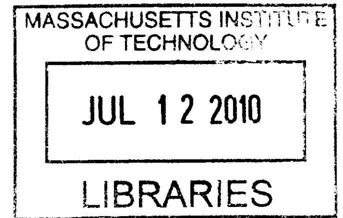
Performance and Energy Efficiency in Simple Simultaneous Multithreading Processor Cores

by

Richard Stephen Uhler

B.S. Electrical Engineering
University of California, Los Angeles 2008


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 12 2010
LIBRARIES
ARCHIVES

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

© Massachusetts Institute of Technology 2010. All rights reserved.

Author . . . .
Department of
Electrical Engineering and Computer Science
March 15, 2010

Certified by . . . .
Jack B. Dennis
Professor Emeritus
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Performance and Energy Efficiency in Simple Simultaneous Multithreading Processor Cores

by

Richard Stephen Uhler

Submitted to the Department of
Electrical Engineering and Computer Science
on March 15, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Simultaneous multithreading, where instructions from different threads share processor resources, has shown promise in delivering high throughput with little area and power overhead. We compare where in the performance energy-efficiency space alternative simple simultaneous multithreading configurations lie, leveraging standard industry tools to estimate area and power from high level hardware descriptions.

We find sharing function units among threads can improve energy efficiency over duplicating the function unit set for each thread. A good choice for the number of threads sharing a function unit ensures the function unit is not overloaded. Sharing front-end pipeline logic does not improve performance or energy efficiency over either duplicating the full pipeline or just duplicating the front-end pipelines for each thread. Different arbitration policies for use of function units do not impact performance much, but they do have a large impact on the power of the core, so the simplest arbitration policy should be used to maximize energy efficiency. Operand bypassing, an obvious optimization for a pipeline which does not share function units, is not obviously better when function units are shared, improving performance at the cost of reduced energy efficiency.

Thesis Supervisor: Jack B. Dennis
Title: Professor Emeritus

Acknowledgments

The Microsystems Technology Laboratories provided CAD tools. This research is supported in part by NSF grants CNS-0719753 and CCF-0937832.

Contents

1 Introduction **9**

2 Experimental Design **14**

2.1 Base Configuration 14

2.1.1 Fetch 15

2.1.2 Decode 17

2.1.3 Issue 17

2.1.4 Function Units 18

2.1.5 Register File 19

2.2 Methodology 20

2.2.1 Evaluating Performance and Energy Efficiency 20

2.2.2 Measuring Performance and Energy Efficiency 23

2.3 Benchmarks 25

2.3.1 MULT 26

2.3.2 IDOT 26

2.3.3 IDOTLU 26

2.3.4 FDOT 27

2.3.5 FDOTLU 27

2.3.6 Workloads 27

3 Initial Multithreaded Configurations **34**

3.1 SEPARATE Configuration 35

3.2 FUSHARED Configuration 35

3.3 ALLSHARED Configuration 36
3.4 Evaluation of the Initial Configurations 37
3.5 Improving FUSHARED 41
3.6 Improving ALLSHARED 45

4 Expanded Configurations **51**

5 Arbitration for Function Units **60**

6 Operand Bypassing **64**
6.1 Integer Bypassing 64
6.2 Condition Code Bypassing 66
6.3 Results 67

7 Conclusion **75**
7.1 Future Work 76

Bibliography **78**

List of Tables

2.1 Definition of workloads. 33

3.1 Load balance of MIX workload on initial configurations. 39

4.1 Load balance of MIX workload of expanded configurations. 51

5.1 Load balance of MIX workload under different arbitration. 61

6.1 Load balance of MIX workload with bypassing. 67

List of Figures

2-1 Single thread core base configuration. 15

2-2 Fetch stage in base configuration. 16

2-3 Toolflow for measuring performance, energy efficiency, area. 24

2-4 Pseudocode for MULT program. 27

2-5 Assembly for MULT program. 28

2-6 Pseudocode for IDOT program. 29

2-7 Assembly for IDOT program. 29

2-8 Assembly for IDOTLU program 30

2-9 Pseudocode for FDOT program 30

2-10 Assembly for FDOT program. 31

2-11 Assembly for FDOTLU program. 32

3-1 SEPARATE configuration. 35

3-2 FUSHARED configuration. 36

3-3 ALLSHARED configuration. 37

3-4 Performance of SEPARATE, FUSHARED, ALLSHARED. 38

3-5 Core area of SEPARATE, FUSHARED, ALLSHARED. 40

3-6 Energy efficiency of SEPARATE, FUSHARED, ALLSHARED. 42

3-7 IPC IPJ plot of SEPARATE, FUSHARED, ALLSHARED. 43

3-8 FUSHARED function unit utilization. 44

3-9 SEPARATE function unit utilization. 46

3-10 FUSHARED function unit conflicts. 47

3-11 FUSHARED2 configuration. 48

3-12 ALLSHARED function unit utilization. 49
3-13 ALLSHARED2 configuration. 50

4-1 Performance with FUSHARED2, ALLSHARED2 configurations. . . . 52
4-2 FUSHARED2 function unit conflicts. 53
4-3 Area with FUSHARED2, ALLSHARED2 configurations. 55
4-4 Energy efficiency with FUSHARED2, ALLSHARED2 configurations. 56
4-5 IPC IPJ plot of FUSHARED, FUSHARED2. 57
4-6 IPC IPJ plot of SEPARATE, FUSHARED2, ALLSHARED2. 58

5-1 Performance of FUSHARED2 using different function unit arbitration. 62
5-2 Energy efficiency of FUSHARED2 using different function unit arbitration. 63

6-1 Example of back to back dependent instructions. 64
6-2 Example of condition code dependency. 66
6-3 Performance of SEPARATE and FUSHARED2 with bypassing. . . . 68
6-4 Area of SEPARATE and FUSHARED2 with bypassing. 69
6-5 Energy efficiency of SEPARATE and FUSHARED2 with bypassing. . 70
6-6 IPC IPJ plot of SEPARATE with and without bypassing. 72
6-7 IPC IPJ plot of FUSHARED2 with and without bypassing. 73
6-8 IPC IPJ plot of SEPARATE and FUSHARED2 with bypassing. . . 74

Chapter 1

Introduction

In the past few decades significant work has gone into building high performance computer processors. The availability of ever increasing numbers of transistors in processor designs has allowed us to go beyond the scalar pipelines of the 1980s into a whole new realm of superscalar architectures[12]. Complex techniques such as out-of-order issue and execution, register renaming, branch prediction, and speculative execution put those extra transistors to work squeezing out every last bit of single thread performance, exploiting instruction level parallelism to bring us to the heroic processors of the 1990s.

However, this form of instruction level parallelism could take us only so far, leading people to investigate another form of parallelism, that of thread level parallelism through multithreading. Rather than sitting idle for those long latency operations common in single thread execution, the latency can be effectively hidden by a fast context switch to a different thread. In [22], Tullsen, et al., note multithreading can reduce what they call wasted horizontal space, where no instructions are available for issue because of dependencies, but they also argue multithreading does little to reduce wasted vertical space, where function units are not utilized because the single thread considered for instruction issue still has limited instruction level parallelism.

Tullsen, et al. then introduce a technique they call simultaneous multithreading, which allows instructions from different threads to be considered for issue in the same cycle. They present in [14] an adaptation of a heroic out-of-order superscalar

processor for simultaneous multithreading and demonstrate it can achieve both the single thread performance of single thread superscalar processors and the latency hiding of multithreading processors while taking advantage of increased function unit utilization. The technique of simultaneous multithreading has opened the door to the next generation of high performance processors in the new millennium.

With the new millennium comes a changing landscape for computing. Embedded and mobile devices are dominating the market once centered around desktop and server systems. Cell phones, smart phones, mp3 players, all strive to run programs with the high performance we have come to expect from our computers, but now a new constraint has come to the forefront: energy. A device with unrestrained energy consumption eats through battery life or heats up to unacceptable temperatures, becoming unusable and costly.

The complex techniques used in out-of-order superscalar processors, attractive in their ability to enhance performance, are suddenly under scrutiny where energy efficiency is desired. Is it worth the excess energy consumption to execute speculative instructions whose results may ultimately be discarded?

Fortunately simultaneous multithreading, the technique used to push performance even beyond what a single threaded superscalar architecture achieves, also shows promise in improving energy efficiency. Burns and Gaudiot in [2] demonstrate the high performance improvements from simultaneous multithreading come with just small to moderate area overhead. Seng, et al. suggest in [20] simultaneous multithreading is inherently energy efficient because of less energy wasted on wrong path instructions for misspeculation and less waste of underutilized resources, and Li, et al. find simultaneous multithreading to provide a substantial benefit for energy efficiency metrics such as the energy delay square metric.

Simultaneous multithreading is not limited to being applied to heoric superscalar architectures as studied in the previous works mentioned. With the trend in computing towards massive parallelism for high performance, where throughput is more important and there are always more threads to work on, it makes sense to apply simultaneous multithreading to simple scalar pipelines, which sacrifice single thread

performance but also do not have the complex, energy hungry structures required by superscalar achitectures. The joining of simultaneous multithreading and simple pipelines, which we refer to as *simple* simultaneous multithreading, shows huge potential for high performance, high energy efficient architectures.

After Tullsen, et al. originally introduce the technique of simultaneous multithreading in [22] they then demonstrate in [14] how it can be implemented as a straightforward extension to a conventional high performance out-of-order superscalar architecture. As a result of this, much research effort on simultaneous multithreading has been on simultaneous multithreading in a heroic processor [9] [6] [10].

Hily and Seznec in [11] consider simultaneous multithreading in a simpler architecture, suggesting that as the number of threads increase there is less advantage to having out-of-order execution over in-order execution.

Davis, et al. in [4] suggest if you fix area, to maximize throughput it is better to use many scalar multithreaded cores, which they call *mediocre* cores, than a few heroic cores.

It is argued by Laudon in [13] that using many simple multithreading cores for throughput oriented server applications has a significant performance per watt advantage over high performance superscalar processors. Both Intel's Atom [8] and Sun's Niagara [16], which target low power, use simple simultaneous cores to do so while still supporting decent throughput.

In this work we take a closer look at the performance and energy efficiency of simple simultaneous multithreading cores.

After describing our experimental design and methodology in chapter 2, we look specifically at

- How the performance and energy efficiency of multiple threads sharing function units compares to that of duplicating multiple simple single thread pipelines, each with their own separate set of function units. Chapters 3 and 4.

- Whether sharing the front end pipeline logic among threads leads to similar performance benefits as sharing function units. Chapters 3 and 4.

- Under what circumstances a function unit should be shared by all threads, only some threads, or not at all. Chapters 3 and 4.
- How different policies for arbitrating threads' use of function units affect performance and energy efficiency. Chapter 5.
- How sharing function units interacts with the high performance optimization technique of operand bypassing, an obvious technique to apply in single thread architectures. Chapter 6.

We conclude with a summary of our findings.

For our evaluation we have implemented in a high level hardware description language a variety of configurations all derived from a single common base configuration described in section 2.1. We limit our configurations to the core pipeline logic needed to execute simple kernels with straight line code and branches, we model memory as taking a fixed small latency, and threads each operate in their own instruction and data memory spaces.

Using standard industry tools we simulate our hardware description for cycle accurate performance results. We synthesize, place, and route our hardware description to get estimates of the area of the various components of our designs and take advantage of low level power simulation tools to estimate average power consumption of our architectures.

We present our evaluation of performance and energy efficiency for the various configurations by plotting them in a performance energy-efficiency space which makes clear which configurations are best and how performance and energy efficiency are traded off without prescribing preference to performance or energy efficiency.

We find sharing function units among threads can improve energy efficiency over duplicating the function unit set for each thread, though the performance can at most match that of duplicating the function unit set for each thread. A good choice for the number of threads sharing a function unit ensures the function unit is not overloaded. Sharing the front end pipeline logic does not improve performance or energy efficiency over either duplicating the full pipeline or just duplicating the front end pipelines for

each thread. Different arbitration policies for use of function units do not impact performance much, but they do have a large impact on energy efficiency, so the simplest arbitration policy should be used to maximize energy efficiency. Bypassing, an obvious optimization for a pipeline which does not share function units, is not obviously better when function units are shared, improving performance at the cost of reduced energy efficiency.

Chapter 2

Experimental Design

This chapter describes the experimental design used in our investigation of the performance and energy efficiency in simple simultaneous multithreading cores.

Section 2.1 describes the simple scalar pipeline used as a common base for all of the multithreaded configurations we experiment with. Section 2.2 describes the methodology we use to evaluate and measure the performance and energy efficiency of our multithreaded configurations, and section 2.3 describes the benchmark programs and workloads we use.

2.1 Base Configuration

This study is motivated by the desire to choose a processor architecture suitable for a multi-core chip capable of supporting composable parallel programming, which is the goal of the Fresh Breeze Project led by Professor Dennis in the CSAIL Computation Structures Group[5].

Each of the configurations we experiment with is a mulitithreaded variation on a single threaded base processor core configuration. The base configuration is a simple, single threaded, pipelined, in-order scalar processor core for the Fresh Breeze instruction set architecture. The Fresh Breeze instruction set architecture, modified with load and store instructions to give it a more traditional memory model, is a typical RISC architecture[18].

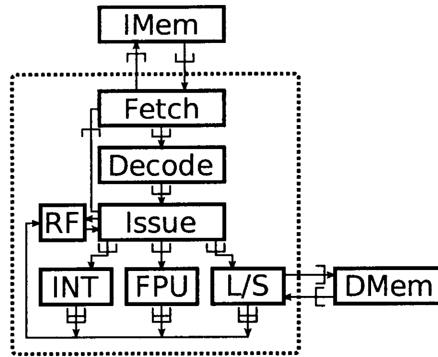


Figure 2-1: Single thread core base configuration.

Two notable features of the Fresh Breeze instruction set are the use of a condition code register for branching, where the condition code register can optionally be set as a side effect of common arithmetic operations, and absolute target addresses in branch instructions, which simplifies the implementation of branch speculation.

Because we are focusing on the processor core, we limit the instructions supported to just those required for straight line code and branches. Instruction and data memory are modeled as taking a small fixed latency.

A high level view of the base configuration is shown in figure 2-1. The core consists of a fetch stage, decode stage, issue stage, an integer function unit (INT), floating point unit (FPU), and a load-store unit (L/S).

2.1.1 Fetch

The fetch stage makes requests for instructions from the off-core instruction memory and forwards the responses to the decode stage. The instruction memory is pipelined and takes two cycles to retrieve an instruction. Figure 2-2 shows the layout of the fetch stage.

To reduce bubbles in the pipeline, fetch has a branch predictor which predicts the next program counter every cycle. The branch predictor makes predictions for all types of instructions, including jumps, branches, and nonbranch instructions. This is because it may take many cycles for the instruction memory to respond to a request, so we do not know at the time of prediction what type of instruction we are predicting

Figure 2-2: Fetch stage in base configuration.

for.

Different branch prediction algorithms can easily be swapped into the configuration. The branch predictor we use throughout our experiments is a simple four entry branch target buffer, mapping the address of previously taken branches to their targets. This branch predictor is sufficient for our benchmark programs to correctly predict most loop branches in steady state.

When the instruction memory response arrives, before fetch forwards the instruction to decode, a simple analysis of the instruction is performed which we refer to as the *fetch analysis*. This analysis checks to see if the instruction is a jump instruction, branch instruction, halt instruction, or other.

If the instruction is a jump instruction, the target is encoded directly in the instruction. If our branch predictor predicted correctly we will already have performed the jump and can drop the instruction, otherwise we notify the branch predictor that it mispredicted the jump and perform misprediction recovery. Jump instructions are never forwarded to the decode stage.

If the instruction is a non-branch instruction, we know it will not branch. If the branch predictor predicted a branch, it predicted incorrectly. We notify the predictor and begin misprediction recovery. The instruction is then forwarded to the decode stage.

Branch instructions are forwarded as is to the decode stage. They do not give any immediate feedback to the branch predictor. Later the issue stage will resolve the

branches and notify the fetch stage if there was a branch misprediction.

When we discover a branch misprediction we have to do some recovery. Branch misprediction recovery involves restoring the program counter, notifying the branch predictor for training purposes, and killing all outstanding instructions requested from the instruction memory. To easily kill outstanding instruction requests we associate with each instruction a single bit fetch epoch. On mispredict we toggle the fetch epoch, and in fetch analysis we drop any instructions that are not of the current epoch.

For the issue stage to know if we made the correct branch prediction for branch instructions, fetch passes the branch prediction made, either branch taken or not, along with the encoded instruction. There is also a single bit issue epoch used in exactly the same manner as the fetch epoch for killing instructions between fetch and issue when issue discovers a branch misprediction.

2.1.2 Decode

The decode stage is simple combinational logic which extracts interesting fields from an encoded instruction. For a branch instruction these fields are the branch operation and target. For a nonbranch instruction these fields include the operation, precision, location of operands, the destination, and whether or not to write to the destination or condition code register.

2.1.3 Issue

The issue stage stalls instructions until their dependencies are resolved, resolves branch instructions, gathers operands, and dispatches nonbranch instructions to their appropriate function unit.

Instructions are issued one at a time and in order. To respect read-after-write and write-after-write dependencies an extra bit for each register, including the condition code register, is used to indicate whether a currently executing instruction is pending a write to that register. To respect write-after-read dependencies operands are read

from the register file at the time of issue.

Specifically, a nonbranch instruction is stalled for issue if it

- Writes a destination register or condition code register which already has a pending write
- Reads an operand from a register which is pending a write

When none of the above conditions are met and there is space at the appropriate function unit, destination registers are marked pending, condition codes are marked pending if written, register operands are read from the register file, and the instruction is dispatched to the appropriate function unit based on the instruction opcode.

Dispatching of the instruction to the appropriate function unit consists of bundling together the operands, destination and other pertinent information and putting the bundle on a queue for the destination function unit. As soon as the function unit is ready to execute another instruction it will read the bundle off the front of the queue.

The issue stage is also responsible for resolving branch instructions. Branch instructions can be resolved if the condition code register is not pending. To resolve branches the issue stage reads the condition code register to determine if the branch should be taken based on the branch type. The issue stage then compares that result with the original branch prediction formed in the fetch stage. If the branch was correctly predicted, the issue stage does not have to do anything. If the branch was incorrectly predicted the issue stage notifies the fetch stage of the misprediction and increments the issue epoch.

Any instructions from an old issue epoch are dropped.

2.1.4 Function Units

There are three function units, each of which handles different types of instructions.

The integer unit performs single precision integer addition, subtraction, multiplication, bitwise AND and OR, left and right shift, and load immediate instructions. The integer unit takes a single cycle to execute any type of operation. Operations are

implemented using their corresponding Verilog operators. The synthesis tool recognizes these operators and uses optimized library implementations for their synthesis.

The floating point unit performs single precision floating point addition, subtraction, and multiplication. Each operation is performed in 4 cycles, fully pipelined. The implementation of the floating point unit is from OpenCores [23] with the division logic removed.

The load-store unit communicates loads and stores to the off-core data memory. It is fully pipelined, taking two cycles for a load.

Each of the function units performs a computation, then uses the result to either write a register, set condition codes, or both. Results are buffered until the actual writeback can occur based on whether they are destined for a left register, a right register, or the condition code register. The results of a single instruction execution may be written back on different cycles depending on the availability of register file write ports.

2.1.5 Register File

The register file contains 32 registers and is broken down into two banks of 16 registers each, corresponding to left and right sides as described by the Fresh Breeze instruction set architecture. Each bank has a single write port and two read ports.

The register file is designed to support writeback of a double precision value each cycle. While our pipeline was designed for and is capable of double precision operations, because our floating point unit only implements single precision operations, we have removed the logic for double precision integer operations, and do not exercise any double precision operations in the programs we use.

After instructions are executed in the function units they wait for a write port to become available then write back their results. Instructions can complete out of order, in which case they may be written back to the register file out of order. When a register is written it is marked as no longer pending a write.

2.2 Methodology

2.2.1 Evaluating Performance and Energy Efficiency

There have been a number of different proposals for a figure of merit to use to compare the performance and energy efficiency of different architectures. A nice summary of some of those proposals is given in [1], which suggests different choices may be valid under different circumstances. A commonly used metric is the energy delay product ED, the product of the energy per instruction and number of cycles per instruction, used in [24], [3] and [7]. The energy delay square product, ED^2, is used as the metric in [15]. The ED^2 metric gives more preference to performance than does the ED metric.

Rather than choose a single figure of merit to describe both the performance and energy efficiency of an architecture, we keep separate measures of performance and energy efficiency and plot them in a 2D space to get an idea of the tradeoffs our microarchitectural variations have on both performance and energy efficiency without suggesting whether performance or energy efficiency is more important. Our plots are similar to those in [24], except we look at energy per instruction instead of energy per cycle, and we do not draw curves of constant energy delay, though one certainly could on our graphs if they had decided the energy delay metric was appropriate for them and wanted to see what the best configuration under that metric is.

Evaluating Performance with IPC

The performance of an architecture is a measure of how much time it takes to run a program on it. To make it possible to compare the performance under different benchmarks programs, which may have different numbers of instructions in them, we normalize the time it takes to execute a program by the number of instructions executed in the program, resulting in the amount of time it takes to execute a single instruction on average for that program.

For this work we assume all the configurations we compare have the same cycle time. This means we can look at performance in terms of the average number of cycles

it takes to execute a single instruction. This is commonly known as CPI, cycles per instruction. Because we expect our architectures to execute multiple instructions per cycle on average, it is convenient to look at the reciprocal of CPI, which is the IPC, or instructions per cycle.

IPC of multiprogram workloads

There is some question as to whether IPC is a good metric for comparing the performance of two different multithreaded architectures, where a single workload consists of multiple programs running on different threads. To make a fair comparison, for a given workload each architecture should execute the same amount of work, so we fix the total number of instructions executed. The problem is, different architectures may devote more of the total instructions executed to one thread over another in the workload, meaning in the end the architectures may not have executed the same amount of work.

For example, consider a workload of two threads where single thread performance of the first thread is 2 IPC and single thread performance of the second thread is 1 IPC. We could imagine a simple multithreaded architecture which simply time multiplexes the two programs on a single thread. A version of this architecture which devotes all the workload instructions running the first thread will have an IPC near 2, while a version of the architecture which devotes all the workload instructions running the second thread will have an IPC near 1. But if the architecture is the same, how can IPC be so different?

One way to get around the problem would be to force each thread in a workload to execute a fixed number of instructions, but then threads may not all finish at the same time, failing to take full advantage of the throughput capabilities of a simultaneous multithreading architecture.

The difficulties of IPC with multiprogram workloads are raised in [19], [21], [15], and [3]. Sazeides and Juan in [19] propose the SMT-speedup metric

$$speedup = \frac{\sum_i L_i}{L}$$

to account for different load balances among threads in a workload, where L is the number of cycles for the multithreaded workload to complete, L_i is the number of cycles for the single-threaded base configuration to execute I_i instructions from thread i, and I_i is the number of instructions for thread i executed in the multithreaded workload.

Snavely and Tullsen in [21] propose their own SMT-speedup metric

$$speedup = \sum_i \frac{IPC_{smt_i}}{IPC_{nonsmt_i}}$$

which is adopted in [15] and [3].

Seng, et al., in [20] fix the number of instructions run in each thread, meaning not all threads will complete at the same time.

We stick with IPC for our measure of performance. For all our workloads except MIX, each thread in the workload runs the same program, so changing the load balance should not have any affect on IPC. For the MIX workload we will present the different load balances which occur naturally from the different configurations along with the speedup metric presented in [19] and explicitly note based on that if it appears load balance was a major contributor to the different IPC.

Evaluating energy efficiency with IPJ

The energy efficiency of an architecture is a measure of how much energy it takes to run a program on it. Using the same argument given for IPC, we normalize the total energy it takes to execute a program by the number of instructions executed and get the amount of energy it takes to execute a single instruction on average. If we represent energy in joules this gives us the number of joules per instruction. To be consistent with IPC we take the reciprocal to get the number of instructions per joule on average, or IPJ.

Once we have measured the IPC and IPJ of two different architectures, we say the *overall performance* or *overall efficiency* of one is better than the other if it has both better IPC and better IPJ. If IPC is better in one architecture and IPJ better

in the other, we can not make any claims about which architecture is better without deciding whether IPC or IPJ is more important, something we want to avoid in this project.

The energy delay product, ED, gives equal weight to IPC and IPJ, as it is just the reciprocal of the product of IPC and IPJ. Using ED^2 is the reciprocal of the square of IPC and IPJ, putting more weight on IPC than IPJ.

2.2.2 Measuring Performance and Energy Efficiency

To obtain measurements of performance, energy efficiency, area, and other features of each configuration, we use our own design and implementation of the different processor configurations.

We have implemented each configuration using Bluespec System Verilog[17], a high level hardware description language. Figure 2-3 shows the toolflow we run on the Bluespec implementations to gather the measurements we are interested in. Bluespec's support for modularity allows us to easily share code common across all configurations without adding any unfair overhead in the generated logic. It is simple to describe new configurations by choosing parameters and instances of the generic, parameterized modules already implemented for previous configurations and connecting them together appropriately.

Bluespec comes with a cycle accurate simulation framework called Bluesim. Bluesim can be controlled with Tool Command Language (Tcl) scripts and provides access to register values and debug output. We use Bluesim for measuring IPC and other timing related information.

To calculate IPC for a configuration given a set of programs to run, we load each thread with its program and simulate the configuration using Bluesim until the number of instructions from each thread issued sums to 100,000. For example, on an architecture which is fair to each thread, each of the four threads will have issued 25,000 instructions, for a total of 100,000 instructions. For unfair architectures, some threads may end up executing more or less than 25,000 instructions. Registers implemented in the core keep track of how many instructions have been issued, so we

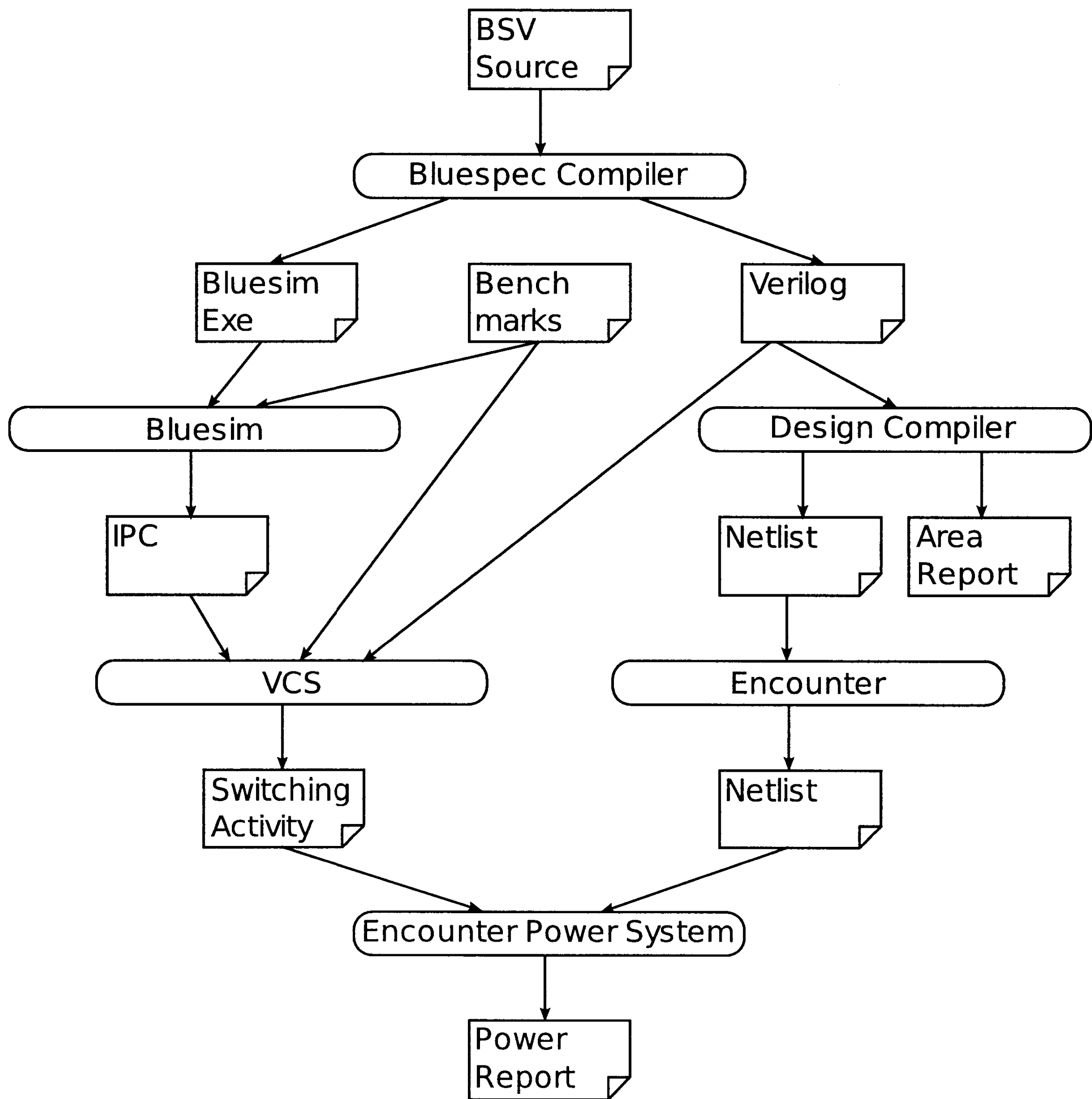


Figure 2-3: Toolflow for measuring performance, energy efficiency, area.

know when to stop the simulation. The IPC for the configuration is 100,000 divided by the total number of cycles simulated.

The Bluespec compiler can also be directed to compile the configurations to synthesizable Verilog, maintaining module hierarchy information where directed. After compiling to Verilog, we synthesize each configuration separately using Synopsis Design Compiler with Taiwan Semiconductor Company (TSMC) 0.13 μm technology libraries. Once synthesized, the configurations are placed and routed with Cadence Encounter, again using the TSMC libraries. Once the design is placed and routed, Cadence reports core area, which we use as our measure of the area of a configuration.

To measure energy efficiency, we first simulate the Verilog code generated by Bluespec using the Synopsys Verilog compiler VCS under the chosen set of programs, generating switching activity for each net in the form of a value change dump(VCD) file. The simulation is done exactly as it was for measuring IPC, 100,000 instructions are run, with each program looped so that it starts over again as soon as it finishes. The VCD file output by VCS is then supplied to Cadence Encounter, which uses it to generate an overall estimate for average power consumed by the configuration.

Once we have power estimates IPJ can be calculated as 100,000 instructions divided by the average power times the total number of seconds taken to execute all 100,000 instructions. For the IPJ results shown we assume an operating frequency of 100 MHz. Mostly we are just interested in relative IPJ and not absolute IPJ.

2.3 Benchmarks

We have selected a handful of benchmarks to exercise our processor cores. Because our cores are multithreaded they can run mixtures of different programs all at once. To avoid confusion we refer to a *program* as the code running on a single thread and a *workload* as the collection of programs run together on a multithreaded configuration.

Because we are focusing on core processor performance without considering memory hierarchy issues from caching or paging we limit our programs to simple kernel codes designed to span integer and floating point computations and the effect of

intense arithmetic use through loop unrolling.

The programs are presented with both high level pseudocode and an assembly representation of the specific instructions in the programs. In the assembly representation arithmetic operations can be marked with `Res` to indicate they set the result register but do not update the condition code register, `Cnd` to indicate they update the condition code register but do not change any other register, or unmarked to indicate they both update the destination register and the condition code register.

2.3.1 MULT

The MULT program is a software implementation of integer multiplication using a common add-shift method. The program pseudorandomly generates two 32 bit integers to multiply, then multiplies those integers together.

The branch predictor in our base configuration is capable of predicting all the loop branches in this program, but does not predict well the condition inside the loop which tests if the product is incremented or not.

Pseudocode for the MULT program is given in figure 2-4. Assembly code for the MULT program is shown in figure 2-5.

2.3.2 IDOT

IDOT is an integer dot product program. Pseudocode and assembly code is shown in figures 2-6 and 2-7 respectively.

2.3.3 IDOTLU

The IDOTLU program is a loop unrolled, hand optimized version of the IDOT program. It performs dot product on a 1024 element vector, with loop unrolling of four. The assembly code for IDOTLU is given in figure 2-8.

```
unsigned int seed = 1;

for (int i = 0; i < 1024; i++) {
    unsigned int x = seed * 0x41c64e6d + 0x3039;
    unsigned int y = x * 0x41c64e6d + 0x3039;
    seed = y;

    unsigned int product = 0
    while (x != 0) {
        if (x & 1) {
            product += y;
        }

        x >>= 1;
        y <<= 1;
    }
}
```

Figure 2-4: Pseudocode for MULT program.

2.3.4 FDOT

FDOT performs a floating point dot product. The elements of the two vectors are initialized in memory ahead of time with randomly generated floats between 0 and 1. Pseudocode and assembly code is shown in figures 2-9 and 2-10 respectively.

2.3.5 FDOTLU

FDOTLU is the loop unrolled version of FDOT. The assembly is given in figure 2-11.

2.3.6 Workloads

Our processor supports 4 threads, which means multiple programs can run at the same time. Table 2.1 lists the programs which make up each workload we refer to when discussing results.

```
// Single Precision Software Integer Multiply
// Use shift-adder algorithm.
// r0: x    an operand
// r1: y    an operand
// r2: s    multiply result
// r3: 'h41c64e6d  constant for use in random number generation
// r4: r    random number seed
// r5: i    current iteration
// r6: 12345       constant for use in random number generation
```

```
// Put 'h41c64e6d into r3
LdImIntSngl  'h41c6 -> r0
LdImIntSngl  'h4e6d -> r1
IShlRes r1, 16 -> r2
IOrRes r2, r1 -> r3

LdImIntSngl  12345 -> r6

// Seed the random
// number generator
LdImIntSngl  1 -> r4

// Main loop
LdImIntSngl  0 -> r5
MAINLOOP:
  ISubCnd r4, 1024
  BrCmpEq END

  // Generate a random
  // value for x
  // r0 <= r4 * r3 + 'h3039
  IMultRes r4, r3 -> r0
  IAddRes r0, r6 -> r0
```

```
  // Generate a random value for y
  // r1 <= r4 * r0 + 'h3039
  IMultRes r4, r0 -> r1
  IAddRes r1, r6 -> r1
  IAddRes r1, 0 -> r4

  // Multiply those two numbers
  LdImIntSngl  0 -> r2
  ISubCnd r0, 0

MULTLOOP:
    BrCmpEq ENDMULT
    IAndCnd r0, 1
    BrCmpEq SHIFTS
    IAddRes r2, r1 -> r2

SHIFTS:
    IShr r0, 1 -> r0
    IShlRes r1, 1 -> r1
    Jump MULTLOOP

ENDMULT:
  IAddRes 1, r5 -> r5
  Jump MAINLOOP

END: Quit
```

Figure 2-5: Assembly for MULT program.

```
unsigned int* mem;
for (int i = 0; i < 16; i++) {
    mem[i] = i;
}

unsigned int sum = 0;
for (int i = 0; i < 16; i++) {
    sum += mem[i] * mem[i];
}
```

Figure 2-6: Pseudocode for IDOT program.

```
    LdImIntSngl  16 -> r0
    LdImIntSngl  0 -> r1

MEMLOOP:
    ISubCnd r1, 16
    BrCmpPos MEMEND
    StoreW r1 -> Mem[r1]
    IAdd 1, r1 -> r1
    Jump MEMLOOP

MEMEND:
    LdImIntSngl  0 -> r1
    LdImIntSngl  0 -> r2

DOTLOOP:
    ISubCnd r2, 16
    BrCmpPos END
    LoadW Mem[r2] -> r3
    LoadW Mem[r2] -> r4
    IMultRes r4, r3 -> r5
    IAddRes r5, r1 -> r6
    IAddRes r6, 0 -> r1
    IAdd 1, r2 -> r2
    Jump DOTLOOP

END:
    Quit
```

Figure 2-7: Assembly for IDOT program.

```
    LdImIntSngl  1024 -> r0
    LdImIntSngl  0 -> r1

MEMLOOP:
    ISubCnd 1024, r1
    BrCmpPos ENDMEM
    IAddRes 1, r1 -> r2
    IAddRes 2, r1 -> r4
    IAddRes 3, r1 -> r6
    IAddRes 1, r1 -> r3
    IAddRes 2, r1 -> r5
    IAddRes 3, r1 -> r7
    StoreW r1 -> Mem[r1]
    StoreW r3 -> Mem[r2]
    StoreW r5 -> Mem[r4]
    StoreW r7 -> Mem[r6]
    IAdd 4, r1 -> r1
    Jump MEMLOOP

ENDMEM:
    LdImIntSngl  0 -> r1
    LdImIntSngl  0 -> r8
```

```
DOTLOOP:
    ISubCnd 1024, r8
    BrCmpPos END
    IAddRes 1, r8 -> r13
    IAddRes 1, r8 -> r15
    IAddRes 2, r8 -> r19
    IAddRes 2, r8 -> r21
    IAddRes 3, r8 -> r25
    IAddRes 3, r8 -> r27
    LoadW Mem[r8] -> r9
    LoadW Mem[r8] -> r10
    LoadW Mem[r13] -> r14
    LoadW Mem[r15] -> r16
    LoadW Mem[r19] -> r20
    LoadW Mem[r21] -> r22
    LoadW Mem[r25] -> r26
    LoadW Mem[r27] -> r28
    IMultRes r10, r9 -> r11
    IMultRes r16, r14 -> r17
    IMultRes r22, r20 -> r23
    IMultRes r28, r26 -> r29
    IAddRes r11, r1 -> r12
    IAddRes r17, r23 -> r18
    IAddRes r12, r29 -> r24
    IAddRes r18, r24 -> r1
    IAdd 4, r8 -> r8
    Jump DOTLOOP

END:
    Quit
```

Figure 2-8: Assembly for IDOTLU program

```
int i = 0;
float sum = mem[i++] * mem[i];
while (i < 30) {
    sum += mem[++i] * mem[++i];
}
```

Figure 2-9: Pseudocode for FDOT program

```
// Single Precision Floating Point Dot Product
// Takes the dot product of floating point values in memory.
// One vector consists of the values from even addresses, the other
// values from odd addresses. Each vector should have 16 elements, for a total
// of 32 floating point values read from memory.

// r0: the dot product sum
// r1: element from first vector
// r2: element from second vector
// r3: i: pointer into vectors.
// r4: r1 * r2

// Step 1. Get the first elements, multiply them, and put the result in r0
    LdImIntSngl  0 -> r3
    LoadW Mem[r3] -> r1
    LdImIntSngl  1 -> r3
    LoadW Mem[r3] -> r2
    FMultRes r1, r2 -> r0

// Step 2. Iterate over the rest of the elements
LOOP:
    ISubCnd r3, 30
    BrCmpPos END
    IAddRes r3, 1 -> r3
    LoadW Mem[r3] -> r1
    IAddRes r3, 1 -> r3
    LoadW Mem[r3] -> r2
    FMultRes r1, r2 -> r4
    FAddRes r0, r4 -> r0
    Jump LOOP

END:
    Quit
```

Figure 2-10: Assembly for FDOT program.

```
// Single Precision Floating Point Dot Product
// Takes the dot product of floating point values in memory.
// One vector consists of the values from even addresses, the other
// values from odd addresses. Each vector should have 16 elements, for a total
// of 32 floating point values read from memory.

// r0: the dot product sum
// r3: i

// Step 1. Get the first elements, multiply them, and put the result in r0
    LdImIntSngl  0 -> r3
    LoadW Mem[r3] -> r1
    LdImIntSngl  1 -> r3
    LoadW Mem[r3] -> r2
    FMultRes r1, r2 -> r0
    LdImIntSngl  2 -> r3

// Step 2. Iterate over the rest of the elements
LOOP:
    ISubCnd r3, 124
    BrCmpPos END
    IAddRes r3, 1 -> r5
    LoadW Mem[r3] -> r1
    IAddRes r3, 2 -> r3
    LoadW Mem[r5] -> r2
    IAddRes r5, 2 -> r6
    LoadW Mem[r3] -> r7
    IAddRes r3, 2 -> r8
    LoadW Mem[r6] -> rA
    FMultRes r1, r2 -> r6
    IAddRes r8, 1 -> r1
    LoadW Mem[r8] -> r2
    IAddRes r8, 2 -> r3
    FMultRes r7, rA -> r8
    LoadW Mem[r1] -> r9
    FAddRes r6, r8 -> rB
    FMultRes r2, r9 -> r1
    FAddRes r0, rB -> rC
    FAddRes r1, rC -> r0
    Jump LOOP

END:
    Quit
```

Figure 2-11: Assembly for FDOTLU program.

Workload	Thread 0	Thread 1	Thread 2	Thread 3
MULT4	MULT	MULT	MULT	MULT
IDOT4	IDOT	IDOT	IDOT	IDOT
IDOTLU4	IDOTLU	IDOTLU	IDOTLU	IDOTLU
MIX	MULT	IDOT	MULT	IDOTLU
FDOT4	FDOT	FDOT	FDOT	FDOT
FDOTLU4	FDOTLU	FDOTLU	FDOTLU	FDOTLU

Table 2.1: Definition of workloads.

Chapter 3

Initial Multithreaded Configurations

Simultaneous multithreading is attractive for high performance and energy efficient computing because instead of duplicating underutilized logic such as the function units for each thread, the logic is shared by all threads. We expect this sharing to result in a smaller area and power footprint without a corresponding decrease in performance because with more independent threads to draw from we can better utilize the shared logic.

To better understand the performance energy-efficiency trade-off for sharing logic among threads we begin by looking at three simple multithreaded processor configurations based on the single-threaded processor described in section 2.1. These configurations differ in how much logic is shared by the threads. In the first configuration no logic is shared, in the second configuration only the function units are shared by the threads, and in the third configuration the entire pipeline, excluding memory, is shared by the threads.

Each of our multithreaded configurations throughout this project runs exactly four threads.

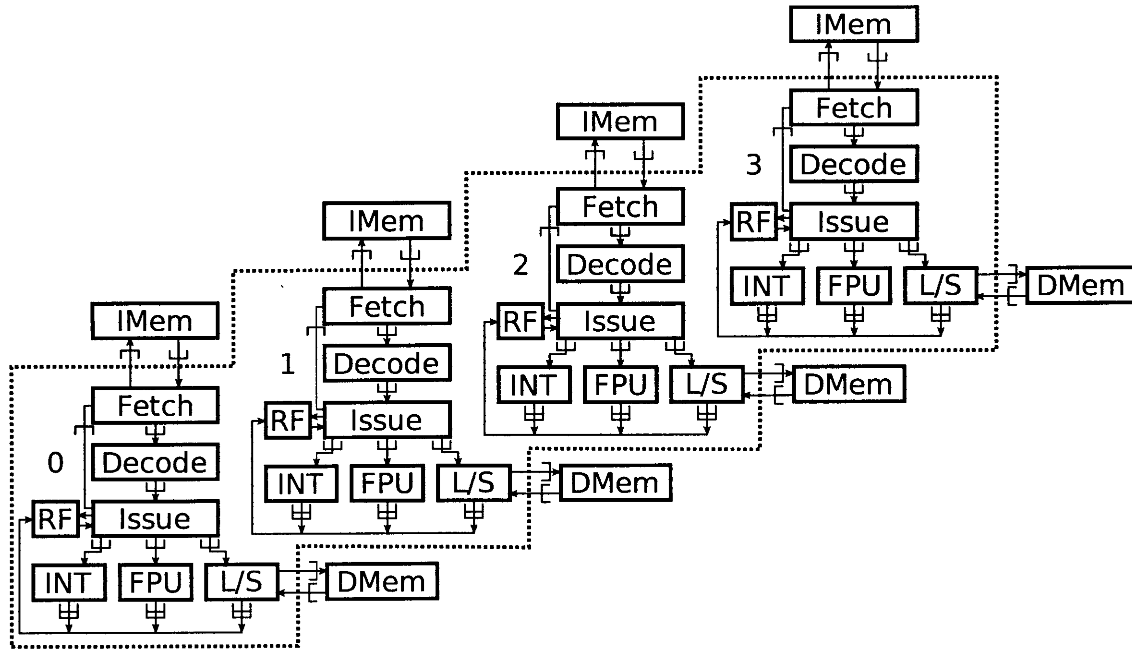


Figure 3-1: SEPARATE configuration.

3.1 SEPARATE Configuration

In the SEPARATE configuration, shown in figure 3-1, no logic is shared by the threads. We duplicate the single-threaded core configuration verbatim for each thread. This configuration serves as a baseline which our other configurations can be compared against. We expect this configuration to have the best performance at the cost of using the most logic, potentially reducing energy efficiency.

3.2 FUSHARED Configuration

The FUSHARED configuration duplicates the front-end pipeline for each thread but shares a single set of function units as depicted in figure 3-2. This configuration represents our simple simultaneous multithreading core, which we expect to have decent performance and energy efficiency.

The fetch, decode, and issue stages of FUSHARED are duplicated verbatim per thread from the single thread core. Each thread's issue stage has its own dispatch queue for each function unit. When a function unit is ready to execute another in-

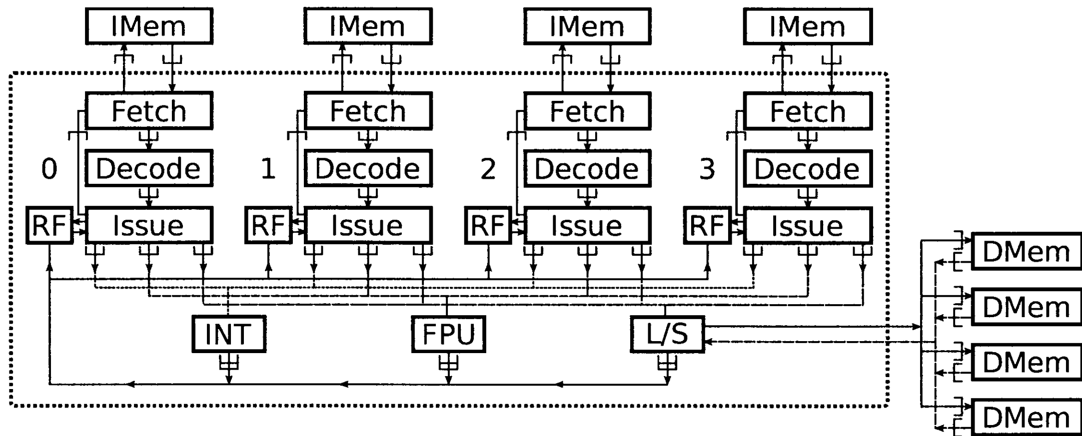


Figure 3-2: FUSHARED configuration.

struction, it searches the dispatch queues from each threads' issue stage for something to execute.

Priority for the function unit is assigned statically to the threads, so, for example, if thread 1 and thread 3 both want to use the same function unit in the same cycle, thread 1 will always win because its identifier is smaller. In chapter 5 we discuss how different arbitration policies for use of the function units affect performance and energy efficiency.

The function units are also augmented from the single thread core to pass the two bit thread identifier from argument to result. Each result is accompanied by the thread identifier, used to direct the result to the appropriate thread's register file.

3.3 ALLSHARED Configuration

The ALLSHARED configuration has all threads sharing the same pipeline and function units. If sharing function units can increase energy efficiency without great loss in performance, perhaps sharing the fetch, decode, and issue logic can too.

Each thread has its own instruction memory, branch predictor, data memory, condition code register and register file. The fetch analysis, decode, issue and function units are all shared by the different threads.

Each cycle a fetch is performed for each thread to their respective instruction

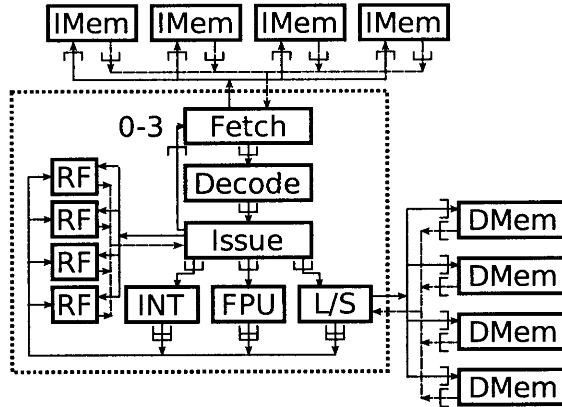


Figure 3-3: ALLSHARED configuration.

memories. The fetch analysis operates on threads in a round robin fashion. The output of the fetch unit now includes the thread identifier for the instruction fetched.

Decode is the same as the single thread core, except it includes the thread identifier in the decoded instruction. Issue is also the same, aside from now using the thread identifier to select the appropriate condition code register and register file for the thread whose instruction is currently under consideration for issue. Issue is still limited to a single instruction per cycle. Even if we have four active threads, we will not issue more than one instruction per cycle.

The function units pass the thread identifier with the result to identify the appropriate thread's register file just as is done in the FUSHARED configuration.

3.4 Evaluation of the Initial Configurations

Figure 3-4 compares the performance of each configuration on our different workloads. We expect the SEPARATE configuration to always have the best performance, because there is no shared logic, which means there are no conflicts for the use of that logic. We are more interested in knowing how much better the SEPARATE performance is than the FUSHARED and ALLSHARED performance. If it is only a little better, sharing logic did not reduce performance too much, suggesting the FUSHARED and ALLSHARED configurations have good potential for being high

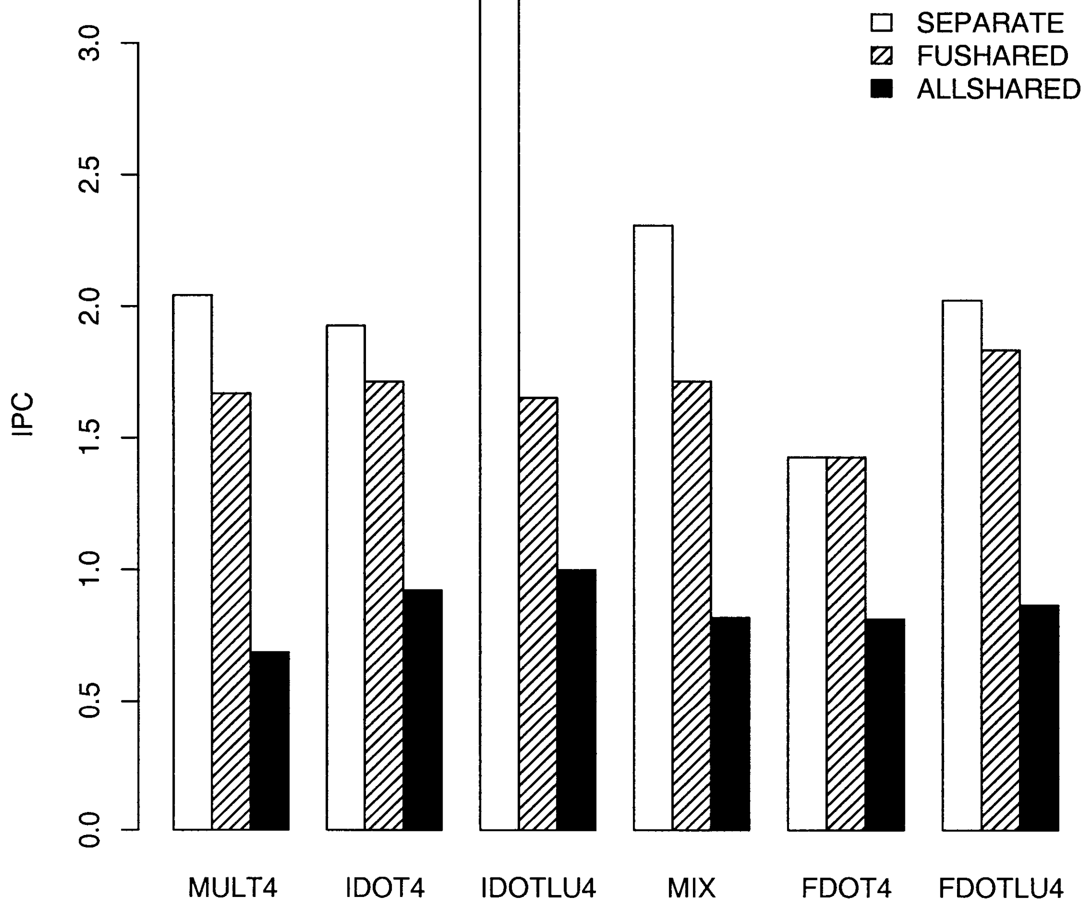


Figure 3-4: Performance of SEPARATE, FUSHARED, ALLSHARED.

	MULT	IDOT	MULT	IDOTLU	Speedup
SEPARATE	22204	20879	22204	34713	4.02
FUSHARED	29650	25906	25410	19035	3.19
ALLSHARED	21349	27889	21348	29414	1.22

Table 3.1: Load balance of MIX workload on initial configurations.

performance, energy-efficient configurations.

What we see in figure 3-4 is for the FDOT4 workload, the FUSHARED configuration performs as well as the SEPARATE configuration, but for the other workloads there is a drop in performance from SEPARATE to FUSHARED, especially large on the IDOTLU4 workload. The performance of the ALLSHARED configuration is not close to either the SEPARATE or FUSHARED configurations.

Table 3.1 shows the number of instruction executed in each thread for the MIX workload and the speedup, calculated as described in section 2.2. The SEPARATE configuration executes a number of instructions proportional to the IPC of each program run on the base single-threaded configuration. The FUSHARED configuration gives preference to the threads with lower identifiers, nearer the left in table 3.1, which have higher priority given our static arbitration policy. The ALLSHARED configuration executes less instructions from the MULT program, which has much worse branch prediction because of the random branch inside the main loop. The speedup shown for the configurations gives a similar impression as the IPC, with SEPARATE the best, FUSHARED a little worse, and ALLSHARED much worse.

Figure 3-5 shows the area of each configuration after place and route, broken down by component. The fetch, decode, and issue logic for each configuration, shown at the bottom of the bars, takes up the same amount of space in the SEPARATE and FUSHARED configurations, and a bit less in the ALLSHARED configuration. The function units dominate the difference in area between the SEPARATE configuration and the other configuration, because the SEPARATE configuration has four of each function unit, where the FUSHARED and ALLSHARED configurations each only have one of each function unit.

Sharing function units saves considerable area going from the SEPARATE config-

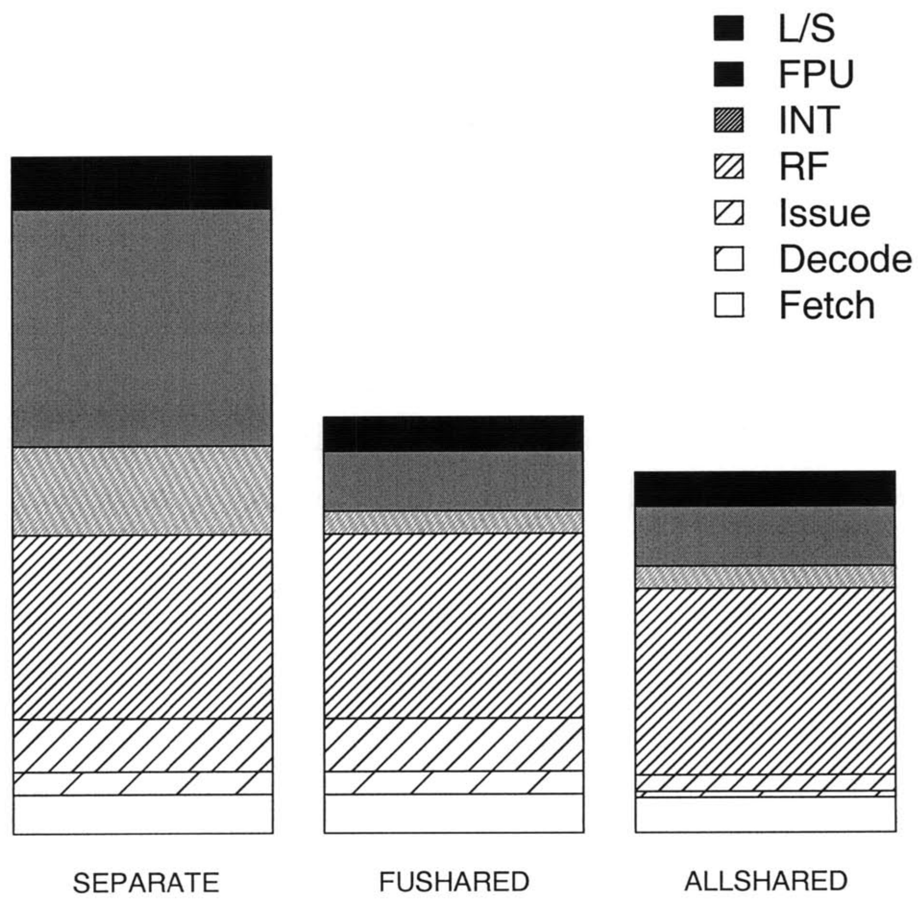


Figure 3-5: Core area of SEPARATE, FUSHARED, ALLSHARED.

uration to the FUSHARED configuration. Sharing the front end pipeline results in a much smaller area savings, because the front end area is relatively small compared to the register file and function units.

Figure 3-6 compares the energy efficiency of each configuration on the different workloads, measured in instructions per picojoule(IPpJ). The ALLSHARED configuration is always worse than the FUSHARED configuration. The SEPARATE configuration is better than the FUSHARED configuration for the IDOTLU4 and MIX workloads.

Figure 3-7 shows a plot of where the configurations are in the performance energy-efficiency space for each workload. The vertical axis shows increasing performance. The horizontal axis shows increasing energy efficiency. Lines connect points from the same configuration. The best configurations will have greater performance and greater energy efficiency, hovering near the upper right corner of the graph.

For every workload, the FUSHARED configuration has both greater performance and energy efficiency than the ALLSHARED configuration. The FUSHARED configuration performance never exceeds the SEPARATE performance, but for all workloads except IDOTLU4 and MIX the FUSHARED configuration has a better energy efficiency than the SEPARATE configuration, tending to take up space down and to the right of the SEPARATE configuration on the plot.

3.5 Improving FUSHARED

We see the FUSHARED configuration is a potentially interesting alternative to the SEPARATE configuration because for many workloads the energy efficiency of the FUSHARED configuration exceeds that of the SEPARATE configuration. Is there something we can do to improve the performance of the FUSHARED configuration to bring it closer in line with the performance achieved in the SEPARATE configuration?

Figure 3-8 shows the average function unit utilization in the FUSHARED configuration. The integer function unit has great utilization, especially for the integer workloads. This is exactly what we hoped to see by having threads share function

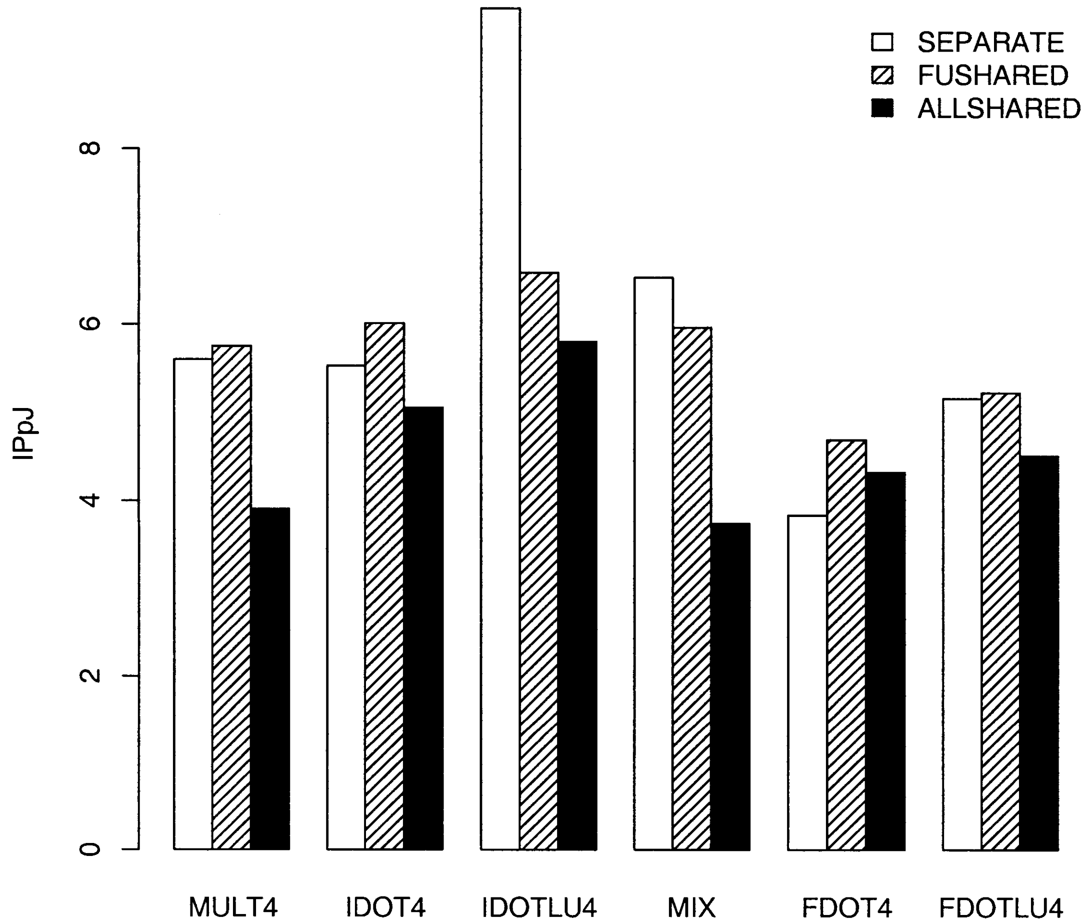


Figure 3-6: Energy efficiency of SEPARATE, FUSHARED, ALLSHARED.

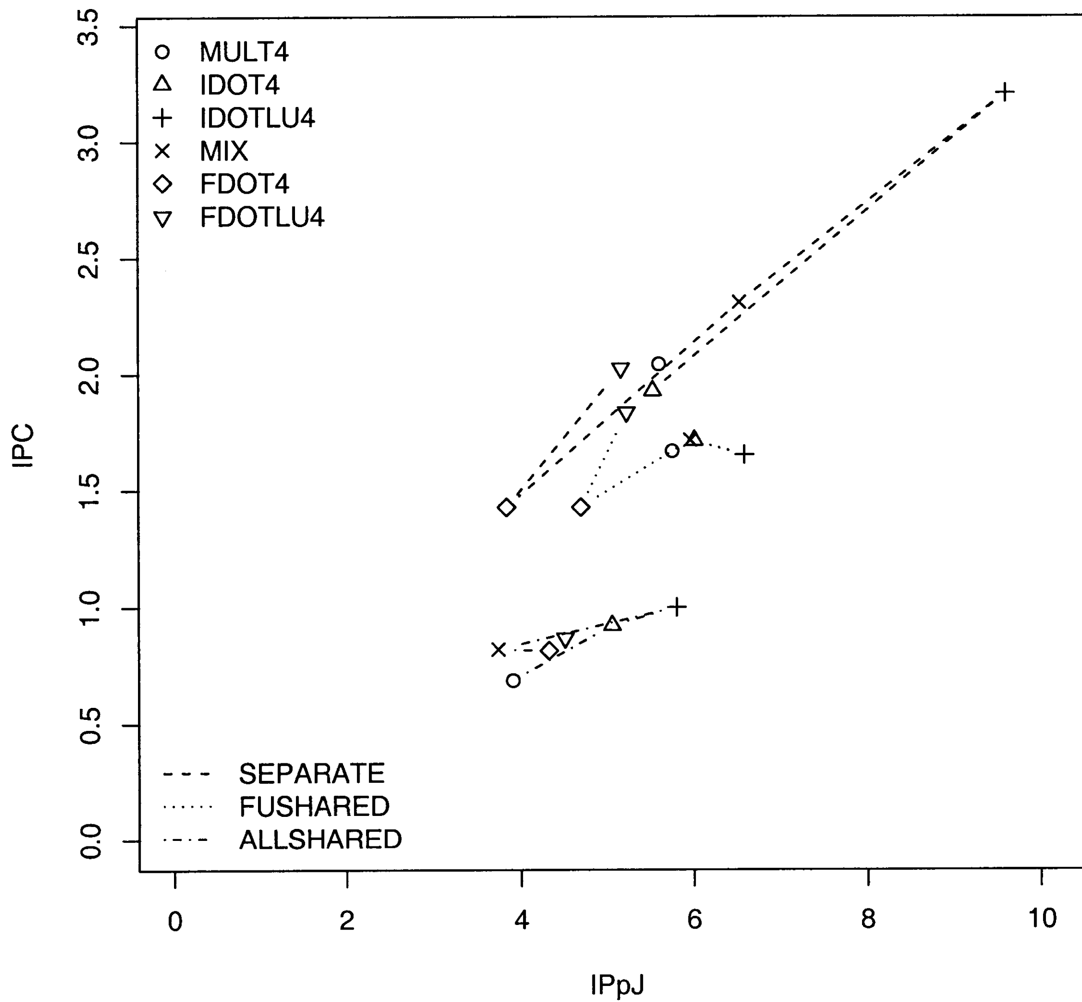


Figure 3-7: IPC IPJ plot of SEPARATE, FUSHARED, ALLSHARED.

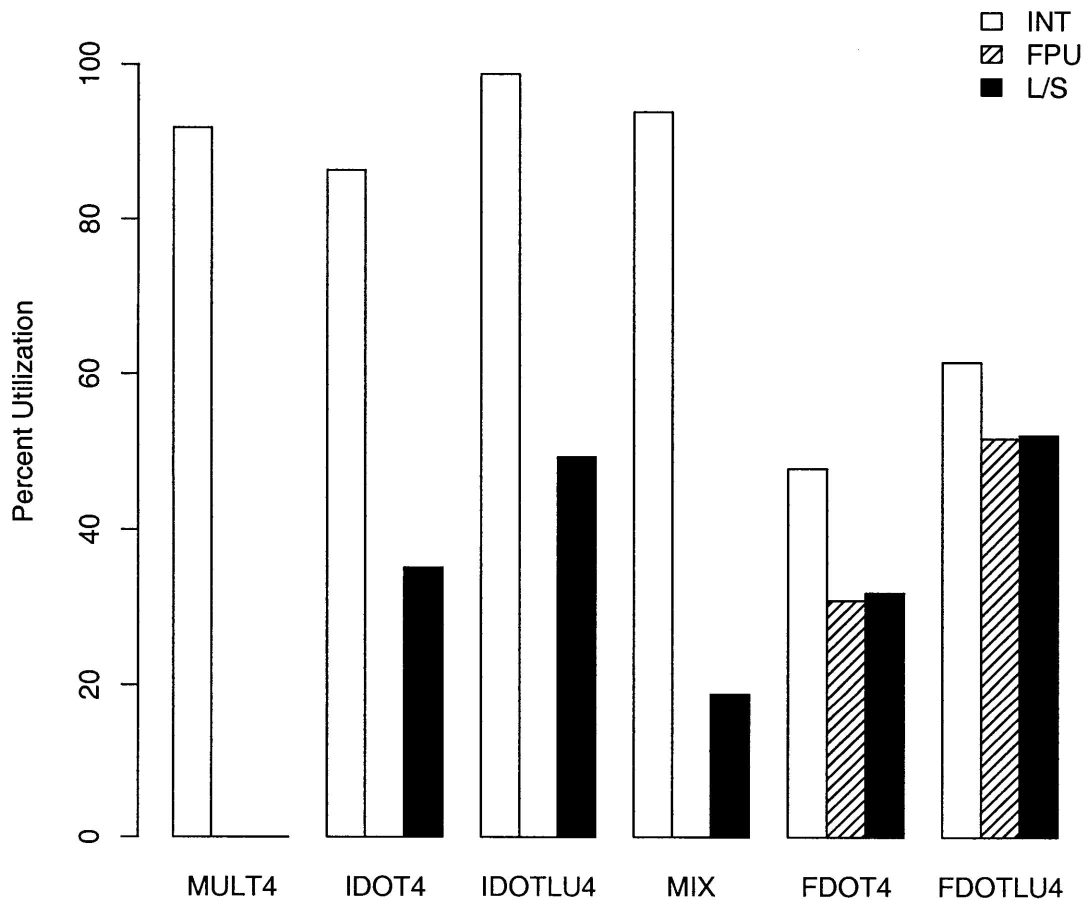


Figure 3-8: FUSHARED function unit utilization.

units. The utilizations are much better than that for the SEPARATE configuration shown in 3-9. The trouble is, the integer function units are being over-utilized.

Figure 3-10 shows the average number of structural conflicts in a cycle for each function unit. A conflict is counted for each thread with a ready instruction that is not executed only because an instruction from a different thread is executed instead. In each cycle there can be at most 3 conflicts. In the SEPARATE configuration there are no conflicts, because each thread has exclusive access to its own function units.

The FUSHARED configuration has a large number of conflicts over its single integer function unit for the integer workloads. The most conflicts are seen in the IDOTLU4 workload, which was one workload where the SEPARATE configuration was clearly better than the FUSHARED configuration. The FDOT4 workload, where FUSHARED does as well as the SEPARATE configuration, has almost no conflicts, neither for the integer nor floating point function units.

Clearly another integer function unit would be beneficial in easing the contention for the sole integer function unit in the FUSHARED configuration. The low number of conflicts for the floating point and load-store function units suggest there is no need to include additional floating point or load-store function units.

This suggests an improved FUSHARED configuration to look at, similar to the FUSHARED configuration except with 2 integer function units instead of one. We will call that the FUSHARED2 configuration. It is shown in figure 3-11.

3.6 Improving ALLSHARED

The ALLSHARED configuration does not perform as well as the FUSHARED configuration in terms of performance or energy efficiency. The way the configuration is setup, at most a single instruction can be issued each cycle, capping the overall IPC at 1. This limit brings down our function unit utilization to only a little better than in the SEPARATE configuration, as shown in figure 3-12.

Because for the most part the SEPARATE and FUSHARED configurations do not exceed an IPC of 2, we introduce another configuration, ALLSHARED2, with two

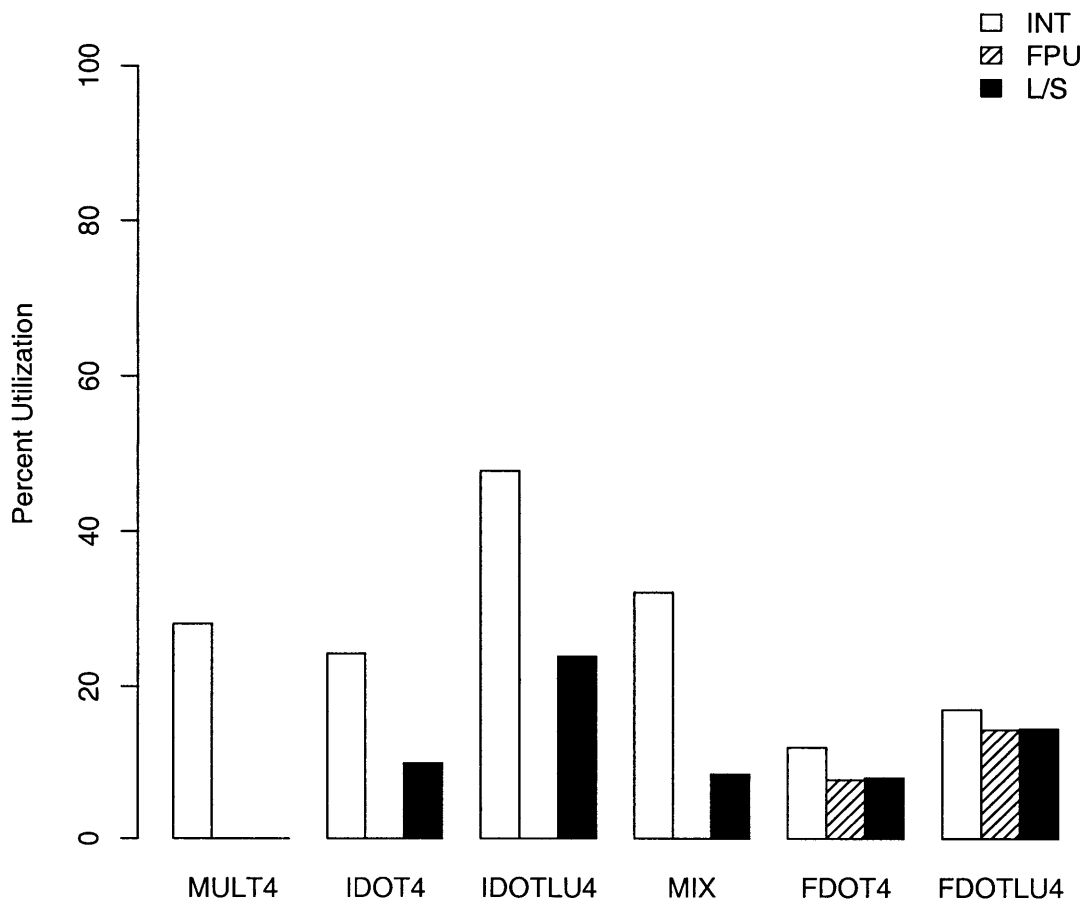


Figure 3-9: SEPARATE function unit utilization.

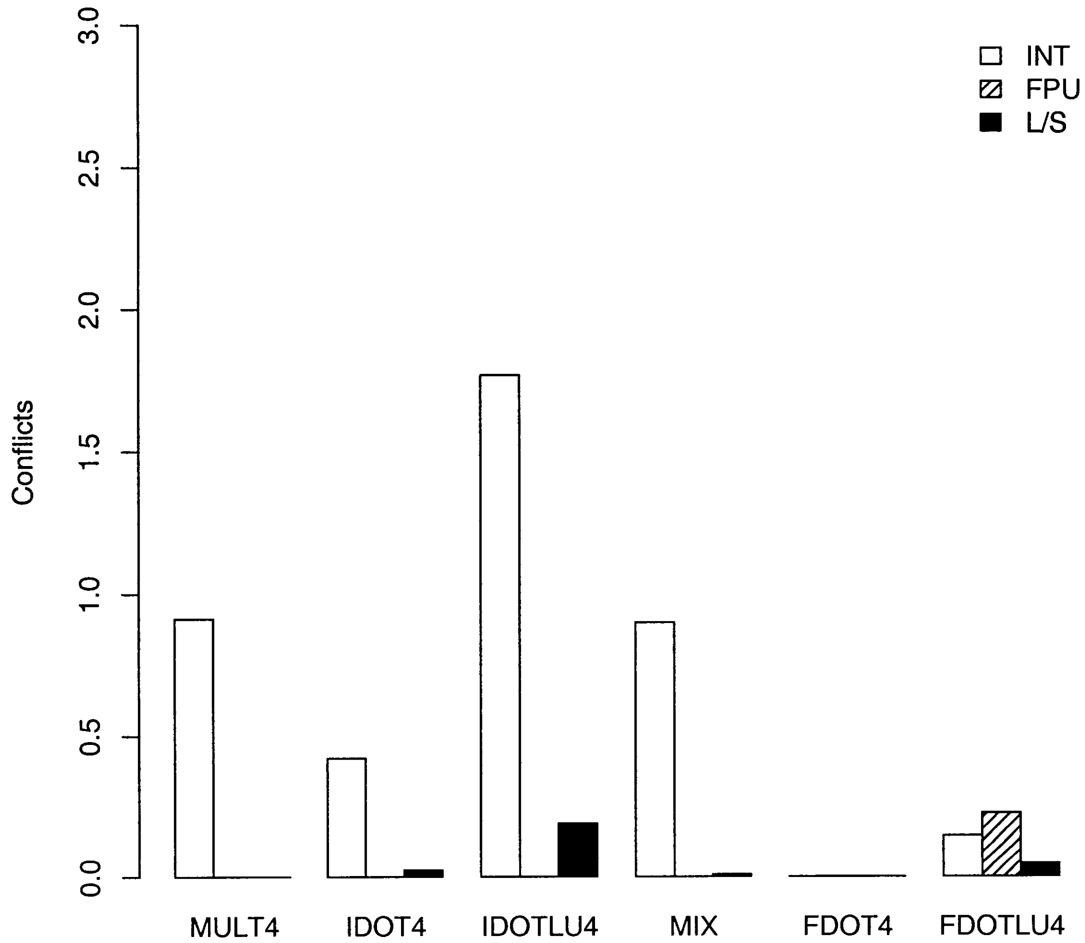


Figure 3-10: FUSHARED function unit conflicts.

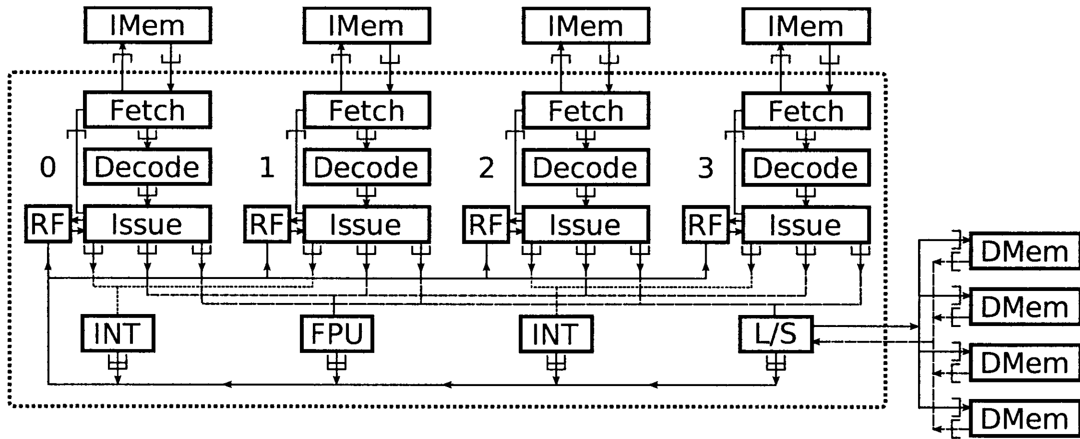


Figure 3-11: FUSHARED2 configuration.

pipelines, each completely shared by just 2 threads. The ALLSHARED2 configuration is shown in figure 3-13. The performance limit of ALLSHARED2 is an IPC of 2, which is hopefully enough to increase its performance to be competitive against the other configurations while still sharing enough logic to save energy.

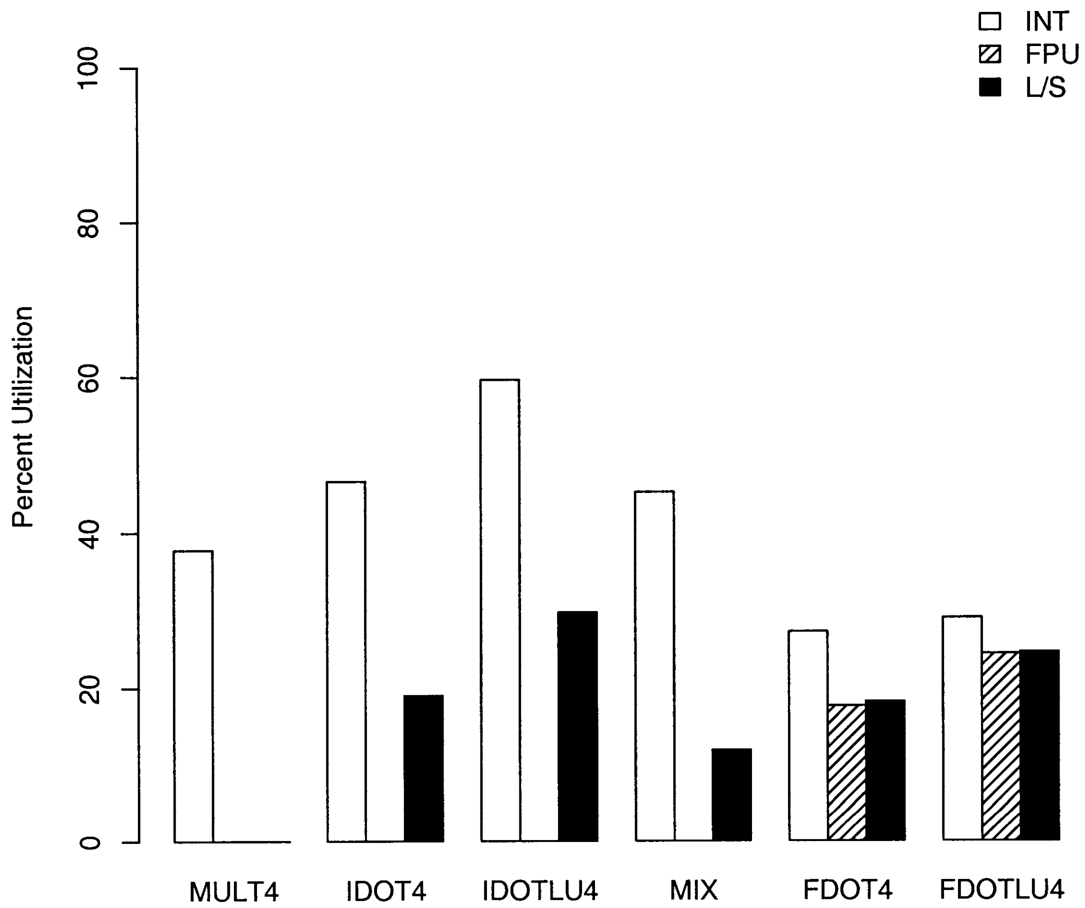


Figure 3-12: ALLSHARED function unit utilization.

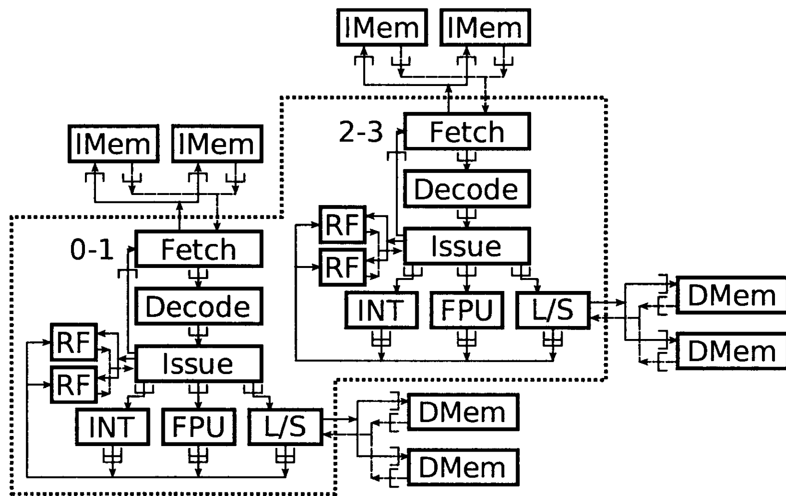


Figure 3-13: ALLSHARED2 configuration.

Chapter 4

Expanded Configurations

In chapter 3 we looked at some initial multithreaded configurations and how well they performed overall. Here we look at the results of adding the new FUSHARED2 and ALLSHARED2 configurations, and we discard the unsuccessful ALLSHARED configuration.

Figure 4-1 shows the performance for each of the configurations on our workloads. We see that the FUSHARED2 has better performance than FUSHARED in all cases, approaching the performane of the SEPARATE configuration. The ALLSHARED2 configuration, while achieving an IPC greater than 1, still fails to match the performance of the initial FUSHARED configuration except on the IDOTLU4 workload, where it approaches its limit of 2 IPC.

Table 4.1 shows the load balance for the MIX workload. With less contention for the integer function unit, FUSHARED2 gives less preference to the highest priority thread under our static arbitration policy for the function units. As before, the speedups give similar results as the IPC for the MIX workload.

	MULT	IDOT	MULT	IDOTLU	speedup
SEPARATE	22204	20879	22204	34713	4.02
FUSHARED	29650	25906	25410	19035	3.19
FUSHARED2	23701	22342	22383	31576	3.80
ALLSHARED2	19700	27160	21631	31510	2.62

Table 4.1: Load balance of MIX workload of expanded configurations.

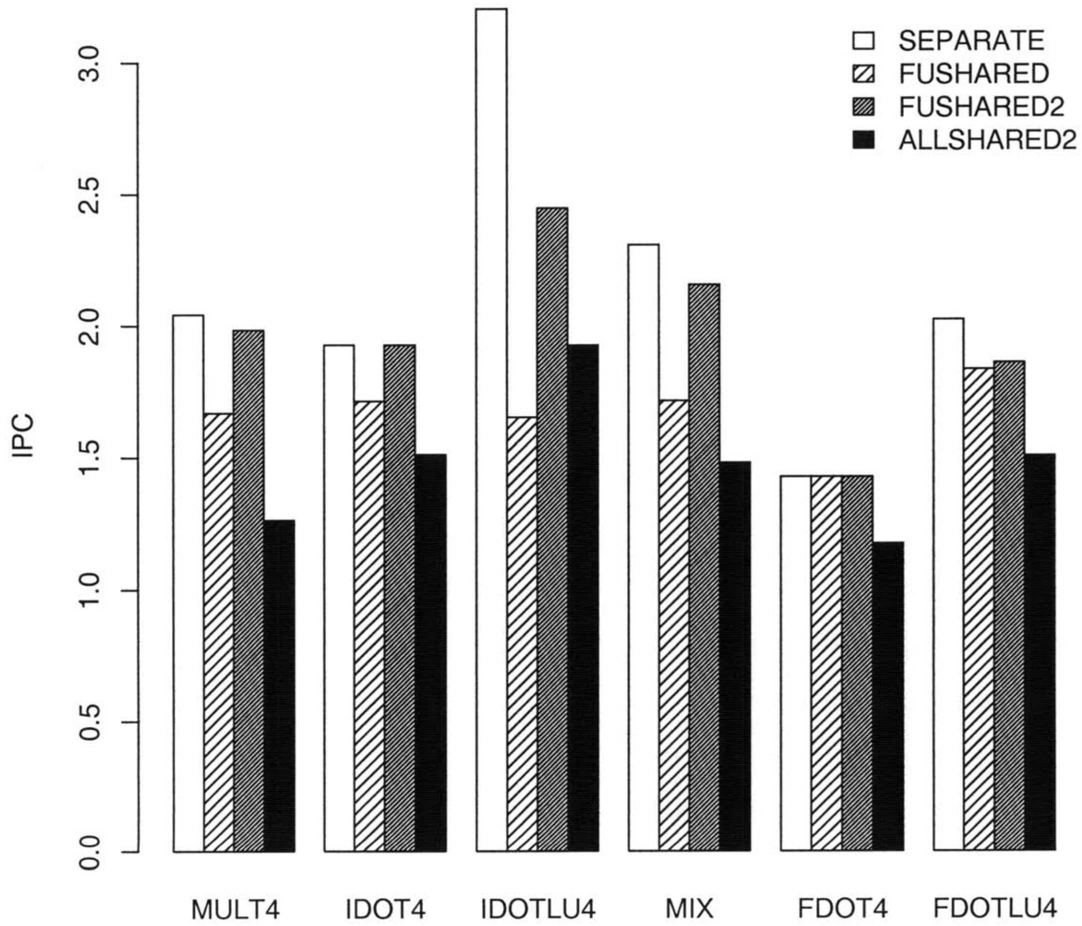


Figure 4-1: Performance with FUSHARED2, ALLSHARED2 configurations.

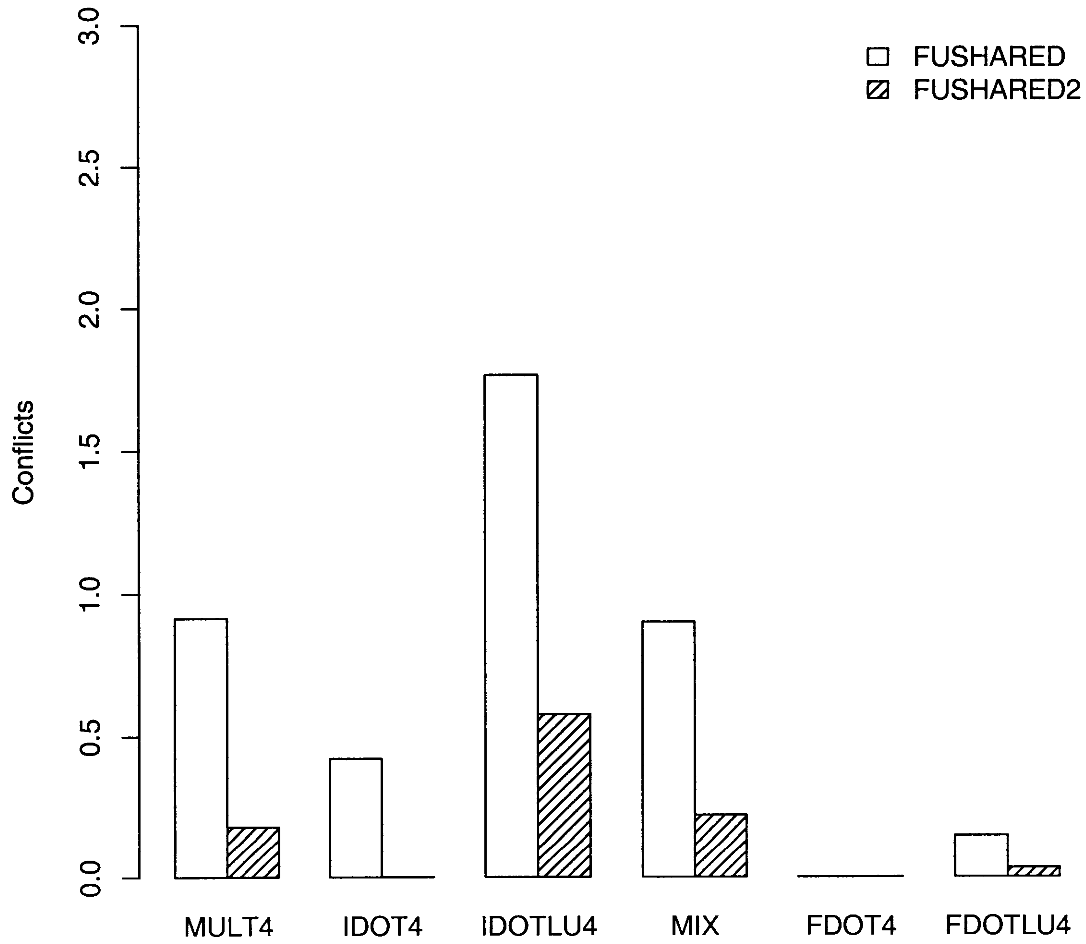


Figure 4-2: FUSHARED2 function unit conflicts.

Figure 4-2 compares the average conflicts over integer function units in the FUSHARED and FUSHARED2 configurations. The FUSHARED2 configuration has many fewer conflicts, averaging no more than about a conflict every other cycle, compared to the FUSHARED configuration, which averaged as much as a conflict every cycle.

Along with the decrease in conflicts over the integer unit in the FUSHARED2 configuration, there is a small increase in the conflicts for the load-store unit on the IDOTLU4 and FDOTLU4 workloads. This is not shown in the figure.

Figure 4-3 shows the area of each configuration, broken down by component. The FUSHARED2 configuration is only slightly larger in area than the FUSHARED configuration, the added area coming from the additional integer function unit, which is relatively small. The ALLSHARED2 configuration is a bit larger than the FUSHARED configurations, because it duplicates all of the function units, including the floating point unit, which takes up a large fraction of the core.

Energy efficiency is shown in figure 4-4. The FUSHARED2 configuration has a greater energy efficiency than the FUSHARED configuration in every case, and exceeds the energy efficiency of the SEPARATE configuration in all but the IDOTLU4 workload. The energy efficiency of the ALLSHARED2 configuration is more variable. ALLSHARED2 appears to have better energy efficiency on the more optimized workloads.

Figure 4-5 shows a plot comparing the performance and energy efficiency of the two FUSHARED configurations. The FUSHARED2 configuration has better overall performance in all cases, and especially so for those integer workloads which overloaded the FUSHARED configuration's single integer function unit.

Figure 4-6 plots the SEPARATE, FUSHARED2 and ALLSHARED2 configurations. The FUSHARED2 configuration is shifted to the right from the SEPARATE configuration in all workloads except for IDOTLU4, so while it still does not achieve the performance of the SEPARATE configuration, the energy efficiency is mostly better.

The ALLSHARED2 configuration is never as good as the FUSHARED2 configuration in terms of performance or energy efficiency. This suggests sharing the front end

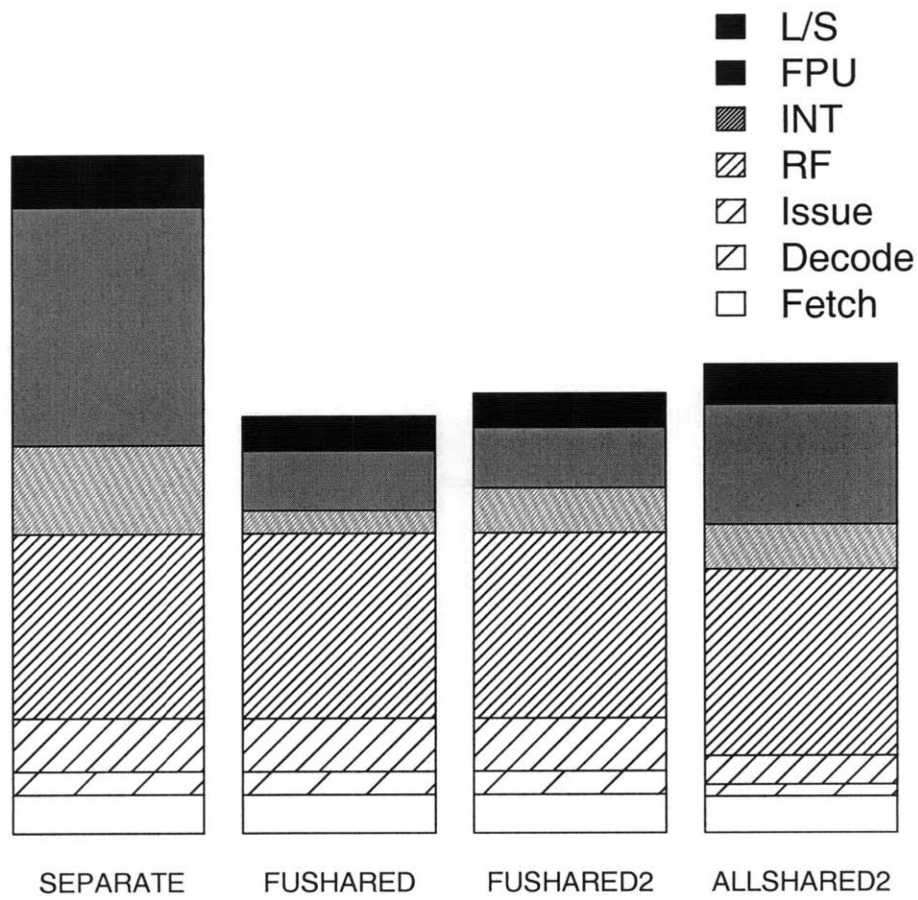


Figure 4-3: Area with FUSHARED2, ALLSHARED2 configurations.

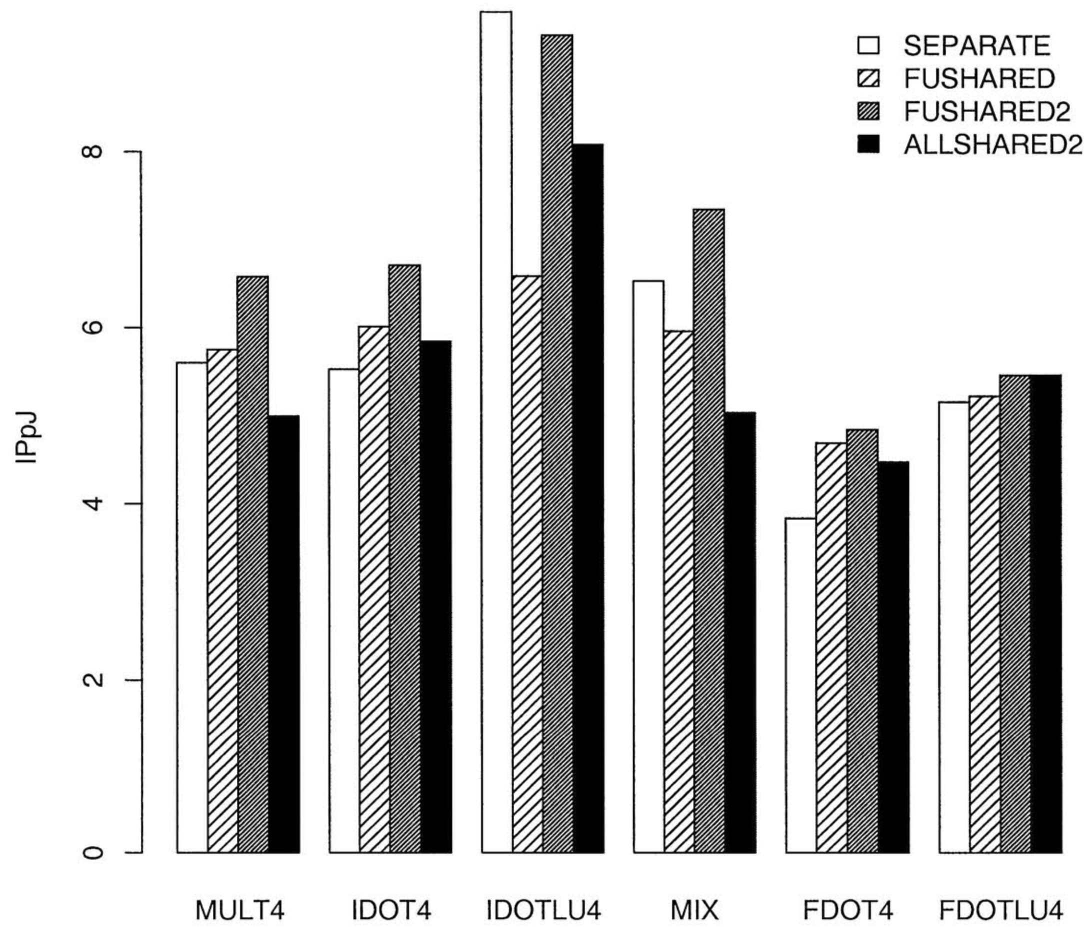


Figure 4-4: Energy efficiency with FUSHARED2, ALLSHARED2 configurations.

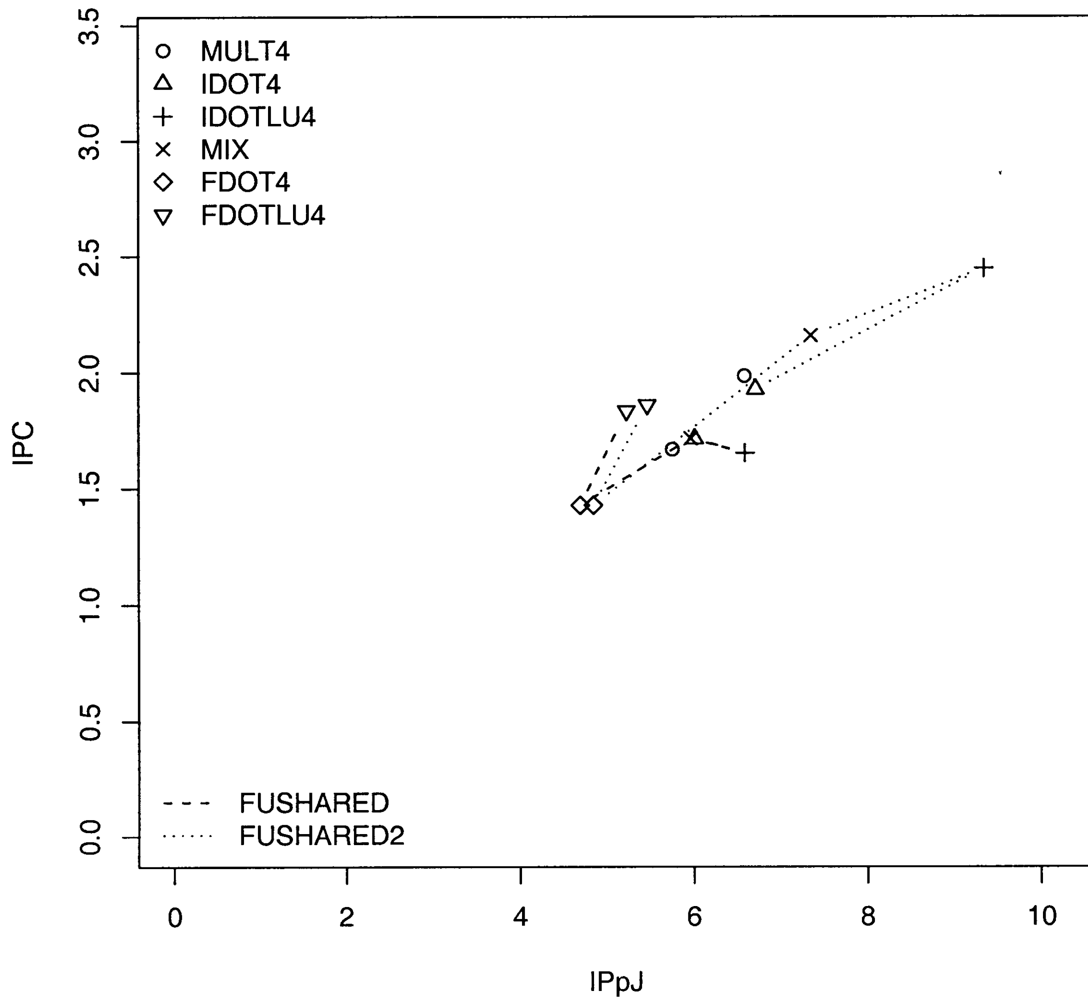


Figure 4-5: IPC IPJ plot of FUSHARED, FUSHARED2.

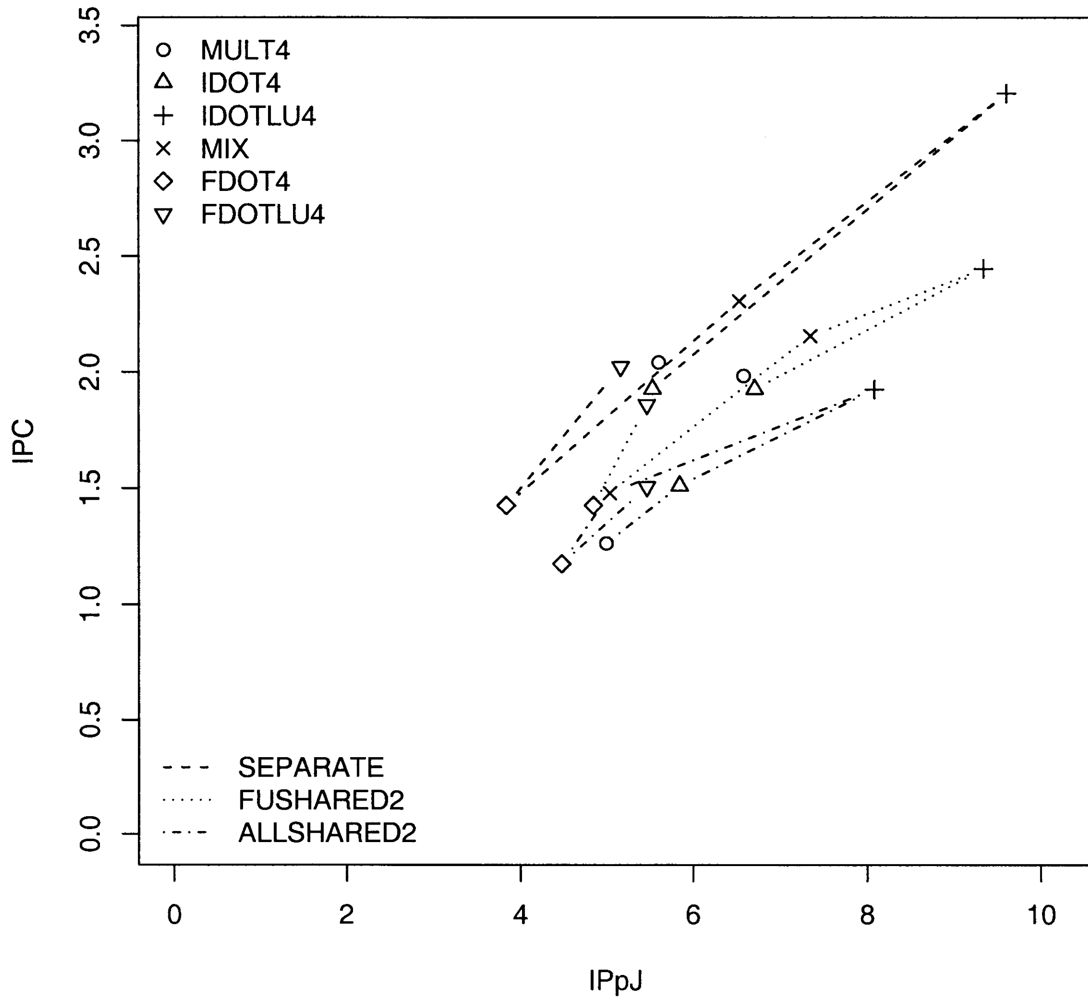


Figure 4-6: IPC IPJ plot of SEPARATE, FUSHARED2, ALLSHARED2.

pipeline in the simple round robin fashion we use for the ALLSHARED configurations does not improve energy efficiency. Perhaps we would get better energy efficiency if we chose the number of each function unit in the ALLSHARED configuration to match that of the successful FUSHARED2 configuration, but that would result in a reduction of the number of function units, which could only degrade performance. Given how poor the performance of the ALLSHARED configuration already, it is doubtful that configuration can be competitive with the SEPARATE and FUSHARED2 configurations.

Chapter 5

Arbitration for Function Units

We saw in previous chapters a big factor in whether or not having multiple threads share a function unit improves overall performance is the number of conflicts for the function unit. If there are too many conflicts, the performance is degraded, overpowering the power saved by sharing the function unit logic.

In chapter 3 we mentioned that priority for the use of a function unit in case of conflict is assigned statically to the thread with lowest identifier. But there are different policies we could use for arbitrating use of the shared function units, and these different policies may have an impact on the overall performance of the machine.

We look at three different arbitration policies for determining which thread gets to use a function unit when there is conflict. The first is STATIC, the policy we have been using in our FUSHARED and FUSHARED2 configurations so far. If multiple threads want to use the same function unit in the same cycle, the thread with the smallest identifier will be allowed to use the function unit and the other threads will have to wait.

We can imagine such a static priority to be unfair, favoring whatever program is running in the highest thread slot at the expense of the others, perhaps hurting the overall performance of the system. We have seen a little of this in the load balancing of the MIX workload for the FUSHARED configuration. To combat this unfairness, we look at a round robin arbitration policy, called ROUNDROBIN, where every cycle priority is assigned to the next thread in round robin fashion. If the thread with

	MULT	IDOT	MULT	IDOTLU	SPEEDUP
STATIC	23701	22342	22383	31576	3.80
ROUNDROBIN	22836	20922	22834	33409	3.84
LASTUSED	22787	18952	22785	35476	3.73

Table 5.1: Load balance of MIX workload under different arbitration.

priority on a given cycle wants to use the function unit, it will use the function unit. If there are multiple threads competing for the function unit, none of which currently have the priority slot, we fall back on our static priority system.

Later, when we introduce operand bypassing into the configurations, we will see another possible arbitration policy which is convenient to implement, LASTUSED. In the LASTUSED policy priority is given to whichever thread most recently used the function unit.

Figure 5-1 shows the performance for the different arbitration policies on the FUSHARED2 configuration. It appears the arbitration policy has little impact on the performance of the system except for in the IDOTLU4 workload, where static priority reduces performance.

Table 5.1 shows the thread load balance on the configurations with different arbitration policies for the MIX workload. Once again what they show mirrors that of the MIX IPC.

The different arbitration policies for use of the function units do not have a noticeable impact on the area of the configuration. The ROUNDROBIN and LASTUSED configurations increase area by less than one percent over the STATIC configuration.

Figure 5-2 shows that in the FUSHARED2 configuration, the ROUNDROBIN and LASTUSED arbitration policies have worse energy efficiency than STATIC arbitration.

When using a configuration designed with enough function units not to overload the function units, the number of conflicts for the function units are small. This means different arbitration policies do not have such a large impact on performance. More complex arbitration policies do consume more energy, so it makes sense to choose the simplest arbitration policy to get the best overall performing configuration.

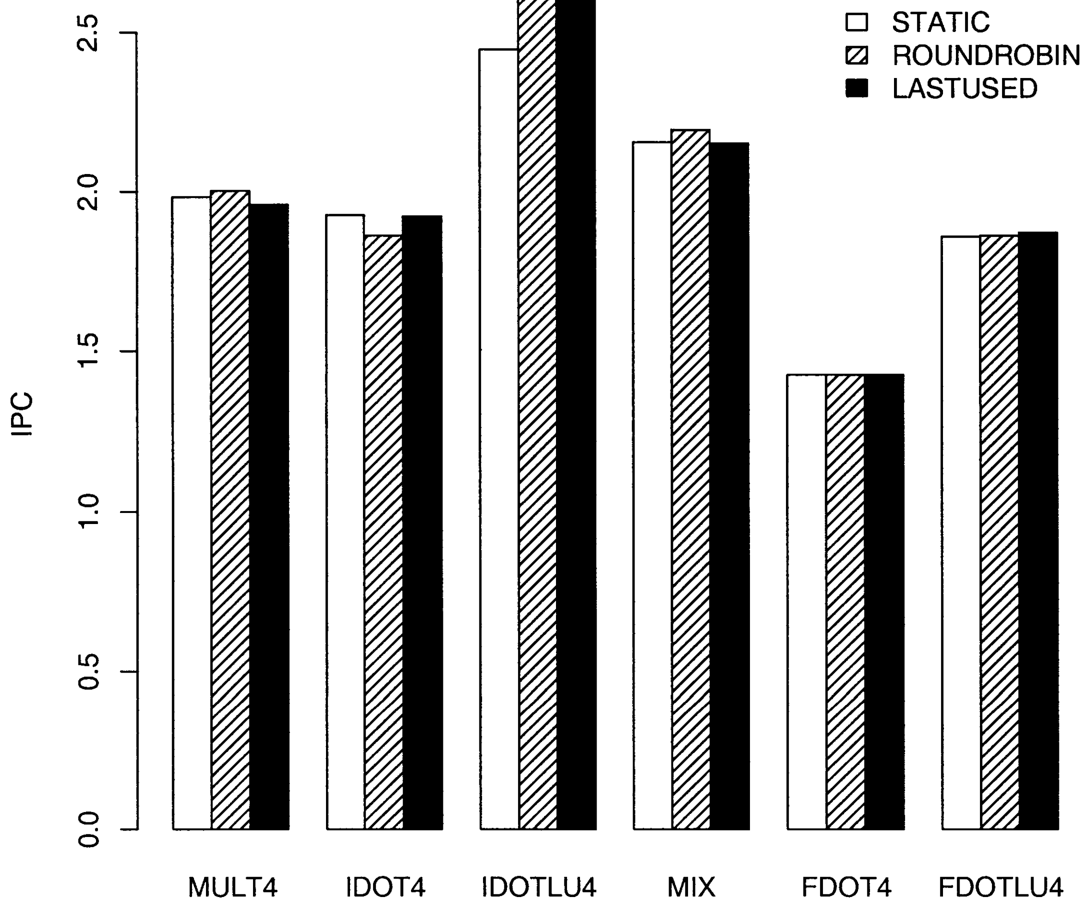


Figure 5-1: Performance of FUSHARED2 using different function unit arbitration.

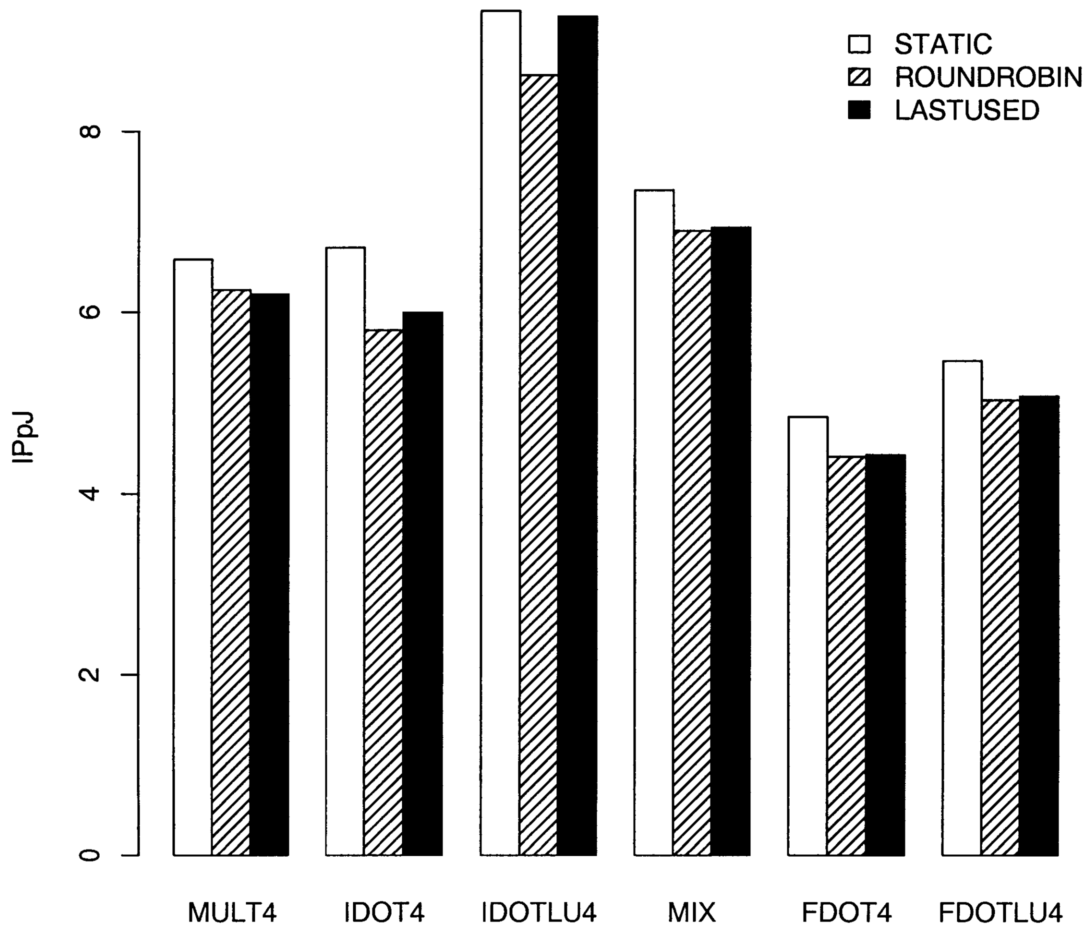


Figure 5-2: Energy efficiency of FUSHARED2 using different function unit arbitration.

Chapter 6

Operand Bypassing

In this chapter we look at how operand bypassing affects the performance and energy efficiency of our configurations.

We implement operand bypassing for the integer function units and bypassing of the condition code register.

6.1 Integer Bypassing

It is common for an instruction to depend on the result of the instruction immediately before it. For example, consider the assembly code in figure 6-1.

The add instruction depends on the value in register `r0` computed by the previous instruction. Normally this would cause a two cycle bubble in our pipelines. The add instruction can not be issued until the `r0` register has been written back to the register file. Instead of issuing the add instruction the cycle after the multiply instruction is issued, it must wait a cycle for the multiply instruction to execute, then wait another cycle for the multiply instruction to write back to the register before it can issue the add instruction on the following cycle.

```
IMultRes r4, r3 -> r0
IAddRes r0, r6 -> r0
```

Figure 6-1: Example of back to back dependent instructions.

This 2 cycle bubble can be eliminated with operand bypassing. We add a path from the output of the integer function unit back to its input. Now when the issue stage goes to issue the add instruction and finds some of the operands are not ready, `r0` in this case, instead of giving up on issuing that cycle it looks at the instruction currently executing in the integer function unit to see if the destination of that instruction matches the missing operand. If the destination does match the missing operand, the issue stage can go ahead and issue the add instruction on that cycle knowing the operand will be ready by the time the add instruction is executed.

The integer function unit is modified to accept bypassed operands. Instead of having the physical operand available, the operand is marked as bypassed, which means the function unit will use its most recently computed value for that operand.

The consequence of this bypassing is the add instruction can be issued the cycle immediately following the issue of the multiply instruction; no cycles are wasted.

This implementation of operand bypassing works as is for a function unit dedicated to a single thread, but difficulties arise when a function unit is shared by multiple threads. We need a mechanism to ensure operands will only be bypassed to the thread they belong to. The issue stage needs to know what cycle a result will be available, and there is limited space to save computed results in the function unit itself (that is what the register file is for).

To make sure bypassing still works for function units shared by multiple threads we require the function unit give priority to the thread which most recently had an instruction executed in that function unit. This is equivalent to using the LASTUSED schedule priority discussed in chapter 5.

To see why this arbitration policy allows bypassing to work for shared function units, consider again the code in figure 6-1. The issue stage issued the multiply instruction the previous cycle and is now considering whether it can issue the add instruction. It finds the operand `r0` is not ready yet, but could potentially be bypassed from the multiply instruction. If the multiply instruction is not executed this cycle, because perhaps another thread has priority for the function unit, there is no place for the add instruction to be issued to, so the issue stage has to wait for the next

```
ISubCnd r2, 16
BrCmpPos 21
```

Figure 6-2: Example of condition code dependency.

cycle to issue anyway. If the multiply instruction is executed this cycle, this thread is guaranteed to have priority for the function unit for the next cycle, so the issue stage can issue the add instruction knowing it will be executed the cycle following the execution of the multiply instruction, and the operand `r0` can be bypassed from the previous result. If we did not have the scheduling restriction, the issue stage would not know whether it could issue the add instruction, because an instruction from a different thread might be scheduled between the multiply and add instructions, overwriting the bypass operand. If the add instruction followed that, it would get data from a different thread which is not correct.

We do not perform this sort of bypassing for the floating point or load-store function units because those operations take multiple cycles to execute. A different form of bypassing would be needed which allows instructions to be issued on the last cycle of their predecessor's execution or allows the issue stage to read operands in the same cycle they are written back to the register file.

6.2 Condition Code Bypassing

As mentioned briefly in section 2.1, the Fresh Breeze architecture uses a condition code register to indicate which path of a branch should be taken. This register can be bypassed similar to the way the general purpose registers can be bypassed.

Figure 6-2 shows a common sequence of instructions which demonstrates the advantage of bypassing the condition code register. The subtract instruction sets condition code register based on the difference between the value in register `r2` and 16. The branch instruction reads the condition code register to determine if it should branch or not.

Without bypassing there is a two cycle bubble between execution of the subtract

	MULT	IDOT	MULT	IDOTLU	SPEEDUP
SEPARATE	22204	20879	22204	34713	4.02
SEPARATE_BY	21591	23422	21591	33396	4.56
FUSHARED2	23701	22342	22383	31576	3.80
FUSHARED2_BY	22142	21538	22142	34179	4.24

Table 6.1: Load balance of MIX workload with bypassing.

instruction and the branch instruction. The subtract instruction takes a single cycle to execute, then another cycle to write back the condition code register before the branch can read the condition code register.

We can save a single cycle by allowing the branch instruction to read the new value of the condition code register in the same cycle the register is updated.

It is harder to remove the remaining single cycle bubble. Unlike the case for integer bypassing, the branch instruction is executed a stage earlier in the pipeline than the previous instruction. That single cycle bubble comes because the branch instruction is blocking the following instruction from being issued while waiting for the condition codes. For this reason we only perform the simple condition code bypassing.

6.3 Results

Figure 6-3 compares the performance of the SEPARATE and FUSHARED2 configurations with and without bypassing. Both the SEPARATE and FUSHARED2 configurations have improved performance with bypassing.

Figure 6-4 compares the area of the configurations with and without bypassing. Bypassing results in a small increase in the area of the integer function units.

Figure 6-5 compares the energy efficiency of the configurations with and without bypassing. The energy efficiency of the SEPARATE configuration improves with the addition of bypassing, but the energy efficiency of the FUSHARED2 configuration degrades with the addition of bypassing for all but the MIX workload.

The degradation in energy efficiency under the FUSHARED2 configuration appears to be partly due to the decreased energy efficiency resulting from requiring the

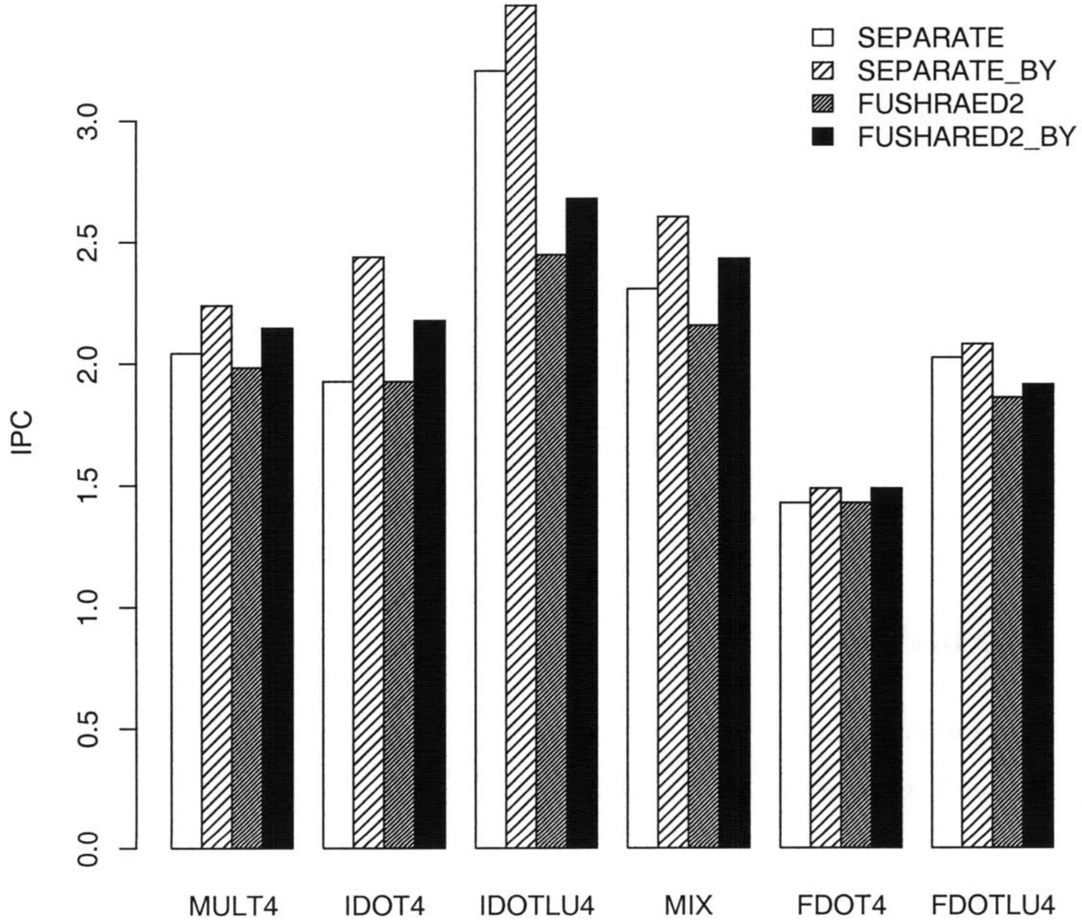


Figure 6-3: Performance of SEPARATE and FUSHARED2 with bypassing.

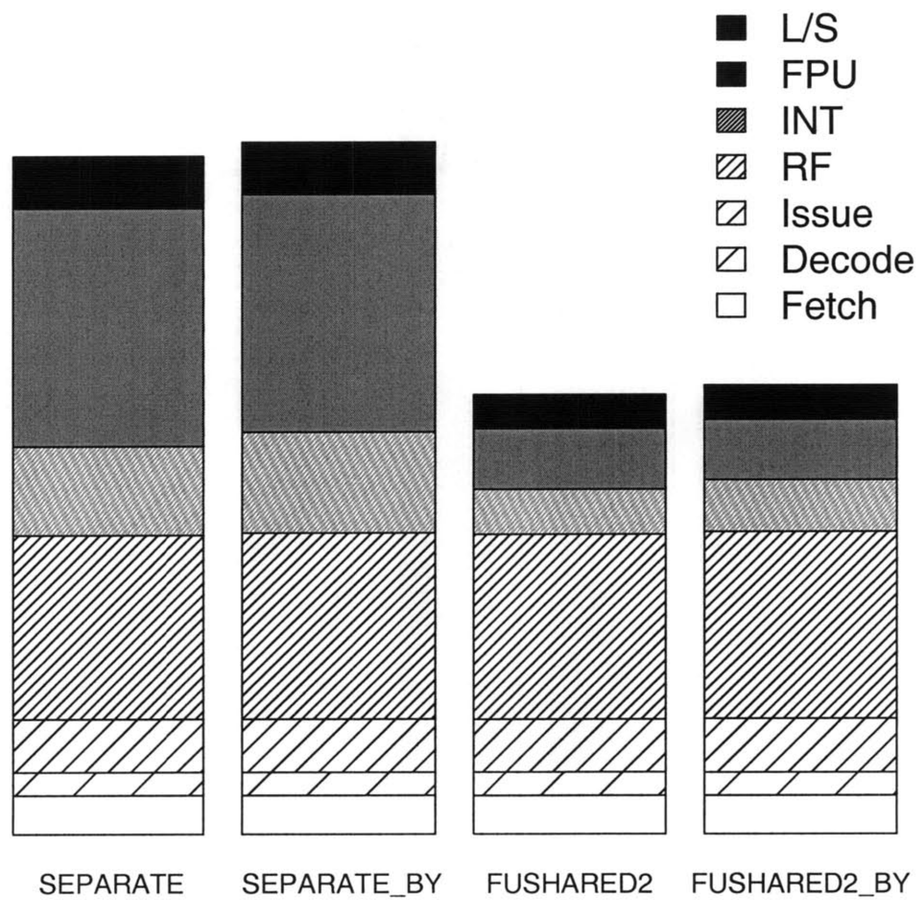


Figure 6-4: Area of SEPARATE and FUSHARED2 with bypassing.

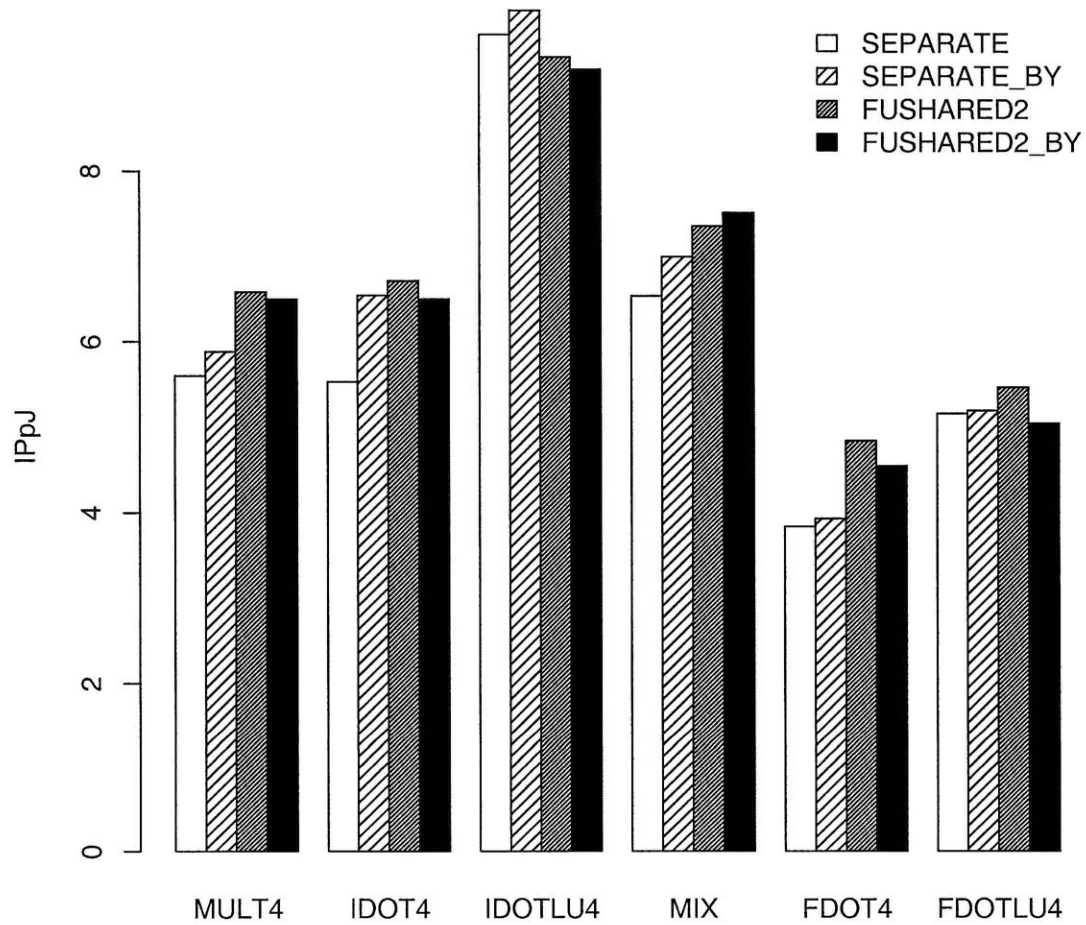


Figure 6-5: Energy efficiency of SEPARATE and FUSHARED2 with bypassing.

LASTUSED arbitration policy over the static arbitration policy. The drop in energy efficiency mirrors that of the drop from STATIC to LASTUSED shown in figure 5-2 back in chapter 5.

Figure 6-6 compares the overall performance of the SEPARATE configuration with and without bypassing. There is a clear shift toward the upper right corner of the graph when adding bypassing, suggesting bypassing is beneficial for both performance and energy efficiency in a configuration where function units are not shared.

Figure 6-7 compares the FUSHARED2 configuration with and without bypassing. In contrast to the SEPARATE configuration, when adding bypassing to the FUSHARED2 configuration it looses energy efficiency, shifting up and to the left.

Bypassing clearly is advantageous for the single threaded case as seen in the overall performance improvement in the SEPARATE configuration when bypassing was added. It is not as obvious, however, that bypassing makes sense for function units that are shared among threads. There are two factors detracting from the bypass overall performance improvement when threads share function units. The first is sharing threads already partially hides the bubbles bypassing aims to get rid of. It does not improve performance any to get rid of a bubble which was hidden anyway. The second is the added complications in ensuring operands are bypassed from the appropriate thread. In our implementation bypassing required us to use a more complicated arbitration policy for the function units, which we saw from chapter 5 has a significant impact on energy efficiency.

Figure 6-8 shows a plot on the performance energy-efficiency space of the SEPARATE configuration with bypassing and the FUSHARED2 configuration with and without bypassing. From the plot we see clearly how performance and energy efficiency can be traded off. The SEPARATE configuration with bypassing offers a higher performing, worse energy efficiency configuration. The FUSHARED2 configuration without bypassing has better energy efficiency at the cost of degraded performance, and the FUSHARED2 configuration with bypassing is between the two.

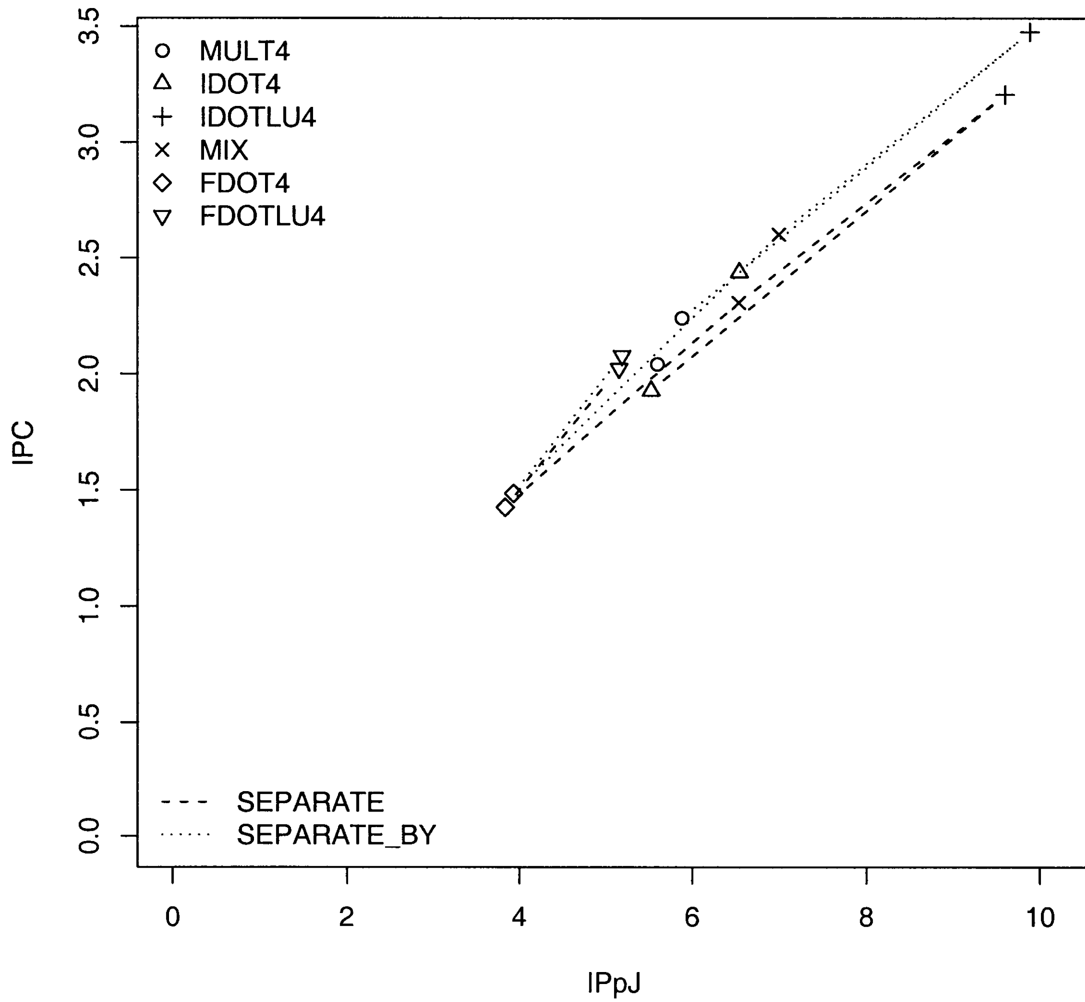


Figure 6-6: IPC IPJ plot of SEPARATE with and without bypassing.

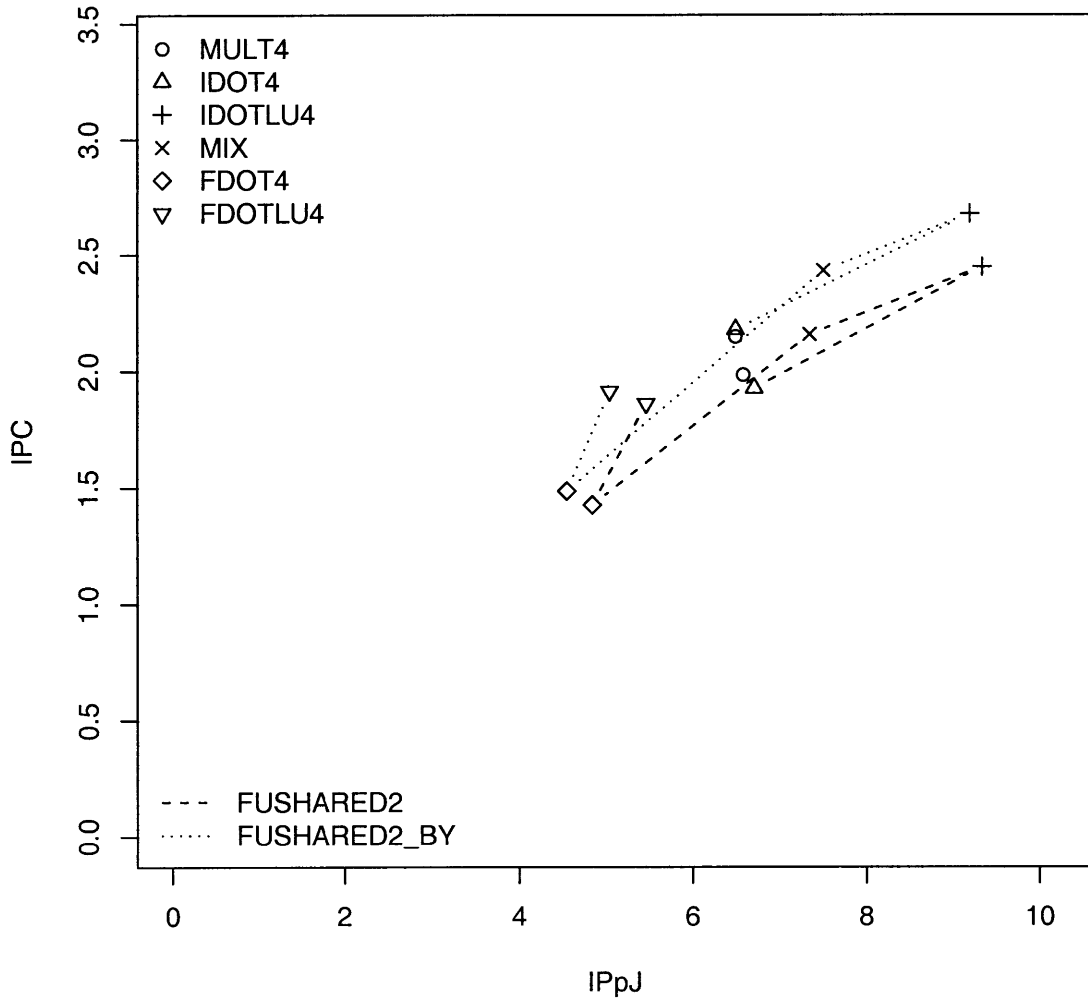


Figure 6-7: IPC IPJ plot of FUSHARED2 with and without bypassing.

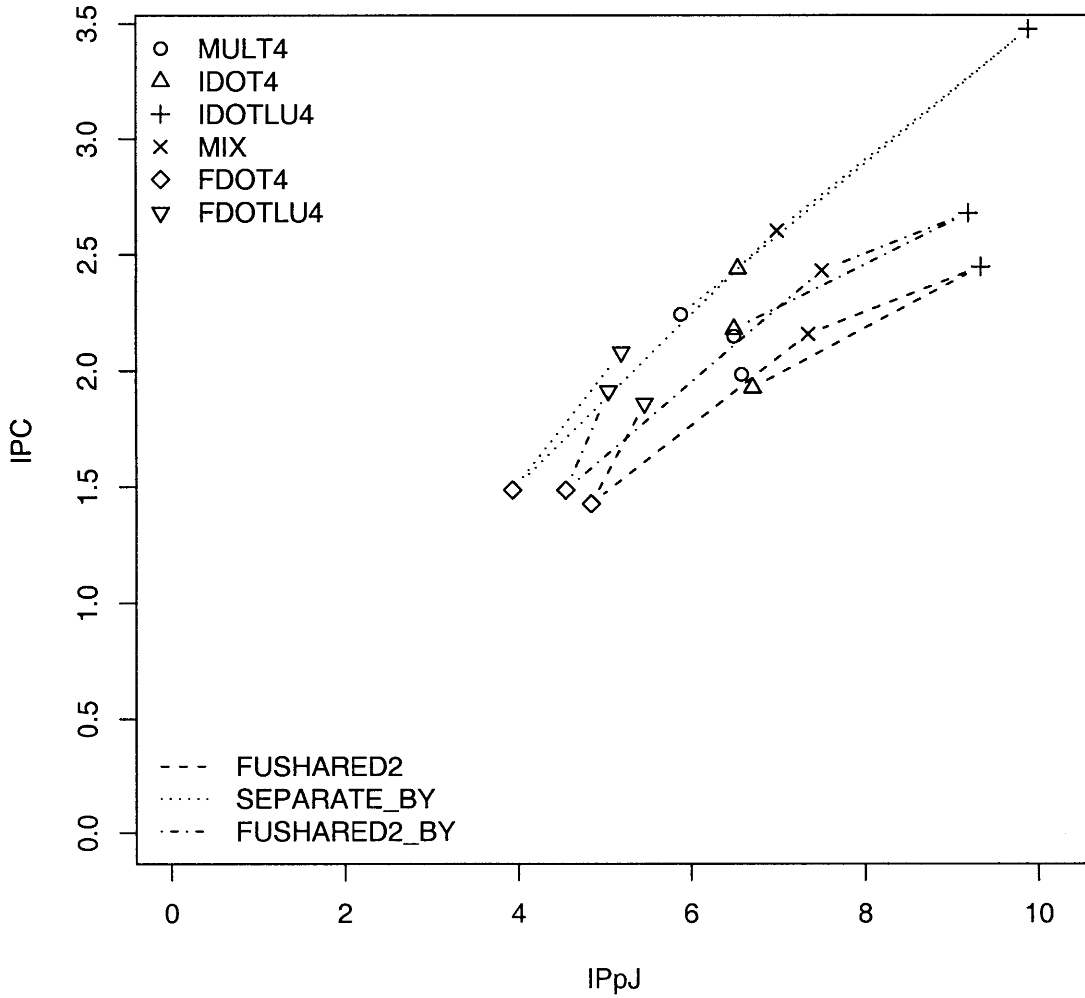


Figure 6-8: IPC IPJ plot of SEPARATE and FUSHARED2 with bypassing.

Chapter 7

Conclusion

This thesis looked at performance and energy efficiency in simple simultaneous multithreading processor cores. The previous focus on solely high performance processors is shifting to a focus on some combination of high performance and energy efficiency, both of which the technique of simultaneous multithreading has the potential to improve. When throughput is more important than single thread performance, simultaneous multithreading joined with simple scalar processors looks very attractive as a way to have high performance and energy efficiency.

To examine the performance and energy efficiency in simple simultaneous processor cores we implemented using a high level hardware description various multithreaded core configurations based on a common single-threaded scalar baseline configuration. With the aid of standard industry tools we augmented cycle accurate simulation with area and power estimates derived from layouts of the configurations.

We evaluated the overall performance of the configurations by plotting performance and energy efficiency together rather than combining them into a single figure of merit, allowing us to see how performance and energy efficiency can be traded off.

We focused on the processor core, modeling memory as having a small fixed latency and using benchmark programs with straight line code and branches.

Specifically we looked at

- How the performance and energy efficiency of multiple threads sharing function

units compares to that of duplicating multiple simple single thread pipelines, each with their own separate set of function units.

- Whether sharing the front end pipeline logic among threads leads to similar performance benefits as sharing function units.

- Under what circumstances a function unit should be shared by all threads, only some threads, or not at all.

- How different policies for arbitrating threads' use of function units affect performance and energy efficiency.

- How sharing function units interacts with the high performance optimization technique of operand bypassing, an obvious technique to apply in single thread architectures.

We found sharing function units among threads can improve energy efficiency over duplicating the function unit set for each thread, though the performance can at most match that of duplicating the function unit set for each thread. A good choice for the number of threads sharing a function unit ensures the function unit is not overloaded. Sharing the front end pipeline logic does not improve performance or energy efficiency over either duplicating the full pipeline or just duplicating the front end pipelines for each thread. Different arbitration policies for use of function units do not impact the performance much, but they do have a noticeable impact on the energy efficiency of the core, so the simplest arbitration policy should be used to maximize energy efficiency. Operand bypassing, an obvious optimization for a pipeline which does not share function units, is not obviously better when function units are shared, improving performance at the cost of reduced energy efficiency.

7.1 Future Work

This study focused on the performance and energy efficiency in the processor core. Interesting future work would be to model a more realistic memory hierarchy with

shared caches. Cache misses are a source of latency which can be partially hidden by simultaneous multithreading, bringing the performance of shared function units closer to that of separate thread configurations. A more realistic memory hierarchy may also put into better perspective the significance of the energy savings possible from sharing function units. If the energy consumed accessing the cache dwarfs that of duplicating a function unit, perhaps sharing function units makes less sense.

Introducing shared memory also complicates the fetch stage, potentially introducing alternative configurations which could be evaluated for their trade-offs in performance and energy efficiency as we have done for shared function units.

We have shown sharing function units alters the affect operand bypassing has on performance and energy efficiency. More interesting future work could be done evaluating how other optimization techniques change when threads share function units, techniques such as register renaming, out-of-order issue, and extended speculation. In this work we focused on the performance energy-efficiency trade-off of various configurations simple simultaneous multithreading cores. It would also be interesting to see how more complex cores compare.

Bibliography

[1] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. *IEEE Micro*, 20:26–44, 2000.

[2] James Burns and Jean-Luc Gaudiot. Quantifying the SMT Layout Overhead - Does SMT Pull Its Weight? In *International Symposium on High-Performance Computer Architecture*, volume 6, pages 109–120, 2000.

[3] Jason Cong, Ashok Jagannathan, Glenn Reinman, and Yuval Tamir. Understanding the Energy Efficiency of SMT and CMP with Multiclustering. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 48–53, New York, NY, USA, 2005. ACM.

[4] J.D. Davis, J. Laudon, and K. Olukotun. Maximizing CMP Throughput with Mediocre Cores. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 51–62, Sept. 2005.

[5] Jack B. Dennis. Fresh Breeze: A Multiprocessor Chip Architecture Guided by Modular Programming Principles. *SIGARCH Comput. Archit. News*, 31(1):7–15, 2003.

[6] A. Falcon, A. Ramirez, and V. Valero. A Low-Complexity, High-Performance Fetch Unit for Simultaneous Multithreading Processors. In *International Symposium on High-Performance Computer Architecture*, volume 10, pages 244–253, February 2004.

[7] R. Gonzalez and M. Horowitz. Energy Dissipation In General Purpose Microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277–1284, Sep 1996.

[8] T Halfhill. Intel's Tiny Atom. *Microprossesor Report*, 22, April 2008.

[9] Liqiang He and Zhiyong Liu. An Effective Instruction Fetch Policy for Simultaneous Multithreaded Processors. In *International Conference on High Performance Computing and Grid*, volume 7, pages 162–168, July 2004.

[10] S. Hily and A. Seznec. Branch Prediction and Simultaneous Multithreading. In *Internaltional Conference on Parallel Architectures and Compilation Techniques*, volume 0, page 0169, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[11] S. Hily and A. Seznec. Out-Of-Order Execution May Not Be Cost-Effective on Processors Featuring Simultaneous Multithreading. In *International Symposium on High-Performance Computer Architecture*, volume 5, pages 64–67, Jan 1999.

[12] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall, Englewood Cliffs, N.J., 1991.

[13] James Laudon. Performance/Watt: The New Server Focus. *SIGARCH Comput. Archit. News*, 33(4):5–13, 2005.

[14] H.M. Levy, Jack L. Lo, J.S. Emer, R.L. Stamm, S.J. Eggers, and D.M. Tullsen. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *International Symposium on Computer Architecture*, volume 23, pages 191–191, May 1996.

[15] Yingmin Li, David Brooks, Zhigang Hu, Kevin Skadron, and Pradip Bose. Understanding the Energy Efficiency of Simultaneous Multithreading. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 44–49, New York, NY, USA, 2004. ACM.

[16] H McGhan. Niagara 2 Opens the Floodgates. *Microprossesor Report*, 20, Nov 2006.

[17] R. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[18] David A. Patterson. Reduced Instruction Set Computers. *Commun. ACM*, 28(1):8–21, 1985.

[19] Y. Sazeides and T. Juan. How to Compare the Performance of Two SMT Microarchitectures. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, pages 180–183, 2001.

[20] John S. Seng, Dean M. Tullsen, and George Z.N. Cai. Power-Sensitive Multithreaded Architecture. volume 0, page 199, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[21] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Mutlithreading Processor. *SIGPLAN Not.*, 35(11):234–244, 2000.

[22] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *International Symposium on Computer Architecture*, volume 22, pages 392–403, New York, NY, USA, 1995. ACM.

[23] R Usselmann. Open Floating Point Unit. Sep 2000.

[24] V.V. Zyuban and P.M. Kogge. Inherently Lower-Power High-Performance Superscalar Architectures. *Computers, IEEE Transactions on*, 50(3):268–285, Mar 2001.