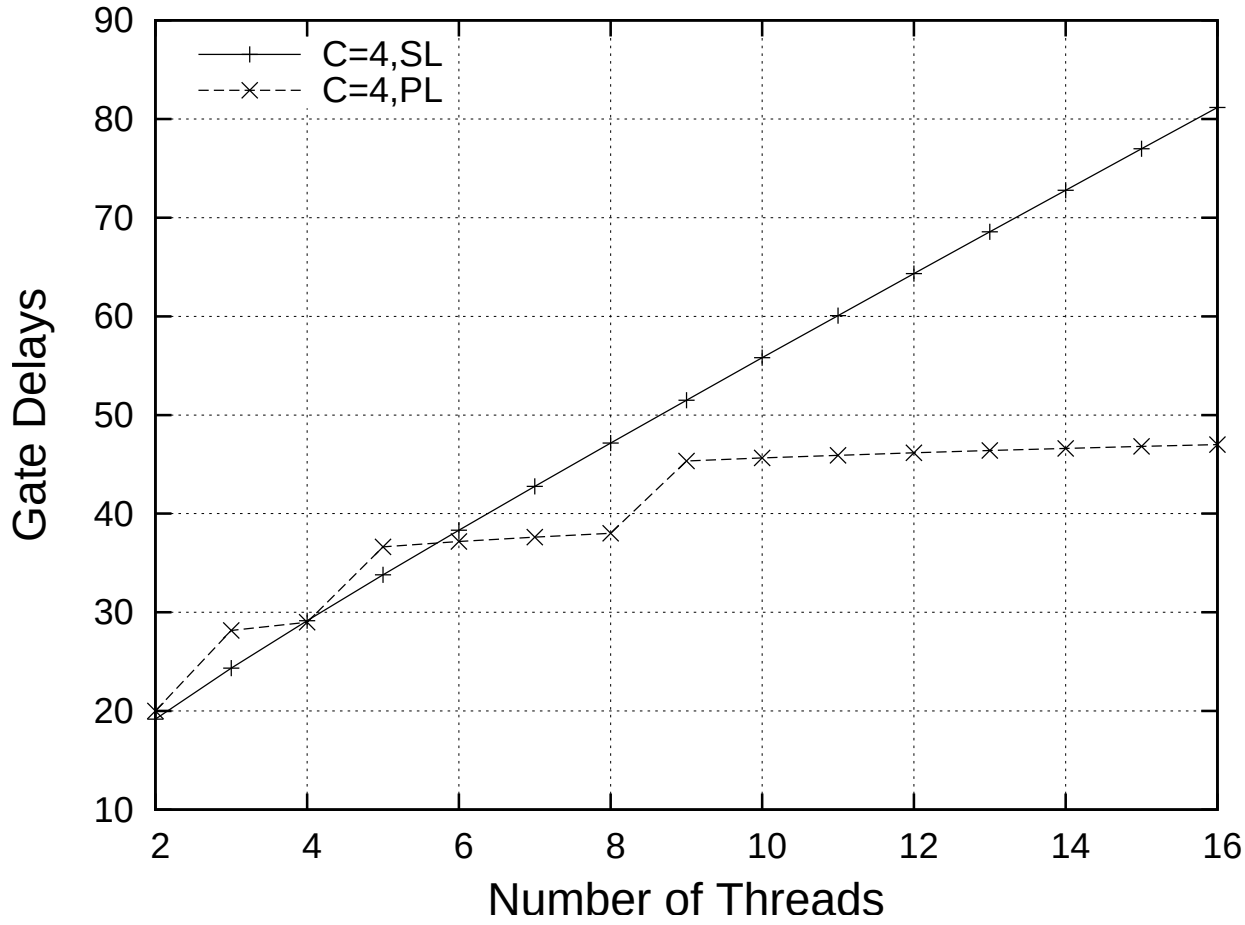


Figure B.10: **OpSMT Gate Delays**

B.3 Results

This section presents a comparison of the OpSMT thread merge hardware cost for serial and parallel implementation. As we already discussed in the previous chapter, are and delay overhead of implementing CSMT is very low and is comparable to low cost multi-threading schemes like IMT. OpSMT, however, has a far greater implementation cost for both serial and parallel logic. Figure B.9 shows the estimated transistor usage of serial and parallel logic for a 4-Issue 4-Cluster architecture with a varying number of threads on a logscale. Note that this graph does not include the area occupied because of wires in routing logic which is a non-trivial amount of area. As shown in the graph, the area for serial logic grows linearly with number of threads while parallel implementation area increases exponentially.

Figure B.10 shows the estimated gate delays for serial and parallel logic for a 4-Issue 4-Cluster architecture with a varying number of threads. Unlike transistor count, gate delays grow linearly for serial implementation while parallel implementation increase is much smaller. Note that even for a 2-thread architecture, the gate delays are around 20 which is far higher than gate delays for CSMT. With more threads, gate delays increase quite a bit and may require multiple pipeline stages to avoid any degradation in target frequency. This is the main reason, OpSMT is not as scalable as CSMT for a large number of threads.