Exploring Multipath Execution on a Distributed Microarchitecture

A Thesis Presented

by

Alireza Khalafi

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the field of

Electrical Engineering

Northeastern University
Boston, Massachusetts

May 20, 2003
Abstract

A common goal of many new microarchitectures is to improve performance by reducing the execution time of programs. One promising approach is to take advantage of the parallelism that is present in code. Extracting instruction level parallelism (ILP) from sequential code is one of the major challenges to effectively exploit large microarchitectures. Technology trends in the design of the future microprocessors suggest we will see an exponential growth in the number of transistors on a single chip. Chips with over one billion transistors will arrive shortly. The challenge is how to modify current design approaches to take advantage of these new manufacturing capabilities. Conventional microarchitectures limit our ability to exploit many of these new opportunities due to the increased wire delays that span large chip distances and the use of centralized resources such as reorder buffer and register files.

There have been a small number of research projects aimed at developing a high-ILP machine. The results of these studies show that branch instructions pose a major impediment to extracting high levels of instruction level parallelism by causing breaks in instruction execution. Reducing the effects of branch mispredictions is therefore crucial to obtain good performance of any large microarchitecture. Branch prediction and predicative execution are two conventional solutions to this problem. Multi-path execution is another approach that, until recently, had not drawn much attention due to its high hardware cost. Using a multi-path approach, instructions from both paths of a branch are speculatively executed, though instructions are only committed from one of the paths. A more cost-effective variation is Disjoint Eager Execution, wherein resources are assigned to the paths that are most likely
to commit. With an increasing number of instructions in flight in the execution window, multi-path execution seems to be a more plausible solution for eliminating branch misprediction penalties.

In this thesis a novel distributed and scalable microarchitecture is introduced that uses dynamic predication and multi-path execution to mitigate branch mis-prediction penalties. We have enhanced the Resource Flow execution model to maintain correct execution in this highly speculative microarchitecture, while contention for centralized resources has been eliminated through the use of distributed resources. Hardware scalability is achieved through the use of time-tags and active stations. Segmented buses form the interconnection fabric for our large execution window. A trace-driven simulator has been developed, which allows for an extended performance evaluation of the microarchitecture.
Acknowledgements

Over the years of my graduate studies, I have been blessed with the support and guidance of many more people to whom I can express my gratitude on a single page. However, I would like to acknowledge the few who have had the most influence on me.

Above all, I am grateful to my wife, Carla, for her sacrifice, encouragement and continuous support over the years. It is hard to imagine how I would have succeeded without all of the support and comfort that she has given to me.

My special thanks goes to my advisor, David Kaedi, whose continued support and direction provided a constructive environment for carrying out this research.

Additionally, I want to thank Augustus Uht, whose novel ideas created the foundation for this research endeavor. His close contribution and inspiration has transformed our initial concepts into a reality.

I have shared many memorable moments over the years with my friends in the graduate program to whom I am indebted for their support and encouragement. Among them, a special thank goes to David Morano for his insight and persistence toward the development of concepts incorporated in this thesis.

As well, I would like to recognize the other members of my defense committee, Fabrizio Lombardi and Waleed Meleis, who have provided their insight and judgment, for which I am equally appreciative.
Finally, I would like to express my gratitude to Zainalabedin Navabi for introducing me to the graduate program at Northeastern University, and for providing guidance in my initial studies as a doctoral candidate.
# Contents

1 Introduction ............................................. 1
   1.1 Hardware Scalability ............................... 2
   1.2 Speculative Execution .............................. 2
   1.3 Control Speculation ............................... 3
   1.4 Thread Level Parallelism ........................... 4
   1.5 Scope and Contribution of this Thesis ............... 4
      1.5.1 Enhancements to Resource Flow Execution Model .... 5
      1.5.2 Multipath Execution ........................... 6
      1.5.3 Speculation Invalidation ......................... 7
      1.5.4 Microarchitectural Simulator .................... 7
      1.5.5 Design Space Evaluation ........................ 7
   1.6 Thesis Organization ............................... 8

2 Related Work ........................................... 9
   2.1 Instruction Level Parallelism ....................... 9
      2.1.1 Early Dynamic Scheduling Mechanisms .............. 11
      2.1.2 High-ILP Machines .............................. 12
   2.2 Control Speculation ............................... 13
      2.2.1 Predicated Execution ........................... 15
2.2.2 Multipath Execution .............................. 18
2.3 Multiple-branch Prediction ........................................ 24
   2.3.1 Trace Cache ........................................ 26
2.4 Future Microarchitectures .................................. 27
   2.4.1 Superspeculative Processor .......................... 27
   2.4.2 Trace Processor .................................. 27
2.5 Summary .............................................. 28

3 Resource-Flow Execution Model .......................... 30
   3.1 Time-Tags ........................................... 31
   3.2 Register-based Transactions ............................. 31
   3.3 Time-Tag Assignment Schemes ............................ 34
   3.4 Register Operation .................................... 38
   3.5 Hardware Predication .................................... 42
   3.6 Memory Operations ..................................... 48
   3.7 Summary ............................................. 52

4 A Resource Flow Microarchitecture .................. 53
   4.1 Microarchitecture Components ........................... 54
   4.2 Execution Window .................................... 55
      4.2.1 Segmented Buses ................................ 56
      4.2.2 Rampant speculation .............................. 60
   4.3 Fetch and Dispatch ................................... 66
   4.4 Memory System ......................................... 68
   4.5 Execution and Commitment .............................. 69

5 Dynamic Multipath Execution .......................... 71
   5.1 General Operation ................................... 72
5.2 Static Multipath Execution ................................. 74
5.3 Dynamic Multipath Execution ............................. 75
5.4 Issues in Multipath Execution ............................ 77
  5.4.1 Fetch Bandwidth Requirements ........................ 77
  5.4.2 Fetch Heuristics ..................................... 77
  5.4.3 Fetch Priority ....................................... 79
  5.4.4 Spawning Heuristic .................................. 80
  5.4.5 Return Address Stack ................................. 81
  5.4.6 Delayed Fetching .................................... 81
  5.4.7 Path Switching ...................................... 82

6 Simulator ................................................. 86
  6.1 Simulation Methodologies ............................... 86
  6.2 Related Work .......................................... 89
  6.3 FastLevo Simulation Environment ....................... 89
  6.4 FastLevo Simulator Architecture ....................... 92
    6.4.1 Limitations ....................................... 96
    6.4.2 Validation ........................................ 97
    6.4.3 Design Space Exploration .......................... 97

7 Experimental Methodology ................................. 100
  7.1 Simulator Configuration ............................... 100
  7.2 Benchmark Characteristics ............................. 102
  7.3 Simulation Results ................................... 103
    7.3.1 Geometry Effects on Performance .................. 103
    7.3.2 Multipath Execution .............................. 110
    7.3.3 Effect of Fetch Bandwidth ......................... 113
8 Conclusions

8.1 Thesis Summary .................................................. 123
  8.1.1 New Time-Tag Assignment Scheme ................. 123
  8.1.2 Memory Nullify and Request Transactions ..... 124
  8.1.3 Dynamic Predication ................................. 124
  8.1.4 Speculation Invalidation ......................... 125
  8.1.5 Dynamic Multipath Execution ...................... 125
  8.1.6 Microarchitectural Simulator ..................... 126
  8.1.7 Design Space Evaluation ......................... 127
  8.1.8 Instruction Fetch ..................................... 127

8.2 Future Work .................................................. 128
  8.2.1 Multipath Execution ................................. 129
  8.2.2 Value Prediction ................................. 129
  8.2.3 Compiler Optimizations ......................... 130
  8.2.4 Detailed Modeling .................................. 130
List of Figures

2.1 multipath Execution (taken from [1]) .............................. 21
2.2 Operation of Adaptive Branch Table (taken from [2]) ......... 22
3.1 Original time-tag assignment scheme .............................. 36
3.2 New time-tag assignment scheme ................................. 37
3.3 Operand snooping logic within an Active Station(Courtesy of D. Morano) ......................................................... 39
3.4 Example instruction execution ...................................... 41
3.5 Branch Domain and Closest Enabled Previous Branch .......... 43
3.6 Example Dynamic Predication ...................................... 47
4.1 High Level View of the Microarchitecture ....................... 54
4.2 Interconnection fabric in the Execution Window ............... 55
4.3 Read operations satisfied in single bus span ................... 58
4.4 Hardware implementation of Speculation Invalidation scheme .... 62
4.5 An example of implementing Speculation Invalidation scheme in a Sharing Group ...................................................... 64
4.6 High level diagram of our microarchitecture ................... 67
5.1 Example of static disjoint paths ................................. 74
5.2 Example of dynamic disjoint paths ............................... 76
5.3 DSAQ table ......................................................... 83
5.4 SDQ table ......................................................... 84
5.5 Example of DSAQ and SDQ tables ............................ 85
6.1 High level view of the simulator modules: ................. 92
6.2 Partial list of the event queue: ............................... 93
6.3 Partial Event List for a Sharing Group .................... 95
6.4 Pseudocode of the flow through the simulator .......... 96
6.5 Partial list of output report .................................. 98
7.1 Effect of the number of Sharing Groups on IPC ........ 104
7.2 Effect of the number of Active Stations per Sharing Group on IPC .. 105
7.3 Effect of the number of columns on IPC ................ 106
7.4 Comparison of ideal and realistic configurations for a 4-8-4 machine
    geometry. .......................................................... 107
7.5 Comparison of ideal and realistic configurations for an 8-4-8 geometry.108
7.6 Comparison of ideal and realistic configurations for an 8-8-8 geometry.109
7.7 Comparison of ideal and realistic configurations for an 8-16-8 geometry.109
7.8 IPC as a function of multipath configurations ............ 110
7.9 IPC as a function of the number of dynamic disjoint paths. .. 111
7.10 Speedup gained by using subroutine call confidence estimation. ... 112
7.11 Effects of varying the speculation invalidation distance for a 8-4-8
    configuration. .................................................... 115
7.12 Effect of the Forwarding Unit delay on performance for an 8-4-8 con-
    figuration. ....................................................... 117
7.13 Effect of bus span size on performance. .................... 118
7.14 IPC as a function of different bus configurations. ........ 119
7.15  Effect of DRAM hit delay on performance. . . . . . . . . . . . . . . 120
7.16  Effect of L1 hit delay on performance. . . . . . . . . . . . . . . . 121
7.17  Effect of L2 hit delay on performance. . . . . . . . . . . . . . . . 121
List of Tables

3.1 Active Station Operation ........................................... 52

6.1 Microarchitectural Parameters ................................. 99

7.1 Default Machine Configuration ................................. 101
7.2 Benchmark Statistics ............................................. 102
7.3 IPC as a function of geometry ................................. 103
7.4 Idealized Configurations ......................................... 107
7.5 Effect of fetch bandwidth on performance ................... 113
7.6 Effect of domain threshold values on performance .......... 115
Chapter 1

Introduction

Technology trends in the design of the future microprocessors suggest an exponential growth in the number of transistors on a single chip. This increase is a direct result of the increased chip density that will be possible. Chips with over one billion transistors will arrive shortly. Our challenge is how to modify our current design approach to take advantage of these new manufacturing capabilities. Conventional microarchitectures do not scale due to the increased capacitance associated with long interconnect wires that must span large chip distances, limiting our ability to exploit many of these opportunities.

A common goal of many microarchitectures is to execute a program in a shorter amount of time. Although the majority of the speedups obtained over the last 20 years have been due to technological advancements, microarchitectural innovations have also played a major role. Extracting instruction level parallelism (ILP) from sequential code is one of the major challenges to effectively exploit large microarchitectures. Conditional control flow dependencies and memory latencies have been acknowledged to be the two major barriers to obtaining ILP.
1.1 Hardware Scalability

Future superscalar microprocessors are expected to issue and execute many instructions in a single cycle. This will require development of new techniques to accelerate instruction fetch, issue and execution. It will be a major challenge to produce a valid window containing many instructions. Trace caches [3, 4] are one of the proposed solutions to producing a sequence of program traces. By storing program basic blocks according to their dynamic execution order, it is possible to load many instructions into the execution window in a single cycle.

Conventional register files and rename and re-order buffers need to be modified to accommodate handling many more instructions. Contention for centralized resources makes it difficult or even impossible to scale current microarchitectures [5]. Future microarchitectures need to be partitioned across multiple processing elements in order to be scalable [6, 3]. Instructions are distributed across these partitions, and data is communicated between clusters utilizing a low-latency communication network.

Palacharla et al. [7] studied the complexity of superscalar processors and showed that the complexity of bypass paths grows quadratically with the number of functional units. The bypass paths require long wires that do not scale well for smaller feature sizes. The length of wires is becoming one of the limiting factors in the design of future microprocessors [8].

1.2 Speculative Execution

Dynamic branch prediction is a commonly used technique for speculating past control dependencies in programs. If we can predict the outcome of a conditional branch, we can issue and execute instructions following a branch, prior to branch
CHAPTER 1. INTRODUCTION

resolution. The accurate prediction of branches can help to eliminate latency and expose further ILP. A branch misprediction results in a pipeline flush. All instructions after the mispredicted branch are discarded and new instructions need to be fetched from the correct path.

Although the current branch predictors have a high prediction accuracy of greater than 95%, we still need to improve prediction rates in order to keep the instruction window filled with useful instructions. Unfortunately there are limits on the predictability of conditional branch prediction schemes. Many branches are very hard to predict and need to be handled using different techniques.

Value prediction is another speculative approach for exposing instruction level parallelism. By speculating on operand and address values at runtime, true data dependencies in program flow could be broken and as a result, more parallelism exposed. The cost of value prediction is not high and it has been shown that a fairly accurate value prediction scheme can be constructed using a moderately-sized hardware table [9].

1.3 Control Speculation

Although the accuracy of branch predictors has been greatly improved over last couple years, it is generally accepted that getting higher prediction accuracy is becoming much harder. Alternative solutions are required to hide the latency of remaining hard to predict branches. Two of the more common solutions are predicated execution and multipath execution. Predicated execution reduces the number of branches and increases the distance between mispredictions. Predication also reduces the frequency of flushing of the execution window as a result of branch mispredictions.

Multipath execution is another technique which reduces the execution time by
executing instructions down both paths of a branch. At branch resolve time, instructions on the incorrect path are squashed and fetch continues from the correct path.

1.4 Thread Level Parallelism

Another approach that has recently become popular is to exploit thread level parallelism using hardware mechanisms such as simultaneous multithreading [10]. Threads from a single program or multiple independent programs are executed simultaneously and share resources in a pipeline. Whenever a delay occurs in one thread, the processor can switch contexts to another thread, hiding the latency associated with the delay. This technique suffers from the need to efficiently manage shared resources.

1.5 Scope and Contribution of this Thesis

This thesis evaluates a novel distributed microarchitecture that uses dynamic predication and multi-path execution to mitigate branch misprediction penalties. The architecture employs time-tags and active stations to maintain correct execution in a highly speculative resource flow execution model.

Aggressive speculation is realized through a large amount of execution resources. A set of segmented shared buses are used to forward values from earlier to later instructions. Hardware scalability is achieved through limiting the maximum bus span and using repeater-like components. Contention for centralized resources such as register file, reorder buffer and execution units are eliminated through distribution of these resources. Chapter 3 and 4 provide a more detailed description of this microarchitecture.
CHAPTER 1. INTRODUCTION

The main contributions of this thesis are as follows:

1.5.1 Enhancements to Resource Flow Execution Model

In this thesis we enhanced the Resource Flow execution model as was originally created by A. K. Uht (90-95%) and D. Morano (5-10%) [11] and was presented in [12, 13]. We are using Resource Flow as an execution model for achieving high levels of IPC through speculative execution. The main idea behind Resource Flow execution is to allow execution of instructions as soon as free resources become available. Instructions that executed with wrong operand values will be re-executed until every instruction has generated its final correct value.

Memory Transactions

A set of new memory transactions are developed to enforce correct execution of the program in the presence of a highly speculative and distributed microarchitecture.

The memory request transaction is proposed as a mechanism for handling memory load instructions. Requests for memory values are sent on the backwarding buses and in response, correct values are forwarded from previous store instructions.

The nullify transaction is the other transaction that is proposed and is needed for correct handling of store operations that become disabled. By adding these two transactions, we are able to integrate memory operations into Resource Flow execution model.

Hardware Predication

Hardware predication is a technique for assigning predicate values to instructions in an architecture without requiring any instruction set support. Since predicates are generated at run time and evaluated by hardware, we can use hardware predication
CHAPTER 1. INTRODUCTION

when running legacy application code.

We are proposing a new predication assignment technique that greatly reduces the cost and complexity of the previously described method for implementing predication \[14\]. The complexity of our predicate assignment technique grows linearly with the number of instructions in the execution window. Unlike the previous scheme, we do not need to allocate null-predicates or predicate-only slots to handle overflow of branch targets to a single instruction.

1.5.2 Multipath Execution

Although predicated execution reduces the branch misprediction penalty by eliminating pipeline flushes, the disabled instructions after a mispredicted branch still need to be executed. This delay can be eliminated by executing instructions from both branch paths concurrently.

We use disjoint eager execution \[1\] to assign resources to the most probable paths. A disjoint path refers to the less probable path in a multipath execution scheme which its execution can be deferred. Disjoint paths execute in conjunction with predicated instructions to further decrease the branch misprediction penalties.

A static disjoint path has the same instructions as the mainline path, but the output predicate of the associated branch is inverted. This allows loading of similar instructions on the not-predicted path to execute concurrently with instructions on the predicted path.

In this thesis, we introduce dynamic multipath execution as an alternative to static disjoint paths. Dynamic multipath execution allows for instructions on the disjoint path to be different from mainline path. Although this scheme requires a more complex fetch unit, its better handling of branches with far targets results in higher performance.
1.5.3 Speculation Invalidation

We propose a hardware mechanism to control the amount of speculation in our microarchitecture. Although speculation is a potential source of speedup, rampant speculation could have an adverse effect on performance. This is a result of limited execution opportunity for instructions on the disjoint paths due to the resource sharing between mainline and disjoint paths.

1.5.4 Microarchitectural Simulator

As part of our research, we developed a simulator to evaluate our microarchitecture and explore its design space. The main structure of our microarchitecture is similar to the Levo [15] but we have made many enhancements over this base architecture.

The simulator is trace-driven with an added execution-driven engine to enable correct evaluation of the speculative features of the microarchitecture. An extended evaluation of the design space has been possible through parameterizing the simulator.

1.5.5 Design Space Evaluation

A set of benchmark programs are used to evaluate the performance of our microarchitecture. In addition to measuring the overall speedup improvements, other issues such as the sensitivity to variations in machine parameters has also been studied. Some example studies are geometry effect studies which measure performance as a function of the execution window size, memory sensitivity analysis and fetch bandwidth analysis.
Limit studies are another set of simulations which measure the upper limit improvement that could be achieved under a certain assumptions. This data is especially helpful for improving the performance through locating the existing bottlenecks.

1.6 Thesis Organization

Chapter 2 presents an overview of other research related to the topic of this thesis. In Chapter 3 the Resource Flow execution model is introduced as a new model for managing program execution on a distributed microarchitecture. Chapter 4 describes our proposed distributed microarchitecture that realizes Resource Flow execution model. Dynamic multipath execution is introduced in Chapter 5. Chapter 6 outlines the architecture of the simulator that has been developed in this thesis and is used to evaluate the proposed microarchitecture. Our simulation results are presented in Chapter 7. We finish this thesis with a summary and suggested extensions in Chapter 8.
Chapter 2

Related Work

In this chapter we review previous work that is related to the ideas presented in this thesis. We begin by discussing previous work in the pursuit of high levels of instruction level parallelism.

2.1 Instruction Level Parallelism

A number of studies into the limits of instruction level parallelism (ILP) have demonstrated that there is a significant amount of parallelism within typical integer programs (e.g., SpecInt-2000). The work of researchers like Lam and Wilson [16], Uht and Sindagi [1], Gonzalez and Gonzalez [17] have shown that there exists a great amount of instruction level parallelism (ILP) that is not being exploited by any existing computer design.

Wall [18] presented an ILP study that assumed dynamic branch prediction in the presence of perfect memory disambiguation, perfect register renaming, unlimited instruction fetch bandwidth, and a large number of functional units.

Lam and Wilson [16] conducted an extensive set of revealing simulations on some of the SPEC89 benchmarks assuming different control dependency resolution
CHAPTER 2. RELATED WORK

techniques. Their most advanced model employed Single-Path branch speculation (like a simple branch predictor) with reduced Control Dependencies (instructions after a forward branch’s target are independent of the branch) and Multiple Flows of control (multiple program counters). For this model, which was named SP-CD-MF, an average speedup of 40X was achieved on the integer benchmarks using unlimited execution resources. For an Oracle predictor (a branch predictor that obtains 100% prediction accuracy), a speedup of 158 was obtained.

The Lam and Wilson study illustrated the need for providing a large instruction window to obtain large ILP. They concluded that a large amount of ILP exists in integer codes, but conceded that it was unlikely to be realized, particularly with high IPC, because of the machine limitations extant at the time. In particular, no commercial machine realized MF, and few realized CD. We view the Lam and Wilson results as a challenge to produce a machine model that will realize high IPC from the available high ILP. Our machine model goes beyond SP-CD-MF in both the data speculation and control speculation dimensions. Thus, we should be able to exploit high levels of ILP, since data speculation was not included in the Lam and Wilson study.

Gonzalez and Gonzalez [17] performed a study on the potential impact of address and data speculation on ILP. Using the SPEC95 benchmark suite, they evaluated the benefits of address and data value prediction. They assumed infinite execution resources and found that they could achieve a speedup of 42. While they assumed multiple predictions could be in flight simultaneously, they did not consider following multiple paths of execution. Riseman and Foster [19], Chen [2] and Uht [1] found much ILP in general purpose code, but none of these studies made use of data speculation.
2.1.1 Early Dynamic Scheduling Mechanisms

Early efforts to speedup program execution was concentrated on relaxing the strict program order through an out of order execution mechanism.

Scoreboarding is a technique for allowing instructions to execute out of order and was introduced by Thornton [20] in the CDC 6600 processor. The goal of scoreboard-
ing is to maintain an execution rate of one instruction per cycle. If an instruction is stalled, other instructions can continue execution. A scoreboard is a centralized unit that performs all hazard detection and resolutions and controls the instruction progression from one step to the next.

Tomasulo’s scheme is another dynamic scheduling scheme that allows execution to proceed in the presence of hazards. It was invented by Tomaso [21] and im-
plemented in the IBM 390/91. It uses register renaming to avoid Write-After-Read (WAR) and Write-After-Write (WAW) hazards and combines it with the key elements of scoreboard. The centralized scoreboard is replaced with distributed control within the processor. Each functional unit has one or more reservation sta-
tions which could hold one instruction. The execution result is passed on the Com-
mon Data Bus to the register file and is also snoop by all reservation stations. The IBM 360/91 also recognized the importance of maintaining an uninterrupted supply of instructions for overall performance. To avoid a disruption when a branch instruction was encountered, it fetched (but did not execute) instructions from both paths of a branch. It also had a loop buffer that supplied instructions in a loop mode.

Both of these techniques are used today in many superscalar processors. We borrow the idea of a reservation station from Tomasulo and use a more intelligent version of this device in our datapath. In the next section we review some more recent efforts to develop a high-ILP machine.
2.1.2 High-ILP Machines

The Multiscalar model of execution [6, 22] divides a program into a collection of tasks using a combination of hardware and compiler techniques. The tasks are distributed to a number of processing elements and are executed in parallel. Although the multiscalar processor has similarities with a Chip Multiprocessor, the processing elements are more closely coupled. A compiler is used to partition tasks between processing elements in order to increase the performance of a single thread program.

A program is represented as a control flow graph (CFG) where each node corresponds to a basic block. The flow of control from one basic block to the another is represented by an arc. A program sequencer speculatively assigns each task to a PE for execution. Tasks are executed in parallel resulting in execution rate of multiple IPC.

To enforce data dependencies among different tasks, register values are dynamically routed among the processing elements using compiler-generated masks. Inter-task communications are synchronized such that the register values produced by earlier tasks are forwarded through a unidirectional link and are consumed by later tasks.

Handling memory operations is more complex because a load instruction does not know if an earlier task has a store to the same address. Multiscalar uses data dependence speculation to perform speculative memory operations. An Address Resolution Buffer (ARB) [23] is used to check memory dependencies and squash incorrect speculations.

Multiscalar processors use multiple program counters to sequence through a program. Information regarding the execution order of tasks can be statically determined and stored in the task descriptors dispersed within the program text.

Nagarajan et al. have proposed a Grid Architecture that builds an array of
CHAPTER 2. RELATED WORK

ALUs, each with limited control, connected by a operand network [24]. Their system achieves a peak IPC of 11 on SPEC2000 and Mediabench benchmarks. While this architecture presents many novel ideas in attempt to reap high IPC, it differs greatly from the ideas in this thesis.

Probably the most successful high-IPC machine to date is Lipasti and Shen’s Superspeculative architecture [25], achieving an IPC of about 7 with realistic hardware assumptions. The Ultrascalr machine [26] achieves asymptotic scalability, but only realizes a small amount of IPC, due to its conservative execution model. The Warp Engine [27] uses time tags, for a large amount of speculation; however their realization of time tags is cumbersome, utilizing large integer or floating point numbers and machine wide parameter updating.

The performance of multipath execution techniques is closely related to the available fetch bandwidth. Section 2.3 reviews the current research in this area and some of the proposed techniques for increasing the bandwidth and reducing the fetch latency. Finally, in section 2.4, we will study some of the proposals for future microarchitectures.

Next we will review specific mechanisms that help to expose ILP that specifically target control flow.

2.2 Control Speculation

Branch instructions pose a major impediment to extracting high levels of instruction-level parallelism. Branches cause breaks in instruction execution since the program may follow different paths of execution. Approximately 15-20% of all instructions are conditional branches.

A number of mechanisms have been proposed to speculate on the outcome of
conditional branch instructions. Dynamic branch prediction techniques use the previous history of the outcome of the branches to predict their future outcome [28]. Branch history is recorded and is used to drive the prediction of future executions of this branch.

Smith [28] introduced a family of dynamic branch predictors. The simplest dynamic branch predictor uses a branch target buffer (BTB) (also referred to as a branch history table (BHT)). Each entry of the BTB contains the past history (taken or not-taken) of the most recent execution(s) of the branch. In current implementations, the BTB is indexed by the lower bits of the branch instruction address exclusive-OR’ed with the outcome of the last n-branches [29]. One common implementation is to use a 2-bit counter to capture the momentum of a branch. The two bits create four states:

1. strongly not taken,
2. weakly not taken,
3. weakly taken, and
4. strongly taken.

Yeh and Path [30] introduced two-level adaptive predictors which uses a global branch history register (BHR) to index into a pattern history table (PHT) of 2 bit counters. The outcome of all branches is used to update the global BHR in order to employ the correlation between branches. The work by Pan et al. [31] on correlation based predictors had shown that using correlation usually results in a higher prediction rate for integer intensive programs. Other variations of the two level adaptive predictors were also introduced by Yeh and Path in [32]. They examined configurations by using combination of global branch history register and per-address branch history table with global or per-set pattern history tables. Global
history schemes were shown to be more effective for the integer programs whereas per-address schemes are better suited for the floating point programs.

There have been a number of variations upon the general design of a BTB. Hybrid Predictors [33, 34, 35] combine multiple separate predictors, each targeted towards a specific class of branches. Dealiased Predictors [36, 37] are proposed to reduce the interference effects resulting from multiple branches sharing the same entry in the predictor table. Skewed [38, 39], Bimodal [40] and YAGS [41] predictors belong to this category. Confidence predictors [42, 43, 44], on the other hand, are used to classify branches as either high confident or low confident according to their prediction accuracy.

Next we look at an alternative approach to solving control flow uncertainty using predication.

2.2.1 Predicated Execution

One solution for reducing the effect of mispredicted branches is the use of predicates. The instruction set architecture of a microprocessor is extended by adding predicate registers and a set of conditional or predicated instruction. The result of each conditional instruction is saved in a predicate register which is used as an additional input operand for predicated instructions. The following code demonstrates how this technique is applied.

```c
if (x == 2)
    c = d + e;
else
    c = d - e;
e = a * b;
```
CHAPTER 2. RELATED WORK

Using a predicate register \( p1 \) and assuming predicated instructions, we can eliminate the speculative execution of \( e = a * b \).

\[
\begin{align*}
    p1 &= x == 2; \\
    c &= d + e \text{ if } p1; \\
    c &= d - e \text{ if } \neg p1; \\
    e &= a * b;
\end{align*}
\]

Predication reduces the number of branches and increases the distance between mispredictions. Predication is most effective for if-then-else constructs with small conditional bodies. By effectively increasing the basic block size, a compiler has more flexibility when scheduling instructions and produces a better schedule. When using predication, we will fetch many uncommitted instructions; predication can waste fetch bandwidth.

Most microprocessors use a partial predication as a limited form of predication as compared to full predication. As examples, Alpha, MIPS, PowerPC, and SPARC processors use conditional move instructions. In a full predication scheme, all instructions are predicated and the their opcode is augmented with predication bits. Cydar, Arm [45] and Itanium all use full predication. Predicated instructions are usually speculatively executed and only committed after their input predicate values are resolved. The Hewlett Packard PA-RISC 2.0 architecture [46] provides a nullification mechanism that can be used as a support for limited predicated execution. The nullified instructions do not update the processor state. Conditional moves are used in the Alpha ISA to support partial predication [47].

If-conversion [48, 49] is a process for eliminating conditional branches by replacing them with comparison instructions which set a predicate. Instructions that are dependent on the branch are then converted to predicated instructions.
CHAPTER 2. RELATED WORK

Mahlke et al. [50] proposed hyperblocks as an extended superblock scheduling technique to support predicated architectures. Branch instructions in a hyperblock are eliminated by introducing conditional instructions through if-conversion. The presence of multiple control flow paths in a hyperblock creates more opportunities for the compiler to apply classical optimizations, such as common subexpression elimination and copy propagation [51].

Loop Peeling is an optimization technique that takes advantage of predicated execution support [52]. This optimization is applied to the inner loops that have a tendency to iterate infrequently. The first couple iterations of the loop are peeled away by the compiler and combined with the code surrounding the inner loop using predication. This creates a single block of code which can be further optimized.

Another optimization techniques such as symmetric back substitution which are based on control height reduction [53] can be applied to shorten dependence chain lengths to compute predicates. Control height reduction techniques reduce the dependence chain length to compute the execution conditions of instructions. Since predicates are a series of data values computed using arithmetic instructions, data height reduction techniques can be applied to optimize predicate dependence chains.

One of the limitations of the hyperblock framework is that the excessive application of predication will result in saturation of the fetch and potentially execution units. August [54] introduced the partial reverse if-conversion compiler technique which operates at schedule time to balance the amount of predication and control flow present in the code by re-introducing some branches to replace predicated code.

The benefits of predication for enhancing branch prediction are studied in the works by Tyson [55] and Mahlke et al [56]. Tyson studied the effect of both full and partial predication including support for speculative loads and stores. They studies
found that using full predication could remove 30% of all dynamic branches with near forward targets.

Chang and Lai [57] proposed the Conjugate Register File to allow for architectural support of compiler based speculation. The register file allows pairs of registers to be swapped in a single cycle. The compiler uses these features to allocate a pair of registers as speculative and non-speculative pairs.

Guarded execution is proposed by Pnevmatikatos [58] where a bit-mask is used to propagate the predication information from a single instruction to the subsequent instructions.

Rau [59] proposed using conditional moves in a dynamically scheduled VLIW architecture. In his approach operations with multiple cycle latency are broken into two parts. The first part performs the actual operation and writes the result into a virtual register file. The second part moves the results from the virtual register file into the physical register file after a delay. The combination of these two operations can be viewed as a way of implementing predicated instructions where the predicate is a function of time.

The Skipper microarchitecture, proposed by Cher et al. [60], handles difficult branches by exploiting control-flow independence. Instead of allocating resources to instructions after a hard-to-predict branch, Skipper avoids the control dependent instructions after the branch by skipping over them, and fetches and executes dataflow independent instructions after the recovery point.

2.2.2 Multipath Execution

Multipath execution is a technique used for reducing branch misprediction penalty by fetching and executing down multiple paths of execution. If a multipath execution
is provided with unlimited resources, we could obtain the same theoretical performance as a perfect branch predictor (all misprediction penalties would be hidden). With realistic resources, however, multipath execution needs to be used carefully to prevent exponential growth in resource consumption. The advent of ever-increasing amounts of speculative execution has pushed us to look for new mechanisms that employ both branch prediction and multipath speculation.

Early work on multipath execution was dominated by IBM in the late 1970s and 1980s [61]. The earliest attempts at multipath execution started with the ability to prefetch down both outcomes of a conditional branch. This idea was expanded on by allowing execution to follow both outcomes of a conditional branch. Aggressively executing down both outcomes of conditional branches has been explored by Wang [62].

Unger et al. [63, 64] proposed a compiler technique in combination with a minimal multi-threaded execution model to enable speculative execution of alternative program paths. The processor is assumed to pursue two or more threads concurrently and is extended by adding fork and sync instructions to create and join new threads.

Wall [65] considered branch fanout as part of an ILP study where execution continues along both paths following conditional branches. A fanout limit is employed to bound the number of simultaneously executing paths. Static prediction probabilities are gathered through profiling and are used to choose the fanout by evaluating the confidence for each path.

In [66], branch hammocks are handled through predicated execution. The implementation does not assume instruction set support for predicated execution. Instructions in both paths of an if-then-else construct are predicated at decode time and are both executed.
CHAPTER 2. RELATED WORK

In a more aggressive approach, Uht and Sindagi evaluated the intersection of both multipath execution and future large-scale microarchitectures capable of executing possibly hundreds of instructions simultaneously [1]. They proposed the *disjoint eager execution* technique for assigning resources to branch paths which are more likely to be committed. In a wide-issue microarchitecture, multiple branches can be speculatively executing before the first branch is resolved. As more branches are speculatively fetched, the probability that a later branch would be on the correct path is reduced. The cumulative execution probabilities can be used to select the next spawned path. The branch path that has the highest cumulative prediction accuracy will be executed. To eliminate dynamic probability calculations, a static tree approach is also proposed that uses average branch prediction accuracy to form a fixed pattern for spawning disjoint paths.

Figure 2.1 demonstrates three different options for assigning resources to the branch paths in a speculative microarchitecture. Each branch path is represented with an arrow and is labeled with its associated cumulative probability. Resource assignments are represented using the circled numbers. In single-path speculative execution, resources are assigned to the most recently predicted branch path. As we can see from Figure 2.1, there is a very low probability that the most recently selected branch path is committed. In the full eager execution scheme, resources are assigned to instructions on both branch paths. The resource requirement in this approach grows exponentially with the number of speculatively executing branches. In the disjoint eager execution approach, resources are only assigned to the paths possessing the highest probability to commit. We utilize many of the results of the Uht and Sindagi approach in our work.

Chen proposed a technique to dynamically keep track of branch paths using an Adaptive Branch Tree [2]. A token assignment strategy is introduced to facilitate the
handling of multiple paths by tagging instructions from multiple paths. Cumulative probabilities are dynamically collected to guide the decision on what paths need to be forked.

Figure 2.2 is an example of the operation of an Adaptive Branch Table (ABT). There are five columns in this table: valid bit, token, cumulative probability (CP), leaf node flag, and branch address. Token assignment is implemented such that once the root branch is resolved, all tokens will be shifted to the right by one bit and one of the child nodes will become the new root node. To use this implementation, the following two token assignment rules are used:

- the root node at the first level always has the token 0001, and

- for any two sibling nodes at the level \( i \) in the graph, the difference between their tokens is only at \((i - 1)\) bit.

Figure 2.2.a shows the token assignment and the associated ABT table before branch is resolved. Once the branch at the root node is resolved, one of the subtrees is squashed (the left subtree in the example) and the other subtree is moved up and
becomes the new subtree (as in shown in part b of Figure 2.2). The valid bits in the ABT are updated accordingly. The tokens on this subtree simply need to be shifted one bit to the right to create the new tree. The new CP values are calculated by dividing the current CP by the probability of the resolved branch (this is not a trivial calculation).

Figure 2.2: Operation of Adaptive Branch Table (taken from [2]).

Work on dual path execution, a form of multipath execution with only two speculative paths, has been done by Heil and Smith [67]. Tyson et al. [68] proposed a restricted form of dual-path execution by using a new approach to gather branch prediction confidence and limiting dual path execution to the branches with low confidence.

Examining multipath execution on the PolyPath microarchitecture,
Klauser et. al. [69] enhanced a superscalar processor by adding limited multipath execution features. They used confidence estimators to guide multipath execution, but did not fully explore the opportunities for fetching instructions across multiple taken branches in each cycle.

An increasingly attractive approach to multipath execution is to use a simultaneous multi-threaded (SMT) processor to provide the resources for essentially executing multiple paths of a single architected program thread. This work follows from the original SMT idea by Tullsen el al. [70]. The work by Tullsen focused on making better use of the processor when branch mispredictions are encountered by filling processor resources with work from other architected threads following a mis-prediction. An extension of this idea is to use resources for executing the alternative path (not-predicted path) of a mispredicted branch. An example of this approach has been discussed by Wallace et al. [10]. This general approach of combining both simultaneous multithreading with multipath execution appears to be a good compromise to the problem of most efficiently using processor resources. This approach also lends itself towards hardware that possibly can be programmed at run-time for providing either maximum single threaded execution speed or maximum multi-threaded throughput.

Aluja et al. [71] explored speedup limits for multipath execution, but their work was still largely restricted to more conventional (modest-sized) microarchitectures. They showed that among the options provided by different branch confidence predictor architectures, the ones-counting architecture was the most effective for their application.
CHAPTER 2. RELATED WORK

2.3 Multiple-branch Prediction

In order to have a large number of instructions available in an execution window we may need to predict multiple branches down a single or multiple paths. The problem of high-bandwidth instruction delivery has been the focus of much research. Accurate instruction fetch depends on a number of factors. Instruction cache misses stall the fetch engine and delay execution. Branch mispredictions can cause pipeline flushes. Instructions fetched from the wrong path need to be squashed and fetch resumed from the committed branch path.

As the issue rate of microprocessors is increased, we need to be able to predict multiple branches in a single cycle. Simultaneous access to multiple cache lines may be necessary since multiple branch targets are not guaranteed to be in neighboring cache locations. Mechanisms for fetching and aligning multiple noncontiguous basic blocks are therefore required. Pipelining the fetch unit is one of the techniques for increasing the fetch rate, but the increased pipeline latency will result in higher branch misprediction penalties.

Yeh et al. [72] consider an extension to the branch target buffer called Branch Address Cache which provides a mechanism for predicting multiple branch targets in every cycle. The multiple branch predictions combined with a hit in the branch address cache are used to address multiple instruction cache lines and fetch multiple basic blocks in a single cycle. A complicated alignment network requires additional pipeline stages.

Dutta and Franklin [73] proposed a similar approach, but used a single prediction to choose among multiple possible paths. For example, instead of predicting two branches, they used a single prediction to choose among four possible paths.

Seznec et al. [74] proposed a modification to the Branch Address Cache called
Multiple-Block Ahead Predictor. Using this technique, the fetch address of the current block is used to predict the target address for the n future blocks. This allows for pipelining the target generation processes.

Conte et al. [75] introduced collapsing buffer. This scheme uses multiple passes through an interleaved branch target buffer to produce a fetch address. This allows multiple nonadjacent cache lines to be fetched. Multiple branches in a cache line are detected using the branch target buffer logic. An interchange and masking network, called a collapsing buffer is used to align and merge instructions from different cache lines. A disadvantage of using this scheme is that it does not scale well with the number of branches.

Reinman et al [76] present a fetch architecture, called Fetch Target Buffer that permits faster cycle time and scales better to future process technologies. The FTB is a multi-level fetch block-oriented predictor and is decoupled from the instruction fetch and decode pipelines.

Klauser and Gruwald [77] evaluated different design tradeoffs for the first level instruction cache. They used a branch confidence estimator and showed that the number of ports needed in a multiported instruction cache is directly proportional to the average number of execution paths. They also found out that multi-banking of the instruction cache has almost the same effect as of multiporting. Multi-banking is however a more efficient approach than multiporting and can be implemented with less complexity. They also showed that merging instructions from two different fetch paths into a single decode unit produces similar results as supporting two decode paths. This could further simplifies the implementation of multipath execution.
2.3.1 Trace Cache

A microprocessor that can issue eight or more instructions per cycle needs to be able to access multiple cache lines in a single cycle. Instructions must be shifted and aligned before being decoded. In addition to multiporting or interleaving an instruction cache, we need another mechanism to handle noncontiguous instruction delivery. A Trace Cache provides a solution to the above problems by caching dynamic instruction sequences. A hit in the trace cache means that there is no need for rebuilding the dynamic sequence on the fly. This eliminates the need to reissue recently fetched and decoded instructions. The Trace Cache, however, suffers from the number of redundant instructions introduced when trace cache entries overlap in the execution flow. The trace miss rate is also another limiting factor for a Trace Cache.

A Dynamic Flow Instruction Cache was the first evaluation of a Trace Cache. It first appeared as a US patent filed in 1994 by Intel Corporation [78]. Each line in their cache contained two blocks from the dynamic instruction stream.

The Expanded Parallel Instruction Cache was proposed by Johnson [79]. This technique is similar to a trace cache where each line of the cache contains instructions from the dynamic order. The main difference is that it is targeted for a statically scheduled microarchitecture. The post-fetch processing of instructions is made simpler by reordering and storing them in the cache.

The performance of a Trace Cache was studied by Rotenberg et al. [3]. They showed that a Trace Cache could be more effective than other mechanisms such as Branch Address Cache and Collapsing Buffer. Research by Patel et al. [4] introduced techniques for increasing the instruction bandwidth delivered by the Trace Cache. They described the implementation of an aggressive multiple-branch predictor to be used in conjunction with the Trace Cache. In later work, the authors propose Partial
CHAPTER 2. RELATED WORK

Matching and Inactive Issue [80], and Branch Promotion and Trace Packing [81] as additional enhancements to the Trace Cache. In yet another approach developed by Vajapeyam and Mitra [82], register dependencies are explicitly encoded in the Trace Cache. They also suggest capturing loops in the Trace Cache using a vector like encoding scheme.

2.4 Future Microarchitectures

In this section we review some of the future microarchitecture proposals that look at control speculation.

2.4.1 Superspeculative Processor

The Superspeculative Processor proposed by Lipasti et al. [83] is another effort for improving the performance of wide-issue superscalar processors. In this design, aggressive data and control speculation is used at every point in the processor pipeline. If a large portion of speculations are correct, the weak dependence model used by a superspeculative processor could result in higher performance.

Source value operand prediction uses a dynamic value prediction table per static instruction to eliminate data dependencies. Value stride prediction is another technique used in the superspeculative processor, where a constant increment in an operand value (i.e., a stride) is detected to improve address prediction accuracy. Predicting load values and load addresses are among other speculative techniques used to accelerate the processing of memory instructions.

2.4.2 Trace Processor

Rotenberg et al. presented the idea of a Trace Processor [84] which uses a trace cache to execute multiple traces of instructions on each processing element. They
CHAPTER 2. RELATED WORK

showed that this will greatly reduce hardware complexity. Control prediction is
moved to the trace level and the next-trace predictor replaces the multiple branch
predictor.

Local forwarding of register values is handled through bypass paths within a
unit. Communication between processing elements is carried through global paths
which are likely to require multiple clock cycles. Physical registers are divided into
two sets, local and global. Local register sets are small and fast with fewer ports.
Global registers are mapped to the trace’s source and destination registers to provide
intra-trace data transfers. Value prediction is also used to decouple traces and allow
for parallel execution. Value prediction is only used to speculate on inter-trace
dependencies.

In the trace processor, processing elements generate a speculative and out-of-
order stream of load and store requests to the main memory. This introduces a
number of memory disambiguation issues that need to be resolved. The Address
Resolution Buffer (ARB) [23], has been proposed for solving this disambiguation
problem, but it still represents a central resource that will not scale as memory
traffic increases.

2.5 Summary

In this chapter we surveyed some of the current microarchitectural advances that
have influenced this thesis. ILP studies have shown a great deal of parallelism
in the current sequential code. Out of order execution, control speculation and
value prediction have been introduced as techniques for exposing higher levels of
parallelism by relaxing dependencies.

Although it is hard to predict the future implications of technology, we believe
that the following challenges need to be addressed in any new microarchitecture.
• Preserving object code compatibility is highly encouraged in order to deal with legacy binaries. It is greatly desirable for a new microarchitecture to run the old applications without having to recompile them.

• To expose higher levels of parallelism, it is necessary to have a large number of instructions in the execution window. The current superscalar microarchitectures suffer from scalability issues due to the centralized units such as register file, reorder buffers and execution units. Future microprocessors need to achieve hardware scalability through a distributed microarchitecture.

• Control flow mispredictions are probably the most limiting factor in a high ILP processor. Unfortunately the accuracy of branch predictors is unlikely to greatly improve in future. Other techniques such as predication and multipath execution are therefore necessary to exceed the control flow limit.

• To satisfy the increase in issue width, Multiple branch predictors along with large trace caches are needed to provide a contiguous instruction stream.

• Increase in memory bandwidth requirements requires application of memory hierarchy techniques such as larger caches, prefetching, bank interleaving and streaming buffers. A great deal of research is focused on solving the processor memory bottleneck.
Chapter 3

Resource-Flow Execution Model

To take advantage of the large amounts of instruction level parallelism available in current sequential programs, it is necessary to go beyond traditional data and control flow dependency constraints to expose parallelism present in code. In current microarchitectures that perform speculative execution, access to the reorder buffer becomes problematic as the number of speculative instructions being concurrently executed grows [7]. The issues that arise are primarily due to contention for centralized resources (e.g., data registers within the reorder buffer).

We are proposing a new microarchitecture that exploits highly speculative execution through a Resource Flow execution model to achieve high levels of ILP. In the Resource Flow execution model, instructions with the highest priority are assigned to free execution resources regardless of whether their input operands have reached their final correct value (data flow constraints) and regardless of the outcome of the previous branches in the instruction stream (control flow constraints). This means that the execution is guided based upon the need for an instruction to execute and the availability of resources. The rest of the execution time is spent re-executing the instructions that executed with an incorrect speculative input or operand, so as
to end up with a programatically correct execution of the code. As demonstrated in the rest of this chapter, the Resource-Flow execution model does not require explicit renaming of registers or any reorder buffer. This model provides methods for executing standard procedural (control-flow guided) programs that go beyond the standard control and data flow model.

3.1 Time-Tags

In the Resource-Flow execution model, instructions are executed speculatively in an out-of-order fashion. We use time-tags to enforce the correct data and control dependencies among the instructions to realize speculative data-flow execution of code. A time-tag is a small integer that uniquely identifies the position of an instruction or an operand in program ordered time with respect to the most recently retired instructions.

Our application of time-tags is to keep track of the original instruction order. Time tags were originally proposed in the Warp Engine [85] for similar purposes. The Warp Engine used floating-point numbers for time-tags since they needed to create an arbitrary number of time-tags between any given pair of time-tags. In our microarchitecture, we are able to use small integer values since these values indicate an issue slot in our execution window.

3.2 Register-based Transactions

Both renaming and time-tagging assume that results from execution of an instruction are put on a bus and snooped by all later instructions. In the resource flow model, the operand address (register number) is used to identify the broadcasted value. Accompanying the address are: the actual value, the instruction time-tag,
CHAPTER 3. RESOURCE-FLOW EXECUTION MODEL

and a path ID [86] (if multi-path execution is allowed). The time-tag and path ID will be used by subsequent instructions in the execution window to determine if the operand with the desired address value originated from the closest previous instruction and therefore should be snařed\(^1\). A snařed value could subsequently trigger the execution or re-execution of the instruction.

Communication between instructions is realized through a set of transactions that we call *Register-based Transactions (RTs)*. There are two general forms of RTs:

1. *forwarding transactions* (FW) and

2. *backwarding transactions* (BW).

A forwarding transaction sends (forwards) information to the instructions later in the program execution order. A backwarding transaction sends requests to the earlier instructions in the program execution order.

In order to correctly execute and process the RTs, along with each instruction we maintain a transaction template. This template is used to hold the transaction specific information as well as intermediate operand values. The combination of an instruction and the associate transaction template are the elements of an *Active Station (AS)*. The term Active Station actually comes from our microarchitectural implementation of the resource flow execution model and refers to the expanded issue slots capable of processing the RTs. Control-flow and data-flow dependencies are enforced through communication between Active Stations using a set of transactions. Active Stations constantly examine transactions that they see on a number of buses, and snařft operand values that are needed for the execution of the instruction in this Active Station. The following list shows different information maintained in an

\(^{1}\)Snařing implies we snoop a bus and if a match on the current bus contents is found, we read the associated data value.
CHAPTER 3. RESOURCE-FLOW EXECUTION MODEL

Active Station $AS_i$, where $i$ is the physical index of the Active Station.

$AS_i.en$ is a binary value which is set to one if the instruction is enabled and its result is used in the execution of the program.

$AS_i.tt$ holds the time-tag value.

$AS_i.op_x$ where $x \in \{1, 2, \ldots\}$ holds the address of input operands of the instruction $I_i$ residing in the Active Station.

$AS_i.op_x.data$ where $x \in \{1, 2, \ldots\}$ holds the data value of the input operands of the instruction $I_i$ residing in the Active Station.

$AS_i.op_x.last_tt$ holds the source time-tag of the last snarfed operand used in the execution of the $I_i$. Since each operand can have a different source, a separate variable for each operand is needed.

$AS_i.D$ is the address of the output operand of $I_i$.

$AS_i.D.data$ is the value of the $AS_i.D$ operand.

$AS_i.mem.addr$ holds the memory address (valid only if $I_i$ is a load or store instruction).

$AS_i.mem.data$ holds the memory value (valid only if $I_i$ is a load or store instruction).

$AS_i.mem.last_tt$ holds the source time-tag of the last memory value that was snarfed. (valid only if $I_i$ is a load instruction).

$AS_i.D.last_data$ holds the output operand value produced by the last enabled instruction before $I_i$. In case $I_i$ is disabled, this value is forwarded as the correct value of the destination register.
**AS\textsubscript{i}.outcome** is a binary value set according to the outcome of the branch (valid only if \( I_i \) is a branch instruction).

**AS\textsubscript{i}.target\_tt** holds the time-tag of the branch target instruction (valid only if \( I_i \) is a branch instruction).

**AS\textsubscript{i}.CEPB** structure holds the information regarding the closest enabled previous branch with respect to instruction \( I_i \) in the static program-order. This is used to handle predicate assignment and evaluation. More information will be provided in Section 3.5.

Before we describe how this information is used to enforce the correct execution of programs, it is helpful to see how time-tags are assigned to instructions and their operands. This issue is the topic of the next section.

### 3.3 Time-Tag Assignment Schemes

The first method used for assigning time-tag values to instructions and their operands is based on the scheme introduced in [12]. In this scheme, the oldest issued instruction in flight that is neither yet committed nor squashed would usually have a time-tag value of zero. More recently dispatched instructions take on increasingly larger values the further ahead they are in the program ordered instruction stream. As a group of instructions with the lowest valued time-tags are retired, all time-tags values for all instructions and operands are decremented by the number of instructions just retired. This will keep the next instruction to be retired in having the time-tag value of zero and therefore prevents time-tags from growing indefinitely.

Although this is a feasible solution, it requires extra hardware in every location where time-tag values are stored. To solve this, we propose a new mechanism for
assigning and maintaining time-tags that does not require global modification of
time-tags. A time-tag adjustment unit is introduced which modifies the time-tag of
the forwarded values from the last physical column to the first physical column. We
will also show that if the number of Active Stations is a power of 2, the function of
this unit will be reduced to just resetting the most significant bit of the forwarded
time tag value.

Global Decrement Time-Tag Assignment Scheme

In this discussion, for simplicity, we only present the assignment of time-tags to a
set of Active Stations. This scheme can be easily extended to the actual microarchi-
tecture with other accompanying hardware units. We first make the following two
definitions:

- ELAS: Earliest Loaded Active Station is the active station with the earliest
  loaded instruction in the program order. This AS will be the earliest one to
  complete and retire as well.

- LLAS: Latest Loaded Active Station is the active station with the latest loaded
  instruction in the program order.

Note that Active Stations can be loaded and retired in groups but ELAS and
LLAS are only defined based on the program execution order. Once the current
ELAS is committed or squashed, the next AS in the program execution order will
be the ELAS. Forwarded values from each active station are sent over forwarding
buses and snooped by all subsequent ASs which are before LLAS. Values are not
forwarded past LLAS because either active stations are not loaded with valid in-
structions or they are loaded with instruction from the earlier program order. In
our microarchitecture, forwarding units are responsible for blocking the forwarded
values past LLAS. Figure 3.1 shows an execution window with eight Active Stations and associated forwarding buses. Instructions are issued to Active Stations and executed according to the resource flow execution model. In the first time-tag assignment scheme, ELAS always has a time tag value of zero. To maintain this, every time ELAS is retired, the time tags of all other Active Stations are decremented by the number of retired active stations. The time-tag values are shown in Figure 3.1.

**New Time Tag assignment scheme**

In our new proposed scheme, assuming an execution window with \( n \) Active Stations, \( AS_0, AS_1, \ldots, AS_{n-2}, AS_{n-1} \), the time-tag assigned to an instruction in \( AS_i \) is always equal to \( n + i \). In addition to this simple static assignment scheme, there is a time-tag adjustment unit after last physical Active Station. This unit simply subtracts \( n \) from the time tag of the forwarded value. This creates an illusion for the subsequent Active Stations that the forwarded time-tag has come from an Active Station with an actual lower time-tag value. Figure 3.2 is an example of this scheme. This figure shows a snapshot of the execution window with eight Active Stations and their associated time-tags. The LLAS and ELAS are also marked in this figure. As an example of how time tags are manipulated, consider the \( AS_6 \) with \( tt = 14 \). If this AS forwards a value, \( AS_7 \) sees the time tag 14 and since its
TT value is 15, assumes that the value is from an earlier AS, which is the correct case. Once this value reaches the time-tag adjustment unit, the new time tag value will be $14 - 8 = 6$. $AS_0$ through $AS_2$ compare this $tt$ with their own (8, 9 and 10 respectively) and since 6 is a smaller value, they assume data has come from an earlier AS, which is again a correct assumption. Note that $AS_3$ and $AS_4$ do not see the forwarded value because they are following the LLAS. In fact they have not yet been loaded with any instruction. Once they are loaded, LLAS will move forward.

A similar unit is used for backwarding buses with the small difference that $n$ will be added to the time tags. This unit is placed on the backwarding buses between the first physical AS and the last physical AS.

If $n = 2^k$ then $tt - n$ operation can be simply calculated by clearing the most significant bit of the time-tag value. In this case there will be no extra delay on the bus and there is no need for any additional hardware. The cost of this scheme is an extra bit in the time tag values which is insignificant comparing with the saving that can be achieved by eliminating the subtraction units and reducing the complexity of its associated control.
3.4 Register Operation

In this section we introduce a set of transactions that are employed in our Resource Flow execution model and are used to move data and predicates between different Active Stations. We will then explain how the above information is used to enforce the correct program execution.

The three most common instruction operands are register, memory and predicate operands. In our microarchitecture, we treat these operands rather differently. In this section we describe how register operations are handled in our Resource Flow execution model.

We introduce the following transactions and show that they are adequate for handling register operations in our microarchitecture.

**Transaction 1**

trans.name ← RegVal
trans.type ← FW
trans.tt ← AS_i tt
trans.op ← AS_i D
trans.data ← AS_i D.data

**Transaction 2**

trans.name ← Relay
trans.type ← FW
trans.tt ← AS_i tt
trans.op ← AS_i D
trans.data ← AS_i D.last_data

The above transactions forward the destination register operand of an instruction, along with its time tag. If instruction \( I_i \) is disabled, its output register operand
value is also invalid. This transaction informs the later instructions of this, and the previous value of the output operand is forwarded.

When a newly computed value is snooped, the latest value is stored in $AS_i.D.last_{data}$ register. If the predicate value for an instruction changes from disabled to enabled, the output operand that was computed by the instruction is forwarded. If instead the instruction predicate changes from enabled to disabled, the previous value of the output operand (before being changed due to the instruction execution) is forwarded. Using this strategy, instructions that are located in the program ordered future will eventually always get the correct input operand value by the time it is ready to be committed. This is a simple and effective forwarding strategy which makes it a reasonable choice for handling register operations. The inclusion of the time-tag in the transaction is the key element that allows for the correct ordering of dependencies in the committed program.

Another task of an active station is to snoop for new operand values and snarf them if necessary. Figure 3.3 shows the detailed snooping hardware for an input

Figure 3.3: Operand snooping logic within an Active Station (Courtesy of D. Morano).
operand of an Active Station. As can be seen, a new operand is snarfed when it has the same address as of the operand in the Active Station and it is the last instruction before the current instruction which produced a value for this operand.

Snarfing rule 1 summarized the above statement. As the description of the snarfing rule shows, the time-tag value of the snooped operand is compared with the time-tag value of the current Active Station and the time-tag value of the last-snared operand. If the newly arrived operand is later in program ordered time than the previously snooped operand, the newer operand will replace the old value.

**Snarfing rule 1**

\[
\begin{align*}
& (\text{trans.name} = \text{RegVal} \quad \text{or} \quad \text{trans.name} = \text{Relay}) \\
& \quad \text{and} \\
& AS_{i}.op_{x}.\text{last tt} \leq \text{trans.tt} < AS_{i}.tt \quad \text{and} \\
& AS_{i}.op_{x}.\text{addr} = \text{trans.op.addr} \\
& \Rightarrow \\
& AS_{i}.op_{x}.\text{last tt} \leftarrow \text{trans.tt} \\
& AS_{i}.op_{x}.\text{data} \leftarrow \text{trans.data}
\end{align*}
\]

A similar snarfing rule applies to the output operand values.

**Snarfing rule 2**

\[
\begin{align*}
& (\text{trans.name} = \text{RegVal} \quad \text{or} \quad \text{trans.name} = \text{Relay}) \\
& \quad \text{and} \\
& AS_{i}.D.\text{last tt} \leq \text{trans.tt} < AS_{i}.tt \quad \text{and} \\
& AS_{i}.D.\text{addr} = \text{trans.op.addr} \\
& \Rightarrow \\
& AS_{i}.D.\text{last tt} \leftarrow \text{trans.tt} \\
& AS_{i}.D.\text{last data} \leftarrow \text{trans.data}
\end{align*}
\]
CHAPTER 3. RESOURCE-FLOW EXECUTION MODEL

Figure 3.4 is an example of how the Resource Flow model is used to enforce correct execution of instructions by enforcing data dependencies. Part (a) shows a fragment of MIPS assembly-level code. Each instruction is assigned to an Active Station with a unique time-tag. Figure 3.4b illustrates the steps required to execute this code fragment. For simplicity, in this example we assume that there are no branches among these instructions and that all the instructions are guaranteed to be executed. In this example we assume that there are no constraints on the number of execution units and the number of transactions that can be forwarded in each cycle. The first column shows the clock cycle times. The next three columns list the instructions that are either executing, forwarding an operand, or snarfing a snooped operand value. The notation that is used is $I_x(r_y)$, where $I_x$ is the instruction label and $r_y$ is the register that is either forwarded or snarfed. Note that snarfing is done in parallel with instruction execution and transaction forwarding.

(a) Code fragment

<table>
<thead>
<tr>
<th>inst. Label</th>
<th>Time Tag</th>
<th>Inst. Mnemonic</th>
<th>Cycle</th>
<th>Execute</th>
<th>Forward</th>
<th>Snarf</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>1</td>
<td>lui r3,0x8002</td>
<td>-1</td>
<td></td>
<td>B(r28),Iy(r1)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td>+0</td>
<td>I1,I3,I5,I7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td>3</td>
<td>lw r8,-29(r28)</td>
<td>+1</td>
<td></td>
<td>I1(r3),I7(r5)</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td>4</td>
<td>addiu r5,r8,16</td>
<td>+2</td>
<td>I8</td>
<td>I3(r8),I5(r3)</td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td>5</td>
<td>lw r3,-26(r28)</td>
<td>+3</td>
<td>I4,I8,I9</td>
<td>I4(r8),I8(r3),I9(r8)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td>+4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I7</td>
<td>7</td>
<td>addiu r5,1,32</td>
<td>+5</td>
<td>I9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I8</td>
<td>8</td>
<td>xor r8,r1,r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I9</td>
<td>9</td>
<td>sw r8,-32(r28)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Execution schedule

Figure 3.4: Example instruction execution.

At cycle +0, instructions $I_1$, $I_3$, $I_5$ and $I_7$ are executed in parallel. The execution is the result of snarfing new values for registers $r1$ and $r28$ in the previous cycle. Assuming two cycles for the execution of load and store instructions, and one cycle for the rest of the instructions in this example, instructions $I_1$ and $I_7$ will produce their results in the next clock. The new value for registers $r3$ and $r5$ are forwarded in cycle +1 and are snarfed by instruction $I_8$. In the next cycle, instruction $I_8$ will
be executed using its newly read register value. Normally $I_8$ should forward the new result, but since $I_5$ is sending out a new value for $r3$, instruction $I_8$ snarfs the new value of $r3$. This results in re-execution of $I_8$ during cycle +3. In the next cycle, $I_4$ transmits a new value for register $r5$ (with a time-tag value of 4), but since the last value of $r5$ received by $I_8$ had a time-tag of 7 (which is greater than 4), the new value is ignored by $I_8$ and no re-execution is performed.

3.5 Hardware Predication

Predicated execution been shown to be an effective approach for handling conditional branches [87]. In our microarchitecture, predicates are generated at run-time. Each instruction computes its own enabling predicate by snooping for and snarfing predicate operands that are forwarded to it by earlier instructions from the program-ordered past. Since predication is done solely with hardware, the use of legacy code is allowed. Full hardware-based predication is a new way to manage and achieve minimal control dependencies (MCD). With MCD, all branches may execute concurrently, and the instructions after a branch domain may execute independently of the branch. A branch domain is defined as the set of instructions that fall between the branch and its target instruction [88].

We propose a hardware predication mechanism that can be implemented using our resource flow methodology. Figure 3.5 shows a program code sequence with two branches $b_1$ and $b_2$, with corresponding targets $t_1$ and $t_2$. The two branches divide the code into 4 regions. The execution of the instructions in regions $R_1$, $R_2$ and $R_3$ depends on the outcome of the $b_1$ and $b_2$, whereas the instructions in $R_4$ are executed independent of the branch outcomes. A new scheme is used for assigning an enabling predicate to every instruction in each region based on the outcome of branches. If the enabling predicate value is one, then the instruction will be executed; otherwise
it is disabled. The following definitions are used in the description of our scheme.

$T(b)$: a binary value, set to one if the branch $b$ is predicted taken

$en(I_j)$: a binary value assigned to each instruction $I_j$ which specifies whether the instruction is enabled for execution.

$D_b(I_j)$: a binary value, set to one if the instructions $I_j$ is in the domain of branch $b$.

$CEPB(I_j)$: Closest Enabled Previous Branch to instruction $I_j$ in the static program-order.

Figure 3.5 shows an example of two overlapping branches along with the value of the $CEPB$ and $D_b$ variables. The $D_b(I)$ function is independent of the outcome of the branch and only depends on the static order of the instructions in the code. $CEPB(I)$, on the other hand, is a function of the outcome of other branches and will change during the course of speculative execution.
CHAPTER 3. RESOURCE-FLOW EXECUTION MODEL

Using the above definitions, we can show that:

**Theorem 1:**

\[
\overline{en(I_j)} = T(b) \cdot D_b(I_j) \text{ where } b = CEPB(I_j)
\]  

**(3.1)**

**Proof:** The proof is through induction. Assume \( b_1, b_2, \ldots, b_n \) are \( n \) earlier branches before instruction \( I_j \).

If \( n = 1 \) then \( CEPB(I_j) = b_1 \). \( I_j \) is disabled iff it is in the domain of \( b_1 \) and \( b_1 \) is a taken branch. In other words: \( \overline{en(I_j)} = T(b_1) \cdot D_{b_1}(I_j) \), which is the same equation as (3.1).

Now let’s assume that equation (3.1) is valid when there are \( n \) branches before \( I_j \). We show that equation (3.1) is also valid when there are \( n + 1 \) branches before instruction \( I_j \). Since there are only \( n \) branches before \( b_{n+1} \), we can write:

\[
\overline{en(b_{n+1})} = T(b_x) \cdot D_{b_x}(b_{n+1})
\]  

**(3.2)**

where \( b_x = CEPB(b_{n+1}) \). Also from the definition of CEPB, we have:

\[
CEPB(I_j) = \begin{cases} 
\quad b_{n+1} & \text{if } en(b_{n+1}) = 1 \\
\quad CEPB(b_{n+1}) & \text{if } en(b_{n+1}) \neq 1 
\end{cases}
\]  

**(3.3)**

If \( en(b_{n+1}) = 0 \), then branch \( b_{n+1} \) will not have any effect on the execution of \( I_j \). From (3.3) we can see that \( b_x = CEPB(I_j) = CEPB(b_{n+1}) \). Since \( b_x \) is within one of the last \( n \) branches before \( I_j \), it follows that \( \overline{en(I_j)} = T(b_x) \cdot D_{b_x}(I_j) \), which is the same as equation (3.1).

Next we consider the case where \( en(b_{n+1}) = 1 \). Since there are no other branches between \( b_{n+1} \) and \( I_j \), instruction \( I_j \) is enabled unless it is in the domain of \( b_{n+1} \) and \( b_{n+1} \) is a taken branch. In other words:

\[
\overline{en(I_j)} = T(b_{n+1}) \cdot D_{b_{n+1}}(I_j).
\]  

But since \( en(b_{n+1}) = 1 \), from (3.3) we see that \( CEPB(I_j) = b_{n+1} \)
and therefore equation (3.1) is valid.

Qed.

This equation simply tells us that if instruction \( I_j \) is in the domain of an enabled branch, and if the branch is taken, \( I_j \) will not be executed. If the instruction is out of the domain of its closest enabled previous branch, then its execution is independent of the outcome of the branch.

Equation (3.1) is the basis for our dynamic predication scheme. The following two transactions, along with the snarling rule, are used to implement a mechanism for dynamic assignment of predicates to each instruction.

**Transaction 3**

\[
\begin{align*}
\text{trans.name} & \leftarrow \text{PredVal} \\
\text{trans.type} & \leftarrow \text{FW} \\
\text{trans.tt} & \leftarrow \text{AS}_i.tt \\
\text{trans.dir} & \leftarrow \text{AS}_i.outcome \\
\text{trans.target} & \leftarrow \text{AS}_i.target
\end{align*}
\]

**Transaction 4**

\[
\begin{align*}
\text{trans.name} & \leftarrow \text{PredInv} \\
\text{trans.type} & \leftarrow \text{FW} \\
\text{trans.tt} & \leftarrow \text{AS}_i.tt \\
\text{trans.new_tt} & \leftarrow \text{AS.CEPB.tt} \\
\text{trans.dir} & \leftarrow \text{AS}_i.CEPB.outcome \\
\text{trans.target} & \leftarrow \text{AS}_i.CEPB.target
\end{align*}
\]

**Snarfing rule 3**
CHAPTER 3. RESOURCE-FLOW EXECUTION MODEL

\[
\begin{align*}
\text{trans.name} &= \text{PredVal} \quad \text{or} \\
\text{trans.name} &= \text{PredInv} \quad \text{and} \\
\text{AS}_i.\text{CEPB}.tt &\leq \text{trans.tt} < \text{AS}_i.\text{tt} \\
\Rightarrow \\
\text{AS}_i.\text{CEPB}.tt &\leftarrow \text{trans.tt} \\
\text{AS}_i.\text{CEPB}.outcome &\leftarrow \text{trans.outcome} \\
\text{AS}_i.\text{CEPB}.target &\leftarrow \text{trans.target}
\end{align*}
\]

To assign the predicate value to each instruction \( I_j \), it is sufficient to find the \( \text{CEPB}(I_j) \) branch and its outcome. This is a simple task in our scheme. Each Active Station has a pair of \( \text{AS}_i.\text{CEPB}.tt \) registers to hold the time-tag of its current \( \text{CEPB} \) branch and corresponding target. If an earlier branch is enabled, a \( \text{PredVal} \) transaction is sent on the forwarding bus with the branch time-tag, its target address and whether it is a taken branch or not. Each subsequent Active Station such as \( \text{AS}_i \) that snoops this transaction checks to see if the time-tag of the newly enabled branch is greater than current \( \text{AS}_i.\text{CEPB}.tt \) value. If so, the new branch is closer to this instruction and therefore it will replace the older \( \text{CEPB} \) branch.

If a branch such as \( b \) becomes disabled, the later instructions need to find another branch as their new \( \text{CEPB} \) branch. This new branch is simply \( \text{CEPB}(b) \), because according to the definition, there is no other enabled branch between \( \text{CEPB}(b) \) and \( b \). In our scheme this is achieved by initiating a \( \text{PredInv} \) transaction at branch \( b \). An invalidate transaction contains the time-tag of \( b \), the time-tag of \( \text{CEPB}(b) \) and the branch domain information of \( \text{CEPB}(b) \). This information was snooped by \( b \) when \( \text{CEPB}(b) \) forwarded its output predicate.

To find the domain of a \( \text{CEPB} \) branch, we can simply compare the branch target address with the instruction address. It is however more efficient to use time-tags for this purpose. A branch needs to forward the time-tag of its target instruction
instead of its target address. The branch target time-tag can be easily calculated by adding the branch displacement to the branch time-tag value.

Another improvement to the above scheme can be made by observing that a not-taken enabled branch does not need to send out a transaction on the forwarding bus. This is due to the fact that a not taken branch essentially acts like a NOP operation. The advantage is a reduction in the number of transactions on the bus.

Figure 3.6 shows an example of how our dynamic predication scheme works. The first column in the table lists the relative clock cycle. The next three columns list the status of each branch. The "D", "T" and "NT" entries correspond to disabled, taken and not-taken status, respectively. The next two columns list the $I_k$ instruction status. "E" and "D" stand for enabled and disabled status, respectively. The $CEPB$ column shows the $CEPB(I_k)$ branch at any cycle. The next two columns show the transactions on the bus. The "PredVal" column lists the predicate forwarding transactions for each branch, along with its status. The "PredInv" column lists the
invalidating transactions. the branch name in the parenthesis corresponds to the new \textit{CEPB} forwarded by the invalidated branch.

In the example, it is assumed that $b_1$ and $b_3$ are initially predicted taken and not-taken, respectively. As a result $I_k$ is predicated enabled. In the next cycle, $b_1$ changes its direction to a not-taken branch and sends a forwarding transaction on the bus. This transaction will enable $b_2$, which is also predicted taken. $b_2$ will send a predication transaction on the forwarding bus which is snooped by $b_3$. As a result, $b_3$ will be disabled in cycle 3 and will send an invalidating transaction with $b_2$ as its \textit{CEPB} branch. Instruction $I_k$ will see this transaction and switch its \textit{CEPB} to $b_2$ and will be disabled. The second part of the table in figure 3.6 shows what will happen if $b_1$ changes back to taken. A new set of transactions will follow that eventually enable $I_k$.

In the previous example we saw that $I_k$ was first disabled and then enabled. A disabled instruction needs to notify \textit{later} instructions that its destination register value, which could have been forwarded earlier, is invalid. Further, it has to send the old destination value from the closest enabled previous instruction as the correct value. In section 3.4 we showed that a \textit{Relay} transaction can be used to send out the previous value as the current correct value. The disabled instruction, in effect, acts like a relay and will forward the earlier value using its own time-tag.

As demonstrated by this example, the cost of hardware predication is low and the extra state storage only takes a few bits in the Active Station. This hardware budget is the same for all Active Stations and columns.

### 3.6 Memory Operations

With the increasing number of instructions in flight, there is a higher probability that the memory values generated by store operations will be used by other load
CHAPTER 3. RESOURCE-FLOW EXECUTION MODEL

operations in the execution window. Our measurements in section 4.2.1 show the over 30% of the load operations can obtain their operand values directly from a store operation in the execution window without having to go to a higher level of memory hierarchy. This means that if we could provide a way to satisfy a portion of the memory operations in the execution window, we should expect less pressure on the higher levels of memory hierarchy and overall performance improvement.

To enforce the true memory dependencies, we use a simple forwarding and snooping mechanism introduced in section 3.4. We cannot however use the same RegVal transaction. Unlike register operands which have a fixed architected address, the address of a memory operand is not fixed. The memory address is generally computed using a fixed displacement and a register value. Because of this difference, the following transaction is used for forwarding memory values to later load instructions.

Transaction 5

\[
\begin{align*}
\text{trans.name} & \leftarrow \text{MemVal} \\
\text{trans.type} & \leftarrow \text{FW} \\
\text{trans.tt} & \leftarrow \text{AS}_i.\text{tt} \\
\text{trans.mem.addr} & \leftarrow \text{AS}_i.\text{mem.addr} \\
\text{trans.mem.data} & \leftarrow \text{AS}_i.\text{mem.data}
\end{align*}
\]

The associated snarfing rule is:

Snarfing rule 4

\[
\begin{align*}
\text{trans.name} = \text{MemVal} \quad \text{and} \\
\text{AS}_i.\text{mem.last.tt} & \leq \text{trans.tt} < \text{AS}_i.\text{tt} \quad \text{and} \\
\text{AS}_i.\text{mem.addr} & = \text{trans.mem.addr}
\Rightarrow \\
\text{AS}_i.\text{mem.last.tt} & \leftarrow \text{trans.tt} \\
\text{AS}_i.\text{mem.data} & \leftarrow \text{trans.data}
\end{align*}
\]
CHAPTER 3. RESOURCE-FLOW EXECUTION MODEL  

There are, however, limitations to the usefulness of the above transaction. The speculative nature of this microarchitecture could result in temporary incorrect register values for the memory operations. As a result the load or store memory address could be wrong. A load operation that snoops for an incorrect memory address will miss the opportunity to snarf the correct memory value forwarded by a previous store operation. It is therefore necessary for a load operation to eventually initiate a memory request using its resolved memory address. In traditional microarchitectures, the request normally goes to the first level D-cache. In our proposed microarchitecture, there is a good chance that the correct memory value is still in the execution window and has not yet been written back to the higher level memory. Based on this observation, we send memory requests to both higher level memory and previous Active Stations. If there is any store operation with the same address in earlier Active Stations, it will forward the memory value, which will subsequently be snarfed by the load instruction. Otherwise the memory value received from the higher level memory is used.

Requests to the previous Active Stations are handled using the following backwarding transaction. This transaction is similar to the MemVal transaction except that it is primarily used to send data requests to the instructions earlier in the program order.

**Transaction 6**  
\[\text{trans.name} \leftarrow \text{MemReq}\]  
\[\text{trans.type} \leftarrow \text{BW}\]  
\[\text{trans.tt} \leftarrow AS_i.tt\]  
\[\text{trans.mem.addr} \leftarrow AS_i.mem.addr\]

The only data that is sent is the time-tag of the Active Station and the memory address. The closest previous store operation with the same memory address will
forward its value upon receiving the above transaction.

Another difficulty with the application of the forwarding and snooping strategy for memory operations is that a store operation with an incorrect memory address could forward an erroneous value that will be incorrectly snarled by another load operation with the same address. This seems to present a problem for the correct enforcement of memory operand dependencies, but the solution is straightforward.

The load instruction that snooped the first MemVal transaction from the store instructions has kept a copy of the store time-tag in the $AS_{mem.last}^{i,t}$ register. When the load snoops the second MemVal transaction from the store, it notices the difference between its memory address and $trans_{mem.addr}$ forwarded by the store instruction. This means that the store has changed its address and the load needs to initiate a MemReq transaction.

There is however a need for another type of transaction. A store instruction which receives a disabling predicate needs to notify the subsequent load instructions to ignore any value that was previously forwarded by that store instruction. We define MemNul transaction as a new type of RF transaction with the property of nullifying the effect of a previous store transaction.

**Transaction 7**

\[
\begin{align*}
\text{trans.name} & \leftarrow \text{MemNul} \\
\text{trans.tt} & \leftarrow AS_{i,t}
\end{align*}
\]

This transaction is seen by all later load instructions. Any load instruction that has snarled a value sent from that specific store instruction will ignore the value and sends a new request to the memory using a MemReq transaction.
### Table 3.1: Active Station Operation

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Snarfed Data</th>
<th>AS Operation</th>
<th>Transaction enabled AS</th>
<th>Transaction disabled AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>a or b, en_pred, dis_pred</td>
<td>execute last_r ← c</td>
<td>RegVal</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>-</td>
<td>-</td>
<td>RegVal</td>
</tr>
<tr>
<td>Load</td>
<td>a, en_pred, dis_pred</td>
<td>calc addr last_r ← r</td>
<td>MemReq</td>
<td>-</td>
</tr>
<tr>
<td>r ← M[a]</td>
<td>r</td>
<td>-</td>
<td>-</td>
<td>RegVal</td>
</tr>
<tr>
<td>Store</td>
<td>a, en_pred, dis_pred</td>
<td>calc addr</td>
<td>MemNul</td>
<td>-</td>
</tr>
<tr>
<td>M[a] ← r</td>
<td>r</td>
<td>-</td>
<td>MemNul</td>
<td>-</td>
</tr>
<tr>
<td>Branch</td>
<td>a or b, en_pred, dis_pred</td>
<td>compare</td>
<td>PredVal</td>
<td>-</td>
</tr>
<tr>
<td>br a,b,target</td>
<td></td>
<td>-</td>
<td>PredInv</td>
<td>-</td>
</tr>
</tbody>
</table>

### 3.7 Summary

Table 3.1 summarizes the operations carried out by an Active Station, based on the type of instruction issued and the snarfed values. The first column in this table shows the instruction type. The second column shows the snarfed data. The contents of these two columns determine the operation carried out by the active station, as is shown in third column. The last two columns list the transactions that will be initiated by the Active Station based on the value of the enabling predicate of the instruction in the Active Station.

As an example, if an enabled load instruction receives a value for its output operand r, it will save its value in AS.Đ.last_data (last_r in the table). If at some time in the future this same load instruction receives a disabling predicate, it will initiate a Relay transaction and send out the last value of register r as its correct value.
Chapter 4

A Resource Flow

Microarchitecture

In this chapter we describe an ISA-independent, distributed microarchitecture that uses Resource Flow execution to achieve high IPC. Some of the techniques employed on this microarchitecture include control flow speculation and data flow speculation to help extract instruction level parallelism.

This microarchitecture is aggressive in terms of the amount of speculative execution it performs. Rampant speculation is realized through providing a large amount of execution resources. Hundreds of instructions can be dispatched into a distributed architecture and speculatively executed. Multi-path execution is used to hide latencies associated with branch mispredictions.

Hardware scalability of the microarchitecture is achieved through its distributed nature along with repeater-like components that limit the maximum bus spans. Contention for major centralized structures such as register file, reorder buffer and centralized execution units are eliminated through distribution of theses resources. Accesses to a centralized memory hierarchy are also reduced by providing local
buffering throughout our distributed layout.

4.1 Microarchitecture Components

A high-level view of our microarchitecture is shown in figure 4.1. Many components in the figure resemble components found in conventional microarchitectures. The L1 cache is interleaved and replicated to provide the necessary bandwidth for servicing a wide-issue microarchitecture. In our present design the L2 cache is unified. The I-fetch unit can fetch from multiple cache lines and perform multiple branch predictions in a single cycle. The instructions are then decoded and stored in the instruction dispatch buffer and are dispatched to the execution window whenever empty slots are available. The execution window is where our microarchitecture differs substantially from traditional microarchitectures. A large number of instructions can be loaded into the execution window in a single cycle.
4.2 Execution Window

Figure 4.2 shows a partial view of the execution window with its subcomponents. The Active Stations are laid out in a two-dimensional grid. Subsequent blocks of instructions are loaded into each column and their transactions are forwarded down the column and to the top of the next column. Active Stations comprise the distributed control logic necessary for our microarchitecture to correctly route data and predicates to instructions, without the need for any prior setup or data
CHAPTER 4. A RESOURCE FLOW MICROARCHITECTURE

dependency initialization. The distributed nature of Active Stations and their local communications also help to ensure a fast cycle time.

Dispersed among the Active Stations are associated processing elements (PE). Processing elements may consist of a unified all-purpose execution unit capable of executing any of the possible machine instructions or, more likely, consist of several functionally partitioned units individually tailored for specific classes of instructions (integer ALU, FP or other), as is typical of most current machines.

Groups of Active Stations, combined with their associated processing elements, are called a sharing group (SG). Similar to a conventional microarchitecture, sharing groups have a relatively high degree of bus interconnectivity within them. Active Stations serve the role of both the reservation station and the reorder buffer found on more conventional machines. The transfer of a decoded instruction and its associated operands from an AS to its PE are isolated to within the given SG. The use of this execution resource sharing strategy allows us to provide much fewer interconnections between adjacent sharing groups. Only operand results need to flow from one SG to subsequent SGs.

4.2.1 Segmented Buses

An interconnection fabric is provided to forward result operands from earlier Active Stations to later Active Stations in program order. The interconnect allows for an arbitrary number of sharing groups to be used in a machine, while still keeping all bus spans to a constant length. All the buses in Figure 4.2 form the interconnection fabric. The width of the buses can be increased by using several buses in parallel, which will increase the overall bus bandwidth.

Active bus repeater components are required to allow for constant length bus spans. Adjacent bus segments are connected via Filter Units (FU). These units
do more than just repeat operand values from one span of a bus to the next. For
registers and memory, operands are filtered so that redundant forwards of the same
value are eliminated. It should be noted that these repeaters do introduce some
delay into the interconnection path. This filtering provides a means to reduce the
overall bandwidth requirements of the forwarding interconnection fabric.

There are 3 types of filter units, one each for the type of operand that needs to
be communicated between ASs: 1) Register Filter Unit (RFU), 2) Memory Filter
Unit (MFU) and 3) Predicate Filter Unit (PFU). All filter units need to snoop the
forwarding buses and snarf values which have a time-tag greater or equal to the
latest snarfed value. This will ensure that filter units always keep the latest value
for any register or memory operand address. Each RFU holds the most recent value
for each of the architected registers. The register values are then forwarded to the
next bus segment.

The MFU has basically the same structure of an RFU, but uses a small cache
(L0 cache) to store the latest forwarded memory values. A distributed L0 cache
proves to be an effective device in reducing the number of requests to L1 D-cache.
Among our filtering units, the PFU has the simplest design; the PFU only needs to
keep track of the latest CEPI branch and forward it to the next bus span.

Register filter units eliminate the use of a centralized register file and simplify
commitment logic. By the time the instructions in a column have finished executing
and the entire column is ready to retire, the RFU’s in this column have already
forwarded their register values to the RFU’s in the later columns. This eliminates
having to save the ISA register state in a separate register file.

As can be seen in figure 4.2, the buses extend from the bottom of each column
to the top of the next column. The bottom of the last column in the execution
window is also connected to the top of the first column. This will create a loop,
Figure 4.3: Read operations satisfied in single bus span.
though there is no physical first or last column but there are *earliest loaded column* (ELC) and the *latest loaded column* (LLC). The operand values are forwarded from each column to the next column, except for the LLC column which will forward its operand values when a new column is loaded with a new set of instructions.

Bus span length has a major effect on the performance of this microarchitecture. A short bus span means that many operand values have to go through multiple filter units until they are snooped by their dependent instructions. On the other hand, a large bus span might result in increased bus delay, which translates into longer clock cycles. To estimate the effect of the bus span, first we define the *Def-Use distance* metric for register and memory operations.

**Definition**: Consider a pair of instructions \((I_i, I_j)\) in the dynamic program order such that:

1. \(I_i\) has the same output operand as one or more input operands of \(I_j\).
2. There is no other instruction \(I_k\) between \(I_i\) and \(I_j\) with the same output operand as \(I_i\).

We define the Def-Use distance \(DU(I_i, I_j)\) between \(I_i\) and \(I_j\) as the number of instruction in the dynamic program order between \(I_i\) and \(I_j\), including \(I_j\) itself.

We also define \(P_{du}(d)\) as the probability that an instruction has a Def-Use distance of \(d\). Using the above definitions, we will estimate the fraction of all the instructions that have a Def-Use distance less than or equal to a given bus span.

We can assume that each instruction has an equal probability of being assigned to any Active Station. Let \(n\) be the number of Active Stations within a register bus span length and \(m\) be the number of Active Stations per sharing group. Referring to figure 4.2, we can see that all reads from instruction \(I_j\) in the last Active Station \(AS_j\), for which \(DU(I_i, I_j) < n\) will be satisfied by an instruction \(I_i\) which is on in the same
bus span. Similarly, all reads from instruction $I_{j-1}$ for which $DU(I_i, I_{j-1}) < n - 1$ will be also satisfied on the same bus span. Extending this argument to all the instructions in a sharing group, we can show that the ratio of all read operations that can be satisfied by a write operation in the same bus span is:

$$\frac{1}{m} \left[ \sum_{i=1}^{n-1} P_{du}(i) + \sum_{i=1}^{n-2} P_{du}(i) + \cdots + \sum_{i=1}^{n-m} P_{du}(i) \right]$$

This same formula can be used for both register and memory operations by using their corresponding $P_{du}$ probability.

To develop an estimate for the def-use probability $P_{du}$, we evaluated the selected set of programs used throughout this thesis, which represent a subset of the SpecInt-2000 and SpecInt-95 benchmark suites. For these programs we measured the Def-Use distance for each operand address.

Figure 4.3 depicts the above formula for different values of $m$ and $n/m$. The x-axis graphs the bus span in terms of the number of sharing groups in a bus span. There are two sets of graphs in this figure, one for register read operations and another for memory read operation. Different lines in each set correspond to different values of $m$ where $m \in \{1, 2, 4, 8\}$. These graphs are especially interesting because they show that as the bus span increases, the sensitivity to the number of Active Stations per sharing group is reduced. A bus span of 8 captures more than 90% of register operation def-uses. The fraction of captured memory operations, however, is not very high. This is due to the larger memory def-use distance.

### 4.2.2 Rampant speculation

Although we had anticipated that executing with the intermediate speculative values might help to boost the performance through a last-value prediction scheme,
in practice it turns out that partial speculation is more effective. The intermediate values, generated by rampant speculation, lead to unnecessary executions and writes on the forwarding buses which wastes the available bandwidth. Another side effect of rampant speculation is that it will take away the opportunity for employing sophisticated value prediction schemes. The speculative values overwrite the predicted ones and eliminate the potential speedup from using a value predictor. In this section we propose Speculation Invalidation (SI) as a technique for eliminating rampant speculation. We will show that the increase in hardware cost is minor considering the potential of the performance improvement.

Speculation Invalidation mechanism

Figure 4.4 shows a portion of the execution window with the associated sharing groups and forwarding buses. For simplicity only one of the forwarding buses is shown. In a resource flow execution model, each Active Station will forward its output operand which is snooped and potentially snarfed by subsequent active stations with the same operand address. As an example in figure 1, assume that instructions in AS_1 and AS_3 both have the same output operand r_x. Further, assume that instruction in AS_7 also has r_x as one of its input operands. Now if AS_1 forwards a value r_{x,3} on the forwarding bus, AS_7 will snarf this value, re-execute and forward its output operand value upon a change in its value. Later, AS_3 will forward a new value r_{x,3} which will again be snarfed by AS_7 and will possibly cause another forwarding transaction. The main reason that AS_7 had to re-execute is because it did NOT know that AS_3 has the same output operand as AS_1.

We introduce the concept of Speculation Invalidation as a mechanism by which AS_3 could notify later instructions that it has the same output operand address as AS_1. This way, all Active Stations after AS_3 will ignore the forwarded value
Figure 4.4: Hardware implementation of Speculation Invalidation scheme.
and will wait for the forward from $A_{S_3}$ or a later Active Station. Implementation of Speculation Invalidation is fairly simple. A single bit bus (the thin lines in figure 4.4) is originated from each sharing group in a bus span and extends to the bottom of the bus span. We call this single bit line the SI signal. Note that for each forwarding bus, there is a separate SI signal originating from each sharing group. If an Active Station snoops an earlier forwarded value with the same operand address as its output operand address, it will assert its SI signal by setting it to one. The SI signals from previous sharing groups in the same bus span are ORed together and checked by later active stations. If the value of the input SI signal to a SG station is 1, the Active Stations in that sharing group will ignore the forwarded value. The SI signals from all Sharing Groups are also ORed together and fed into the Forwarding Units at the bottom of the bus span. When a FU receives the forwarded value, it also ignores the forwarded value.

An issue that needs to be addressed, is the timing requirement for SI signals. The SI signals can only be set after an Active Station snoops and verifies that the forwarded operands has the same address as its output operand. In our microarchitecture, this will take one cycle and therefore SI signal might not be set until end of the current bus cycles which could be late for other active stations to see it. For this reason, we assume that SI signal is snooped on the next bus cycle. In other words, each Active Station that sniffs an operand, checks the input SI signals on the next bus cycle. If the SI signal is high, the Active Station will discard the snarfed value. This method, however, does not introduce extra delays. Even if the Active Station has already sent the value for execution, it can either abort it or discard the result. In short, the SI signal can be safely snooped on the next bus cycle without imposing any extra delays.

Throughout this section we have assumed that each SI signal originates from
Figure 4.5: An example of implementing Speculation Invalidation scheme in a Sharing Group

A sharing group, although it seems that each Active Station needs to send out an individual SI signal. To reduce the cost of implementing SI, it is quite possible for the active stations in a Sharing Group to use a local simplified version of SI mechanism between themselves and use a shared SI output signal for the whole sharing group. Figure 4.5 shows one way of implementing this mechanism.

The cost for Speculation Invalidation is fairly low. Assuming $n$ sharing groups in each bus span, for each forwarding bus, we only need $n$ SI signals. For larger bus spans, if we assume the bus delay is much larger than gate delay, it is possible to reduce the number of signals to $n/p$ by inserting an OR gate between every $p$ sharing groups.

**Implications of using Speculation Invalidation**

A summary of the advantages of using Speculation Invalidation are as follows:

- Reduced bus activity lowers the bus bandwidth requirements.

- Reducing the amount of re-executions. A direct consequence of this is to
reduce power consumption and provide an opportunity to reduce the total number of Processing Elements or floating point units through sharing them across multiple columns.

- Providing opportunities for implementation of sophisticated value prediction techniques.

- A more efficient resource sharing between mainline and disjoint paths by reducing the execution demand from mainline path instructions.

Some of the features that are affected by Speculation Invalidation are as follows.

**Value Prediction**

Speculation Invalidation can be easily extended to support value prediction. If the forwarded value is generated through value prediction, the transactions can be stamped as being predicted versus being a normal speculative transaction. Future Active Stations that snoop a predicted transaction will not raise their SI output signal. This means that later instructions will use the predicted speculative value as the new value for their input operands and will execute and forward their speculative results. If later on they receive a non-speculative operand value, they compare it with the older snooped value and will only re-execute if the values are different.

**Predicated Execution**

Speculation Invalidation scheme should only be applied to Register and Memory operations. We would like to keep the speculative nature of predicated execution intact in order to allow for control independence. Note that due to the fewer speculative changes in the register values, we expect less intermediate changes in the predicate values as well. If an active station becomes disabled, we use the regular resource flow rules (nullifying and relay forwarding) to resolve dependencies.
Partial Speculation Invalidation

Our simulations have shown that in the absence of a value predictor, partial application of Speculation Invalidation scheme is more beneficial from a performance point of view. Applying Speculation Invalidation to the whole execution window suppresses almost all speculative executions and results in performance loss. It is possible to apply Speculation Invalidation to only a subset of columns that are closer to latest loaded column. As columns get closer to commitment, there is less instructions that need to be executed and therefore speculation can be less costly in terms of bandwidth and resource utilization. We found out that in a configuration with eight columns, it is more advantageous to apply Speculation Invalidation to all columns except the last two columns closer to commitment.

4.3 Fetch and Dispatch

The challenge in high bandwidth instruction fetch is mainly concerned with the problem of multiple-branch prediction and aligning and collapsing of multiple fetch groups. One approach for solving this problem has been to enhance the instruction cache by using multi-ported, multi banked copies of the instruction cache [72, 75, 74, 89]. A more recent approach suggests the use of a trace cache [3, 90, 91].

One view of our microarchitecture is shown in figure 4.6. We are using a set of replicated multiported I-caches to satisfy our high bandwidth I-fetch requirements. Instruction are fetched and loaded into the Active Stations. The key here is that instructions with small branch domain size are normally fetched in the static or memory order. This will simplify the aligning and collapsing logic in the I-fetch unit and allows for higher instruction fetch rate.

To guide the fetch unit, a branch predictor capable of simultaneously predicting
Figure 4.6: High level diagram of our microarchitecture.
multiple conditional branches is used. Examples of this sort of predictor can be found in [74, 92]. We are also using a set of heuristics to guide i-fetch. These heuristics are developed to take advantage of the specific features of our microarchitecture such as dynamic predication and multipath execution. Section 5.4.2 presents our heuristics in context of multipath execution.

4.4 Memory System

Many studies have shown that memory latency tolerance is critical for obtaining high levels of IPC. Look-up-free caches [93], prefetching [94, 95], stream buffers [96] and load prediction [97] are among the many techniques which have been proposed for reducing the number and impact of cache misses.

In our microarchitecture, we introduce a new level in the memory hierarchy right into the execution window. We provide for caching memory values through the use of MFU's and their associated L0 caches. This extra caching also reduces memory bandwidth requirements by satisfying a fraction of loads from the stores still present in the execution window. The store operations forward their memory values which are snooped by MFU's and load instructions. The MFU's save the memory values in their L0 cache to be used for servicing future backwarding requests from load instructions. Cache coherency in L0 becomes a non-issue when we consider that every value has an accompanying time-tag associated with it.

For those memory accesses that do not hit in the L0 cache, the requests need to go to the higher levels of memory. Since a load does not know whether the memory value is in L0 or in L1 cache, to ensure low memory latency, the load needs to send the request both on the backwarding bus as well as to the higher levels of memory. This insures that no cycles are wasted waiting for a response from an MFU in case the memory value is in the L1-D cache.
The next level of memory hierarchy is the L1-D cache. To provide high memory bandwidth for our speculative microarchitecture, L1 data caches are configured as a replicated and interleaved non-blocking cache. Replication provides the necessary bandwidth required by the large execution window.

Committed writes to memory are handled through the Previous Column Buffer (PCB) [12]. Upon commitment of the instructions in a column, the corresponding MFU's in that column are invalidated. This is necessary for limiting the maximum time-tag values in our microarchitecture as well as suppressing unnecessary forwards of old MFU values. The committed stores need to write their memory values to the first level data cache. During the course of execution, the PCB snoops for all forwarded stores in the previous column and saves the latest value for each memory location. Multiple writes to the same memory location are converted into a single write with the last value. At commit time the PCB holds the latest memory values which are written to the L1 data cache through a set of interleaved buses. Updates are made to all copies of the L1 data cache at the same time.

4.5 Execution and Commitment

Instructions that are fetched along one or more predicted paths are decoded and staged into an instruction load buffer and will be available to be loaded into the execution window. Adjacent columns are consequently loaded until no more empty columns are left. From then on, every time the ELC column is retired, a new set of instructions are loaded into the column.

Branch mispredictions can affect instruction execution in different ways. Mis-predicted branches that have their targets within the execution window result in a change of predicates. These changes are handled through application of the resource flow dynamic predication scheme. A branch with a disjoint path which is resolved
as a mispredicted branch, will cause a switch from the mainline to the disjoint path. If the branch is resolved to be mispredicted and the instructions after the branch target had been fetched into the window, the execution window has to be flushed and fetch will resume after the branch.

Active Stations issue instructions to the Processing Elements for execution. In each sharing group, Active Stations with smaller time-tag values are closer to commitment and therefore have a higher priority to execute. Between the mainline and disjoint paths, mainline has higher priority for using a processing element.

A column is retired when all the instructions in the column have been executed and there are no more transactions on any interconnecting buses between the sharing groups in the column. This is locally verified by each Active Station. For example, an Active Station $AS_i$ would check the status of the previous Active Station $AS_{i-1}$. If $AS_{i-1}$ is either committed or disabled and $AS_i$ does not have any more transactions to send, then $AS_i$ is ready to commit. The commitment check is performed during the course of the execution and does not impose any extra delay. Instructions are gradually committed from top of the column while the instructions at the bottom of the column might be still waiting for execution.
Chapter 5

Dynamic Multipath Execution

The impact of control-flow misprediction penalties upon overall performance has been a major issue in the architecture research community [28, 30, 98]. In order to be able to obtain increasing performance as we increase size of instruction window, a large percentage of the loaded instructions need to be on the committed path. Although sophisticated branch prediction schemes increase the probability of fetching instructions from correct execution path, branch misprediction rate is still far from 100%, and remains as one of the major obstacles to achieving improved performance.

Multipath execution techniques have been proposed to reduce the effects of branch mispredictions by executing instructions from both paths after a branch [71]. When the branch resolves, the results computed by instructions on the wrong path are discarded. Given the current trend in increasing the size of instruction window, it is more likely for the issue unit to come across multiple branches before the first branch is resolved. Forking additional disjoint paths from these branches could be beneficial due to coverage of branch mispredictions penalties.

In [1], it was shown that both minimal control dependencies and disjoint eager
execution techniques are needed to achieve high ILP. Disjoint eager execution gives execution priority to the instructions with the highest likelihood of being committed. In our microarchitecture, this likelihood is not explicitly calculated; instead the \textit{static tree} heuristic of [1] is used. This is a form of multi-path execution in which there is a predicted or mainline path, as well as several much shorter not-predicted or \textit{disjoint} paths \textit{spawned} from the mainline path at some conditional branches.

In this chapter we describe the general operation of our microarchitecture in the presence of multipath execution schemes. We will describe both \textit{static} and \textit{dynamic} multipath execution as two alternative approaches for realizing multipath execution on our microarchitecture. Static multipath execution exploits the ability to create a disjoint path by copying instructions from mainline path and as a result simplifies the implementation of the fetch unit.

Dynamic multipath involves a more complex fetch and issuing mechanism, but allows for more flexibility in assigning disjoint paths and overcomes some of the restrictions imposed by the use of static multipath approach.

5.1 General Operation

In order to reduce the resource requirements for the alternate paths and to reduce the complexity of our design, we have picked a set of design options. In the current implementation of our machine, a disjoint path will be spawned at the first unresolved branch, given that a disjoint column is available. Disjoint paths are created by issuing instruction to a column of Active Stations within a SG column. Instructions from both mainline and disjoint paths share the same PE and bussing resources in each sharing group. Active Stations on the mainline path, however, always have priority for the use of resources. In our present microarchitecture, we always have two columns of Active Stations within a SG. The first AS column is reserved for the
main-line path of the program. The second column of Active Stations is reserved for the possible execution of a disjoint path. It is important to note that this is not the only possible implementation for realizing multi-path execution. We did however found this approach to be an efficient and effective scheme.

To identify the transactions associated with each disjoint path, path\_ID’s are used. These are small integer numbers used to distinguish between values generated by each disjoint path. The mainline path has a path\_ID of zero. When a disjoint path is spawned, a transaction with the time-tag of the fork point fork\_t is forwarded. Instructions on the disjoint path snoop for operand values and only snarf those values from mainline path that have a time-tag less than fork\_t. The values with a time-tag greater than fork\_t are only snarfed if they originated from an instruction on the same disjoint path.

In a Sharing Group the mainline and disjoint paths share resources such as PE and forwarding buses. Mainline and disjoint paths from different Sharing Groups execute concurrently and greatly reduce branch misprediction penalties. Path\_ID is used to distinguish instructions and their operand values from multiple paths in an execution window. Conditional branches are assigned to a free disjoint path (the path is spawned) after they enter the execution window. The disjoint path can be assigned to any free column in the execution window, and is not restricted to the column containing the mainline path. We can then issue multiple disjoint paths associated with a single mainline path. At branch resolution time, if the disjoint path turns out to be the correct path, the disjoint path will become the new mainline path and execution will continue from this point.
CHAPTER 5. DYNAMIC MULTIPATH EXECUTION

5.2 Static Multipath Execution

In this thesis we have developed two different approaches for realizing multipath execution. In the first approach, static multipath execution, mainline path and its associated disjoint columns are loaded with the same set of instructions.

The example shown in Figure 5.1 shows a code snippet containing two forward branches, $b_1$ and $b_2$. In each column, enabled instructions are shown in bold. As we mentioned in section 3.5, we have implemented dynamic hardware predication in our microarchitecture. The outcome of the branch predictor is used to initially enable or disable all issued instruction. Instructions on the mainline and disjoint paths are predicated differently according to the prediction of the branch from which the disjoint path was forked. On a disjoint column, predicates are assigned such that instructions on the non-predicted path of each branch are enabled. At branch resolve time, if the branch turns out to be mispredicted, its associated disjoint path will become the new mainline path and fetch will continue from the last instruction in the disjoint column.
CHAPTER 5. DYNAMIC MULTIPATH EXECUTION

In Figure 5.1 $b_1$ is predicted taken and $b_2$ is predicted not-taken. The mainline path $M'$ residing in column $C$, shows enabled instructions. Columns $D_1^c$ and $D_2^c$ are two disjoint columns associated with $b_1$ and $b_2$ branches, respectively.

The main advantage of static multipath execution is that it does not put any extra pressure on the fetch unit. Instruction are fetched according to their static order in memory. This will greatly simplify the design of the fetch unit.

5.3 Dynamic Multipath Execution

Although static disjoint execution has the advantage of simplicity and does not increase fetch bandwidth requirements, it is limited in its ability to handle branches with far targets. In order for a static disjoint path to be spawned from a branch with large domain size, the instructions in the domain of the branch need to be loaded into the instruction window for both mainline and disjoint paths. The domain of a branch includes instructions that fall between the branch and its target instruction. If the branch is resolved to be taken, all the instructions in the branch domain are discarded. For branches that possess a large domain size, the discarded instructions can constitute a high percentage of total instructions in the instruction window.

Another disadvantage of static multipath execution is that it cannot efficiently handle unconditional branches such as jumps and subroutine calls. If a subroutine call is present in the domain of a conditional branch, in order to spawn a disjoint path from the conditional branch it is necessary to load the branch domain and instructions after the branch target. However, the alternate path needs to be fetched from the target of the subroutine call. This is however not possible in an static multipath approach as it would violate the identically of mainline and disjoint paths.

An alternative approach to static multipath execution is to dynamically fetch instructions down multiple paths. Instruction from the predicted path are issued to
the mainline path, and instructions from the non-predicted path of the branch are issued to a disjoint column. The main difference between this approach versus static multipath execution is that the instructions on the mainline and disjoint paths are no longer identical. Mainline instructions follow the predicted path while disjoint path is fetched from the non-predicted path.

Figure 5.2 shows a code snippet, along with its associated dynamic mainline and disjoint paths. We can see that the mainline and disjoint paths are no longer identical. On the mainline path, $M^C$, it is assumed that $b_1$ is predicted not-taken and $b_2$ is predicted taken. Instructions are then fetched according to this prediction. On the disjoint path $D^C_1$, instructions after $b_1$ are fetched from the target of the branch as opposed to being identical to column $C$. In column $D^C_2$, both $b_1$ and $b_2$ are predicted not-taken and fetch continues from instructions in the body of the subroutine.

In order to limit the number of disjoint paths and only execute the most probable paths to commit, additional disjoint paths are not allowed to be spawned from another disjoint path. Therefore, the number of potential disjoint paths for each
CHAPTER 5. DYNAMIC MULTIPATH EXECUTION

mainline path is equal to the number of branches present in the mainline path.

5.4 Issues in Multipath Execution

In this section we examine some of the design issues that are related to supporting multipath execution. We describe new features and algorithms developed in this thesis.

5.4.1 Fetch Bandwidth Requirements

Dynamic multipath execution requires us to perform instruction fetch along multiple paths of execution. This implies that we might need to provide higher fetch bandwidth.

The major challenges in high-bandwidth instruction fetch are: 1) how to accurately predict multiple branches, and 2) how to align/collapse multiple fetch groups. Our approach for increasing instruction fetch bandwidth is to use a multi-ported, multi-banked instruction cache [89]. We evaluate fetch bandwidth requirements using two sets of parameters. The first parameter reflects the width of the fetch unit by specifying the maximum number of instructions that can be fetched in a single cycle. The second parameter specifies the number of non-adjacent basic blocks that can be fetched in one cycle. In chapter 7, we will explore the effects of varying these parameters.

5.4.2 Fetch Heuristics

In our microarchitecture, we have developed a set of heuristics to guide instruction fetch. These heuristics are designed to take advantage of the specific features of resource flow, such as dynamic predication and multipath execution. In the case of forward branches, we define two threshold values: near and far. We then compare
the branch domain size with the threshold values to categorize a branch as *near* (less than the near value), *moderate* (between near and far) or *far* (greater than the far value).

Since the size of the branch domain for a forward branch possessing a near target is small and can be easily fit within the execution window, the fetch unit will load the instructions following the not-taken path of the branch, regardless of its prediction. It is interesting to note that instructions following the not-taken path of a conditional branch will capture hammock-styled branch constructs \[99\]. Simple single-sided hammock branches generally have near targets and so can be captured within the execution window. Our dynamic predication scheme provides an efficient way for handling hammock branches by exploiting control-flow independence. If the branch is mispredicted, only data dependent instructions after the branch domain will need to be re-executed.

For a conditional branch with a moderate-distance target (neither near nor far), we use a confidence predictor similar to the one described in \[90\] to measure the accuracy of the branch predictor. If the branch can be predicted with high confidence, the instructions will be fetched from the predicted path. If the confidence predictor indicates low confidence in this branch, fetch continues from the sequential instructions after the branch. Fetching sequentially is less costly and in case the branch is mispredicted there is no need to flush the execution window. The dynamic predication scheme will change the input predicates of the instructions within the domain of the branch (to become enabled) according to the outcome of the branch and those instructions are re-executed as necessary.

In case of a conditional branch with a far target, we will always follow the predicted path to avoid loading a large number of useless instructions into the execution window. The threshold values are parameters that can be either fixed at design
time or adaptively changed during the course of execution. While we have only experimented with fixed threshold values, we are interested in evaluating dynamic schemes.

If a conditional backward branch is predicted taken, it will be treated as a loop, and fetch/dispatch will follow the taken path of the branch. This policy will basically unroll the loop and fill the execution window with successive iterations of the loop. These instructions will become one of multiple possible speculative paths that can be accommodated simultaneously. The instructions that constitute successive iterations of the loop will make up the mainline speculative path. A disjoint path is also spawned to speculatively execute the instructions following the not-taken outcome of the looping conditional branch. Note that multiple disjoint paths are possible from the not-taken outcome of the conditional loop branch on each speculative loop iteration.

5.4.3 Fetch Priority

Another important issue is how fetch bandwidth is shared between mainline and disjoint paths. Since instructions on the mainline path have a higher probability of reaching commitment than those on a disjoint path, we should give higher fetch priority to mainline path instructions. On the other hand, if mainline fetching totally dominates the available fetch bandwidth, disjoint paths could starve and we will end up with a single-path behavior.

To create a balance, we assign fetch priority such that the number of loaded mainline columns is always equal to or slightly more than total number of loaded disjoint columns. Ideally, a disjoint path will be spawned as close in time to the execution of the associated mainline path. However, since each mainline path could
have many disjoint paths, spawning and fetching all disjoint paths could cause starvation for the mainline path. By keeping the number of disjoint paths close to the number of mainline paths, we are trying to equally divide the bandwidth between them.

5.4.4 Spawning Heuristic

Spawning a separate disjoint path for each branch can be quite costly in terms of resource requirements. Spawning will reduce the available fetch bandwidth for the mainline path, as well as spending resources that could otherwise be used for forking other branches. Ideally, we would like to fork disjoint paths only at mispredicted branches. However, at fetch time we do not know in advance whether a branch would be mispredicted or not. As an alternative, branch confidence predictors are used to predict the correctness of the outcome of a branch predictor. The JSR predictor [90] is an example of such a predictor. Ahuja et al. [71] investigated some other mechanisms that use counters to count the number of mispredictions for each branch.

Our spawning heuristic is based on threshold values defined in 5.4.2. In case of a near branch which has a domain size less than near threshold, we do not spawn a disjoint path. Dynamic predication could effectively handle branch misprediction associated with near branches.

As for moderate branches, a branch confidence estimator is used to determine whether a disjoint path needs to be forked. If the branch has low confidence, a disjoint path will be assigned to it. Otherwise, as was mentioned in 5.4.2, fetching continues from the predicted path without forking any disjoint path.

We always fork a disjoint path for far branches. This is to avoid the penalty of flushing the window in case of a misprediction.
Note that the above classification provides us with the flexibility to choose the best scheme for spawning disjoint paths. Through running simulations, we would be able to find values for near and far thresholds that maximize performance. For example, it might turn out that we get the best performance when near and far targets are equal. This would mean that not using a confidence predictor results in higher performance.

5.4.5 Return Address Stack

One of the requirements to support multipath execution is to have a separate return address stack for each execution path. A unified stack cannot properly operate in a multipath context due to speculative push and pops that would corrupt the contents of the stack.

Although mechanisms for repairing a return address stack after mis-speculation are proposed [100], they are not effective in a multipath execution environment. A simple and effective solution to this problems is to keep a copy of return address stack for each path [101]. We have adopted this approach in our microarchitecture. The main overhead of this approach is due to copying the contents of the disjoint path stack to the mainline path stack. This is only required if switch to a disjoint path is made.

5.4.6 Delayed Fetching

In our microarchitecture, the execution window could be quite large. Thus, the window can contain a large number of conditional branches. The number of simultaneously active disjoint paths is however limited to the number of columns in the execution window. To-be-spawned disjoint paths need to wait for resources to become available. It is even possible for a branch to resolve before its disjoint path
has had a chance to execute. To correctly handle fetch and spawning of disjoint paths, we maintain the information on disjoint paths in two queues.

Each time a branch is fetched on the mainline path, its time-tag and the address of the first instruction on its non-predicted path are stored in *Disjoint path Starting Address Queue* (DSAQ). This address is the starting address of the *waiting* disjoint path corresponding to a branch on the mainline path. Instruction fetch for the disjoint path is deferred until there is an empty disjoint column. The time-tag is used as an index to distinguish between different disjoint paths. If the branch is resolved before the fetch unit is ready to start fetching instructions for the disjoint path, the disjoint path will be discarded and removed from the DSAQ.

Figure 5.3 shows the detailed structure of DSAQ. Two counters, *read address* and *write address*, are used to index the circular queue. Each time a conditional branch is fetched from the mainline path, its time-tag, along with the address of the first instruction on its non-predicted path, are stored in the queue. The write address counter will then point to the next empty entry.

When the fetch unit is ready to fetch instructions for the next disjoint column, it will read the starting address of the disjoint path from DSAQ and will increment the read address counter to point to the next waiting disjoint path. If the branch corresponding to the current entry in the DSAQ is resolved, the read address counter is incremented to point to the next entry.

### 5.4.7 Path Switching

The committed execution will switch from the mainline to the disjoint path whenever a branch is resolved to be mispredicted. The disjoint column will become the new mainline column. Fetching will then then resume from the last instruction in the disjoint column.
One issue that needs to be addressed is how to spawn new D-paths from a disjoint column that is transformed to mainline column. Normally, in order to prevent the number of disjoint paths from growing exponentially, additional disjoint paths are not allowed to spawn from a disjoint path. However, when a switch is made to a disjoint column, we need to spawn additional disjoint paths for the branches in the new mainline column. Since the new mainline path was treated earlier as a disjoint path, it becomes necessary for fetch unit to keep a history of branches and their predictions for each disjoint column.

We provide a *Spawned Disjoint path Queue* (SDQ) to save predictions made for branches on every disjoint path as they are fetched. Once a switch to a disjoint path is made, this queue is accessed to fetch disjoint paths for the current mainline column.

Figure 5.4 shows the hardware implementation of SDQ. Each entry is indexed using the time tag of the mainline branch. SA is the starting address of the first instruction in the new disjoint path. It is the address of the first instruction on the non-predicted path of the first branch on the disjoint path. $P_i$ is the prediction for the $i$th branch on the disjoint path. A valid bit is also maintained for each entry in
The table. Parameter $n$ is the maximum number of branches in a column.

Read address and write address are two counters that perform a similar function as to the counters in DSAQ. Each time a disjoint path $D_k$ is spawned, the time tag of its associated branch ($t_k$), along with the target address of the first branch on $D_k$, and the predictions for all other branches in the disjoint column are written to SDQ.

Each time a disjoint path is squashed, the read address is incremented to point to the next entry in the table. After a switch occurs to a disjoint path, the current entry is read and the table is reset by clearing all entries.

Figure 5.5 shows an example of how the DSAQ and SDQ are used. The first column on the left side of the picture shows a portion of the static code as it is laid out in memory. There are three conditional branches, $b_1$, $b_2$ and $b_3$, which are predicted not-taken, taken and not-taken, respectively. There are three disjoint columns, one associated with each of the three conditional branches. $D_1$ resides in the same column as $M$, and therefore its first AS is disabled. This is necessary for the correct functioning of the resource flow execution model. The prediction for $b_1$ is inverted and fetch continues from the target of $b_1$. DSAQ keeps track of each
### Program Code

<table>
<thead>
<tr>
<th>Program Code</th>
<th>tt</th>
<th>M</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>1</td>
<td>I1</td>
<td>-</td>
<td>b2</td>
<td>b3</td>
</tr>
<tr>
<td>b1 &gt; I5</td>
<td>2</td>
<td>b1</td>
<td>b1</td>
<td>I4</td>
<td>I11</td>
</tr>
<tr>
<td>I2</td>
<td>3</td>
<td>I2</td>
<td>I3</td>
<td>I5</td>
<td>I12</td>
</tr>
<tr>
<td>I3</td>
<td>4</td>
<td>I3</td>
<td>I6</td>
<td>I6</td>
<td>...</td>
</tr>
<tr>
<td>b2 &gt; I7</td>
<td>5</td>
<td>b2</td>
<td>I7</td>
<td>I7</td>
<td>...</td>
</tr>
<tr>
<td>I4</td>
<td>6</td>
<td>I7</td>
<td>I8</td>
<td>I8</td>
<td>...</td>
</tr>
<tr>
<td>I5</td>
<td>7</td>
<td>I8</td>
<td>I9</td>
<td>I9</td>
<td>...</td>
</tr>
<tr>
<td>I6</td>
<td>8</td>
<td>I9</td>
<td>b3</td>
<td>b3</td>
<td>...</td>
</tr>
<tr>
<td>I7</td>
<td>9</td>
<td>b3</td>
<td>I10</td>
<td>I11</td>
<td>...</td>
</tr>
<tr>
<td>I8</td>
<td>10</td>
<td>I10</td>
<td>I11</td>
<td>I12</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DSAQ</th>
<th>SDQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>tt</td>
<td>SA</td>
</tr>
<tr>
<td>2</td>
<td>I5</td>
</tr>
<tr>
<td>5</td>
<td>I4</td>
</tr>
<tr>
<td>9</td>
<td>I11</td>
</tr>
<tr>
<td></td>
<td>Sa</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 5.5: Example of DSAQ and SDQ tables

branch in the mainline column, along with the address of the first instruction on its non-predicted branch path. SDQ keeps track of the branches on each disjoint path in case a switch from the mainline path to one of the disjoint paths is made. In our example, the first entry in SDQ corresponds to the disjoint path of $b_1$ (time-tag = 2) and indicates that $b_3$ on $D_1$ is predicted not-taken. If later, $D_1$ resolves to be the correct path, this entry is used to create a disjoint path for branch $b_3$. 
Chapter 6

Simulator

To evaluate the potential benefits of our new machine model and explore the design space, we have developed $FastLevo$, an architectural simulator that uses a combination of trace-driven and execution-driven techniques to achieve high levels of precision and accuracy. This workload-driven simulation model provides a flexible environment to measure the impact of various design trade-offs. In this chapter, we present the design of our simulator, as well as a discussion of related simulators.

6.1 Simulation Methodologies

The complexity of modern microprocessors has been rapidly increasing. Today’s designs include hundreds of millions of transistors; the growth in complexity has resulted in increasing the design cycle.

Simulators play an important role in predicting the performance of new microarchitectures. They provide a flexible way to explore various design alternatives. Although obtaining highly accurate simulation results is always desirable, there is a direct relationship between simulation accuracy and simulation speed. Highly accurate simulators are typically slow and can result in extending the design cycle.
CHAPTER 6. SIMULATOR

Accelerating the simulation usually requires modeling the system at a higher level of abstraction. In microarchitectural simulators, a commonly used simplification is to only model user-level code, omitting the effects of operating system on the performance. To provide varying levels of accuracy, some simulation environments provide a range of accuracy to select from. Detailed and fast simulation modes are provided to simulate either a complete or sampled run of the application [102].

The most commonly used approach for evaluating a new microarchitectural features is to use a trace-driven or execution-driven simulator. A trace-driven simulator reads instructions and their associated memory references from a trace (typically stored in a file). A trace of a workload can be created by instrumenting a binary and then executing the program. Examples of binary instrumentation tools are ATOM [103] for the Alpha instruction set and Pixie [104] for the MIPS instruction set. Traces can also be generated by emulating a program using software. This approach requires emulation of the operating system calls and library calls that are issued in the workload. Duplicating the effects of these calls can be a challenging task.

Utilizing a trace-driven approach has some advantages. The simulations are highly repeatable (i.e., the trace input does not change over different runs of a simulation). Traces which are created using hardware tracing can include operating system code and interrupt service routines. Another advantage is that a trace-driven simulator only has to implement the subset of instructions that are included in the input traces.

There are however some disadvantages in using a trace-driven simulation approach. First and foremost is the poor performance of this technique. Disk I/O can be the limiting performance factor of the simulator. Also, a fixed trace does not account for the asynchronous nature of some events, especially in multiprocessor
systems. When considering speculation in a microarchitecture, there may be insufficient information provided in a trace to accurately model speculative execution (typically the trace only includes the committed execution).

Execution-driven simulation has been proposed as a method for addressing some of these issues. Using this approach, a program is directly executed on a system and the execution-driven simulation is performed in the same runtime. The main advantage of using execution-driven simulation over trace-driven simulation is the increase in performance. We will avoid having to read trace data from a file stored on disk. Another key advantage of execution-driven simulation is its ability to model speculative execution. Since the entire instruction image is available in the runtime environment, it may be possible to accurately simulate the impact of speculatively-executed instructions. This provides a clear advantage of using execution-driven simulation over using a trace, especially as microprocessor architectures become more speculative in nature.

The increased accuracy and performance provided by an execution-driven simulator is not without a price. Including the effects of speculative execution is not trivial and can introduce significant overhead in the simulation. Since the simulation is performed in the same runtime as the execution of the program, this time dilation could perturb the behavior of the application.

Another disadvantage of using execution-driven simulators is that it will not know the committed execution state until an instruction is actually committed. This information is necessary when assessing perfect prediction or speculation. For example, oracle studies for perfect branch prediction or memory address prediction are more complicated to perform when using execution-driven environments.
6.2 Related Work

Traces have been used to drive such tools as Dinero [105], Cheetah [106] and the Pentium Pro simulator [107]. These environment take as input a stream of instructions or memory addresses. SimpleScalar [102], SMTSIM [108] and Turandot [109] create an instruction trace through emulating instructions. RSIM [110] is an execution-driven simulator designed to study shared memory multiprocessor architectures. RSIM models out-of-order execution through the use of an event-driven engine, but cannot be used for oracle studies such as perfect branch prediction due to the lack of instruction trace information. MAZE [111] was designed as an attempt to add timing information on top of the SimpleScalar simulator. It also uses a checker to compare the result of the execution-driven core with the instruction trace. SimOS [112] is a system-level simulator that provides for different levels of simulation speed and accuracy. MINT [113] is a fast simulator designed to facilitate the process of constructing event-driven memory hierarchy simulation for multiprocessors.

6.3 FastLevo Simulation Environment

To validate the design of our new microarchitecture and evaluate its performance, we developed FastLevo. Our framework uses a combination of trace-driven and execution-driven techniques, and provides features similar to those in other microarchitecture simulation environments such as SimpleScalar and RSIM. Our simulator presently supports the MIPS instruction set architecture and runs under Solaris. The simulator framework can easily be ported to another runtime environment, and additional instruction sets can also be simulated with nominal effort. The output of the simulator includes the number of instructions executed per cycle (IPC), along with rich set of program execution and microarchitecture performance statistics that
include: memory and cache access/hit rates, branch prediction rates, bus utilization, etc..

We decided to add a trace-driven engine to our simulator in order to provide us with early results of instruction execution in order to carry out oracle studies. This feature allows us to explore the limits of instruction level parallelism as a function of perfect hardware components. Knowledge of performance boundaries of our new microarchitecture has allowed us to focus our efforts on enhancing features that would result in the greatest performance gain.

Execution traces are created either by using the Pixie tracing tool on SGI environment or through the use of SimpleSim\(^1\). Pixie traces do not provide register and memory values, but they provide a stream of the committed instructions that include library routine execution and most of the system calls. SimpleSim is a functional simulator that provides register and memory values for executed instructions. The system calls are handled through emulation.

The execution-driven core is needed in order to validate the microarchitecture and gather realistic performance measurements. Instructions are executed and their results are used to speculatively execute other instructions (using speculative values). Accurate timing models are achieved through detailed modeling of bus latencies, memory hierarchy latencies and mispeculations latencies.

To achieve higher flexibility in evaluating the design space, hardware components are modeled at a functional level and use parameters that can be specified at compile-time. Functional level modeling provides a concise interface of a component while eliminating the need for its detailed implementation. Timing constraints are precisely modeled and delays could be parameterized.

Using compile-time parameters has the advantage of speeding up the simulation

\(^1\)SimpleSim is an execution driven simulator, developed by David Morano in conjunction with this research.
by eliminating run-time checking overhead. The extensive use of parameters also creates a flexible framework that allows us to efficiently evaluate a large range of design alternatives.

Another attractive feature of our simulator is that at commit time, the state of the execution-driven simulator is compared with the state of the trace-driven simulator. This checking process provides a powerful debugging tool that can catch errors early in the simulation process. Self-checking logic has been developed to test the integrity of the simulator during the execution of the program.

Execution speed is another factor that was considered during the design of our simulator. Workload-driven simulation requires simulation of many benchmarks over a large number of instructions, and across a variety of different microarchitecture configurations. We have paid attention to reducing the execution time of the simulator, including:

- Instructions are decoded and stored in an internal simulator buffer to be used for future references. This state caching technique greatly reduces the runtime decoding overhead. A similar approach was used in the Shade simulator [114]. Future references to a particular instruction use a pointer to index into the decoded data structure.

- Compiler directives (#ifdef and #endif) are used extensively to conditionally build the simulator based on the selected configuration. This eliminates the overhead of performing dynamic checks to determine the processor configuration and leads to more efficient execution.

- Multiple representations of data structures are provided in order to facilitate search and update operations in the event queues and event lists.

Using these optimizations, we are able to execute 5-15 KInst/Second on a SUN
Ultra-Enterprise workstation with 248MHz Sparcv9 processor and 1.5 GB memory, running under Solaris 5.8.

In the next section we will describe the internal structure of FastLevo in more detail.

### 6.4 FastLevo Simulator Architecture

Figure 6.1 shows the high-level organization of our simulator. micro architectural modeling is achieved through modeling of hardware components at a functional level, along with a detailed modeling of the interconnection bus fabric.

Simulation is carried out through the use of event lists and a global clock. The system activity is modeled as a series of events. Based on the delay of each hardware component, events are scheduled to occur at a scheduled time in future. At each
Figure 6.2: Partial list of the event queue.

clock cycle, events that have a time stamp less than or equal to the current simulation time are activated and will trigger the re-evaluation of their corresponding hardware models. In each simulation pass, the state of each hardware component is examined. Input events trigger an evaluation of the simulator component and possible assignment of new events on the output lines. A Processing Element is a good example of a component, where the output value is evaluated and assigned as a result of an event on the input ports.

In addition to active simulation components, passive components such as buses also have an associated event list to indicate what values will appear on them at each clock cycle. New assignments will be delayed until there are empty slots on the bus.

Another element in our simulator is the event queue, which is used to keep track of the set of pending events for each simulation component. Each component updates its corresponding entry in the event queue using the time stamp of the next event in its event list. The event queue is also used to keep track of certain events, such as memory store operations that may need to be referenced by a later memory load operations. The event queue reduces the simulation time by only evaluating
hardware components that have an active event. Figure 6.2 shows a partial list of the Event queue and its entries. Each entry represents the list of hardware components that have a pending event at the specified simulation time.

Using event lists has the advantage of reducing the simulation time by only updating components that have a pending event on their inputs. Another advantage of using the event queue is that it reduces the number of simulation passes. For example, in the case of an Active Station, in a single simulation pass a set of events can be added to the event list. Each event corresponds to one or more events on the inputs of the Active Station. This is possible because earlier Active Station have priority over later ones. The only exception is when an Active Station sends a backwarding request. In this case, another pass of the earlier Active Stations might be required. The event queue can effectively manage these special cases.

Figure 6.3 shows a Sharing Group with two Active Stations, a Processing Element, and a forwarding bus with a forwarding unit. Associated with each simulation component is an event list. Lists are sorted by time, so that events can be indexed and processed serially. At each clock tick the first element of the event list is examined, and if the event time is equal to current clock cycle, the first event is activated.

The memory system is modeled through simulation of level-1 and level-2 instruction and data caches. In the case of a level-2 cache miss, an additional delay for accessing the main memory is modeled. To be able to study realistic memory performance, all bus contention for accessing memory banks is modeled faithfully.

Figure 6.4 shows the pseudo code for the main loop of the simulator. During each pass, if the fetch queue is empty, an attempt is made to fetch a new column of instructions. Upon availability of a column, instructions are dispatched to the empty active stations. The simulator will then walk through each simulator component to update their state based on the input values of activated events. A new set of events
Figure 6.3: Partial Event List for a Sharing Group
CHAPTER 6. SIMULATOR

```
while (number of simulated instructions < number of instructions to simulate) {
    if (EIC column is ready to commit)
        commit column;
    if (there is an empty column)
        load a new column;
    if (free slots in fetch buffer)
        fetch instructions;
    for each SIMComponent in EventQueue[PresentTime] {
        Evaluate(SIMComponent); /* updates Event List and EventQueue */
    }
    ++PresentTime;
}
```

Figure 6.4: Pseudocode of the flow through the simulator.

are then scheduled. Once there are no other active events in a column, the column
is ready to commit and loaded with a new set of instructions.

6.4.1 Limitations

Although we have tried to develop a very detailed simulator, there are certain fea-
tures that we thought might not be essential for the purpose of evaluating the
performance of our microarchitecture and could be implemented in future versions.
Among these features include:

- machine exceptions,

- multi-threading support and

- a TLB.

In the present implementation we provide limited floating-point MIPs instruction
support since we have been mainly focused on integer workloads. Also, our simulator
currently does not include a model for value prediction, but this could be added
easily since we model all data values faithfully.
6.4.2 Validation

Presently there is no register transfer level model for our microarchitecture to compare against. Manual verification, along with our cross-checking logic, is used to detect inconsistent results. By performing detailed modeling of the microarchitecture and adopting a hybrid approach to perform runtime checking, we have high confidence in the results generated by our simulation environment.

To facilitate the validation process we developed an execution visualization feature that facilitates detailed tracking the execution of a workload. This feature generates a list of Active Stations, along with their committed execution timing. Figure 6.5 shows a snapshot of this output file. Using this information, we can check the timing of each operation.

6.4.3 Design Space Exploration

The two main criteria for evaluating a simulation environment are the level of detail provided, and the overall simulation performance. Using a range of configurations that included ideal and realistic design assumptions, we were able to explore the design space by measuring IPC. Using ideal design assumptions provides us with an upper limit on the amount of performance gain that can be achieved through with a particular optimization. Of course, we then use realistic values to compare our microarchitecture with other modern architectures.

In Chapter 7, we will present simulation results for a set of benchmarks and a set of configurations. These results are only available because of the availability of effective simulation tools to determine the performance of our microarchitecture.
---SG 0 -------
F1: R18 [R36, R35+fu(1)]*arb(0)+bus(1)=937
F1: R15 [R36, R35+fu(1)]*arb(2)+bus(1)=939
F1: R24 [R36, R35+fu(1)]*arb(3)+bus(1)=940
F1: M0 xf f2e8 [load(0), R36+fu(1)]*arb(0)+bus(1)=937, die at 1
F1: M0 xf f2e4 [load(0), R36+fu(1)]*arb(1)+bus(1)=938, die at 0
P=1 Ox:66 ffa0 move 12, -1, 2;
op [R36, R35]*x(1)*arb(4)+bus(1)=941
P=1 Ox:66 ffa8 addiu 29, -1, 29;
op [R39, R35]*x(1)*arb(1)+bus(1)=942
P=1 Ox:66 ffa8 addiu 31, -1, 29:
op [R39, R35]*x(1)*arb(0)+bus(1)=944
---SG 1 -------
F2: R28 [R36, R35+fu(1)]*arb(0)+bus(1)=937
F2: R28 [R36, R35+fu(1)]*arb(1)+bus(1)=938
F2: M0 xf f2f4 [load(0), R36+fu(1)]*arb(1)+bus(1)=938, die at 1
P=1 Ox:66 ffa0 move -1, -1, -1;
op [R36, R35]*x(0)*arb(0)+bus(0)=936
P=1 Ox:66 102c lw 29,0x f2f08, 7; 
mbus(1)*arb(0)+cache(0)+op[R42, R43]=x(1)+arb(0)+bus(1)=946 
P=1 Ox:66 1030 beq 2, 0x661048:
op [R41, R35]=x(1)+arb(0)+bus(1)=946, bsp .D0 (336) 
P=1 Ox:66 1034 li -1, -1, 8;
op [R36, R35]=x(1)+arb(2)+bus(1)=939
---SG 2 -------
F3: R3 [R36, R36+fu(1)]*arb(2)+bus(1)=939
F3: R11 [R36, R35+fu(1)]*arb(2)+bus(1)=939
F3: R12 [R36, R35+fu(1)]*arb(3)+bus(1)=940
F3: M0 xf f2b0d [load(0), R36+fu(1)]*arb(0)+bus(1)=937, die at 1
F3: M0 xf f2e4 [load(0), R36+fu(1)]*arb(0)+bus(1)=937, die at 0
F3: M0 xf f2a0 [load(0), R36+fu(1)]*arb(1)+bus(1)=938, die at 0
F3: M0 xf f20 [load(0), R36+fu(1)]*arb(1)+bus(1)=938, die at 0
P=0 Ox:66 104c lw 29,0x f2f0c, 9; 
mbus(1)*arb(0)+cache(0)+op[R42, R42]=x(1)+arb(0)+bus(1)=946 
P=0 Ox:66 104c sw 8, 29,0x f2f08; 
op [R36, R39, R42]=x(1)+arb(0)+bus(1)=944 
P=0 Ox:66 106c move 7, -1, 4; 
op [R36, R46, R36]=x(1)+arb(0)+bus(1)=947 
P=0 Ox:66 106c addiu 29, -1, 5; 
op [R36, R42, R36]=x(1)+arb(0)+bus(1)=944
---SG 3 -------
F4: R18 [R36, R37+fu(1)]*arb(0)+bus(1)=939
F4: R29 [R36, R36+fu(1)]*arb(0)+bus(1)=944
F4: M0 xf f2e4 [load(0), R36+fu(1)]*arb(0)+bus(1)=940, die at 0
P=0 Ox:66 106c sw 9, 29,0x f2f08; 
op [R36, R44, R36]=x(1)+arb(0)+bus(1)=946 
P=0 Ox:66 106c move 1, -1, 1; 
op [R36, R36, R36]=x(1)+arb(0)+bus(1)=938, ND (336) 
P=0 Ox:66 106c addiu 29, -1, 29; 
op [R36, R44, R36]=x(1)+arb(0)+bus(1)=946

Figure 6.5: Partial list of output report
### Table 6.1: Microarchitectural Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sharing Groups</td>
<td>8</td>
</tr>
<tr>
<td>AS per SG</td>
<td>4</td>
</tr>
<tr>
<td>Columns</td>
<td>8</td>
</tr>
<tr>
<td>SG bus span</td>
<td>3</td>
</tr>
<tr>
<td>Maximum D-path columns</td>
<td>4</td>
</tr>
<tr>
<td>Register forwarding bus width</td>
<td>2</td>
</tr>
<tr>
<td>Memory forwarding bus width</td>
<td>2</td>
</tr>
<tr>
<td>Predicate forwarding bus width</td>
<td>1</td>
</tr>
<tr>
<td>Memory backing interleave bus width</td>
<td>1</td>
</tr>
<tr>
<td>Memory forwarding interleave factor</td>
<td>2</td>
</tr>
<tr>
<td>Bus delay</td>
<td>1</td>
</tr>
<tr>
<td>Forwarding Unit delay</td>
<td>1</td>
</tr>
<tr>
<td>Main memory access delay</td>
<td>100</td>
</tr>
<tr>
<td>Near branch threshold</td>
<td>4</td>
</tr>
<tr>
<td>Far branch threshold</td>
<td>4</td>
</tr>
<tr>
<td>Instruction fetched per cycle</td>
<td>32</td>
</tr>
<tr>
<td>Noncontiguous basic blocks fetched per cycle</td>
<td>4</td>
</tr>
<tr>
<td>L1 cache interleave factor</td>
<td>4</td>
</tr>
<tr>
<td>L1 cache set associativity</td>
<td>4</td>
</tr>
<tr>
<td>L1 cache block size</td>
<td>32</td>
</tr>
<tr>
<td>L1 cache size</td>
<td>64 MB</td>
</tr>
<tr>
<td>L1 cache hit delay (cycles)</td>
<td>1</td>
</tr>
<tr>
<td>L2 cache block size</td>
<td>32</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>2 MB</td>
</tr>
<tr>
<td>L2 cache hit delay</td>
<td>10</td>
</tr>
<tr>
<td>L1 instruction cache associativity</td>
<td>4</td>
</tr>
<tr>
<td>L1 Instruction cache size</td>
<td>64 KB</td>
</tr>
<tr>
<td>L1 Instruction cache block size</td>
<td>32</td>
</tr>
<tr>
<td>PBHT size</td>
<td>1024</td>
</tr>
<tr>
<td>GPHT size</td>
<td>4096</td>
</tr>
<tr>
<td>BTB size</td>
<td>256</td>
</tr>
</tbody>
</table>
Chapter 7

Experimental Methodology

In this section we evaluate our microarchitecture using FastLevo simulator. We have chosen to use as input to our simulator a subset of the SPECInt 95 and 2000 benchmark suites. This suite represents a set of control-dominated programs (i.e., dominated by difficult to predict branch instructions), that are difficult to obtain instruction level parallelism from directly. Our results show that by using both disjoint execution and data path caching with our machine microarchitecture we can expose higher degrees of instruction level parallelism.

7.1 Simulator Configuration

The simulated microarchitecture supports the MIPS-1 ISA, with some MIPS-2 instructions also supported in order to accommodate code residing in the SGI system libraries. The simulator is extensively parameterized, which allows evaluation of many aspects of the design space. All architectural features of the microarchitecture are accurately modeled, taking into account timing delays and queuing effects. The modeling of Forwarding Units, Active Stations and Processing Elements is done
at the functional level. The delays in this model are programmable to allow for extensive sensitivity studies.

Table 7.1 shows the default configuration of the simulated machine. These machine characteristics are representative of design parameters for a 2003-vintage microprocessor such as the recent Pentium-4 (.13 um at 2.4 GHz) processor.

A particular machine is generally characterized using the tuple:

- Sharing Group rows
- Active Station rows per Sharing Group
- Sharing Group columns
Table 7.2: Benchmark Statistics

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>22.5%</td>
<td>10.1%</td>
<td>10.9%</td>
<td>6.1%</td>
<td>89.5%</td>
</tr>
<tr>
<td>compress</td>
<td>18.4%</td>
<td>9.7%</td>
<td>11.2%</td>
<td>8.0%</td>
<td>84.6%</td>
</tr>
<tr>
<td>crafty</td>
<td>23.3%</td>
<td>17.5%</td>
<td>8.0%</td>
<td>7.4%</td>
<td>87.2%</td>
</tr>
<tr>
<td>gcc</td>
<td>23.5%</td>
<td>15.0%</td>
<td>14.8%</td>
<td>11.1%</td>
<td>92.0%</td>
</tr>
<tr>
<td>go</td>
<td>21.3%</td>
<td>8.6%</td>
<td>11.8%</td>
<td>9.6%</td>
<td>80.8%</td>
</tr>
<tr>
<td>gzip</td>
<td>20.7%</td>
<td>8.4%</td>
<td>8.0%</td>
<td>5.7%</td>
<td>84.3%</td>
</tr>
<tr>
<td>jpeg</td>
<td>18.2%</td>
<td>7.9%</td>
<td>5.7%</td>
<td>3.5%</td>
<td>86.4%</td>
</tr>
<tr>
<td>mcf</td>
<td>23.6%</td>
<td>9.5%</td>
<td>17.3%</td>
<td>13.8%</td>
<td>97.4%</td>
</tr>
<tr>
<td>parser</td>
<td>28.5%</td>
<td>8.8%</td>
<td>11.1%</td>
<td>8.8%</td>
<td>90.6%</td>
</tr>
<tr>
<td>vortex</td>
<td>29.1%</td>
<td>19.7%</td>
<td>8.8%</td>
<td>7.2%</td>
<td>98.4%</td>
</tr>
<tr>
<td>Avg</td>
<td>22.9%</td>
<td>11.5%</td>
<td>10.8%</td>
<td>8.1%</td>
<td>89.1%</td>
</tr>
</tbody>
</table>

These three parameters define the size and performance of a machine configuration, and together they are called the *geometry* of the machine. The tuple is given as three concatenated numbers (e.g., the geometry of the machine in Figure 4.6 would be abbreviated 3-4-3.)

### 7.2 Benchmark Characteristics

We have selected a subset of two SPECInt benchmark suites to drive the simulator. Table 7.2 lists the programs used, as well as a breakdown of the dynamic count of load, store and branch instructions. Seven programs were selected from the SPECInt-2000 suite and three from the SPECInt-95 suite. All programs were compiled using the SGI compiler on the SGI IRIX 6.4 operating system. Programs were compiled with standard optimization (-O) using (-mips1) switch. The first 600 million instruction of all programs were executed and data was only collected after the execution of the first 100 million instructions.

Using the simulator, we measured the effects of changing different microarchitectural parameters to characterize the IPC obtainable on this microarchitecture and quantify the sensitivity of the IPC to the variations in the parameters of our microarchitecture.
CHAPTER 7. EXPERIMENTAL METHODOLOGY

Table 7.3: IPC as a function of geometry

<table>
<thead>
<tr>
<th>SG</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>8</th>
<th>8</th>
<th>8</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>8</th>
<th>8</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS/SG</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>Cols</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>beisp2</td>
<td>3.1</td>
<td>3.5</td>
<td>4.3</td>
<td>5.7</td>
<td>5.9</td>
<td>5.9</td>
<td>6.3</td>
<td>6.0</td>
<td>6.3</td>
<td>7.0</td>
<td>7.3</td>
<td>7.7</td>
</tr>
<tr>
<td>compress</td>
<td>2.6</td>
<td>3.1</td>
<td>3.8</td>
<td>5.1</td>
<td>5.3</td>
<td>5.4</td>
<td>5.4</td>
<td>5.6</td>
<td>5.8</td>
<td>6.3</td>
<td>6.7</td>
<td>6.7</td>
</tr>
<tr>
<td>crafty</td>
<td>2.5</td>
<td>3.2</td>
<td>3.6</td>
<td>4.9</td>
<td>5.2</td>
<td>5.2</td>
<td>5.5</td>
<td>5.6</td>
<td>5.8</td>
<td>6.2</td>
<td>6.9</td>
<td>7.4</td>
</tr>
<tr>
<td>gcc</td>
<td>2.7</td>
<td>3.1</td>
<td>3.7</td>
<td>5.1</td>
<td>5.2</td>
<td>5.2</td>
<td>5.6</td>
<td>5.6</td>
<td>5.8</td>
<td>6.1</td>
<td>6.6</td>
<td>7.0</td>
</tr>
<tr>
<td>go</td>
<td>2.3</td>
<td>2.5</td>
<td>3.0</td>
<td>3.8</td>
<td>3.9</td>
<td>4.2</td>
<td>4.0</td>
<td>4.1</td>
<td>4.5</td>
<td>4.7</td>
<td>4.8</td>
<td>4.8</td>
</tr>
<tr>
<td>gzip</td>
<td>2.7</td>
<td>2.7</td>
<td>3.5</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>4.8</td>
<td>4.6</td>
<td>4.6</td>
<td>5.1</td>
<td>5.3</td>
<td>5.4</td>
</tr>
<tr>
<td>jpeg</td>
<td>2.7</td>
<td>3.7</td>
<td>4.0</td>
<td>6.9</td>
<td>7.5</td>
<td>7.8</td>
<td>7.3</td>
<td>7.8</td>
<td>8.1</td>
<td>9.5</td>
<td>10.8</td>
<td>11.3</td>
</tr>
<tr>
<td>ncf</td>
<td>3.0</td>
<td>3.4</td>
<td>4.1</td>
<td>5.4</td>
<td>5.8</td>
<td>6.2</td>
<td>5.6</td>
<td>5.8</td>
<td>6.1</td>
<td>6.6</td>
<td>7.3</td>
<td>8.0</td>
</tr>
<tr>
<td>parser</td>
<td>2.8</td>
<td>3.2</td>
<td>4.0</td>
<td>5.2</td>
<td>5.5</td>
<td>5.6</td>
<td>5.6</td>
<td>5.7</td>
<td>6.0</td>
<td>6.7</td>
<td>7.4</td>
<td>7.9</td>
</tr>
<tr>
<td>vortex</td>
<td>3.1</td>
<td>3.9</td>
<td>4.5</td>
<td>6.4</td>
<td>7.0</td>
<td>7.4</td>
<td>7.0</td>
<td>7.7</td>
<td>8.4</td>
<td>8.8</td>
<td>10.1</td>
<td>10.8</td>
</tr>
<tr>
<td>hmean</td>
<td>2.7</td>
<td>3.2</td>
<td>3.8</td>
<td>5.2</td>
<td>5.4</td>
<td>5.5</td>
<td>5.6</td>
<td>5.6</td>
<td>5.8</td>
<td>6.4</td>
<td>6.9</td>
<td>7.2</td>
</tr>
</tbody>
</table>

7.3 Simulation Results

We have collected a range of statistics that characterize the execution of the target programs. The default simulation parameters are listed in Table 7.1 (except in the cases where we measure the effect of changing particular design parameters).

7.3.1 Geometry Effects on Performance

We have collected IPC results for a range of machine sizes. All machines simulated contained two forwarding buses for register operands, two forwarding buses for memory operands and one forwarding bus for predicates. All forwarding buses incur a bus transfer delay of one cycle. Each Forwarding Unit also has a minimum latency of one cycle. All simulated machines, regardless of the size, have a forwarding bus span of eight SG’s. The constant bus span is one of the important features of this microarchitecture that allows for physical scalability of the machine.

Table 7.3 lists the IPC for all simulated configurations sorted by harmonic mean of the IPC. To clearly illustrate the effect of changing each geometry parameter, we use three sets of graphs. In each set of graphs, two geometry parameters are kept constant while the third parameter is changed.
Figure 7.1 shows the effect of changing the number of Sharing Groups on IPC. For each geometry, the maximum number of static disjoint paths which are allowed to be spawned is equal to the number of columns in the machine. We use the previously described tuple to label different machine geometry. Starting from left to right, the tuple contains the number of rows of Sharing Groups, the number of Active Stations per Sharing Group and the total number of columns, respectively. The product of the three numbers gives the total number of Active Stations in the machine, which is also the maximum number of instructions that may be in flight.

As can be seen from figure 7.1, by increasing the number of Sharing Groups, the resulting IPC also increases. The main gain in IPC is achieved by going from a 4-4-8 configuration to an 8-4-8 configuration. Further increases in the number of Sharing Groups has less impact on the IPC. As the size of the machine increases, it becomes more difficult to fill the instruction window with instructions from the correct execution path. Although multipath execution reduces the ill effect of mispredictions,
it cannot completely eliminate them.

Figure 7.2 shows the effect of increasing the number of Active Stations per Sharing Group. Again we see an increase in performance as the machine size increases, although the rate of speedup reduces as the machine size increases.

The last graph in this group shows the effect of increasing the total number of columns on IPC. This graph is shown in figure 7.3.

Figure 7.2: Effect of the number of Active Stations per Sharing Group on IPC.
Figure 7.3: Effect of the number of columns on IPC.
CHAPTER 7. EXPERIMENTAL METHODOLOGY

Table 7.4: Idealized Configurations

<table>
<thead>
<tr>
<th></th>
<th>Ideal Fetch / Mem</th>
<th>Ideal Fetch and Memory, 100% cache hit rate, Oracle Branch Predictor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ideal Fetch</td>
<td>Ideal Fetch, Realistic Memory, Oracle Branch Predictor</td>
<td></td>
</tr>
<tr>
<td>Ideal Mem</td>
<td>Ideal Memory, Realistic Branch Predictor, 100% data cache hit rate</td>
<td></td>
</tr>
<tr>
<td>Realistic</td>
<td>Realistic Fetch and Memory</td>
<td></td>
</tr>
</tbody>
</table>

The reason behind the diminishing returns on IPC as the size of the machine increases is due to the limits imposed by conditional control flow and centralized memory on our microarchitecture. In the next set of experiments, we evaluate an ideal branch predictor and an ideal memory system to gain a better understanding of the potential speedup for this microarchitecture. We ran simulations for the 4-8-4, 8-4-8, 8-8-8 and 8-16-8 geometry using four different configurations, as shown in Table 7.4.

![Figure 7.4: Comparison of ideal and realistic configurations for a 4-8-4 machine geometry.](image)

Our ideal fetch model assumes an oracle branch predictor model (a branch predictor with 100% prediction accuracy) for the fetch unit and includes unlimited fetch bandwidth. Our ideal memory model assumes a 100% data cache hit rate and
unlimited memory bandwidth. Performing simulations using ideal fetch and memory models will help to establish bounds on the amount of parallelism that can be extracted from the code using this microarchitecture. This bound, however, does not take into account the potential speedup due to the application of data value prediction technique.

Figure 7.4 shows the IPC obtained for a 4-8-4 configuration. This graph shows there is still more opportunity for improving the IPC, especially through the reduction of branch mispredictions.

Figure 7.5, 7.6 and 7.7 show that by increasing the execution window size, the gap between ideal and realistic IPC is increased. These results show that there is still potential for a better branch handling schemes.

Another conclusion that can be drawn from these results is that the memory system performs well, regardless of whether we use ideal memory or not. This leaves the I-fetch unit as the primary target for improvement.

![Bar Chart](image.png)

Figure 7.5: Comparison of ideal and realistic configurations for an 8-4-8 geometry.
Figure 7.6: Comparison of ideal and realistic configurations for an 8-8-8 geometry.

Figure 7.7: Comparison of ideal and realistic configurations for an 8-16-8 geometry.
7.3.2 Multipath Execution

Figure 7.8 shows the benefits of performing multipath execution on the performance. Simulations are performed using an 8-4-8 geometry. The No D-path column assumes single-path execution, along with the use of predicated execution. Forward branches with a domain size less than or equal to 4 instructions are fetched statically and predicated. All other branches are fetched from the predicted path. Our dynamic multipath models limit the maximum number of disjoint paths to 8.

For comparison purposes, we also simulated a multipath configuration with static disjoint path spawning. In our static model, both mainline and disjoint paths are loaded with the same set of instructions. The results show a substantial improvement for multipath execution over single-path execution. We can also see that dynamic multipath execution outperforms static multipath execution at the expense of a more complex fetch unit.
Figure 7.9: IPC as a function of the number of dynamic disjoint paths.

**Number of Disjoint Paths**

The performance of our multipath execution scheme is dependent on the number of disjoint paths that are allowed to concurrently execute. Figure 7.9 shows the relative speedups with respect to a single path configuration as a function of the number of disjoint paths. Our results show that performance increases as the number of paths increases. Performance, however, does not significantly improve for configurations with more than five disjoint paths. One explanation for this behavior is that as the number of disjoint paths increases, the physical distance between later branches on the mainline path and their associated disjoint paths are also increased. This results in an increased delay between execution of instructions on a mainline path and its associated disjoint path. This extra delay is a result of having to forward operands across multiple bus segments. If the branch is resolved before instructions on the disjoint path had a chance to execute, the gain provided by spawning a disjoint path would be minimal.

Another issue which is related to this limited performance gain is due to the
fact that disjoint paths are always spawned from the predicted mainline path. It is possible to have a branch misprediction follow an earlier branch misprediction. In [115] it is shown that about 14 percent of mispredictions occur after another mispredictions.

In our scheme, in order to prevent an exponential growth in the number of disjoint paths, we do not spawn additional disjoint paths from a disjoint path. This policy has the disadvantage of not allowing us to effectively handle two sequential mispredictions.

**Subroutine Call Confidence Estimator**

In section 5.4.2 we had mentioned that if a branch has a domain size less than near threshold, it will be predicated and no disjoint path will be assigned to it. Fetch will continue from the not-taken path of the conditional branch regardless of its prediction.

![Speedup graph](image)

Figure 7.10: Speedup gained by using subroutine call confidence estimation.

However, if a subroutine call is in the domain of such a branch, the fetch unit
would be faced with making a decision as whether to follow the target of the subroutine call or instead continue fetching from the domain of the conditional branch. If fetch follows the target of the subroutine and the branch is resolved to be not-taken, the execution window needs to be flushed.

To help the fetch unit, We used a subroutine call confidence estimator to identify those subroutines that are never or rarely called. An example of the latter kind of subroutines is an error handling routine. Using the confidence predictor, the fetch unit will determine whether or not to continue fetching from the target of the subroutine call.

We have implemented subroutine call confidence estimation in our model to help guide when to follow a call path. Figure 7.10 shows the speedup for each benchmark. The gcc and parser benchmarks obtain the most benefit from this predictor, which follows since these two programs have the largest number of calls in our suite.

### 7.3.3 Effect of Fetch Bandwidth

To evaluate the effect of the modifying the bandwidth of our fetch unit on performance, we explored two sets of parameters. The first parameter is the maximum number of instructions that can be fetched in each cycle, which is a measure of the width of the fetch unit. The second parameter is the maximum number of non-adjacent basic blocks that can be fetched in one cycle. This parameter is a measure of complexity of the collapsing and aligning mechanism in the fetch unit.
CHAPTER 7. EXPERIMENTAL METHODOLOGY

Table 7.5 shows the performance of our microarchitecture relative to a configuration with a fetch width of 128 instructions and no limit on the number of basic-blocks that can be fetched in each cycle.

From this table, we can see that our default configuration with a bandwidth of 32 instructions and the ability to fetch up to 4 non-adjacent basic blocks per cycles achieves almost 96% of the performance. Note that the fetch bandwidth is shared between mainline and disjoint paths. A wide fetch unit is necessary in order to prevent paths from starvation.

7.3.4 Domain Threshold Values

Domain threshold values, as discussed in section 5.4.2, are two parameters used in our fetch heuristics. We ran experiments with different near and far threshold values and discovered that the best performance for an 8-4-8 geometry is achieved when both near and far threshold values were set to 4. This is an interesting observation because it means that we do not need to use a confidence predictor for conditional branches. In other words, for branches with a domain size greater than 4 we should always fork a disjoint path. One explanation is the fact that branch confidence predictors do not have a high accuracy. By failing to fork a disjoint path at a mispredicted branch we are exposing ourselves to branch misprediction penalties that can degrade performance.

Another reason why we do not need a confidence predictor is related to the fact that we are using more than one disjoint path. It turns out that by forking at each branch, we are essentially outperforming a branch confidence predictor.

Table 7.6 shows the speedup (or slowdown) obtained for configurations with different threshold values relative to a configuration where near and far thresholds are both set to 1. The negative numbers express a slowdown compared to the base
configuration. Note that if the confidence estimator incorrectly labels a branch as highly predictable, instruction fetch will continue from the branch target. In the case of a misprediction, the instruction window is flushed.

### 7.3.5 Speculation Invalidation

![Bar chart showing percent speedup vs speculation distance]

**Figure 7.11**: Effects of varying the speculation invalidation distance for a 8-4-8 configuration.

To measure the effect of *speculation invalidation* on performance, we define *speculation border column* (EBC) as the earliest loaded column where speculation invalidation mechanism is disabled. We also define *speculation distance* as the number

<table>
<thead>
<tr>
<th>Speculation distance</th>
<th>Speculation distance</th>
<th>Speculation distance</th>
<th>Speculation distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-6%</td>
<td>-4%</td>
<td>-2%</td>
</tr>
<tr>
<td>2</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>3</td>
<td>-6%</td>
<td>-6%</td>
<td>-6%</td>
</tr>
<tr>
<td>4</td>
<td>-1.3%</td>
<td>-1.3%</td>
<td>-1.1%</td>
</tr>
<tr>
<td>5</td>
<td>-1.4%</td>
<td>-1.4%</td>
<td>-1.3%</td>
</tr>
<tr>
<td>6</td>
<td>-1.4%</td>
<td>-1.4%</td>
<td>-1.3%</td>
</tr>
</tbody>
</table>

*Table 7.6: Effect of domain threshold values on performance.*
of columns between the Earliest Loaded Column (ELC) and the EBC. All other columns that are between EBC and Latest Loaded Column (LLC) also have their speculation invalidation mechanism turned off.

Speculation distance is a mechanism for limiting the speculation to a portion of the execution window. Instead of having a fully speculative machine, we limit the speculation to only a few columns and by using the speculative invalidation mechanism introduced in 4.2.2.

We measured speedup as a function of speculation distance and show the results in Figure 7.11. Speedups are normalized with respect to a configuration where speculation is not allowed in any column (i.e. the speculation distance is zero). The first set of bars is associated with a multipath configuration (with D), whereas the second set of bars belong to a single path configuration. A speculation distance of zero corresponds to a machine with no speculation. For an 8-4-8 geometry, a speculation distance of 8 corresponds to a fully speculative machine where all columns are speculatively executed.

From Figure 7.11 we can see that the most benefit for a dynamic multipath configuration is obtained for a speculation distance of two. Speedup starts to decline for speculation distances greater than two. This decline in performance is a result of missed execution opportunity for disjoint paths. Mainline path instructions have execution priority over disjoint paths. In columns where speculation is allowed, disjoint paths have less opportunity for execution, even though the instructions on the disjoint path are closer in time to commitment.

In a single path configuration, the performance increases with the number of speculative columns. Speculation in the single path configurations does not have any side effects and in case the speculative value is wrong, a re-execution is needed. From the graph in Figure 7.11, it can be seen that the performance stays the same for
a speculation distance of greater than 5 columns. This is because other resources such as PE and Bus bandwidth are constrained and would delay the speculative execution of instructions.

7.3.6 Sensitivity to Data Path and Memory Latencies

In this section we study the effect of changing various microarchitectural parameters related to data path and memory latency. The goal of this study is to study the sensitivity of our microarchitecture to changes in technology-dependent structures. This information will also aid the designer when considering design tradeoffs.

![Figure 7.12: Effect of the Forwarding Unit delay on performance for an 8-4-8 configuration.](image)

**Forwarding Unit Delay**

First, we study the impact of Forwarding Unit delay on performance. Figure 7.12 shows the resulting performance degradation as the Forwarding Unit delay is increased from 0 to 8 cycles in an 8-4-8 configuration. From the graph, we can see
that an increase of 1 cycle delay results in a 10% performance loss. In order to keep the performance at an acceptable level, it is therefore important to keep the Forwarding Unit delay small.

**Bus Span**

In section 4.2.1 we showed how we selected the bus span length based on the def-use characteristics our suite of programs (which should be characteristics of typical integer programs). Figure 7.13 presents the performance improvement obtained if we increase the bus span length on a 16-4-8 geometry. In this experiment, a bus delay of 1 cycle is assumed for all bus span configurations. The performance improvement obtained for moving from 4 SGs to 8 SGs is around 29%, whereas we only get an 11% improvement by increasing from 8 SGs to 16 SGs in each bus span. This helps to justify that our choice of a bus span with 8 SGs is a reasonable choice.

![Figure 7.13: Effect of bus span size on performance.](image-url)
CHAPTER 7. EXPERIMENTAL METHODOLOGY

Figure 7.14: IPC as a function of different bus configurations.

Parallel Buses

Figure 7.14 shows the IPC as a function of the number of parallel buses in an 8-4-8 geometry. The three numbers in each entry of the legend correspond to the number of parallel buses for register, memory and predicate operands. Increasing the number of buses improves the performance by reducing the contention and allowing multiple transactions to be executed in parallel. The cost of providing parallel buses, on the other hand, is high. We have selected a 2-2-1 configuration as the default bus configuration for our microarchitecture.

7.3.7 Memory Latency

In this section we evaluate the effect of memory latency on the performance of our microarchitecture. Figure 7.15 shows the impact of changing the DRAM hit delay on performance. From this graph, we can see that even if the hit penalty increases from 0 cycles to 800 cycles, the performance in our programs only decreases by 5%
on average.

Figure 7.15: Effect of DRAM hit delay on performance.

Figure 7.16 shows the effect of L1 cache hit delay on performance. As can be seen in this graph, with a hit delay of 2 cycles, the performance drops to about 88% of an L1 with a 0 cycle delay. Our microarchitecture running the selected set of application is sensitive to the L1 hit penalty time, so it is important keep the L1 hit delay small.

Figure 7.17 shows the effect of modifying the L2 cache hit delay on overall performance. The default value in our simulations was 10 cycles which, corresponds to about 90% of peak performance (with a 0 cycle penalty). If we increase the clock frequency, we expect to see a higher hit delay (in cycles) for L1 and L2 caches.
Figure 7.16: Effect of L1 hit delay on performance.

Figure 7.17: Effect of L2 hit delay on performance.
Chapter 8

Conclusions

Extracting instruction level parallelism (ILP) from sequential code is one of the major challenges to effectively exploit large microarchitectures. Technology trends in the design of the future microprocessors suggest we will see an exponential growth in the number of transistors on a single chip. Chips with over one billion transistors will arrive shortly. The challenge is how to modify current designs to take advantage of these new manufacturing capabilities. Conventional microarchitectures limit our ability to exploit many of these new opportunities due to the increased wire delays that span large chip distances and the use of centralized resources such as reorder buffer and register files.

This thesis introduced a novel distributed and scalable microarchitecture that uses dynamic predication and multi-path execution to mitigate branch misprediction penalties.

This chapter presents a summary of this thesis and highlights its major contributions. We will then point to some of the possible directions in which this work can be extended.
8.1 Thesis Summary

In this thesis we enhanced the Resource Flow execution model [12] and applied it to the base microarchitecture proposed in [12]. We explored additional microarchitectural enhancements to our base model through development of a trace-driven simulation model.

Resource Flow execution model has been introduced as an alternative execution model to extract high IPC from sequential programs. Previous studies have shown that there exists a high degree of ILP in ordinary sequential programs (e.g., SPECint). The challenge is how to expose and extract this ILP. This thesis has obtained much higher IPC through exploring new microarchitectural techniques.

The Resource Flow execution model is based on the concept of aggressive speculation and re-execution of instructions. Instructions are speculatively executed regardless of whether they have correct operand values or not. If one operand of a speculatively executed instruction resolves to have the different current value of the operand (a single operand can have multiple speculative values during execution), the instruction will be re-executed using the updated value for the operand.

The Resource Flow execution model maintains correct execution in this highly speculative microarchitecture, while also effectively eliminating contention for centralized resources. Hardware scalability is achieved through the use of time-tags and Active Stations along with segmented buses, that form our large execution window.

The next few sections summarize the most important contributions of this thesis.

8.1.1 New Time-Tag Assignment Scheme

As part of the Resource Flow execution mode, time-tags are used to enforce correct data and control dependencies among instructions. We proposed a new scheme to assign and manipulate time-tags that does not require global adjustments as was
used in [13].

The new scheme effectively reduces the cost and complexity of managing time-tag values through static assignment. Each Active Station is assigned a unique time-tag value that does not change over the course of execution.

8.1.2 Memory Nullify and Request Transactions

Memory nullify and request transactions were added to the Resource Flow execution model to enable the execution of memory operations in the execution window. In the original model, memory operations were not an integrated part of the Resource Flow model.

The memory request transaction provides a mechanism to make a request for a memory value on the backwarding buses. A previous Active Station that has the same value would provide the memory value.

The use of nullify transactions is required when a store operation is disabled and needs to invalidate its forwarded memory value that has been used by later instructions.

By adding these two transactions, we were able to integrate memory operations into Resource Flow execution model and allow the speculative execution of memory operations.

8.1.3 Dynamic Predication

One of the other major contributions of this thesis is the design of a novel dynamic predication technique for efficient evaluation and application of predicate values at execution time. This technique provides for full predication without the need for explicit predicate registers or compiler intervention. Our proposed scheme is especially interesting due to its scalability and ease of implementation.
In this scheme each enabled branch forwards its predication to future instructions in the program execution-order future. Each instruction only needs to keep track of its closest enabled previous branch (CEPB). Disabled branches use an invalidate transaction to notify future instructions of the change in their CEPB.

8.1.4 Speculation Invalidation

We developed Speculation Invalidation as a mechanism for throttling rampant speculation in our microarchitecture. Using intermediate values generated by rampant speculation led to many unnecessary executions and forwarding transactions. The wasted bandwidth is particularly detrimental to the execution of disjoint paths. We found out that, although rampant speculation would improve the performance in a single path configuration, for multipath execution it would be more beneficial to limit the rampant speculation to the two first columns closest to the commitment column.

8.1.5 Dynamic Multipath Execution

Substantial effort in this thesis was devoted to developing a dynamic multipath execution scheme. We had already found out that predicated execution and static disjoint paths, on average, improve performance by over 45%. Although this level of performance improvement is significant, we were interested in exploring mechanisms capable of even larger gains.

Our limit studies with ideal fetch and memory demonstrated the potential for exposing higher level of ILP. Multi-path execution proved to be a very effective mechanism for reducing the latencies imposed by branch mispredictions.

In the static disjoint scheme, paths execute in conjunction with predicated instructions to hide branch misprediction penalties. A static disjoint path contains
CHAPTER 8. CONCLUSIONS

the same instructions as the mainline path, but the output predicate of the associated branch is inverted. This allows instructions on the not-predicted path to execute concurrently with instructions on the predicted path. Employing static disjoint paths has the advantage of reducing the complexity of the fetch unit since both paths contain the same instructions.

Dynamic multipath execution provides for a more aggressive execution by fetching down both paths of the branch. Dynamic multipath execution is used when the target of a branch is a far target (far means that the address is at a distance that is larger than a far threshold value). The mainline path follows the predicted path and the disjoint path follows the not-predicted path. This is similar to other multi-path techniques [1, 2, 61, 62, 68]. To reduce the number of dynamic disjoint paths, we evaluated the merits of using a branch confidence mechanism to throttle the generation of disjoint paths associated with high-confidence branches. Surprisingly we found out that not using a branch confidence predictor would actually result in better performance. We attribute this to the low prediction accuracy of confidence predictors and also due to the fact that in our microarchitecture there is a greater opportunity for spawning disjoint paths than on other microarchitectures that only consider a single alternate path. It turns out that by taking advantage of the disjoint columns and spawning on every branch with a domain size greater than 4 instruction, we obtain a speedup.

8.1.6 Microarchitectural Simulator

To evaluate different micro architectural trade-offs, we developed a simulator for our microarchitecture. The simulator uses a combination of trace-driven and execution-driven techniques to achieve a high degree of accuracy and precision. Using our parameterized simulator, we were able to explore the performance of a fairly large
design space.

8.1.7 Design Space Evaluation

We simulated our microarchitecture using a set of programs selected from the SPEC-Int 95 and SPEC-Int 2000 benchmark suites. We were faced with simulating a large design space due to many design tradeoffs needed to be considered in the microarchitecture. Among the more important parameters, the geometry studies evaluate the performance as a function of the number of Active Stations, Sharing Groups and columns. Although our results show an increase in performance with the increase in the size of the microarchitecture, we saw diminishing returns as the size of the machine was increased.

In addition to geometry studies, we also measured the effects of changing the number of buses, number of disjoint paths, memory hierarchy and Forwarding Unit delays. From these simulations, we found out that our microarchitecture has only minor sensitivity to memory latencies. Forwarding unit delays, on the other hand, have a rather large effect on performance and need to be minimized.

8.1.8 Instruction Fetch

To fully understand the tradeoffs between fetch unit complexity and overall speedup of multi-path execution, it was important to quantify the effect of fetch bandwidth. One of the side-effects of employing multi-path execution is the need for increased fetch bandwidth. The execution core of the processor needs a steady stream of instructions in order to keep all functional units busy. In a multi-path processor where the fetch unit is potentially shared between multiple execution paths concurrently, it is important to ensure that none of the execution paths are starved for instructions.

Fetching across multiple basic blocks requires the ability to fetch across many
CHAPTER 8. CONCLUSIONS

Cache lines. Cache replication and address interleaving are two different ways of addressing this issue.

In this thesis we measured the effect of fetch unit on overall performance using the following two parameters:

- The maximum number of instructions that can be fetched in one cycle.
- The maximum number of branch paths that can be fetched in one cycle.

We then considered the effect of varying the above two parameters and studied their effect on performance. We found that a fetch unit with a 32 instruction bandwidth and the ability to fetch from 4 non-adjacent basic blocks is able to achieve 95% of the maximum bandwidth provided by a very wide, single cycle latency, fetch unit.

8.2 Future Work

This section summarizes some of the directions for future research in the area of Resource Flow microarchitecture. Our implementation of a Resource Flow microarchitecture is one of many possible implementations. One alternative, as an example, would be to apply the Resource Flow execution model to a more conventional superscalar microarchitecture. Although we do not expect to achieve the same CPI performance on a conventional superscalar, we believe that many of our ideas could simplify the complexity found on current microarchitectures. We expect to improve the performance by allowing more aggressive speculative execution through the use of time-tags. We also believe that time-tags have value for power-aware processor implementations.
CHAPTER 8. CONCLUSIONS

8.2.1 Multipath Execution

In this thesis we showed that multipath execution is by far the most important factor in improving the performance. Minimizing the effect of branch mispredictions and improving the percentage of instructions that are fetched from the right path will have a substantial effect on the overall performance.

Our limit studies that used an ideal fetch unit showed that there is still a lot of room for improving performance. Since the accuracy of branch predictors can not be improved beyond the current state-of-the-art, we believe that multipath execution will become more important in the future.

In theory, an Eager Execution model should provide the highest speedup. However, Eager Execution suffers from exponential growth of concurrent alternate paths at each branch point. It would be interesting to explore using a limited form of Eager Execution, where we allow a disjoint path to be forked from another disjoint path. In our disjoint static tree, we assumed that disjoint paths are only forked from the mainline path. This approach, however, does not efficiently handle multiple successive branch mispredictions. Studies need to be done to consider the right number of disjoint paths that should be spawned from a disjoint path. If we cannot effectively throttle the production of disjoint paths, we will overwhelm the available resources.

8.2.2 Value Prediction

Speculation Invalidation creates an opportunity for application of sophisticated value prediction techniques. By limiting the speculative execution and preventing overwriting of predicted values, we expect to achieve higher speedup.

In a large instruction window, the distance between committing and newly loaded instructions could be quite high. It would then be interesting to evaluate the
effect of speculative updates to the value predictor. There is a high probability that
the accuracy of value predictor will degrade as the size of the instruction window is
increased.

8.2.3 Compiler Optimizations

In this thesis we did not consider the effects of employing compiler optimizations to
take advantage of our micro architectural features. Although we do not need special
assistance from a compiler, it would be interesting to see if new optimizations could
be proposed that improve the performance in our microarchitecture.

Profile-guided compilation would greatly aid in the prediction of hard to predict
branches. Profile information could also be used to guide multipath spawning heuris-
tics. By skipping easily predictable branches, we could create more opportunity for
spawning a disjoint path for hard to predict branches. It is however important to
note that as we saw in the case of a branch confidence estimator, a low prediction
accuracy could actually degrade the performance.

8.2.4 Detailed Modeling

Some units in our simulator have been modeled at a functional level. An example
of this is the Forwarding Unit. It would be a challenging task to implement
the Forwarding Units with the minimum delay and extended snooping capabilities.
Although Active Stations are the most complex part of this microarchitecture, the
delay of a Forwarding Unit is a critical design parameter in achieving higher per-
formance.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


[63] Unger A., Ungerer T., Zehendner E., “A compiler technique for speculative execution of alternative program paths targeting multithreaded architectures,”


[101] Kevin Skadron and Pritpal S. Ahuja and Margaret Martonosi and Douglas W. Clark, “Improving prediction for procedure returns with return-address-stack


