PERFORMANCE LIMITATIONS IN WIDE SUPERSCALAR PROCESSORS

By

ASWIN RAMACHANDRAN

Bachelor of Engineering in Electronics and
Communication
University of Madras
Madras, Tamil Nadu
2001

Master of Science in Electrical Engineering
Oklahoma State University
Stillwater, OK
2003

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 2008

PERFORMANCE LIMITATIONS IN WIDE

SUPERSCALAR PROCESSORS

Dissertation Approved:

Dr. Louis G. Johnson

Dissertation Adviser

Dr. R. G. Ramakumar

Dr. Marvin Stone

Dr. Charles Bunting

Dr. Sohum Sohoni

Dr. A. Gordon Emslie

Dean of the Graduate College

Dedicated to my mother

ACKNOWLEDGEMENTS

As Abraham Lincoln quipped, "*It's not the years in your life that count. It's the life in your years.*" My life seemed to attain a meaning only after I met a few wonderful people. As my meager mind realizes, these people not only inspired and motivated me but also shared a portion of their life.

Following a chronological order, the influence my parents and my grandmother have on me is significant. My grandmother's determination to raise four children with modest income and my parents' hard work to provide a good education for their children is something that makes me wonder everyday. My elder brother's excellence in education also induced a spirit of motivation in me – may be this is what is known in animal kingdom as *Sibling Rivalry!* My high school teacher of Chemistry is an epitome of hard work and it is through him that I learned to channelize and try to time myself properly.

Life at Oklahoma State University brought exciting opportunities that I would have never experienced if it were not so. I still remember the day when I walked into Dr. Marvin Stone's sensor lab and the research work that I undertook with interest for the next two years. Just watching him at work is enough to motivate a person. Dr. Rama Ramkumar's witty comments always made me look to the other side of life and Dr. Sohum Sohoni's suggestions always kept me to improve and to achieve more.

Besides research, I experienced a new phase of graduate school when I began to teach! Dr. Charles Bunting's unmatched enthusiasm for students and in research has always

been a tremendous inspiration to me. I began to enjoy the art of teaching and continued it for about 9 semesters.

My first chance to meet Dr. Louis Johnson came through the digital VLSI class that I had taken under him. Further, I continued with my *master's* thesis on digital CMOS design. The class on *Superscalar* processors that intrigued me a lot, especially on a rename-register file design. I had suggested some design enhancements then that I never thought that I would later incorporate them in my dissertation work. We would discuss for several hours in his office about design aspects in computer architecture. Later, these discussions formed the basis of my dissertation. Dr. Louis Johnson has a profound impact on my life and will continue to be so for which I'm indebted to him forever.

Apart from growing in my school life, my friends circle also began to grow. Interestingly, as I look back, I have found friends at all ages from 8 to 80 years. I try to work with the international friends' ministry in a local church and also practice taekwondo both of which expanded my circle of friends in the community of Stillwater.

The joy and qualities of some of my friends, Gerard, Simon, Rajaguru, Vijayaraja, Majunu, Aravind, Grisha, Shyam and many others are remarkable. The help from Bob and Bettie through the church ministry played a significant role. I also enjoy the child-like playful times with Robert and inspiration thoughts from Marley and Mei Ling. All these people and many more have made my life meaningful and beautiful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FLOW CHARTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1. Performance Studies

The design space of microarchitecture is bound to grow significantly as multi-threaded and multi-core architectures are investigated by computer researchers. Typically, computer architecture studies can be classified into 2 categories – *Performance Evaluation* and *Performance Estimation* metric studies.

Studies involving Performance Evaluation simulates the entire microarchitecture design and provides an accurate performance metric for the simulated microarchitecture. The simulation of the microarchitecture is cycle-accurate and involves detailed description of the microarchitecture blocks. This method of detailed cycle-accurate performance analysis takes tens of thousands of host machine's clock cycles.

The SPEC CPU benchmark programs have become the de facto standard to evaluate computer architecture designs. However, with the number of instructions in the SPEC benchmarks mounting to more than a trillion instructions, it is not feasible to simulate the complete set of benchmark programs in a reasonable amount of time.

For example, to execute 1 trillion instructions (assuming it takes 10,000 machine cycles for a simulated cycle) using a typical CPU operating at 1.5 GHz clock speed and 3 instructions per cycle, it takes about 77 days to evaluate the microarchitecture design. As different design trade-off studies have to be carried out by researchers, such long computing wait time becomes a huge impediment for research. Hence, several techniques have been proposed to circumvent the cost of increased simulation time. Reduced input data-set and trace-driven evaluations are a few of the techniques to reduce simulation time for cycle-accurate simulations. However, the similarities of these simulations with the actual simulation are still under investigation.

On the other hand, *Performance Estimation* models are proposed to probabilistically estimate the performance of the architecture design. The performance estimate of the microarchitecture is determined in a short time and this ensures the possibility of several microarchitecture design trade-off studies. But, the accuracy of the probabilistic model that describes the microarchitecture is debatable. Several assumptions are made to describe the microarchitecture model and such abstraction undermines the results of the performance estimations.

1.2. Motivation

The flow of instructions is measured in instructions per clock, IPC_i, at some point i in the data path. Usually what we are interested in is the average *IPC* which can be determined as,

$$IPC_i = \frac{1}{N_c} \sum_c IPC_i(c) \qquad (1)$$

where, N_c is the total number of clock cycles when running a bench mark program and $IPC_i(c)$ is the number of instructions passing a point in the data path during clock cycle, c.

High level processor simulations can calculate IPC in this manner, but they are forced to simulate the processor behavior for billions of clock cycles which is very expensive. Instead a stochastic model for IPC can be used which avoids simulating the processor architecture cycle by cycle.

The data path structure and the hazard control logic determine the IPC(c) when hazards occur. Suppose the cycle by cycle simulation calculates $N(IPC_i = 0)$, $N(IPC_i = 1)$, ..., $N(IPC_i = s_i)$ which is the number of clock cycles that $IPC_i(c) = 0, 1, ..., s_i$, where s_i is the local superscalar width (instruction parallelism) at point i in the data path. The IPC model can be made stochastic by defining the probability that $IPC_i(c) = 0, 1, ..., s_i$ as

$$P(IPC_i = n) = \frac{1}{N_c} N\left(IPC_i = n\right) \qquad n = 0,1,...,s_i \tag{2}$$

so that,

$$IPC_i = \sum_{n=0}^{s_i} n \cdot P(IPC_i = n) \tag{3}$$

The same system of equations from the structural model that determines $IPC(c)$ will give a system of equations that can be solved for $P(IPC_i = n)$ without running a cycle by cycle simulation.

The level of detail of this approach is such that individual instructions are not tracked as they flow through the data path structure. Instead the probability of an instruction flow rate is determined at each point in the data path structure. Many of the hazard control equations require knowledge about certain types of instructions at certain locations during certain clock cycles. The probability of an instruction of a certain type can be determined from instruction frequency analysis of the benchmark programs.

$$\begin{aligned} P(type = t \ at \ i) &= P(type = t \mid instruction \ at \ i) * P(instruction \ at \ i) \\ &= f_t * P(instruction \ at \ i) \qquad (4) \end{aligned}$$

The instruction frequency of type t instructions, f_t, can be reused for performance calculations of many different structural models. The stochastic model determines *P(IPC*$_i$ *= n)* only. The reuse of instruction frequency data greatly reduces the complexity of the stochastic model.

Calculating the effects of hazards is complicated since hazards are not mutually exclusive and that stalls from different hazards can overlap in time. The same stall can be produced by more than one stall at one time, and we must be careful to avoid counting the same stalls more than once. To apply the *IPC* formula, we must include not only individual hazards, but also all possible combinations of hazards with all possible overlaps in time.

In order to accurately estimate the performance of a complex microarchitecture design, we must understand the dynamic relationship between its instruction flow and the hazards due to structural, control and data dependence through its statistical information. An extremely fast microarchitecture simulator with detailed module descriptions that is closely related to hardware behavior is necessary to gather this statistical information. Hence, *OSU AbaKus* – a cycle-accurate microarchitecture simulator is developed to address this issue.

1.3. Cycle-Accurate Simulation Engine Concept

The basic idea behind the cycle-time simulator is that all clocked modules are evaluated for every simulation cycle. This idea is in direct relationship with the pipelined design of the microarchitecture design, as all stages in the pipeline are evaluated similarly for each clock cycle.

1.3.1 Processing Elements and Signals:

As shown in Figure *1.1*, each *Data Processing Element* has an input buffer and an output buffer. The *Data Processing Element* takes the necessary input data for evaluation and produces the output data that is then stored in the output buffer. The flow of data in the buffers is controlled by the *stall signal*. The processing elements can also introduce *forward-propagating stalls* or *bubbles* in the pipelines. The propagation of bubbles in the pipeline occurs when there is insufficient amount of data stored in the input buffers to feed the *processing elements*. The bubbles can be related to the *no-operations* (NOPS) in the microarchitecture design.

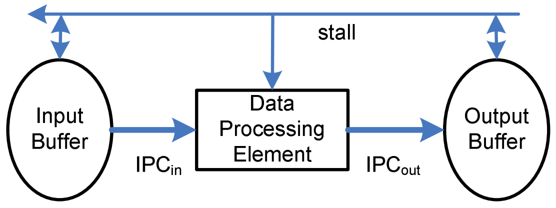


Figure 1.1 Structure of a Simple Data Path Representation

1.3.2 Buffer Design:

The design of the buffer offers the most discreet part of the simulation engine. It defines both the simulation engine's flexibility as well as its simulation speed. The buffer in the simulation act as information sources and sinks for the *Data Processing*

Elements. They maintain the network of connections through which the processing elements communicate with other processing elements in the design.

If the buffer gets filled, it can initiate a *stall signal* that stalls the up-stream buffers. The *IPC* of the processor is directly affected by these stall signals. The stall signals that stall up-stream buffers are called *backward-propagating stalls* or *up-stream stalls*. A major task in designing microarchitecture involves keeping a steady flow of information in the pipeline and to prevent buffers from being filled up.

The IPC_{out} and IPC_{in} are related in the eqn (5) and eqn (6) and are illustrated in Figure *1.2*, where N_b is the total number of instructions that the buffer can store, s_{in} and s_{out} are the number of instructions that are flowing into and out of the buffer in a clock cycle, $bubbles_{in}(c)$ is the number of bubbles that come into the buffer at cycle, 'c' and $bubbles_{out}(c)$ is the number of bubbles that leave the buffer at cycle, 'c'.

$$\mathrm{IPC}_{\mathrm{in}}(c) = \begin{cases} s_{in} - bubbles_{in}(c) & \text{if } \mathrm{stall}_{\mathrm{in}}(c) = 0 \\ 0 & \text{if } \mathrm{stall}_{\mathrm{in}}(c) = 1 \end{cases} \tag{5}$$

$$\mathrm{IPC}_{\mathrm{out}}(c) = \begin{cases} s_{out} - bubbles_{out}(c) & \text{if } \mathrm{stall}_{\mathrm{out}} = 0 \\ 0 & \text{if } \mathrm{stall}_{\mathrm{out}} = 1 \end{cases} \tag{6}$$

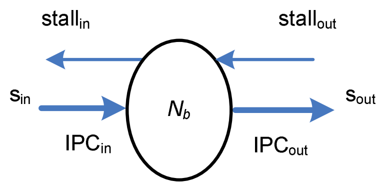


Figure 1.2. Relationship between IPCin and IPCout

As defined in section *1.3.1*, *bubbles* define the *NOP* instructions. The buffers can both propagate as well as initiate stall signals, $stall_{in}$ *and* $stall_{out}$. The condition at which the buffer is filled initiates the *stall$_{in}$(c) signal* at cycle 'c'. This is shown in eqn. (7), where *I$_b$(c)* is the number of instructions present in the buffer at cycle '*c*'.

$$\text{stall}_{in}(c) = \begin{cases} 1 & \text{if } \mathrm{I}_b(c) + s_{in} - s_{out} > N_B \text{ and if } \text{stall}_{out} = 0 \text{ and } \mathrm{I}_b(c) > s_{out} \\ 1 & \text{if } \mathrm{I}_b(c) + s_{in} > N_B \text{ and if } \text{stall}_{out} = 1 \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

The state of the buffer for the next cycle can then be calculated and is given in eqn. (7). Thus eqn. (8) describes that the state of the buffer for the next cycle is only defined by the current state of the buffer.

$$I_b(c+1) \quad = \quad I_b(c) + IPC_{in}(c) - IPC_{out}(c) \quad (8)$$

This simplistic view of the buffer is established from the pipeline model and more succinctly relates to the *Moore State Machine* of the architecture design. Furthermore, for stochastic performance analysis, this step can be extended to a discrete-time *Markov* model and thus future state of the buffer can be estimated.

1.4. Implementation of the Clocked Buffer Model

In this section, the implementation of the *buffer model* and the *Data Processing Elements* that are otherwise known as *modules* is discussed. As discussed in section 1.2, it is important that this cycle-accurate simulator is simple and fast. As shown in Figure *1.3,* the functionality of the architecture is defined by the two modules *A* and *B.* Two separate simulation data structures are maintained at its interface. The simulation methodology is a 2-step process.

The first step is to evaluate all the modules in the *evaluate phase*. In the first cycle, module *A* uses *Data Structure 1* as the output while module *B* uses *Data Structure 2* as the input. The second step, i.e. at the end of the evaluate phase, is the *update phase*. The pointers of *Data Structures A* and *B* are alternated. Hence, during the second cycle module *A* uses *Data Structure 2* as the output while module B uses *Data Structure 1* as the input. This buffer interface mechanism avoids transfer of huge amounts of simulation data during each cycle. This concept is further explained in detail in the following sections.

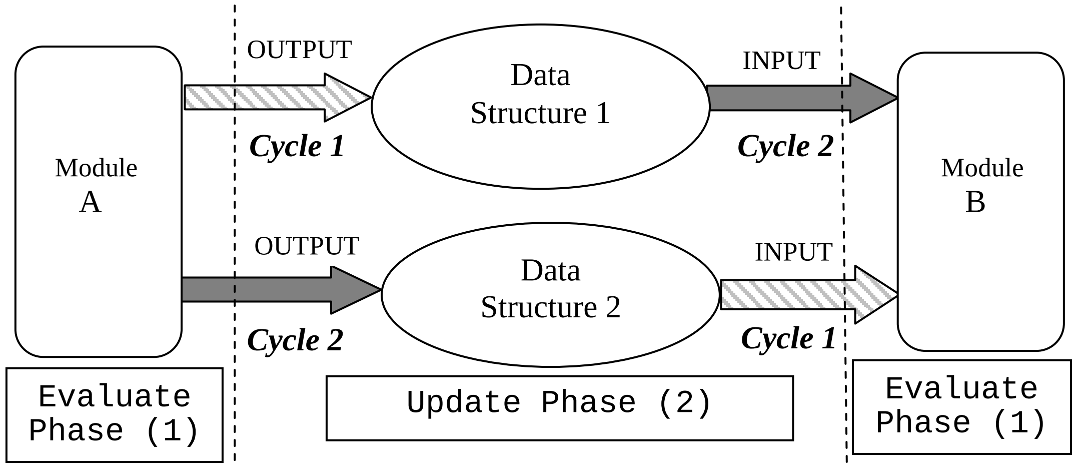


Figure 1.3. Pipeline Register Interface Model. A and B are modules defining the functionality of the architecture.

1.4.1 Module Interfaces through Port Definitions:

The modules descriptions are based on ISO C++ standard constructs. The modules describe the behavior of the *Data Processing Element.* The functional behavior of the module is described using C++ language definitions as in a sequential programming. However, the difference between sequential and modular programming is brought by *port* definitions that are used to interface with other modules. As a result, as shown in Figure *1.4,* the modularity in the design is achieved through *ports* that are used as communication interfaces between modules and the buffer.

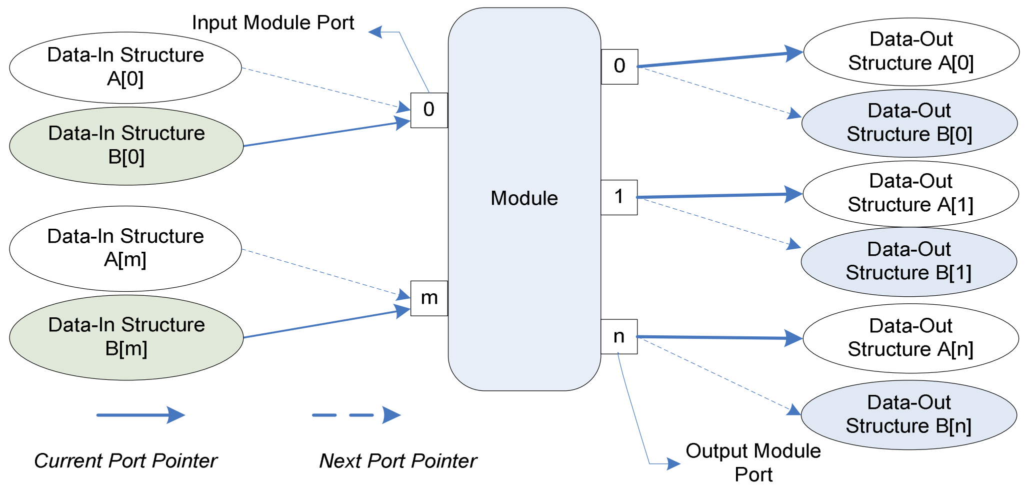


Figure 1.4. Module Interface

Similar to an HDL, *ports* are specified in a module to be an input or output port. In Figure *1.4*, each *port* has *2* pointers, the *current port pointer* and the *next port pointer*. The input data to the module is read from the *Data-In Structure* that is pointed by the *current port pointer* while the module's output data is written into the *Data-Out Structure* that is pointed by *next port pointer*. In the following cycle, the pointing location of the pointers is alternated, thus the outputs written during the previous cycle can be read as inputs in the following cycle and vice versa. This simple alternating of pointers avoids the overhead of copying the entire data structure that leads to slow simulations as in OSCI SystemC *2.1*.

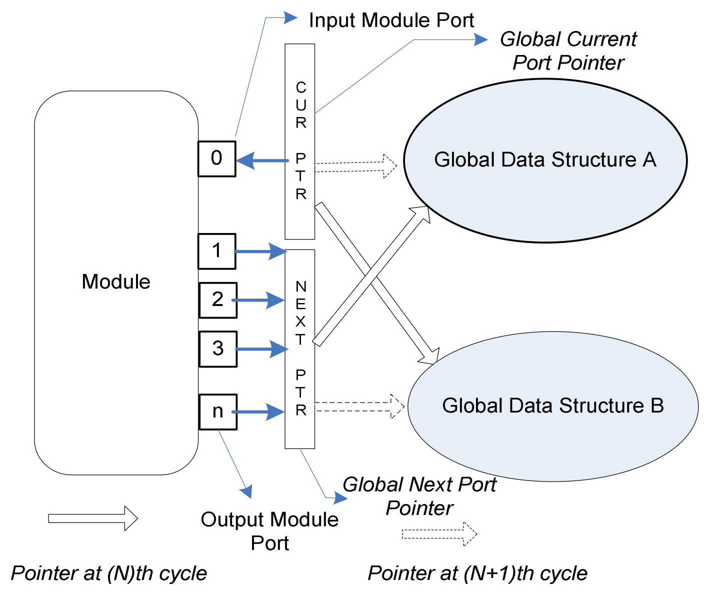


Figure 1.5. Improved Mechanism with Global Pointers for Global Data Structures

The number of update operations in alternating pointers between *Data-In and Data-Out Structures* is directly proportional to the number of ports in a module. Hence, to avoid this additional computational cost, two *Global Pointers* for the *Global Data Structure A and B* are created as illustrated in Figure *1.5*. Furthermore, the outputs of all the modules in the simulation are referenced to the *Global Next Pointer* and similarly, the inputs of all the modules are referenced to the *Global Current Pointer*. These pointers alternate between the *Global Data Structures A and B* for each clock cycle. Thus, the output data structure at clock cycle '*N*' becomes the input data structure at clock cycle '*N+1*' and vice versa. This mechanism not only avoids copying data between the *Global Data Structures* but also makes the number of update operations independent of the number of ports in the modules. As a result, it maintains the computational time for updating the pointer locations a *constant*.

1.4.2 Register and Memory Element Interface Model:

The update phase that is shown is Figure *1.3* is also extended to update the registers in the *register file* and other *memory elements.* As shown in Figure *1.6*, the data in the memory elements are accessed through *ports* similar to the actual memory access. The location of the *write and read* is determined by the *write and read addresses* respectively. Therefore, a *write data or read data* occurs on the referenced register/memory location depending on the logic.

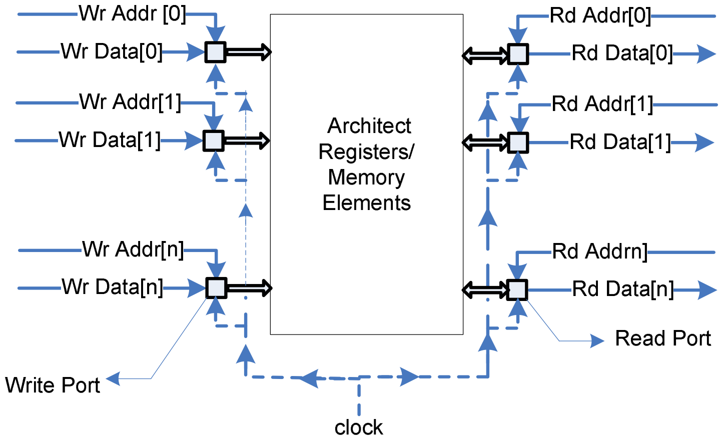


Figure 1.6. Memory Access through Ports

As shown in Figure *1.7*, both the *write port* and *read port* have two in-built data structures defined as *Port A* and *Port B*. On the *write port* interface, the data to be stored are written into *write port A*, while the data from the *write port B* are transferred to the *memory element*. Their corresponding pointers are alternated during the *update phase* that is triggered by the clock cycle. Hence in the following cycle, the functionalities of *write ports A* and *B* are interchanged. Similarly, on the *read port* interface, data is read from the *read port A* and the data from the *memory element* is transferred to the *read port B.* The functionalities of *ports A* and *B* are similarly interchanged for each cycle.

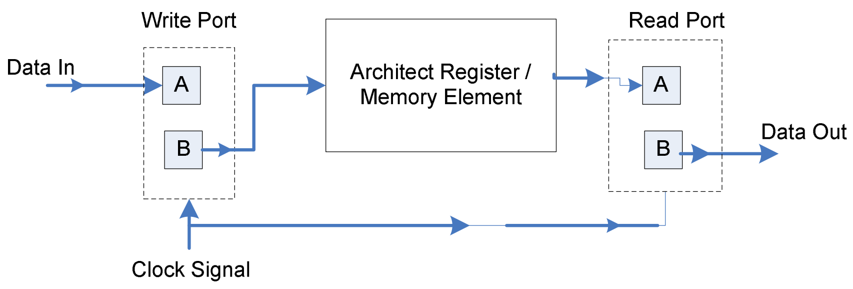


Figure 1.7. Port Access for a Memory Element

This functionality of the *write ports* and *read ports* described in this section corresponds to the *D-flip flop register* that is used in the actual hardware design. Hence, designing the memory structures with *port* interfaces provides this simulator the capability to perform both functional as well as timing verifications as in an HDL, and yet with a much greater simulation speed.

1.4. Organization of this Dissertation

Chapter 2 reviews the simulation mechanism on existing simulators. It reveals the benefits and drawbacks of each simulator. Chapter 3 presents the simulation approach of AbaKus simulator and also compares its performance with existing simulators. Chapter 4 discusses the modeling details of the superscalar architecture. It then presents about the load-store dependence prediction schemes used in AbaKus. Chapter 5 presents a case study on register write-back buses and identifies the characteristics of different bus scheduling mechanisms. Chapter 6 presents another case study on control dependencies problem in superscalar cores. Finally, Chapter 7 summarizes the design of AbaKus and limitations of superscalar processors.

CHAPTER II

LITERATURE REVIEW

2.1 Simulation

Hardware simulation is a process of describing the behavior of hardware logic using computer programming languages and verifying the hardware behavior with test input sets. Its use and adaptation depends on the accuracy of the results obtained using simulated hardware compared with actual behavior, speed of simulation and flexibility to design.

Computer architecture simulators are needed for the following reasons:

- Perform extensive design space exploration because it is cheaper to experiment with simulated designs.
- Verify hardware logic with respect to both functionality and timing, and
- Aid in the simultaneous development of support software tools such as compilers and operating systems.

There is a plethora of computer architecture simulators and the next section discusses some of the widely used computer architecture simulators.

2.2 Simplescalar Tools

Simplescalar tool set (Burger and Austin [1], 1997) has been one of the most widely used computer architecture simulator both in research as well as in class projects. It is an open-source and free-of-charge tool for non-commercial academic users. It provides a baseline out-of-order simulator known as the *sim-outorder* and most of the processor design aspects including the reorder window size, number of functional units and latency of memory ports can be defined at compile time. In addition, it integrates simplistic cache models to its processor and the cache design parameters can also be varied.

Simplescalar package has a set of simulators ranging from simple functional simulator to complex out-of-order processor simulator. It supports MIPS IV based Instruction Set Architecture (ISA) with minor changes to the instruction opcodes and also provides cross-compiler for its ISA to run on host computer machines. The advantage of Simplescalar tool set is its speed of simulation. On *sim-outorder* simulations the simulation speeds can average about 200 K instruction/s on a typical modern day desktop machine. Hence, it has been widely popular to execute SPEC benchmarks with Simplescalar tool sets that would normally be executed on real processors.

One of the main drawbacks of *sim-outorder* is that it is weakly related to the actual hardware behavior. For example, *sim-outorder* does not model the effects of write-back buses in the processor core. The contention among the write-back buses is important as it may increase the latency of dependent instructions. Another weakness of *sim-outorder* is that the actual execution of the instruction is in-order and only the control flow of execution is simulated. The concept of pipeline register timing is not simulated and it is important to maintain accuracy. Besides, code changes in Simplescalar have also proven to be difficult (Vachharajani et. al [2], 2002) and hence it has reduced flexibility.

2.3 Liberty Simulation Environment

In order to the address the problems of accuracy in simulations and to reduce the development time for logic design Vachharajani et. al. [3], 2002 developed the Liberty Simulation Environment (LSE). It is free and is a component-based model designed to reuse code usage.

Modularity in module definitions is well enforced by allowing modules to communicate through ports. Each port as shown in Figure 2.1 handles 3 signals: *data, enable,* and *ack.* The *data* is sent forward and the *enable* indicates that the receiving module should process the data. If the receiving module can process the data then an *ack* signal is transmitted. This simulates effectively the pipeline stalls and timing of data in an architecture simulation.

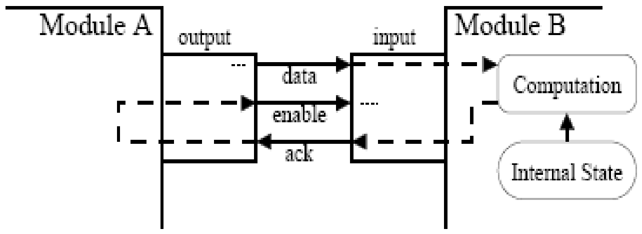


Figure 2.1 Port Communications in Liberty, Vaccharajani et. al. [3], 2002

The advantages of LSE are that it is modular and through the use of a graphical user interface, designers can drag, drop and connect modules. However, the modularity comes at the cost of simulation speed. The number of hand-shaking signals increases with the increase in ports (Vachharajani et. al, 2002].

The order in which the modules are invoked depends on the scheme called *Heterogeneous Synchronous Reactive* (HSR) scheme. It is different for the discrete-event scheduling in that a partial order of module invocation is generated statically using several optimizing scheduling polices and later can change similar to the discrete-event scheduling. In general, the HSR reduces the problem suffered by discrete-event scheduler which invokes repeated module evaluations.

2.4 ASIM

The key feature of ASIM is its modularity (Emer [4], 2002). The performance models in ASIM are mainly developed using C++ and is a proprietary of Intel [4]. Modularity is achieved through ports that are FIFO queues. The model of FIFO ports helps ASIM to simulate the latency between pipeline stages and also wire delays.

ASIM is considered to offer a high degree of module reuse. However, ASIM is likely to suffer in the speed of simulation as it is based on discrete-event scheduler. Although, these schedulers enable designers to simulate realistic hardware signal flow, they suffer from additional computation time. Since, ASIM is considered to be closely related to simulate hardware behavior; an extension of ASIM known as A-Ports (Pellauer et. al. [5], 2008) has been developed to emulate the behavior through FPGAs.

2.5 SystemC Based Simulators

SystemC is a C++ based modeling language with several model libraries for specifying the digital logic of the hardware and has a discrete-event scheduler to simulate the timing details. The popular version of SystemC is maintained by Open SystemC Initiative (OSCI) [6].

2.5. 1 UNISIM

Unified Simulation environment (UNISIM) is an open-source SystemC add-on that focuses on modularity and code reusability. It also supports cycle-level and transaction-level models. Several groups such as *Liberty*, *Microlib* (Perez et. al. 2004, [7]) and SystemC model developer are actively involved to develop architecture models of the computer system.

One key feature in UNISIM is its interoperability which means that it is considered to be possible to integrate with different simulation environments. It also supports full system simulation that includes operating systems such as Linux. Virtutech® Simics™ [8] is another simulation environment that performs full system simulation and supports various operating systems. But the disadvantage of Simics is that it is commercial with source code restrictions. UNISIM currently supports a host of processor model including PowerPC and ARM. The drawback on UNISIM is that it is an even-driven simulation environment and is slower than cycle-time based simulations.

2.5.2 .ArchC

ArchC [9] is an open-source architecture description language based on SystemC. It defines several wrapper class structures to enable designers to specify the architecture parameters instead on the actual module descriptions. Module descriptions are also possible to extend its model libraries. It supports various models including PowerPC, Intel 8051 and SPARC V8 architectures.

2.6 FPGA-based system emulation

Research Accelerator for Multiple Processors (RAMP) [10] aims to emulate dozens of processor cores in multiple FPGAs whose cells are being densely packed. Validating multiple processors is difficult in simulations because of the increase in the level of simulation as well as the number of test inputs. Emulation using FPGA technologies can lead to significant improvements in validating such architecture designs. However, the cost involved in emulation is also significantly higher compared to computer simulations.

2.7 Other Simulators

There are number simulators available for the computer architecture research community to simulate various components of a computer system. Depending on the simulator's characteristic it is the choice of the researcher to select a simulator. Simulators such as M5 (Binkert et. al. 2006, [11]) and SESC [12] model both CPU as well as support network I/Os of a computer system. PTSim (Yourst, 2007, [13]) is an event-based simulation for x86 architectures. Numerous variants of Simplescalar tools such as *sim-mase* (Larson et. al. 2001, [14]) is developed to further increase the level of simulation details in Simplescalar tool set.

2.8 Discussion

Computer architecture simulators available for researchers are abundant. The choice of the simulator comes down to the details of architecture that the researcher is interested to model. The nature of the simulator depends on its modularity/flexibility, speed and accuracy.

Although most of the simulators focus of modularity and reusability, it comes at the cost of simulation speed. Simulation speed is important to enable researcher to test and validate the architecture with numerous test input sets and also to explore more design alternatives.

FPGA based system emulation can provide speed and accuracy but at an increase cost. AbaKus simulator is developed to address the issues of speed, accuracy and modularity and in an affordable way. In the next few chapters, the internals of AbaKus simulation engine and its models are discussed.

CHAPTER III

SIMULATOR PERFORMANCE

3.1 Simulator Design

Simulators strive to achieve the three important parameters - accuracy, flexibility and speed in the best possible way as depicted in Figure *3.1*. The simulators described in Liberty [2], MASE [14] and ASIM [4] emphasize on each of these parameters.

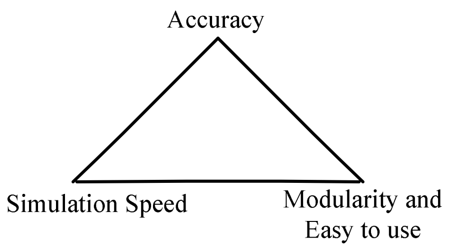


Figure 3.1 Objectives of a Microarchitecture Simulator

Microarchitecture functionality can be visualized as a group of modules triggering dependent modules to be evaluated each cycle. In general, it is modeled as a state machine. Therefore, the signals that are generated in a module propagate and modify the state as they traverse through various module structures. The two common types of simulations are considered to explain interface mechanism,

- Event-Driven Simulation
- Cycle-Time Simulation

In an event-driven simulation, a process queue maintains a list of modules that are to be evaluated for each cycle. The process queue is updated for each finite simulation cycle time. Consider A, B, C, D and E are hardware functional modules connected as shown in Figure *3.2*.

Evaluation of each module triggers its dependent modules and is added in the process queue. For the structural logic shown in Figure *3.2*, the process queue collects copies of same modules to be evaluated repeatedly as shown in Table *3.1*.

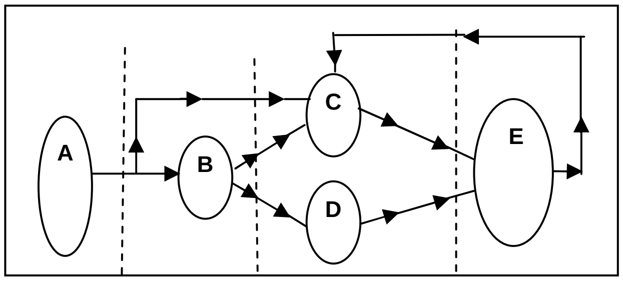


Figure 3.2 Module Executions

TABLE 3.1 EVENT-DRIVEN SIMULATION PROCESS

Cycle	**Evaluate**	**Trigger**	**Process Queue**
1	A	B, C	B, C
2	B C	C, D E	C, D, E
3	C D E	E E C	E, E, C

Modules C and E are evaluated multiple times.

Although, this ensures a more realistic hardware logic evaluation, repeated module execution results in a lot of computing time. Simulation kernels of HDLs such as Verilog, VHDL and SystemC are based on this mechanism. Techniques to reduce the number of redundant module executions in SystemC by acyclic scheduling have been proposed by Perez et al. [15].

On the other hand, the cycle-time simulation has a simpler approach. All the modules in the simulation are evaluated only once on each simulation cycle. This provides a more straightforward solution to avoid redundant module evaluations. The functional verification is typically provided by enforcing sequential order of module executions, as in SimpleScalar. The challenge in a cycle-time simulation is to provide both functional as well as timing verification that is provided by the event-driven simulation. There are two cycle-time simulators that are developed in this study.

[1] OSU SystemC

[2] OSU Aba*K*us

3.1.1 OSU SystemC

As SystemC has grown to be one of the frameworks for developing system-level architectures, a new cycle-time simulation model based on SystemC language construct – OSU SystemC – is developed in this research.

The models developed in SystemC v2.1 from Open SystemC Initiative (OSCI) are compared OSU SystemC. SimpleScalar version 3.0 tool-set provides the base-line model to compare the performance of the simulators as it is widely used for academic research and studies. The syntax of OSU SystemC is same as IEEE 1666 standard described for SystemC v2.1, but with restriction on usage of thread modules. The following summarizes the kernel of OSU SystemC,

- The old and new values have pointer that are switched on each delta cycle instead of values being copied [15].

- The scheduler is cycle-time based and hence, it evaluates all the modules that are declared with SC_METHOD in a delta cycle. SC_THREAD definitions are not handled as it needs synchronization of all thread modules after each delta cycle.

3.1.2 OSU AbaKus Simulator:

On the other hand, the syntax of OSU AbaKus is in standard C++ and is developed such that it is adaptable to any hardware description language. The OSU AbaKus Microarchitecture Design Simulator is developed to address the issues of flexibility and speed. The design flow for each microarchitecture simulator is illustrated in Figure 3.3.

OSU AbaKus provides a much simplified simulator with a new simulation kernel and is completely different from that of SystemC 2.1 kernel. Thus, by having a new simulation kernel, the redundant codes present in the existing OSU SystemC version and its class hierarchical design is avoided.

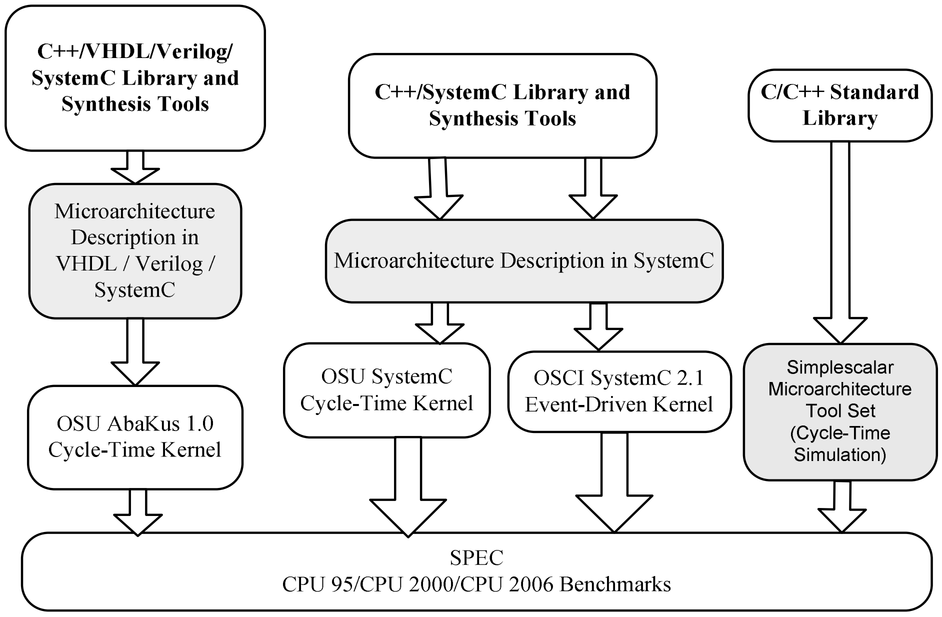

C++/VHDL/Verilog/ SystemC Library and Synthesis Tools
C++/SystemC Library and Synthesis Tools
C/C++ Standard Library
Microarchitecture Description in VHDL / Verilog / SystemC
Microarchitecture Description in SystemC
OSU AbaKus 1.0 Cycle-Time Kernel
OSU SystemC Cycle-Time Kernel
OSCI SystemC 2.1 Event-Driven Kernel
Simplescalar Microarchitecture Tool Set (Cycle-Time Simulation)
SPEC CPU 95/CPU 2000/CPU 2006 Benchmarks

Figure 3.3 Design Flow of Simulation

3.1.3 Comparison between OSCI SystemC, OSU SystemC and OSU AbaKus

A simple three-stage scalar pipeline model was tested with SPEC *95* benchmark programs on an AMD Duron *750* MHz processor running Linux kernel *2.4.2*. As shown in Figure *3.4*, the instruction execution rate of the new simulators using the SimpleScalar's instruction-execution engine is *10* times faster than the model developed in *SystemC 2.1*. This results in *25%* increase in simulation speed between *OSU AbaKus* and *OSU SystemC*. The throughput of the simulators is compared in Figure *3.5*. SimpleScalar's sim-safe executes all instructions in a clock cycle i.e. the instruction execution latency is *1* and it represents the most ideal execution engine.

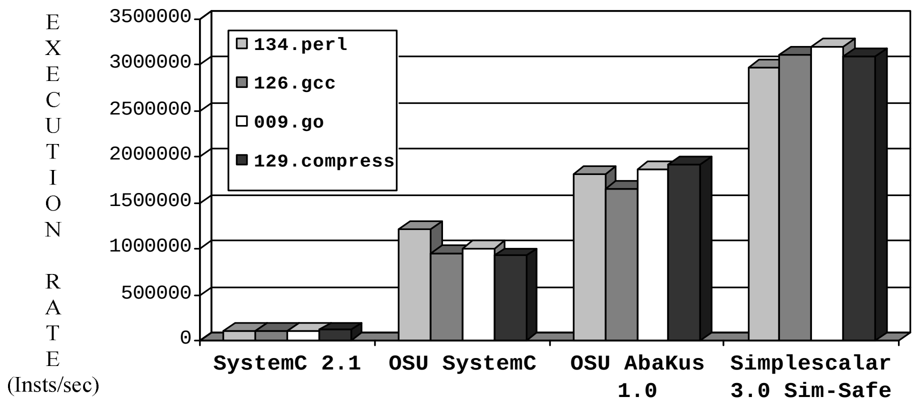


Figure 3.4 Performance Comparison between the Simulators for a simple 3-stage scalar MIPS architecture

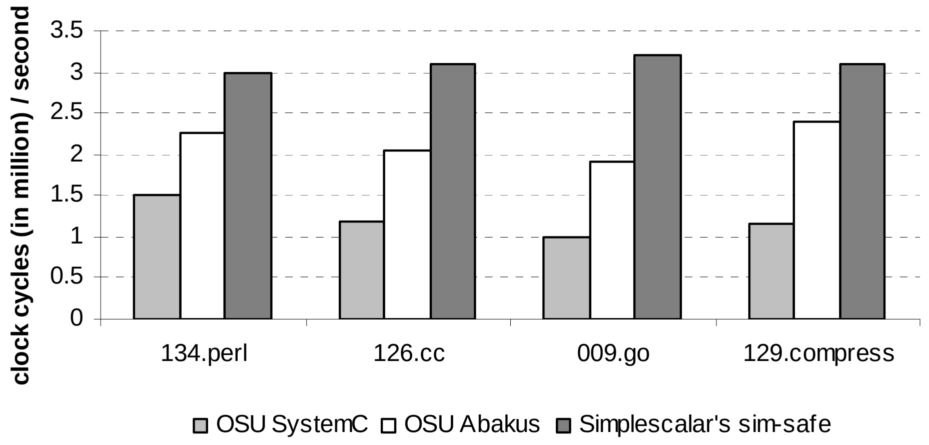


Figure 3.5 Comparison of Simulator Through-put

It is observed from Fig. *3.5* that *OSU AbaKus* has *40%* more throughput than OSU *SystemC*. The simulation kernel differences such as implementation of advanced object-oriented concepts cause the asymmetric distribution of execution rate seen in Figure *3.5*. To further investigate the performance of the simulators on complex designs, a superscalar architecture is built using *OSU AbaKus*.

3.2 Comparison with Superscalar Designs

A modular description of superscalar architecture design is written to accurately model the functionality of the microarchitecture during each clock cycle. The modules are described in C++ and reuse Simplescalar's execution core and memory models. Figure *3.6* shows the details of the simulated superscalar architecture. It accurately models the stall signals and in addition, the pipeline registers are parameterized to simulate different superscalar architecture widths. The finish stage encompasses the issue logic, instruction execution and write-back buses to update the register file. The microarchitecture is designed to explicitly model the *rename register mechanism* using Rename Register

Pointers and Architect Register Pointers that is not modeled in SimpleScalar's sim-outorder *3.0*. Moreover, unlike SimpleScalar *3.0*, all executions are true out-of-order.

As shown in Figure *3.6*, the microarchitecture uses SimpleScalar's *memory model* to fetch instructions and to perform memory related operations. The dynamic instruction scheduler with single instruction window includes instruction wake-up logic and out-of-order issue logic. As a test case perfect branch prediction is used to determine the throughput of each simulator and limit the architectural differences between the two simulations. But, it is found that in SimpleScalar, the next program counter is determined at dispatch and hence it encounters conditional stalls even during perfect branch conditions.

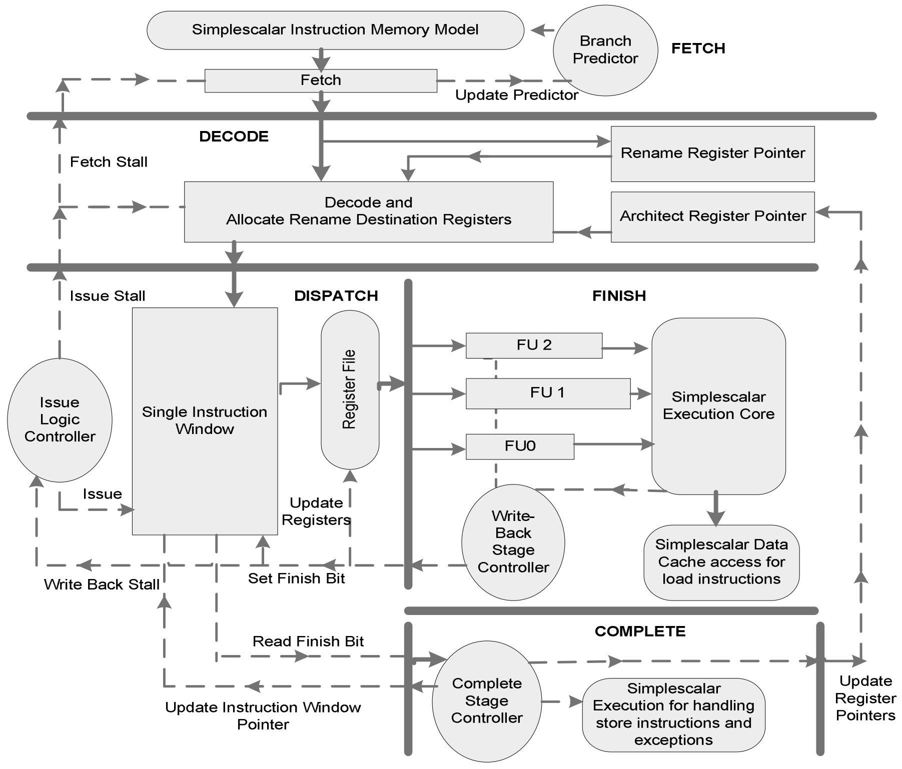

Simplescalar Instruction Memory Model
Branch Predictor
FETCH
Fetch
Update Predictor
DECODE
Fetch Stall
Rename Register Pointer
Decode and Allocate Rename Destination Registers
Architect Register Pointer
Issue Stall
DISPATCH
FINISH
FU 2
FU 1
FU0
Register File
Simplescalar Execution Core
Issue Logic Controller
Single Instruction Window
Issue
Update Registers
Write-Back Stage Controller
Simplescalar Data Cache access for load instructions
Write Back Stall
Set Finish Bit
COMPLETE
Read Finish Bit
Complete Stage Controller
Simplescalar Execution for handling store instructions and exceptions
Update Instruction Window Pointer
Update Register Pointers

Figure 3.6 Simulated Superscalar Architecture

As there is no floating-point unit incorporated in *OSU AbaKus*, it handles floating-point instructions as a precise exception. Due to this simplification, it is expected to have a lower IPC than SimpleScalar. Instructions that cause exceptions have three-cycle functional unit latency. The recovery mechanism then recovers the processor to the original machine state. However, the number of recovery cycles depends upon the state of the processor at the time of exception and this is not modeled in detail with SimpleScalar *3.0*. Besides, no explicit register rename mechanism is implemented in SimpleScalar *3.0*.

A more detailed out-of-order architecture model is developed in Simplescalar *4.0*/MASE [14]. The renaming register logic is included and a distributed reservation station model is incorporated. An in-order execution queue is maintained and hence it does not incur a *2*-cycle penalty for perfect branch prediction studies as in Simplescalar *3.0*.

Another architecture difference between *Simplescalar* and *OSU AbaKus* is the register write back bus model during the finish stage. This is an important module that defines the number of instructions that can finish in a clock cycle. This aspect is not considered in Simplescalar versions (Vachharajani et. al. 2006, [3]). *OSU AbaKus* solves this problem by providing an explicit parameter for the write-back bus bandwidth and simulates realistic stalls encountered during instruction finish. The pre-compiled SPEC binaries from SimpleScalar and our own compiled binaries with ref and train input data-sets were run to completion. Due to the long running time not all benchmarks are incorporated in the test.

3.3 Comparison of Simulation Speed

Figure *3.7* compares the simulation speed of the three microarchitecture simulators, sim-mase, *OSU AbaKus* and sim-outorder. In order to correctly compare the simulators, every effort is made so that the simulated hardware architectures are as similar as possible. In addition, the processor model that is simulated in the three simulators is designed to have similar average Instructions per clock cycle (IPC). The simulators are compiled with gcc *3.4.5* with the O0 optimization level and are executed in a *64*-node cluster each with *3.2* GHZ Intel Xeon™ processor running a Linux *2.6.9* kernel. The simulated architecture details are listed in Table *3.2*.

SPEC CPU 2006/2000 integer benchmarks with reference input datasets are used to compare the simulators. A total of *6* billion instructions are executed in each of the selected benchmarks. Only the benchmarks that compiled successfully with Simplescalar's *sslittle-na-sstrix-gcc* are used in this research.

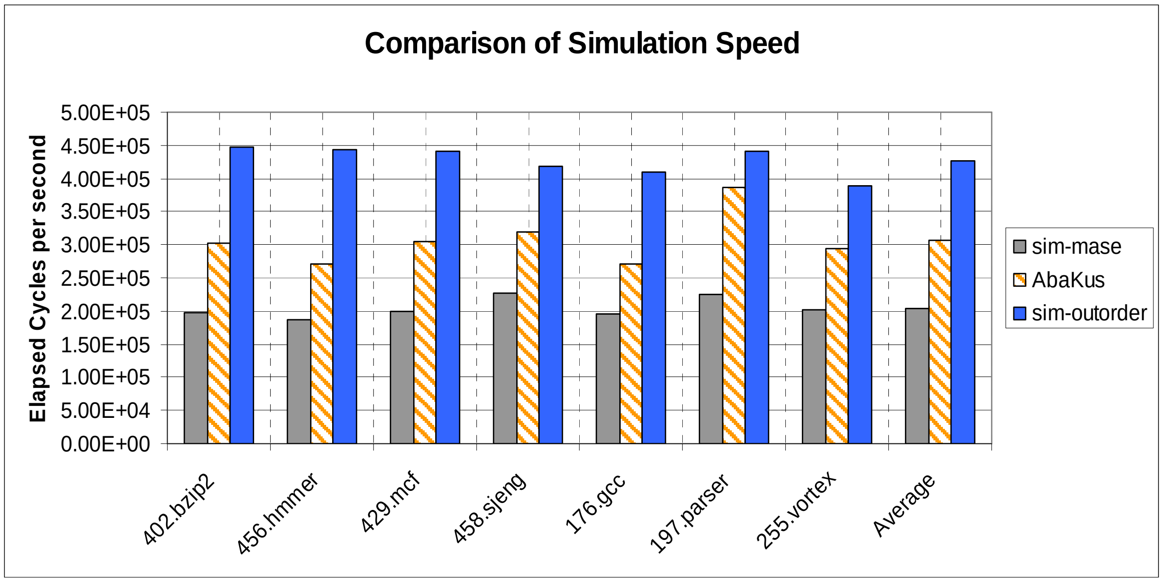


Figure 3.7 Comparison of Simulator Performance for a Superscalar Architecture Model

Due to the inherent dissimilarities between the simulated architectures, it is more pertinent to compare the elapsed simulation *cycles/s* between the simulators instead of instructions/s. Both *sim-mase* and *OSU AbaKus* have more detailed architecture simulation than sim-outorder. As seen in Figure 1, the new simulator *–OSU AbaKus-* is on average *50.27%* faster than sim-mase while sim-outorder is on average *39.04%* faster than OSU AbaKus.

TABLE 3.2 SIMULATION DETAILS OF THE THREE DIFFERENT SIMULATION MODELS

Design Parameters	**sim-outorder**	**AbaKus**	**sim-mase**
Instruction Fetch Width	4 inst/cycle	4 inst/cycle	4 inst/cycle
Instruction Window Size	Single Window: 64	Single Window: 64	Split Window: 64
Physical Registers	32	100	100
Issue Width	8	8	8
Commit Width	8	8	8
Branch Predictor	Perfect	Perfect	Perfect
Integer ALU units (Latency =1)	3	3	3
Mul/Div Unit (Latency = 6)	1	1	1
Float ALU units	4	Exception call	4
Float Mul/Div units	1	Exception call	1
Write Back Bus Width	Not Modeled	4	Not Modeled
Exceptions	Not Modeled	Precise	Precise
Memory Latency	1	1	1
Number of Executed Instructions	6 Billion	6 Billion	6 Billion
Average IPC	2.165	1.794	1.884
Average Simulation Time	6520.1 seconds	11268.7 seconds	15655.0 seconds

Although, other simulators such as publicly available *Liberty* and Intel's *ASIM* also focus on modularity, their over-head time on port communication is significant as the number of signals increase. The *3*-way hand-shake port communication in *Liberty* and multiple event-driven executions in ASIM slow the simulations as the complexity of the design increases (Vachharajani et. al. 2002 [2]). In contrast, *OSU AbaKus* is a cycle-time simulator similar to Simplescalar. Thus, the simulation speed of *OSU AbaKus* is compared only to those of Simplescalar *3.0* and MASE. The AbaKus simulator can be succinctly defined as an HDL that is familiar to hardware designers, but with a cycle-time based simulation environment.

The *OSU AbaKus* simulation tool set enjoys the advantages of modularity and simulation speed. Modularity is achieved by writing module descriptions as done in typical hardware description languages such as Verilog or VHDL. Changes to its modules are simplified because the modules are not sequentially dependent as is the case with Simplescalar tools. In the following section, the flexibility of the *OSU AbaKus* simulation tool will be demonstrated by studying the effect of write-back bus widths. The write-back bus is a natural part of our model because of the direct correspondence of simulation modules with real hardware modules; whereas the write back bus is not included in Simplescalar or MASE module descriptions.

3.4 Discussion

Simulation design objectives of AbaKus computer architecture design tool are provided below followed by a brief discussion.

1. Modularity: Breaks down performance modeling into different pieces.
2. Reusability of modules: Increases productivity and robustness of the software.

3. Familiarity with HDL programming.

4. Fast Simulation Speed.

1. Modularity:

All the modules in the simulation are accessed by either the *Global Input Port Pointer* or the *Global Output Port Pointer*. As indicated in Figure 3.8, the data elements the 2 global pointers point are switched for each cycle. Thus the updated values are read by the read ports while the write ports have a temporary data location to write its entries.

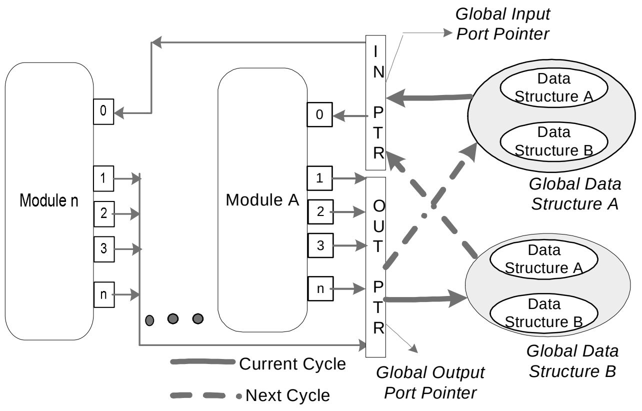


Figure 3.8. Illustration of Module Port Communication

2. Reusability:

Because our module port implementation is fully synchronous, much less simulation time is required to verify the architecture. The flexibility, i.e. reusability of modular code of the *AbaKus* simulation tool will be demonstrated by studying the effect of write-back bus widths. The write- back bus is a natural part of our model because of the direct correspondence of simulation modules with real hardware modules; whereas the write back bus is not included in Simplescalar or MASE module descriptions.

3. Familiarity of HDL Programming:

The following code structure format is similar to a behavioral HDL that is familiar to hardware developers.

```
<module_name>(){
OUTPUT* <output_struct_pointer>;      // Output Port Definition
INPUT* <input_struct_pointer>;        // Input Ports
INPUT_STALL* <input_stall_pointer>;   // Propagating Input Stall Signals
/*module descriptions*/
OUTPUT = module_function( INPUT, INPUT_STALL );     // Module Descriptions
}// End of Module
```

4. Fast Simulation Speed:

Both *sim-mase* (Larson et. al., 2001 [14]) and *AbaKus* have similar and more detailed architecture description than *sim-outorder*. The machine state of *AbaKus* and *sim-mase* architectures recovers from exceptions at the complete stage and write-back stage respectively, while *sim-outorder* recovers from exceptions at the dispatch stage. The functional units are matched both in terms of number of units and its latencies. In the Chapter 4, the hardware and software logic of architecture modeling are discussed in detail.

CHAPTER IV

SUPERSCALAR ARCHITECTURE MODEL

4.1 SUPERSCALAR DESIGN

This chapter describes about the basic structure of AbaKus' superscalar processor models in detail. The models are described in a structure similar to an HDL that is discussed in Chapter 3. The functional description of the modules is in standard C++. The basic modules of the 7-stage pipeline are the fetch, decode, dispatch, issue, finish, write-back and complete. The implementation details of each of these modules follows below,

Fetch Stage

In the CPU architecture core, the fetch stage of AbaKus architecture interfaces with the memory. The memory unit can be a cache module or the main memory. In a simple interface model, the fetch is interfaced to the main memory. Although, the memory interface architecture is a weak relationship with the actual CPU-Memory behavior, it can be extended to be interfaced with caches.

The following statement is a macro described in Simplescalar (Burger and Austin, 1997, [1]).

```
MD_FETCH_INST(inst, mem, fetchPC);
```

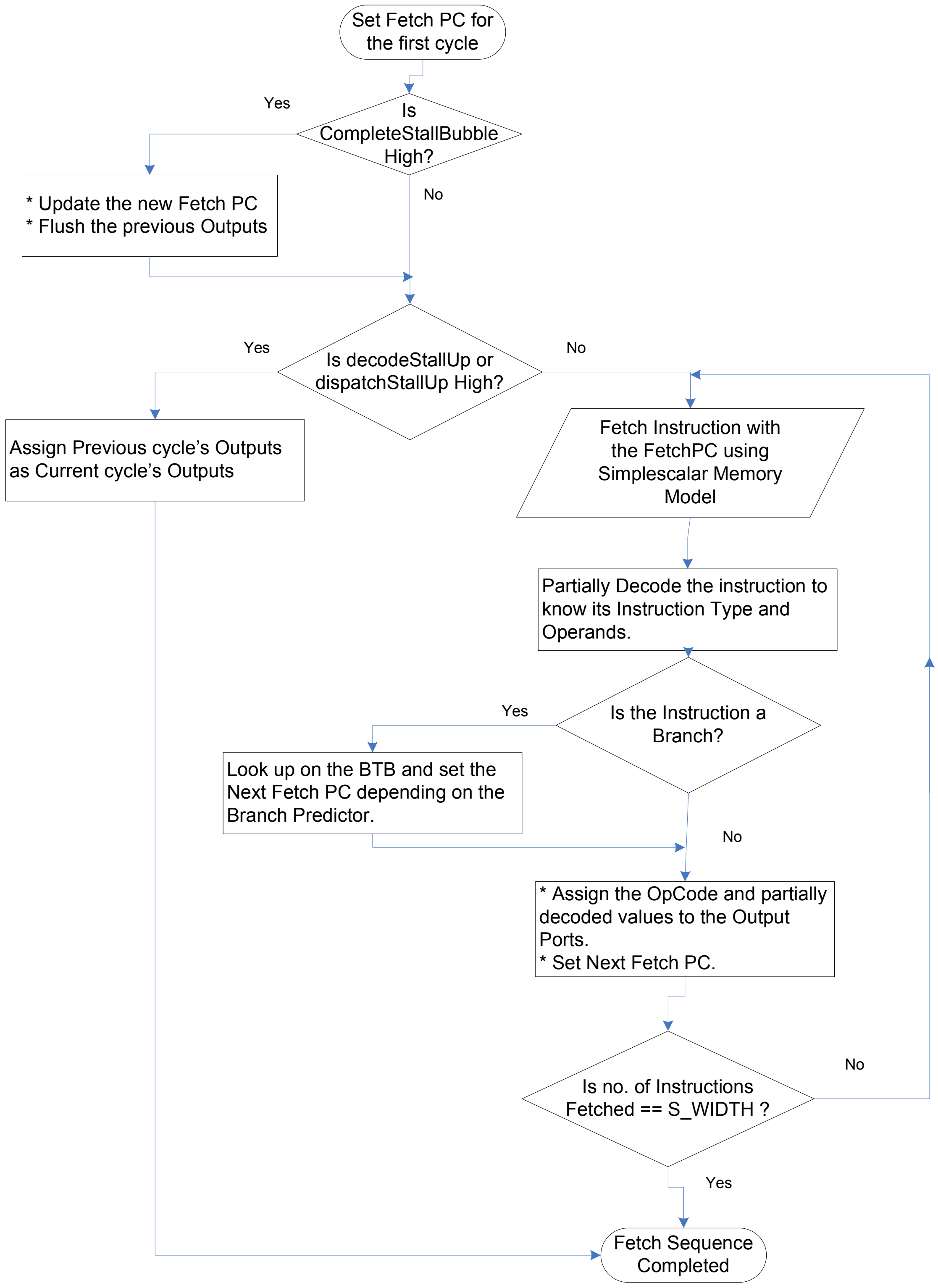
It is a direct interface to the main memory requiring only 3 arguments, the instruction object, main memory pointer and the Program Counter (PC) to fetch. A cache functional module can replace this statement in the fetch module. However, 2 additional signal arguments are required, if the cache functional module is interfaced, that is shown below,

```
cache_func(inst, cache_mem, fetchPC, stallUp_signal, hit_signal);
```

The 2 additional signals, `stallUp_signal` and `hit_signal` are required to ensure both timing as well as data coherency respectively. Following the instruction fetch of the corresponding PC, the instruction is partially decoded to identify its type and operands. This is done for simulation speed-up and also to balance the work-load.

Since the branch predictor look-up can have a significant adverse effect on the simulation time, it is necessary that only branch instructions need to be searched in the look-up table of the Branch Target Buffer (BTB). Hence, after the type of instruction is known through the partial decoder, only the branch instructions are allowed to access the BTB and the branch predictors. This is described in Flow Chart 1.

The work-load between these stages must be balanced because the *decode stage* has override logic, free-register priority encoder and register renaming where as the *fetch stage* only has the function of instruction fetch interfaced to the main memory. However, depending on the required timing, the fetch stage can be further easily be super-pipelined into instruction fetch and partial decoder stages.


Set Fetch PC for the first cycle
Is CompleteStallBubble High?
Yes
No
* Update the new Fetch PC
* Flush the previous Outputs
Is decodeStallUp or dispatchStallUp High?
Yes
No
Assign Previous cycle's Outputs as Current cycle's Outputs
Fetch Instruction with the FetchPC using Simplescalar Memory Model
Partially Decode the instruction to know its Instruction Type and Operands.
Is the Instruction a Branch?
Yes
Look up on the BTB and set the Next Fetch PC depending on the Branch Predictor.
No
* Assign the OpCode and partially decoded values to the Output Ports.
* Set Next Fetch PC.
Is no. of Instructions Fetched == S_WIDTH ?
No
Yes
Fetch Sequence Completed

Flow-Chart 4.1 Fetch Functional Module

Decode Stage

As mentioned earlier, the main functionality of this stage as implemented in AbaKus architecture is selecting *free rename register, register override logic* and *register renaming logic*.

Selecting Free Rename Register:

This functional block selects the next free register available to be renamed. The instruction set architecture registers are renamed to avoid name dependency stalls in the superscalar architectures. Basically, the number of required renamed registers is equal to the sum of instruction window width and instruction fetch width.

Selecting the free register is simple. It only requires determining the bit that is not set from the list of busy bits. The corresponding index of the busy bit is the register pointer for the free register.

Override Logic:

This is a special case where the operands of one or more subsequent instructions in an instruction decode group refer to the destination register of any of its previous instructions. In this case, the override logic makes sure that the newly renamed register that would only be updated in the next cycle get referenced to the operand that matches its pointer in the same cycle. This logic is discussed by Shen and Lipasti, 2005 [16] and is implemented in the AbaKus architecture model.

Register Renaming:

Register renaming is done by having 2 register pointer files – Architect Register Pointer File and Rename Register Pointer File. This is best explained with the help of the following diagram in Figure 4.1.

Register File (RF): Holds the values of the computed data.

Architect Register Pointer (RRP): Holds permanent register pointers for the 32, LO and HI registers of the Instruction Set Architecture (ISA). The updates are made at the complete stage.

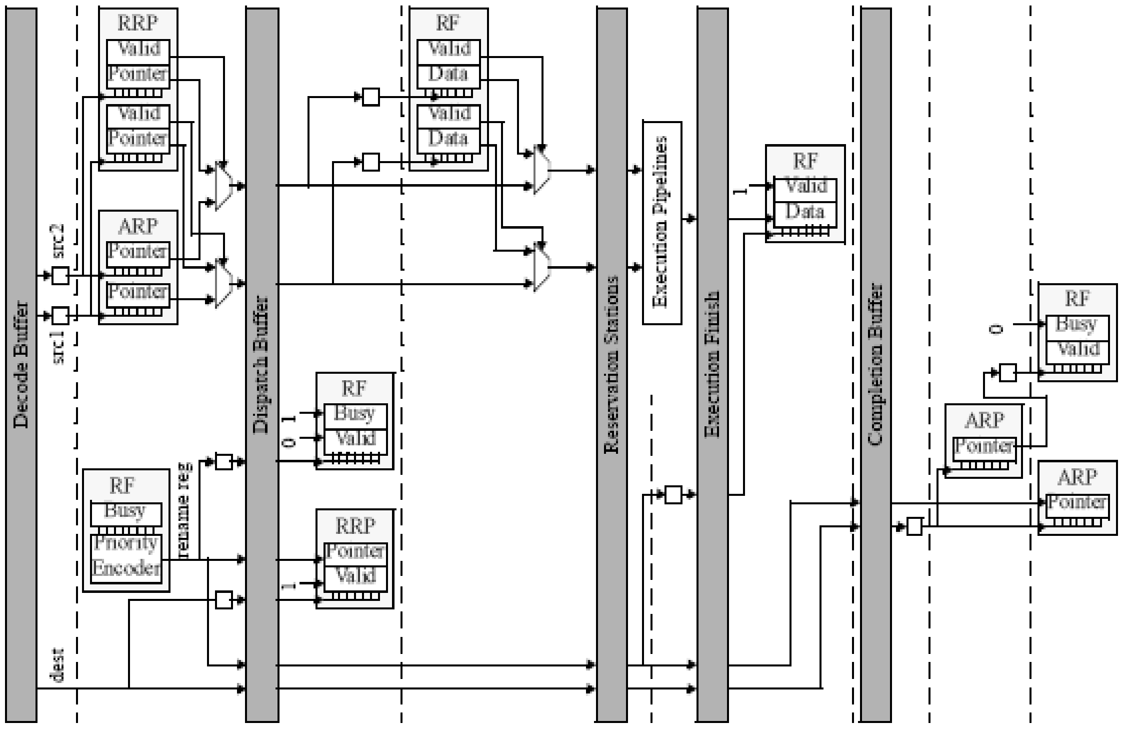


Figure 4.1 Design of Rename Register Logic

Rename Register Pointer (RRP): Holds temporary register pointers for all the destination registers of in-flight instructions in the pipeline and is updated at the dispatch stage.

Hence, instructions with dependent source operands refer to the RRP at the decode stage to find out the correct dependent register pointers.

Dispatch Logic

Instructions are dispatched to a special instruction window buffer after the decode logic. The number of entries in the instruction window is fixed during compilation time. The fields of the instruction window entry are shown in Figure 4.2.

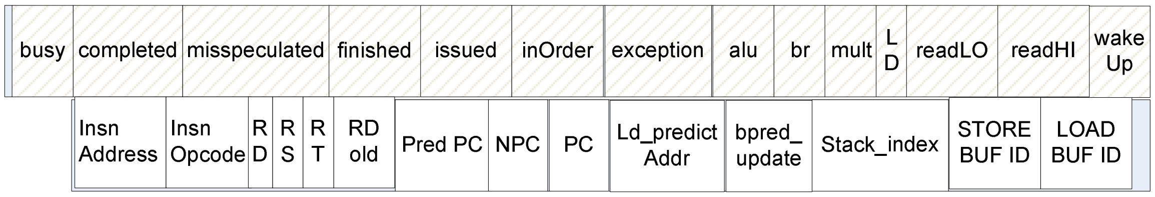


Figure 4.2 Fields of Instruction Window Entry

The implementation of the instruction window buffer is a choice of the designer. For hardware logic implementations such as FPGA or custom IC, it is efficient to implement the instruction window buffer as a fully-associative memory. On the other hand, for a software simulation it is efficient to implement this special buffer as a direct-mapped cache.

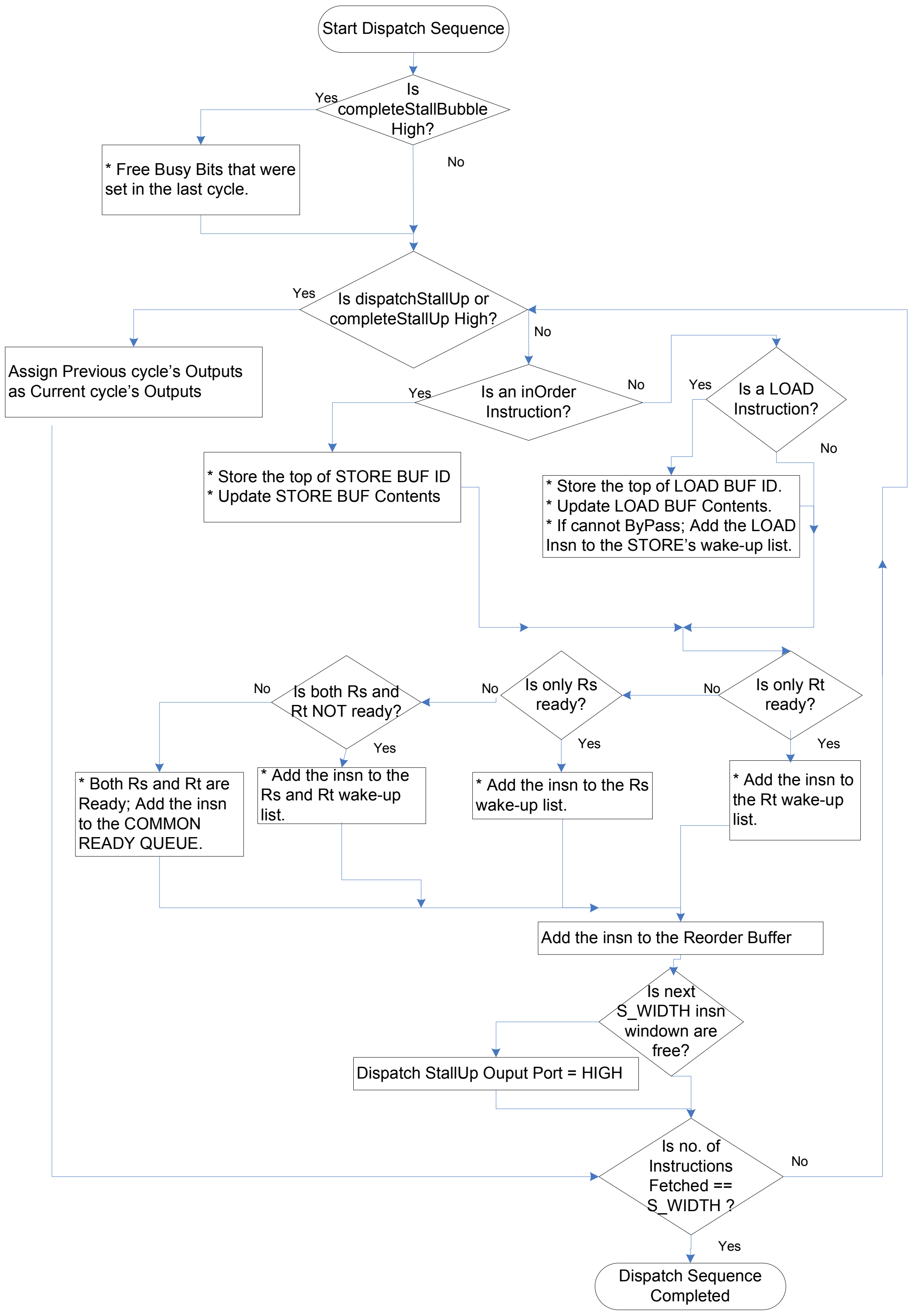
In Figure 4.2, the hashed fields represent a single bit field and the remaining fields are represented by 32 bits in the software implementation. However, the number of bits should be discerned carefully for the hardware implementation depending on the requirement. The summary of description of each field is described in Table 4.1.

TABLE 4.1. SUMMARY OF DESCRIPTION ON THE FIELDS OF THE INSTRUCTION WINDOW

Name of the Field	**Description**
Busy	Indicates the entry is busy or free.
Completed	Indicates the instruction is completed/committed
Mis-speculated	Indicates the instruction is misspeculated and have to thrown out.
Finished	Indicates the instruction has finished execution
Issued	Indicates the instruction has its operands ready and is issued in the issue queue.
InOrder	Indicates the instruction enforces order of fetch, i.e. the STORE instruction forces all other instructions fetched before it must be completed, if no load prediction/memory disambiguation is turned on.
Exception	*syscall* or any special instructions that is not implemented in the hardware to be treated as an *exception.*
ALU	Indicates an ALU type of instruction.
Br	Indicates a BRANCH/JUMPtype of instruction.
lD	Indicates a LOAD type of instruction.
Mult	Indicates a Multiplication/Division type of instruction.
readLO	Indicates a lower 32-bit of the 64-bit multiplication result.
readHI	Indicates a higher 32-bit of the 64-bit multiplication result.
Insn Address	Instruction Address (32-bit) of the instruction

Insn Opcode	Instruction Opcode (insn A & insn B) of the instruction
RD	Destination Register
RS	Source Operand A
RT	Source Operand B
Rd_Old	Old Destination Register
PC	Program Counter
NPC	Next Program Counter
Pred PC	Predicted Program Counter at the branch instruction
Ld_Predict Address	Predicted Load Dependent Address at Fetch
Bpred_update	Branch Update Structure Pointer
Stack_index	Index of the Branch Stack for a direct jump instruction.
STORE BUF ID	Index of the top of STORE Buffer
LOAD BUF ID	Index of the top of LOAD Buffer
Wake-Up	0 – Indicates both Rs and Rt are not Ready. 1 – Indicates either Rs or Rt is Ready.

The dispatch logic is described in the Flow Chart 4.2. The head pointer of the instruction window is incremented and it is determined if the next 'S_WIDTH' of instruction window entries are available. If not, then the output port of the *dispatchStallUp* signal is raised high.


Start Dispatch Sequence
Is completeStallBubble High?
Yes
No
* Free Busy Bits that were set in the last cycle.
Is dispatchStallUp or completeStallUp High?
Yes
No
Assign Previous cycle's Outputs as Current cycle's Outputs
Is an inOrder Instruction?
Yes
No
* Store the top of STORE BUF ID
* Update STORE BUF Contents
Is a LOAD Instruction?
Yes
No
* Store the top of LOAD BUF ID.
* Update LOAD BUF Contents.
* If cannot ByPass; Add the LOAD Insn to the STORE's wake-up list.
Is only Rt ready?
No
Yes
* Add the insn to the Rt wake-up list.
Is only Rs ready?
No
Yes
* Add the insn to the Rs wake-up list.
Is both Rs and Rt NOT ready?
No
Yes
* Add the insn to the Rs and Rt wake-up list.
* Both Rs and Rt are Ready; Add the insn to the COMMON READY QUEUE.
Add the insn to the Reorder Buffer
Is next S_WIDTH insn windown are free?
Dispatch StallUp Ouput Port = HIGH
Is no. of Instructions Fetched == S_WIDTH ?
No
Yes
Dispatch Sequence Completed

Flow Chart 4.2 Dispatch Logic

Issue Logic

The issue logic reads the instructions in the *Common Ready Queue* and adds it to the separate *issue queue* that is specific for each instruction type. Basically, there are 5 categories of *issue queue* – ALU, BR, LD/STORE, MULT/DIV and Other instructions such as *syscall*, *DLW*, *DSW* and other floating-point instructions. It is important that the instructions have individual queues because a *stall* in one of the functional unit would not stall-up the entire queue. The functional block of the *dispatch logic and the issue logic* is illustrated in Figure 4.3.

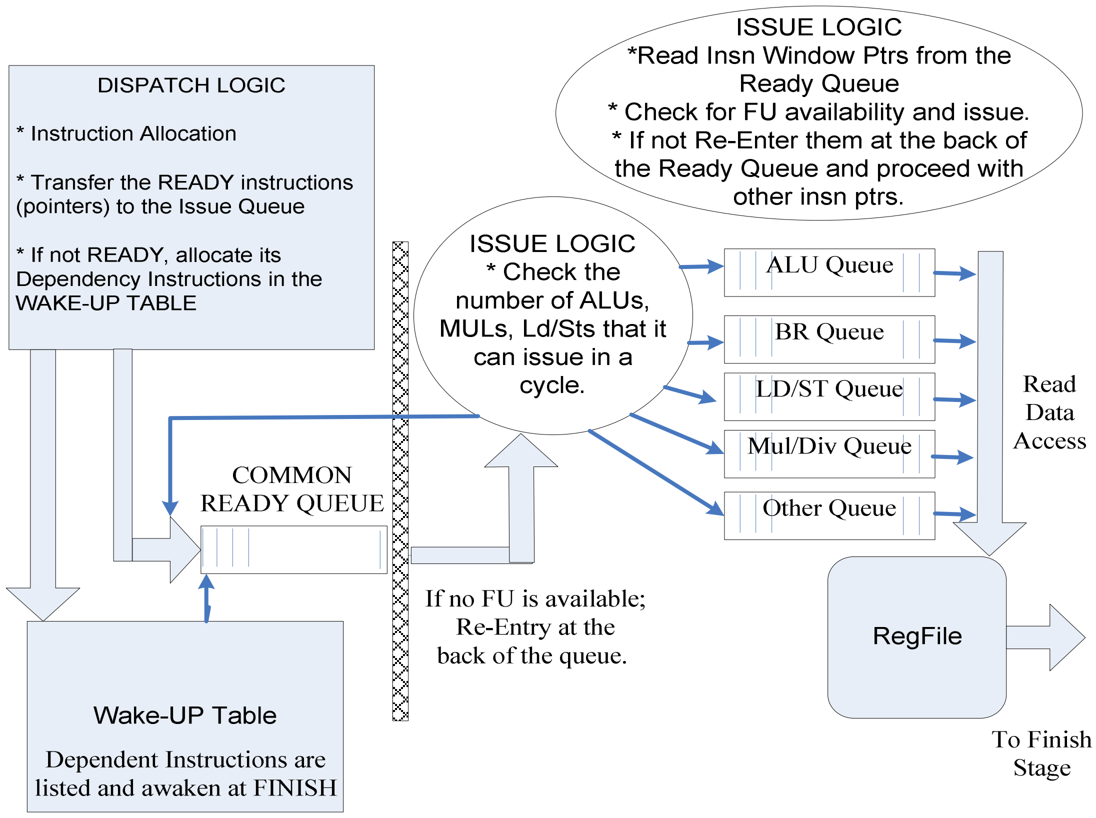

ISSUE LOGIC
*Read Insn Window Ptrs from the Ready Queue
* Check for FU availability and issue.
* If not Re-Enter them at the back of the Ready Queue and proceed with other insn ptrs.
DISPATCH LOGIC
* Instruction Allocation
* Transfer the READY instructions (pointers) to the Issue Queue
* If not READY, allocate its Dependency Instructions in the WAKE-UP TABLE
ISSUE LOGIC
* Check the number of ALUs, MULs, Ld/Sts that it can issue in a cycle.
ALU Queue
BR Queue
LD/ST Queue
Mul/Div Queue
Other Queue
Read Data Access
COMMON READY QUEUE
If no FU is available; Re-Entry at the back of the queue.
RegFile
To Finish Stage
Wake-UP Table
Dependent Instructions are listed and awaken at FINISH

Figure 4.3. Functional Block Diagram of the Issue Logic

[1] The pointer of the ready instruction that is put into the *Common Issue Ready* is read and its corresponding instruction window entry and the instruction type are determined.

[2] Depending upon the type of instruction, it is then added to the respective instruction issue queue. Step 1 and 2 are continued until all the ready instructions in the *Common Issue Ready* are added into its specific instruction type queues.

[3] Finally, if there is no *stall-up* signal for the corresponding issue queue then the instruction is assigned to the output ports for issue. Although, the instruction is assigned for issue, it is only finalized, i.e. the *issue bit* is set only in the next cycle because there can be a stall in the *execute stage* that is propagates to the *issue stage* only in the next cycle.

A *Round-Robin priority* issue is implemented in order equally distribute the instruction issue among the different instruction types. The number of instruction issues is set as a compilation parameter in the *sc_datatypes.h* file. The issue queue stalls due to unavailability of functional units and finalization of the instruction issue are determined in the next stage – *Execute Stage*. The number of entries in the individual issue queue is a compilation parameter and is set equal to the number of entries in the instruction window. Stall-Up signals due to unavailability of issue queue entry is not implemented, however, *the optimal number of entries in the individual issue queue* is a topic of future research.

Execute Stage

The execute stage consists of ALU, Mult/Div, BR, LD/STORE and Float/Other instructions functional units. Each of functional units has latency, ‘m’, which is a compilation parameter. Besides, the number of functional units, ‘n’, of each instruction type is also a variable that is defined during the program compilation.

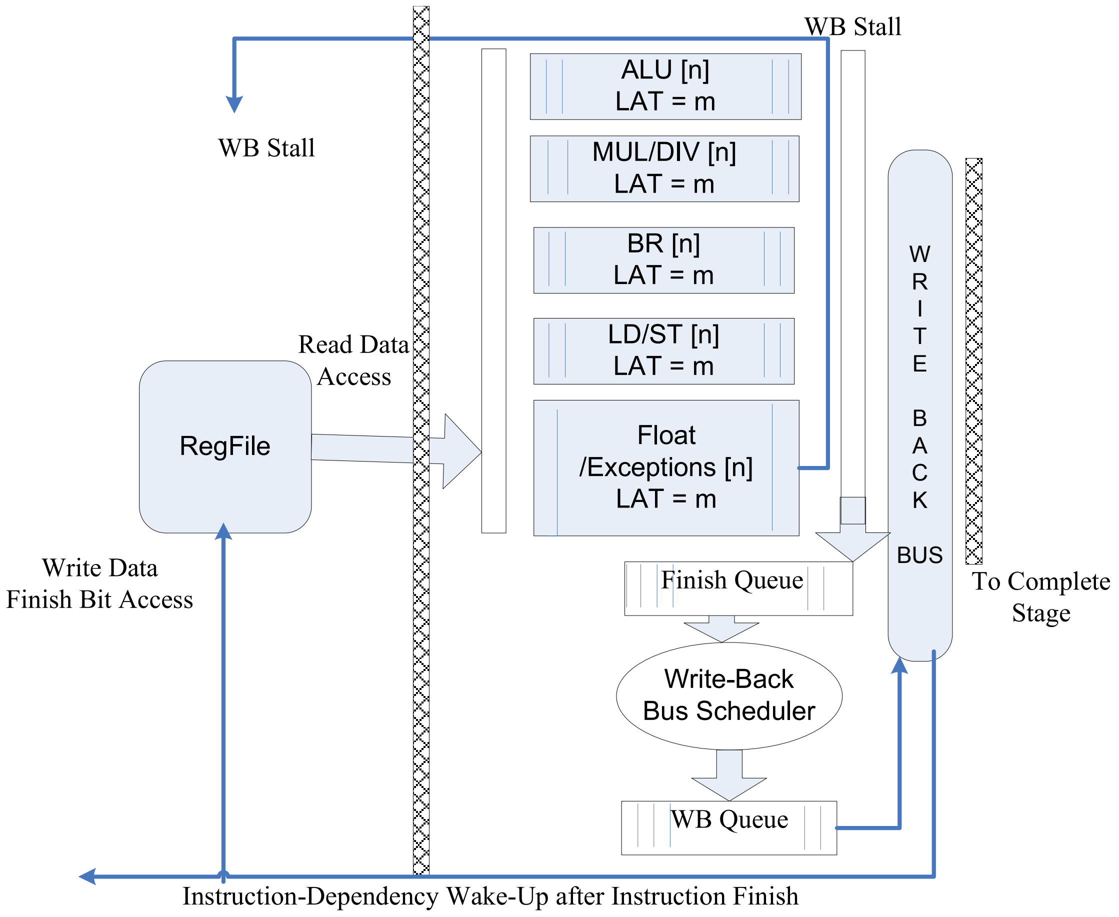
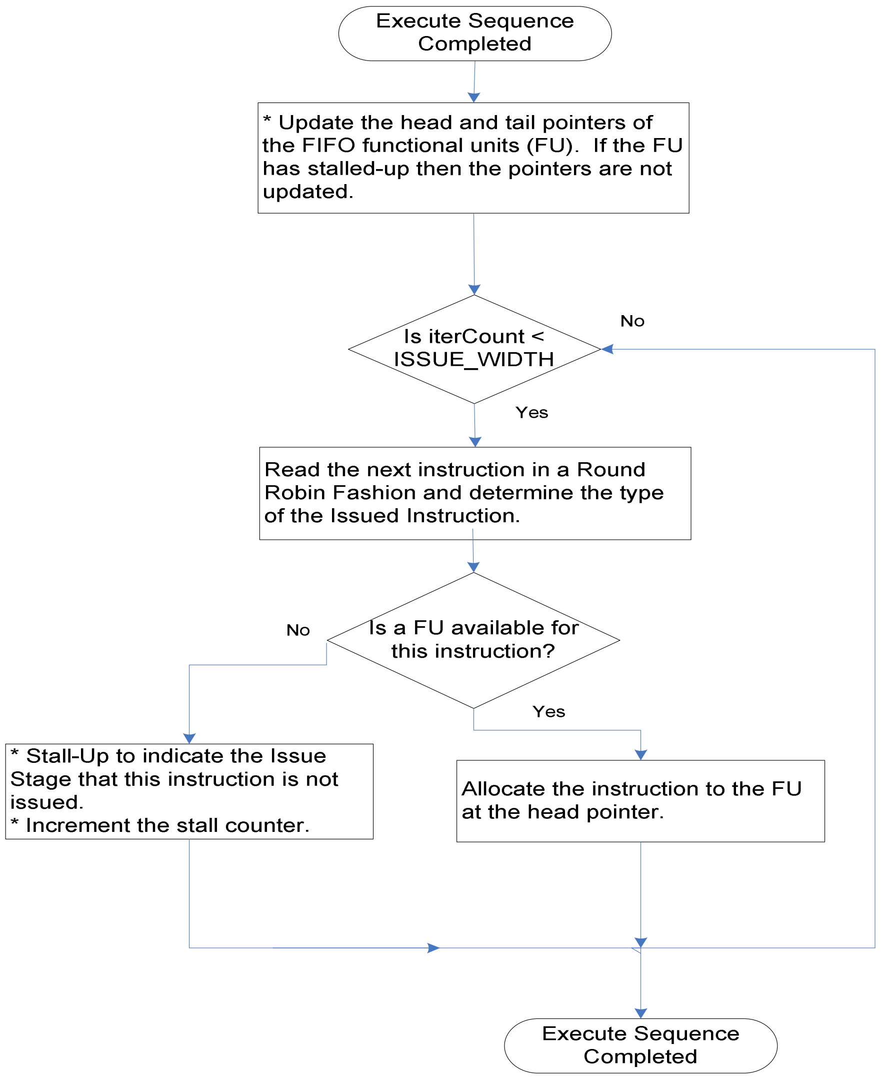


Figure 4.4. Functional Block Diagram of the Execute Stage.

It should be noted that at the execute stage only the latency of the instruction execution is simulated but the actual instruction execution takes place only at the finish/write-back stage. The pipeline stage of the functional unit is implemented as a circular FIFO queue. The head and tail pointers of the queue are updated at each cycle.

The instructions are read from the issue queue and are assigned to the corresponding functional units in a round-robin fashion. If no functional units are available then a *stall-up* signal is for the corresponding functional unit is raised high. After a fixed number of cycles, the inserted instructions in the circular FIFO queue at the head pointer propagate to the tail pointer. Once the instruction i.e. the instruction window ID reaches the tail pointer, it is determined to be finished the execution. As shown in Figure 4.4, the instruction is then inserted into the *finish queue*.


Execute Sequence Completed
* Update the head and tail pointers of the FIFO functional units (FU). If the FU has stalled-up then the pointers are not updated.
Is iterCount < ISSUE_WIDTH
No
Yes
Read the next instruction in a Round Robin Fashion and determine the type of the Issued Instruction.
Is a FU available for this instruction?
No
Yes
* Stall-Up to indicate the Issue Stage that this instruction is not issued.
* Increment the stall counter.
Allocate the instruction to the FU at the head pointer.
Execute Sequence Completed

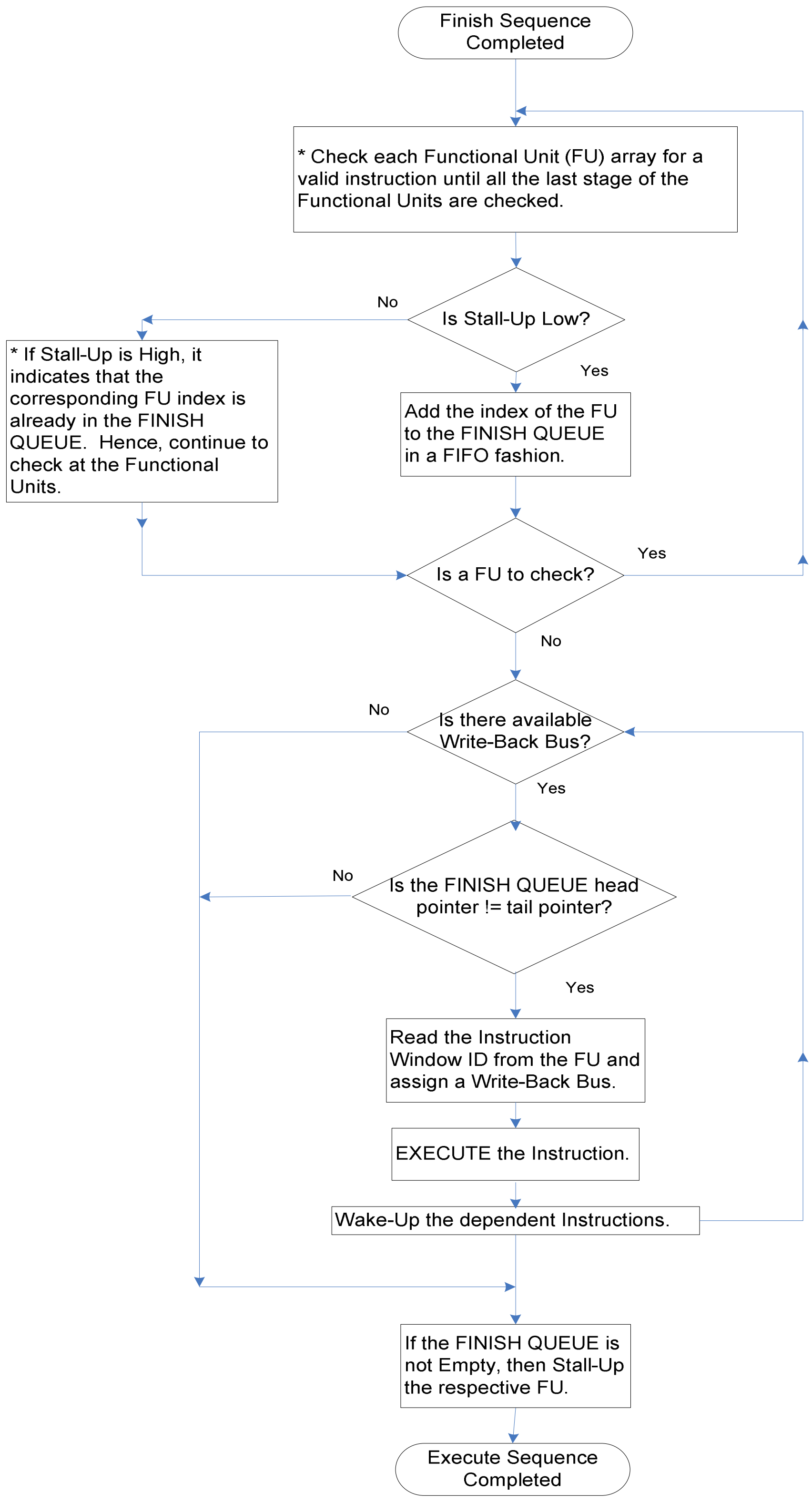
Flow Chart 4.3 Execute Stage

Finish/Write-Stage:

The *finish-stage* is an important module as instructions are scheduled to finish by accessing the write-ports of the *Register File*, if required, and also its dependent instructions are waken-up. In addition, the function execution of the instruction takes place at this stage through a subroutine macro call – *SYM_CAT( ).* As shown in Figure 4.4, the finished instructions are inserted into the FINISH QUEUE. The order in which the instructions are scheduled into this queue determines the write-back bus scheduling order. By default, the instructions are arranged in the FINISH QUEUE in a FIFO fashion. A more detailed study of scheduling the instructions in the FINISH QUEUE is discussed later in Chapter 5.

The instructions are read from the FINISH QUEUE and assigned a write-back bus, if available. If the write-back bus is not available, then that specific Functional Unit is stalled-up. Once assigned a write-back bus the instruction is set to finish, i.e. the instruction is functionally executed and the results are updated in the register file in the next cycle.

Once the instruction finishes execution, its dependent instructions are found by walking through the dependent list of the wake-up structure. The wake-up structure has a list of instruction window pointers. Either the wake-up bit of the dependent instruction window slot is set to 1 or the instruction is directed to the READY QUEUE depending upon its operands validity. The finish-stage functionality is further illustrated in Flow Chart 4.4.


Finish Sequence Completed
* Check each Functional Unit (FU) array for a valid instruction until all the last stage of the Functional Units are checked.
No
Is Stall-Up Low?
Yes
* If Stall-Up is High, it indicates that the corresponding FU index is already in the FINISH QUEUE. Hence, continue to check at the Functional Units.
Add the index of the FU to the FINISH QUEUE in a FIFO fashion.
Yes
Is a FU to check?
No
No
Is there available Write-Back Bus?
Yes
No
Is the FINISH QUEUE head pointer != tail pointer?
Yes
Read the Instruction Window ID from the FU and assign a Write-Back Bus.
EXECUTE the Instruction.
Wake-Up the dependent Instructions.
If the FINISH QUEUE is not Empty, then Stall-Up the respective FU.
Execute Sequence Completed

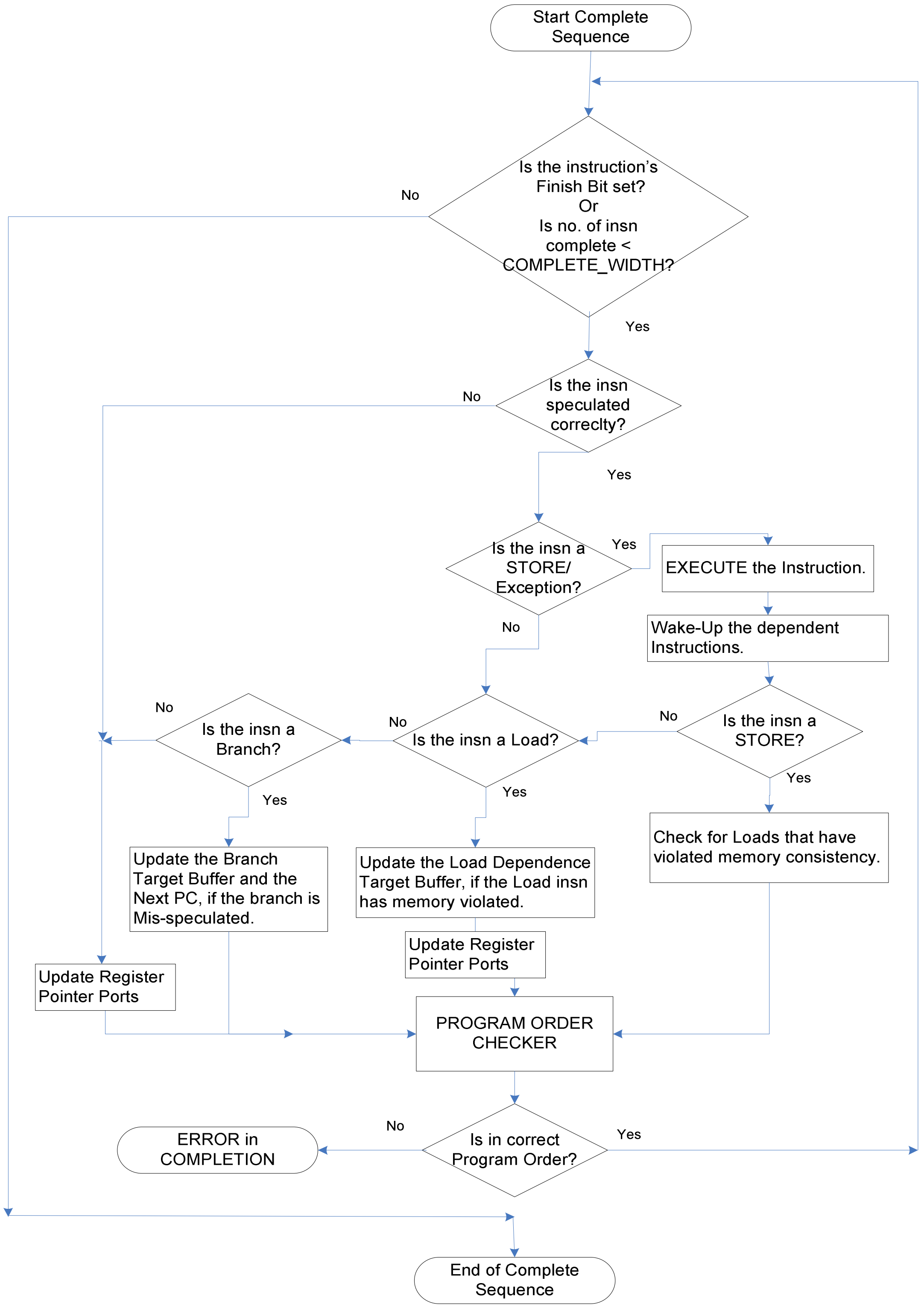
Flow Chart 4.4 Finish Stage Logic

Complete Stage

The *complete stage* includes retiring the STORE instructions, executing the FLOAT and exception causing instructions, waking up dependent instructions and resetting the instruction window and register pointer entries. In addition, the complete stage also has *in-order instruction checking mechanism* to ensure that the completing instruction is the instruction in the program order. Apart from these functionalities, the complete stage also takes care of memory disambiguation that is discussed later in this chapter. The instructions are ready for complete, when they have finished execution and assigned for completion based on the program order. The maximum number of instructions that can be completed is defined as COMPLETE_WIDTH in the *sc_datatypes.h* header file.

The flow chart in Figure 4.5 illustrates the complete stage logic. When STORE complete, it checks for data memory address violation in the load buffer, i.e. it check if a load instruction after the store instruction had already finished execution. If it were the case, then the finished load instruction would have a stale value. Hence, those loads that have memory violated are identified and are marked as '*memory violated'* in the instruction window. Later, as the load instruction completes, if the '*memory violated'* bit is set, then, all subsequent instructions following the load instruction are not completed and initiates processor recovery state.

Similarly, as branch instructions complete, the '*next program counter'* is checked for equality with the '*predicted program counter'*, if not equal, then all instructions following the branch are not completed and processor recovery is initiated.


Start Complete Sequence
Is the instruction's Finish Bit set? Or Is no. of insn complete < COMPLETE_WIDTH?
No
Yes
Is the insn speculated correclty?
No
Yes
Is the insn a STORE/ Exception?
Yes
No
EXECUTE the Instruction.
Wake-Up the dependent Instructions.
Is the insn a STORE?
No
Yes
Check for Loads that have violated memory consistency.
Is the insn a Load?
No
Yes
Update the Load Dependence Target Buffer, if the Load insn has memory violated.
Update Register Pointer Ports
Is the insn a Branch?
No
Yes
Update the Branch Target Buffer and the Next PC, if the branch is Mis-speculated.
Update Register Pointer Ports
PROGRAM ORDER CHECKER
Is in correct Program Order?
No
Yes
ERROR in COMPLETION
End of Complete Sequence

Flow Chart 4.5 Complete Stage Logic.

4.2 Store Buffer and Load Dependence Prediction Mechanism

Memory consistency in a computer system has become a vital part in the design of the multiprocessor systems. Although, there is only one process executing in a single-threaded superscalar machine, the fact that instructions executed out-of-order introduces the challenge to maintain memory coherency. The order of reads and writes into the cache or the main memory must be maintained in program order by the hardware logic and any violation of this order can cause erroneous result in execution.

In the case of simple scalar pipelines, the memory consistency is satisfied because the writes and reads are inherently executed according to the program order. However, in the case of the out-of-order execution this order of accessing the memory must be enforced by a special logic and buffers – *Load Finish Buffer* and *Store Finish Buffer*.

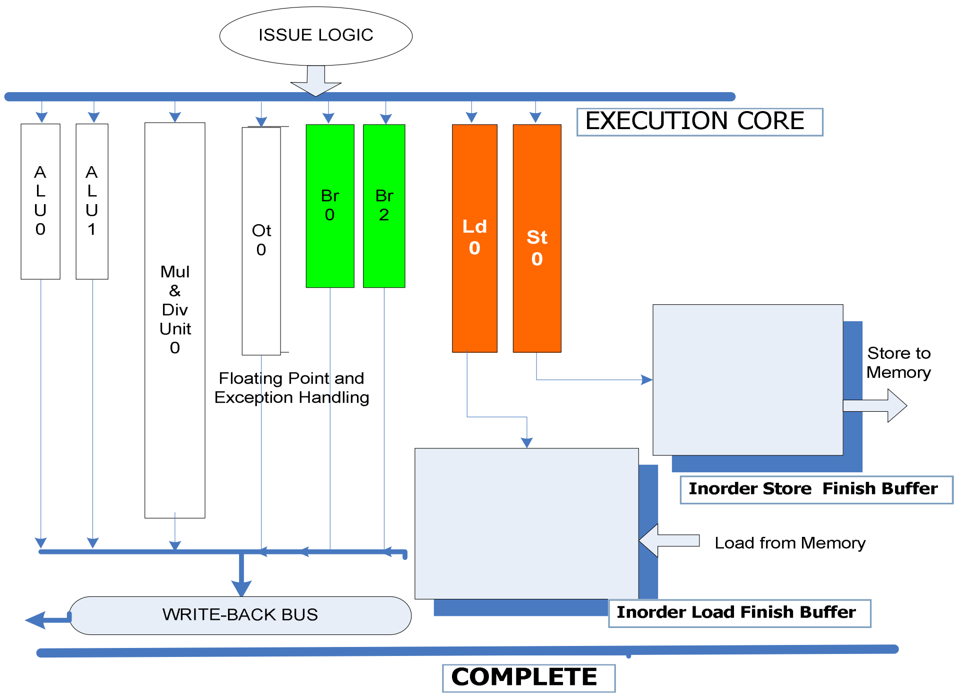


Figure 4.5 Load Finish and Store Buffer Models

Load Forwarding

This is a scheme in the out-of-order machines to reduce the latency of the load instruction by forwarding the data from the *store finish buffer* instead of accessing it from the data cache. Store instructions write their destination value into the store finish buffer in program order and are then updated into the cache or the main memory – store retirement. If a load instruction follows a store instruction before the store instruction is retired then the data can be forward to the load instruction from the store finish buffer.

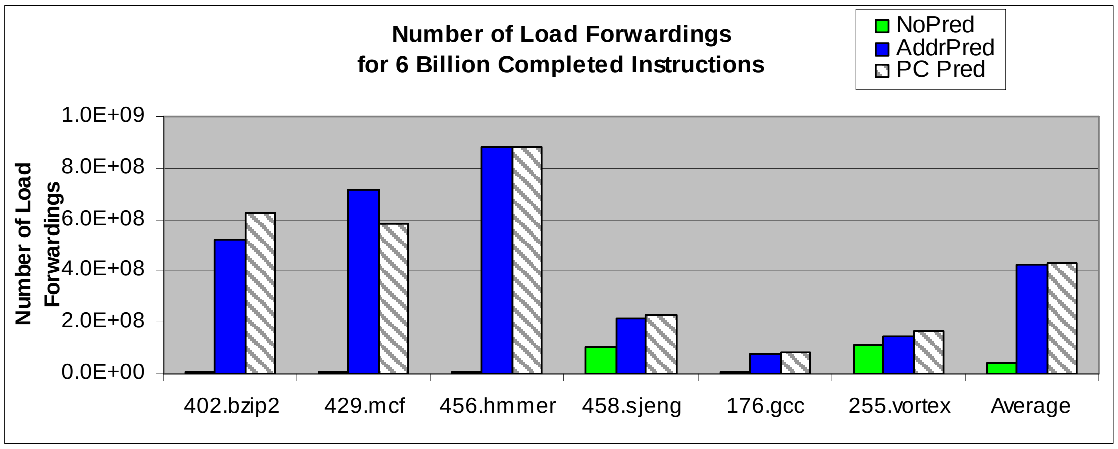
4.2.1 Load – Store Address Dependence Prediction

Resolving a load instruction quickly can result in increased speed-up because large percentage of instructions in the program is dependent on the load instructions. Hence, by predicting the dependence between a store and load instruction, a load instruction can be allowed to by-pass a store instruction, if there is no dependence between the pair of instructions. The store finish buffer is used to determine if the load instruction can by-pass the preceding store or has to wait till the store instruction is executed.

The dependence is based on previous machine recoveries due to load-store memory violation, i.e., a load instruction had executed even before the store to that location can update the data. The load-store prediction buffer is placed at the fetch stage and in most cases it is similar to the operation of the branch target buffer (BTB) except that the target address is the speculated memory load address. The relationship between the load and store instruction can be determined in 2 ways.

1. By matching the instruction address between the load and store instruction.
2. By matching the destination memory address between the load and store instruction.

In the case of no prediction, loads by-pass store instructions without any restriction. The number of load forwarding and recoveries due memory violation are illustrated in Figures 4.6 and 4.7.



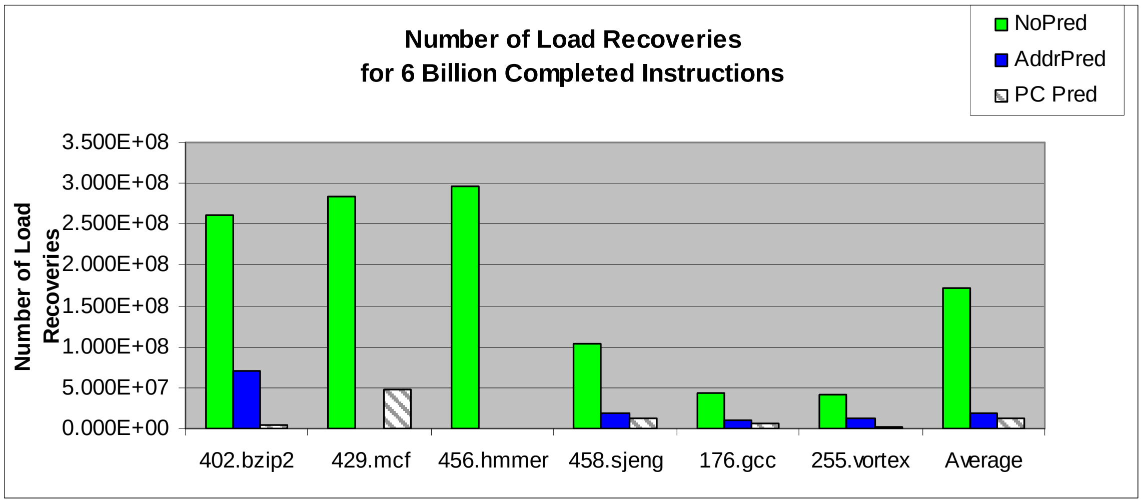


Figure 4.6 No. of Load Forwarding and Memory Recoveries with destination memory address prediction and instruction address (PC) prediction.

As seen in Figure 4.6, the number of load forwarding is about 4 times more than the case with no prediction. As the number of load forwarding increases the instructions that depend on the load can be issued quickly and hence results in increased IPC as seen in Figure 4.7.

Similarly, the number of recoveries in the case of no prediction is about 15 times more than with the prediction. This shows that load and stores are highly dependent and it is important to have some schemes like load-store dependence prediction in the machine to improve the performance of the processor.

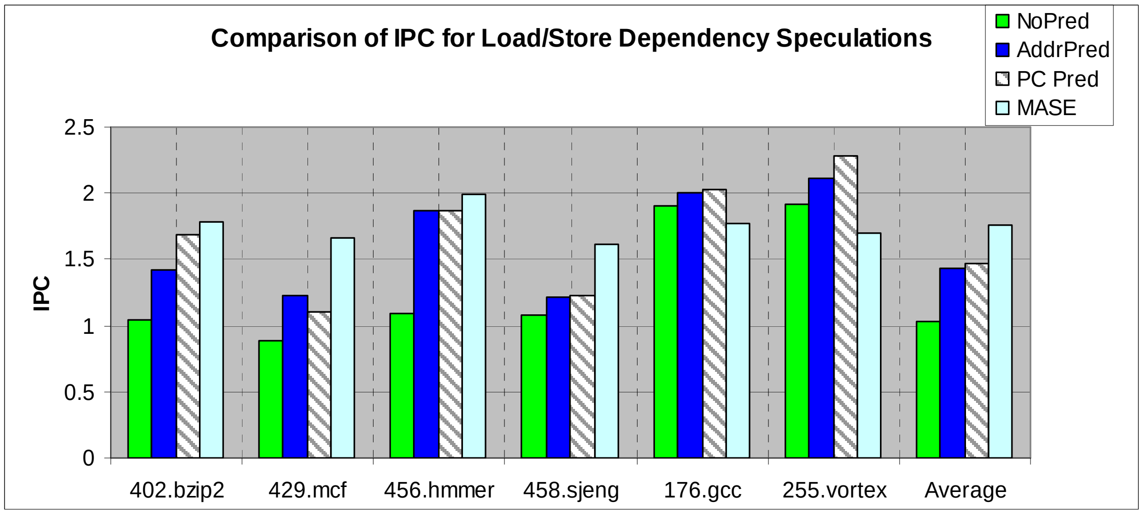


Figure 4.7 IPC with and without Load-Store Dependence Prediction

As seen in Figure 4.7, there is about 50% improvement on average IPC which is significant considering the simplicity of the scheme. In addition, the low percentage of recoveries with load-store address prediction also reveals that loads and stores dependencies can be predicted with high degree of accuracy. However, in some cases both the dependence approaches fail resulting in a machine-state recovery. These cases are as follows,

1. Instruction Memory Address Prediction (PC):

```
for(i=1 to 1 x 10^6) {
    if ( i mod 2 = 0 )
      R3 = Load(&Y);
    else
      R3 = Load(&Z);
   Store(&X) = R3;
}
```

In the above lines of codes, the relationship between a single load and store instruction for a loop unrolled code cannot be established because the loading memory address toggles for every count. Hence, more dependence entries have to be stored to predict dependence over number of memory addresses or a combination of data memory address prediction can be used.

2. Problem with Data Memory Address Prediction (AddrPred)

```
for(i=1 to 1 x 10^6) {
   R3 = Load(&Y);
 Store(&X + i) = R3;
}
```

In the above case, the load memory data is stored into different store location in the iteration. Hence, it is not possible to establish a relationship because of ever changing store address. In this case, a sophisticated logic using stride predictor or instruction addressed based prediction can be used.

4.3 Summary

Memory violation due to load instructions can be detected when the store instruction completes by simply checking the load finish buffer. The completing store instruction checks for a memory address match and then for a data match. If the data of store does not match the data of the following load in the load finish buffer, then the load instruction had violated its order of execution, it set a bit and the processor machine-state has to be recovered once the load instruction is ready for completion.

A more interesting challenge arises when a store to a byte is followed by a load to a word of the same address. Since, at this only a byte address is present in the store finish buffer. These cases are detected and the memory violation bit in the instruction window in set, initiating the machine-state recovery when the load completes. Such occurrences are not common and compiler can take care of it by changing the store to a byte to store to a word

CHAPTER V

WRITE-BACK BUS SCHEDULING MECHANISMS FOR MULTI-PORT REGISTER FILE DESIGN

In a superscalar processor each execution unit, with the exception of the store unit, requires a write-back bus to update the state of the register file. Ideally, each execution unit has a write-back bus both to update the register file as well as to forward the results to the waiting instructions. In order to reduce the cost of the register file and the cost of instruction wake-up logic, we explore the effect on IPC by having fewer write-back buses than the total number of execution units. Furthermore, the performance of various write-back bus scheduler algorithms is also studied. A major bottleneck in the instruction flow is the size of the register file write-back bus. The size of the write-back is critical for the following reasons:

a. The number of write-back buses is proportional to the number of write-ports in the register file. Multi-port register files are expensive to fabricate as they require more transistors and chip area. The cost of the multi-port SRAM increases as n^2, where n is the number of write-ports in the register file.

b. In the instruction scheduler design, special wake-up issue logic circuitry has to be designed for each write-back bus to determine if the operands are ready for the waiting instruction. Hence, the complexity and cost of the hardware increases with the size of the write-back bus.

c. For an architecture design that is only limited by data dependencies, the number of register write-back buses limits the flow of instructions. This exacerbates the data dependency problem as the instructions wait to update the results in the register file.

In order to emphasize only the effects of the write-back bus width, a sufficient number of execution units is simulated. Many stalls that are incurred at the finish stage are only due to lack of sufficient write-back buses, eventually stalling the upstream instruction fetching. The problem of insufficient write-back buses is more pronounced in a Simultaneous Multi-Threaded (SMT) processor. As an SMT type processor maximizes the utilization of the execution core, there is much more demand on the write-back buses than with a superscalar processor. Hence, it is important to understand the size and the write-back scheduling logic for these buses can be expensive but when lacking will tend to limit the instruction flow.

5.1 Related Work

A delay write-back queue strategy similar to a load and store buffer is proposed by Kim and Mudge, (2003 [17]) to reduce the number of write ports. In their paper, they show a 20% savings in energy for a modest penalty in IPC. A multi-level register bank is proposed by Cruze et. al., (2000 [18]), as an alternative to reduce the register file write-ports. This scheme is further extended by Balasubramonian et al., (2003 [19]), with a register-file allocation policy to increase the hit rates in level 1 register file. A low-power 12-port multi-bank register file is designed by Sueyoshi et al., (2004 [20]), and shows a 72% decrease in area compared to a 12-port-cell-based register file.

Kim and Mudge, (2003 [11]), use the more common FIFO scheduler between the functional units and the write-back buses. Our paper shows the relationship between various bus schedulers and its effect on CPU performance in a detailed manner. Contention between write-back buses is identified by Smotherman et al., (1993 [22]), and takes a heuristic approach to reduce this problem.

5.2 Write-Back Bus Model

In this section we describe architecture details that are associated with simulating the write-back buses. In our paper, the write-back bus allocation policy is used as an example of this capability since such subtle but important aspects of computer architecture design are not always modeled in Simplescalar tool sets [1].

A detailed model of the finish stage is illustrated in Figure *5.1*. The write-back busses in our simulations can be considered to be extensions of the "common forwarding data bus" in Tomasulo's classic algorithm. The write-back busses not only access the ports of the register file but also update the control information for store and branch instructions.

5.2.1. Round-Robin Issue Logic

The write back bus allocation strategy cannot be studied independently of the issue strategy, but we did not intend for our paper to be a detailed study of pipeline scheduling algorithms. We are attempting to show the ease with which detailed studies of hardware design trade-offs can be done with our modular approach. The instructions in the issue ready queue are inserted by the dispatch stage. During issue, the instructions are taken out of the issue queue and are issued to appropriate type execution units in a round-robin fashion. This results in the instructions being distributed equally in their set of execution units.

5.2.2. Execution Units:

The execution units are grouped into three sets and are scalable. All the execution units in each of the sets have individual stall signals.

a) ALU: Executes integer add, sub and bit-wise type of instructions.

b) MUL/DIV: Executes integer multiply and divide type of instructions.

c) Other Execution Units (OEU): Executes other remaining instructions, such as *load, branch* and *float* instructions. Besides, it also handles *store* instructions that are processed by simplified store buffer logic. Since, the simulated architecture does not model the floating-point register file all the float instructions in the SPEC CPU INT 2000/2006 benchmarks are treated as *exceptions* and are handled precisely.

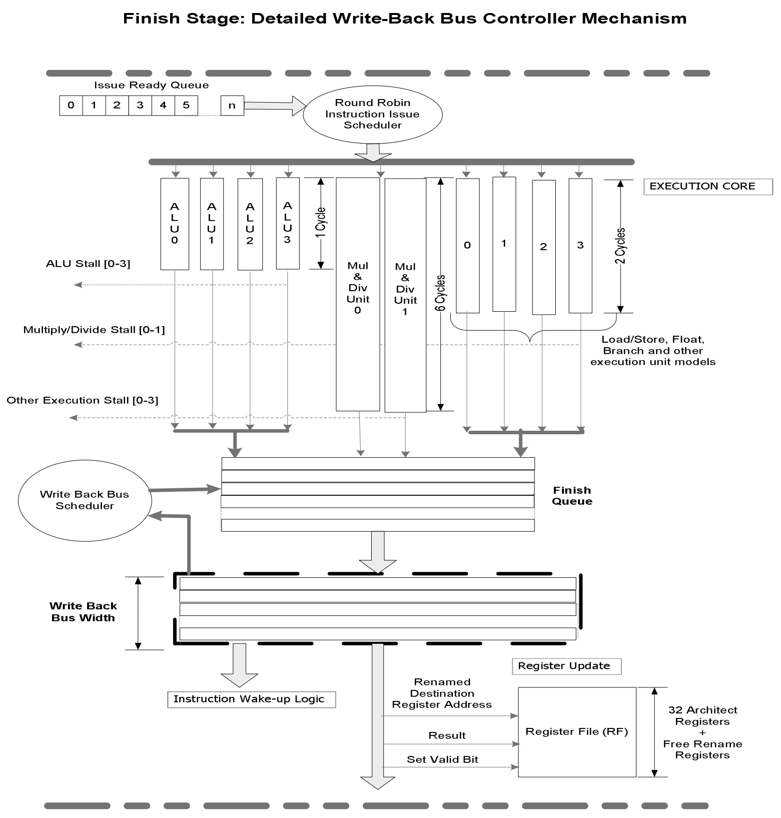

Finish Stage: Detailed Write-Back Bus Controller Mechanism
Issue Ready Queue
0 1 2 3 4 5 n
Round Robin Instruction Issue Scheduler
EXECUTION CORE
ALU 0
ALU 1
ALU 2
ALU 3
1 Cycle
Mul & Div Unit 0
Mul & Div Unit 1
6 Cycles
0 1 2 3
2 Cycles
ALU Stall [0-3]
Multiply/Divide Stall [0-1]
Other Execution Stall [0-3]
Load/Store, Float, Branch and other execution unit models
Write Back Bus Scheduler
Finish Queue
Write Back Bus Width
Instruction Wake-up Logic
Register Update
Renamed Destination Register Address
Result
Set Valid Bit
Register File (RF)
32 Architect Registers + Free Rename Registers

Figure 5.1. Detailed Architecture Model describing the Write-Back Buses at Finish Stage

C. Finish Queue and Write-Back Bus:

As shown in Figure *5.1*, the finish queue is a part of the write-back bus scheduler implementation. It is not an extra storage space but only models the last stage of the execution units. The write-back bus scheduler inserts the finished instructions that are waiting for the write-back bus in the finish queue. The write-back bus width is parameterized to study the effects of IPC on varying the write-back width. As the size of the write-back bus increases, there is a proportional increase in the number of write-ports in the register file and the forwarding bus lines for instruction wake-up.

5.2.3. Distribution of Write-Back Bus Size:

The maximum possible IPC for hypothetical processor architecture, limited only by fetch width and data dependencies is an interesting starting point for the study of the effects of write-back bus width. All other structural hazards and control dependencies are ignored in order to focus the study on the write-back bus width. As mentioned in section 2, only those benchmarks which could be successfully compiled for Simplescalar MIPS IV instructions are used in this study. Moreover, the subset of SPEC CPU INT 2006/2000 benchmarks actually represents a balanced instruction mix (Phansalkar, 2007[5]).

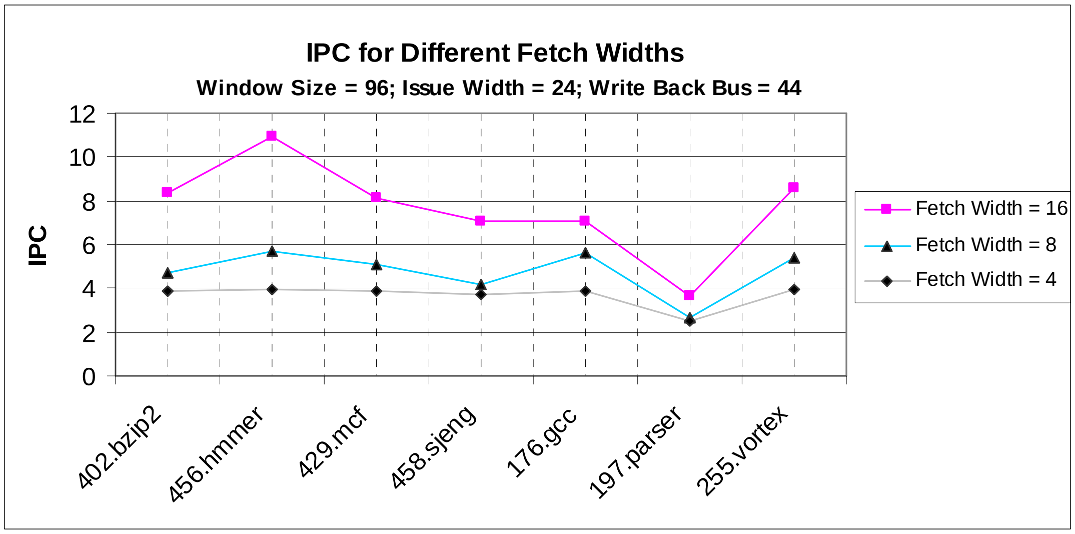


Figure 5.2. IPC of a hypothetical processor using SPEC CINT 2006/2000 Benchmarks

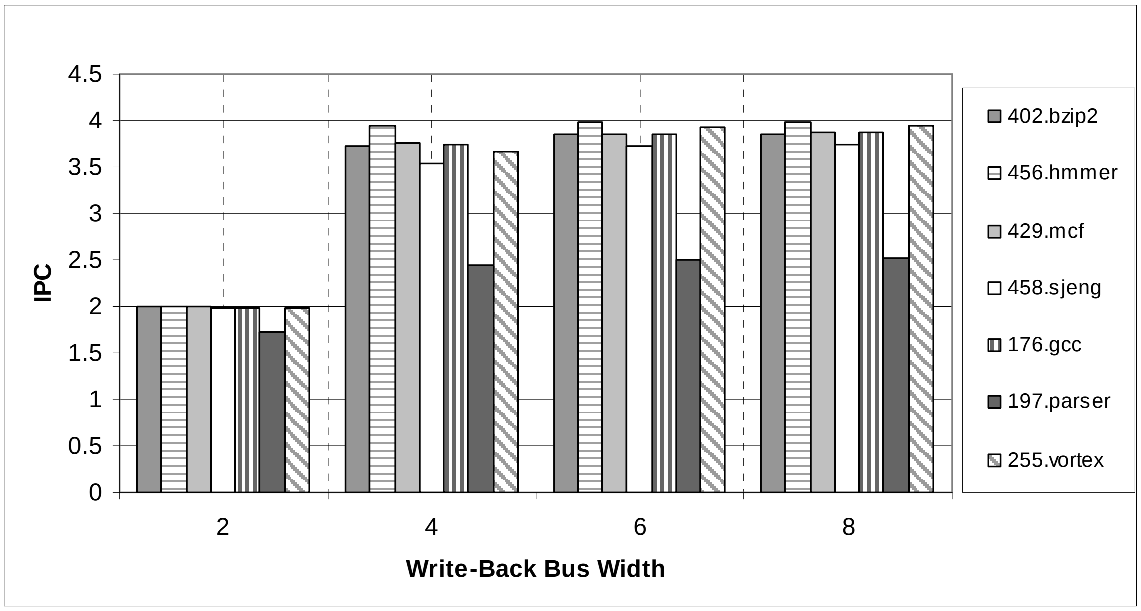


Figure 5.3. Comparison of IPC for Different Write-Back Bus Widths for fetch width of 4

Figure *5.2* shows that for a hypothetical processor that is only limited by a fetch width of 4 and data dependencies, an average IPC of 3.681 is achievable. With this as the base-line, the write-back bus width is varied to obtain the sensitivity of IPC to write-back bus widths. The sensitivity of the write-back bus width to IPC is shown Figures *5.3* and *5.4*. For small write-back widths, there is a linear relationship between IPC and the write-back bus widths.

As the write-back buses are a critical and expensive part of the design of the microprocessor, it becomes important to verify if any of the bus scheduling algorithms would result in a better IPC. As shown in Figure *5.4*, a Round-Robin write-back bus scheduling logic is used and its average IPC is measured. An important well known constraint is that the IPC of the microprocessor cannot be greater than the width of the number of the write-back bus.

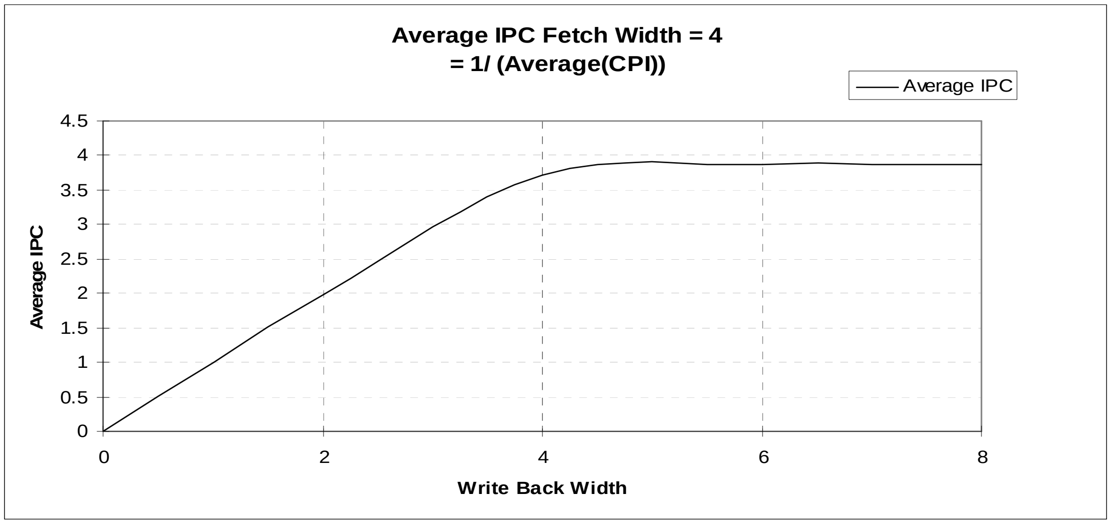


Figure 5.4. Average IPC for Fetch Width of 4

But, it is curious to find the type write-back scheduling algorithm that is chosen to maximize the IPC for a given number of write-back buses. As indicated in Figure *5.4*, since the write-back bus width of 3 falls in the linear range of IPC, we select this width to analyze the scheduling algorithms in sections 6 and 7.

5.3 Write-Back Scheduling Logic

In this section, the various write-back scheduling algorithms that are tested in simulations are discussed.

5.3.1. First-In First-Out (FIFO):

a) Strategy:

First-In First-Out (FIFO) logic is most common queuing model in memory systems. It is simple to implement as it naturally follows the pipeline model of the architecture design. At the last stage of the execution pipeline, the FIFO scheduler schedules the instructions to the write-back bus depending on the order in which the instructions finish. At the finish stage, the scheduler keeps track of execution units that are ready to finish. High priority is given to those instructions that finished in the previous cycles and are waiting for the write-back bus than is given to instructions that finish in the current cycle. An execution unit stalls in a given cycle, if it has an instruction that is ready to write-back its results but is unable to access the write-back bus.

b) Benefits:

The implementation of the FIFO scheduler is simple and requires less hardware. It removes long waiting times for accessing the write-back bus and hence keeps the execution units from stalling the pipeline.

c) Pitfalls:

The FIFO scheduler is likely to give priority to the execution units that have less latency than other execution units, since they are more likely to finish first in the execution core. Hence, ALU type of instructions is given more priority than other categories of instructions.

5.3.2. Round-Robin (RR):

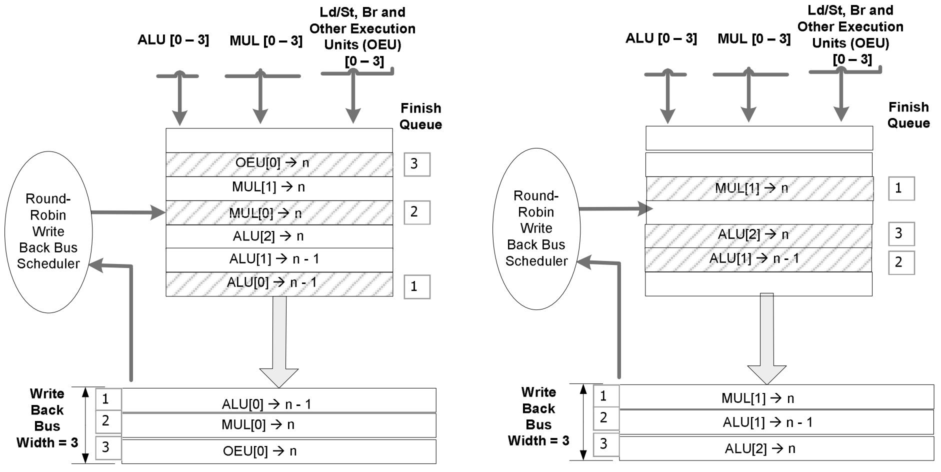
a) Strategy:

Round-Robin (RR) scheduling logic is an unbiased bus scheduling logic as it gives equal priorities to all the execution groups. The instructions scheduled to the write-back bus alternate between the ALU, MUL/DIV and LD/ST/BR execution pipelines.

b) Example:

Figure *5.5a* shows the write-back bus state at the nth cycle. The RR scheduler starts by giving priority to the ALU, MUL and OEU instruction type (shaded boxes). As shown in Figure *5.5b*, in the (n+1)th cycle the scheduler starts by giving priority to the MUL, OEU and ALU instruction type.

c) Benefits:



a. Bus State at nth cycle b. Bus State at (n+1)th cycle

Figure 5.5 Round-Robin Write-Back Bus Scheduler

Any dominance by a particular type of instruction that would normally result in instruction window stalls due to data dependence is reduced since the priorities are normally distributed.

d) Pitfalls:

Figure *5.6* shows the dynamic instruction mix in SPEC 2006/2000 benchmarks. Designing a bus scheduler that allocates the instructions to the bus with equal priorities may not always yield the best

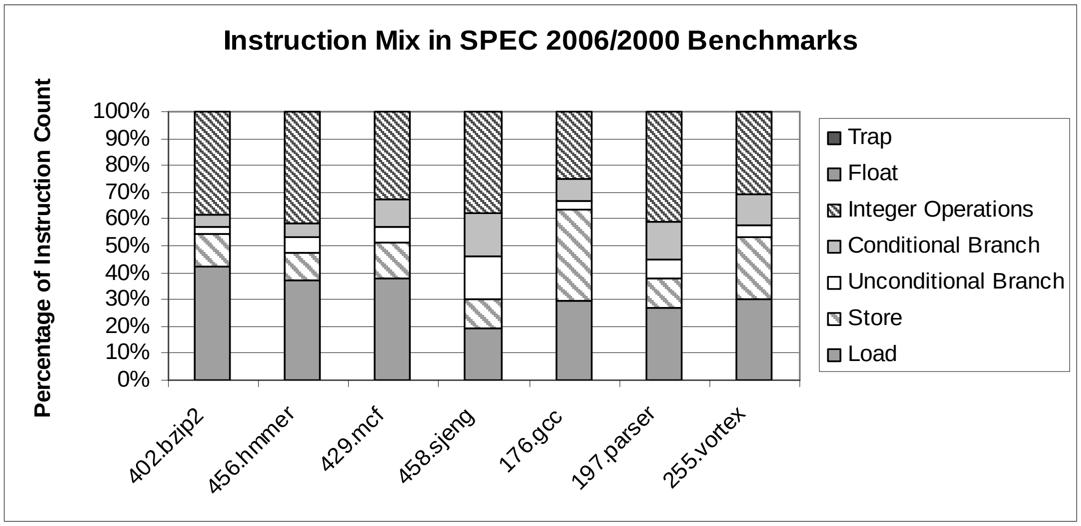


Figure 5.6 Instruction Mix in SPEC 2006/2000 Benchmark

results, considering the variations in instruction frequencies and the dynamic behavior of the program during run-time.

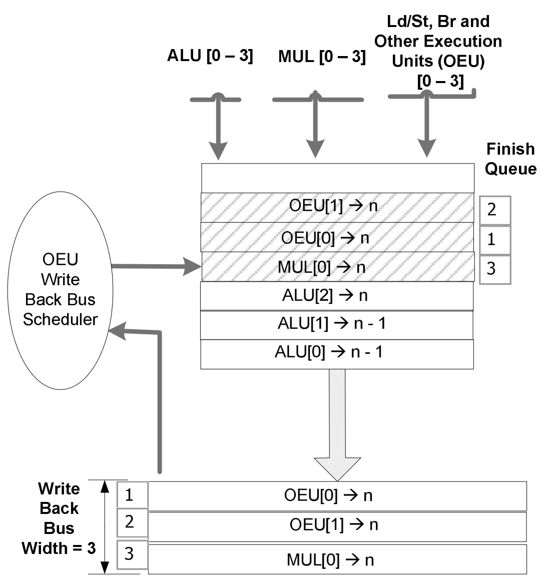
5.3.3. Priority to Load/Store, Multiply/Divide and ALU instructions (LMA):

a) Strategy:

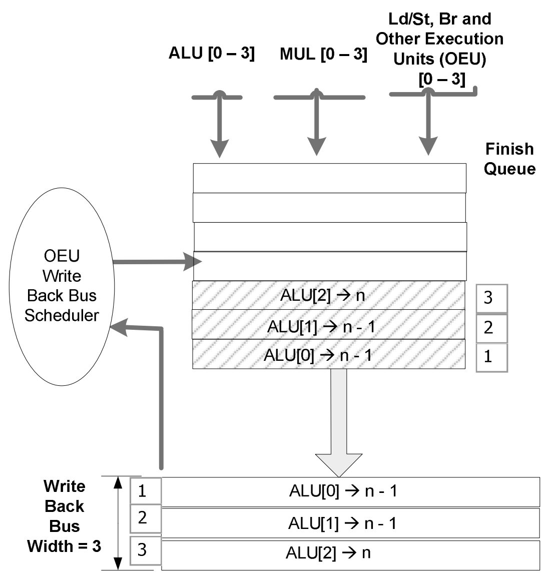
This strategy is designed to exploit the high frequency of Ld/Br/float instructions in the SPEC 2006/2000 benchmarks when compared to integer type instructions as seen in Figure *5.7*. Hence, in this strategy order of priority is given by OEU (Ld/Br/float) instructions followed by MUL/DIV instructions and ALU instructions – LMA priority.

b) Example:

As shown in Figure *5.7a*, in the nth cycle the OEU type of instructions gets access to the write-back bus and is followed by the MUL/DIV instruction. The ALU execution pipelines get stalls, until they get access to the write-back bus in the (n+1)th cycle.



a. Bus State at nth cycle



b. Bus State at (n+1)th cycle

Figure 5.8 LMA Write-Back Bus Scheduler

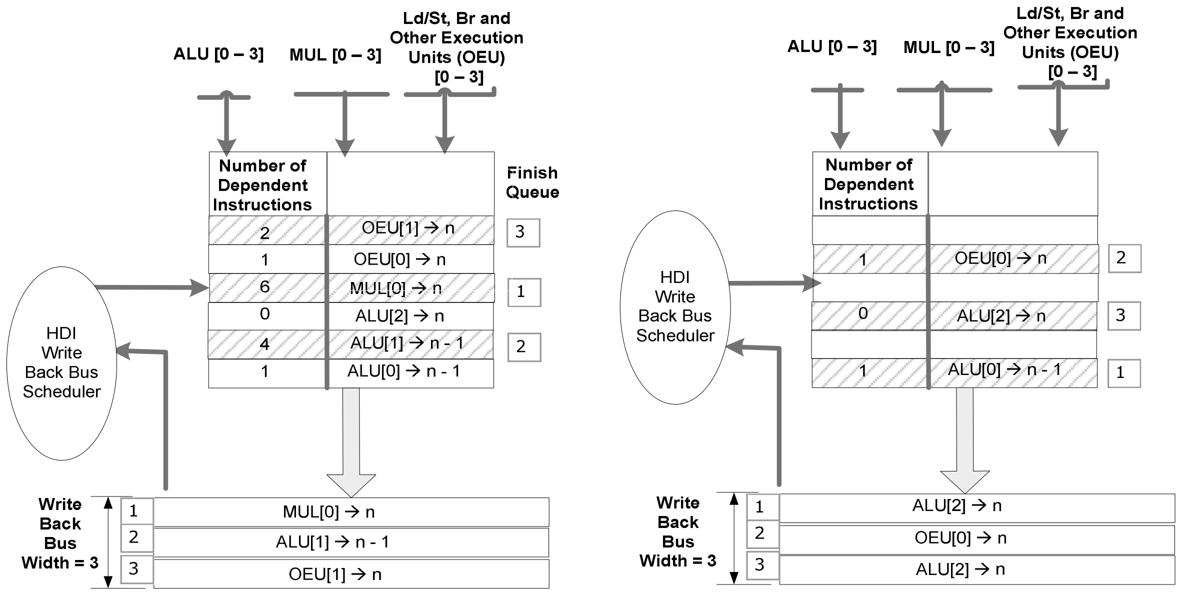
c) Benefits:

It is likely that the Ld/Br/float instructions that use OEU execution pipelines stall often due to their high instruction frequency as indicated in Figure 5.7. Hence, providing a high priority to this group of instructions reduces the number of stalls in the OEU execution pipelines.

d) Pitfalls:

Providing high priorities to only load instructions causes the ALU execution pipelines to be starved for access to the write-back bus. This results in the instruction window stalling the dispatch and fetch logic until the ALU execution pipelines get access to the write-back bus.

5.3.4. *Priority to the instruction that has Highly Dependent Instructions (PHD):*



a. Bus State at nth cycle b. Bus State at (n+1)th cycle

Figure 5.9 PHD Write-Back Bus Scheduler

a) Strategy:

In the Priority to Highly Dependent instruction (PHD) scheduler logic, the scheduler checks the wake-up table to determine the number of instructions that depend on the instruction that is ready for write-back. High priority is provided to the instruction that has high dependency on it.

Table *5.1* shows the number of times exactly 2 instructions, 3 instructions and more than 3 instructions are woken-up in the processor using RR write-back scheduler. Since, on average 4.6% instructions out of 6 billion instructions are dependent on 2 or more instructions, high priority is given to those instructions that have 2 or more instructions depending on them.

TABLE 5.1 INSTRUCTION WAKE-UP FREQUENCY USING ROUND ROBIN WRITE-BACK SCHEDULER

Benchmarks	Instruction Wake-Up Frequency			Total Instruction Wake-Up Frequency
	2 Instructions	3 Instructions	More than 3 instructions	
402.bzip2	6.58E+07	1.27E+07	1.98E+07	9.83E+07
456.hmmer	3.39E+07	4.66E+05	2.31E+05	3.46E+07
429.mcf	9.61E+07	2.21E+07	5.65E+07	1.75E+08
458.sjeng	1.72E+08	3.85E+07	4.21E+07	2.52E+08
176.gcc	1.69E+08	8.35E+07	9.58E+07	3.49E+08
197.parser	2.93E+08	2.09E+08	8.48E+07	5.86E+08
255.vortex	3.45E+08	5.22E+07	5.74E+07	4.55E+08

b) Example:

Figure *5.8* shows the example of a PHD scheduler. In Figure *5.8a*, the instructions that have high dependency win the write-back bus in the nth cycle and then in the (n+1)th cycle the other instructions get the bus allocation.

c) Benefits:

As data dependency is the main problem causing instruction window stalls, the HDI scheduler reduces these stalls by providing accesses to the instructions that have dependent instructions waiting on them. Hence, this scheduler is designed to issue a group of data independent instructions that are just waiting on one instruction to finish.

d) Pitfalls:

The PHD scheduler cannot determine the dependency length of a chain of instructions that are waiting on one another. Figure *5.9* highlights the problem of chain data dependency. The PHD scheduler fails to allocate that highest priority to instruction in slot 1, since a chain of instructions in the instruction window all have a dependency length of 1 in their wake-up table.

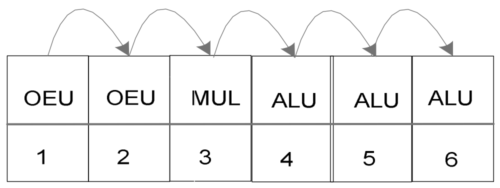


Figure 5.9 Chain of Data Dependency in an Instruction Window

5.3.5. Priority to Program Order Instructions (PO):

a) Strategy:

In this strategy, high priority is given to the instruction that is dispatched first i.e. in the program order. It is likely that later instructions in the program code are dependent on the instructions that are issued earlier. It encompasses the characteristics of order dependent FIFO and data dependent PHD schedulers to allocate priority to access the write-back bus.

b) Benefits:

The problem of chain data dependency as shown in Figure *5.9* is solved by simply scheduling the instruction to the write-back bus in the program order. This is an effective algorithm that allays the problem of long waiting times in the instruction window.

c) Pitfalls:

As there is no check on the number of instructions on which the scheduled instruction is dependent, there can be instances where the instruction scheduled to write back had no dependent instruction on it. Moreover, the hardware to implement the PO scheduler is expensive, requiring many comparators and ALUs in order to select the instruction in the program order at the finish stage.

5. 4. Simulation Methodology and Implication of scheduler mechanisms

A write-back width of 3 is selected to compare differences between the write-back strategies that are discussed in section 5.3. The benchmarks are selected from SPEC CPU CINT 2006/2000 suite and are complied with Simplescalar's *sslittle-na-sstrix-gcc* compiler. The other benchmarks in the SPEC CINT benchmark suite have compilation problems and hence are eliminated. However, based on the SPEC suite similarity analysis Phansalkar, 2007 [28], 402.bzip2, 456.hmmer and 429.mcf are determined to be dissimilar and hence unique in program characteristics. All the benchmarks are supplied with reference data input sets and are run until 6 billion instructions are executed. Table *5.2* provides the details of the simulated microarchitecture design.

TABLE 5.2 MICROARCHITECTURE DETAILS OF THE SIMULATED PROCESSOR

Design Parameters	**OSU AbaKus**
Instruction Fetch Width	4 inst/cycle
Instruction Window Size	96
Physical Registers	100
Issue Width	14

Commit Width	8
Branch Predictor	Perfect
Integer ALU units (Latency =1)	3
Mul/Div Unit (Latency = 6)	1
Ld/St/Float/Br Unit (Latency = 2)	4
Write Back Bus Width	3
Exceptions	Precise
Memory Latency	1
Number of Executed Instructions	6 Billion

As shown in Figure *5.4*, a write-back bus width of 3 is chosen since it is in the linear range of IPC. All control dependencies are eliminated by considering a perfect branch prediction in the simulation. Memory latencies are kept at 1cycle. These assumptions are made to focus the study on the effects of the write-back bus width on IPC. As a width of 3 is the bottleneck of the architecture, the IPC can not be higher than 3. Figure *5.10* shows that performance of the simulated architecture with various write-back scheduling algorithms. The implications of each scheduler are discussed below.

5.4.1. Round-Robin Schedule (RR):

Since the Round-Robin (RR) write-back bus scheduler give all the execution units equal priority, it provides a base-line scheduler for effective comparison with other bus schedulers.

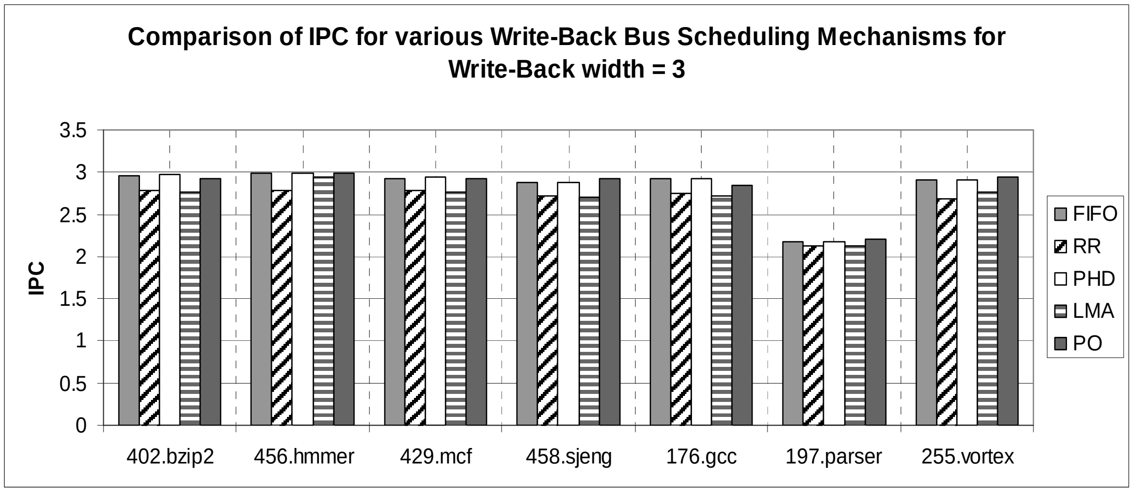


Figure 5.10 Comparison of IPC over various Write-Back Bus Scheduling Mechanisms

TABLE 5.3 IMPROVEMENT IN IPC FROM ROUND-ROBIN BUS SCHEDULER

Improvement = (IPC - IPC$_{RoundRobin}$) / (IPC$_{RoundRobin}$)	**FIFO (%)**	**PHD (%)**	**LMA (%)**	**PO (%)**
402.bzip2	6.145	6.681	-0.398	5.088
456.hmmer	7.529	7.549	5.628	7.582
429.mcf	5.481	5.984	-0.288	5.400
458.sjeng	5.973	6.300	-0.305	7.735
176.gcc	6.209	6.556	-1.404	3.305
197.parser	2.093	2.437	-0.124	3.884
255.vortex	7.960	8.124	3.029	9.475
Average Improvement in IPC (%)	5.913	6.233	0.881	6.067

5.4.2. FIFO Write-Back Bus Scheduler:

As indicated in Table *5.3*, the FIFO bus scheduler is superior to the Round-Robin (RR) bus scheduler with an improvement of approximately *6%*. This increase can be attributed to the priority that the FIFO scheduler gives the finished instructions in execution order. Hence, there is less waiting time for an instruction that is waiting for the write-back bus. On the other hand, RR scheduler may result in a condition where an instruction that finishes in the nth cycle waits for the bus, while the instructions that finishes in (n+1)th cycle gets access to the write-back bus. This leads to a stall in the instruction window and fetch stages.

5.4.3. Priority to High Dependence (PHD) Write-Back Bus Scheduler:

The Priority to High Dependence (PHD) scheduler logic also performs well as it schedules an instruction to the write-back bus that has a high instruction dependency. Hence, more instructions are issued as their data dependencies are resolved with priority. This effect can be seen in Figure *5.11*, where there are more write-back stalls in the PHD scheduler than the RR scheduler. This implies that due to the increase in issue rate more instructions can finish than the RR scheduler and are waiting for the write-back bus.

Conversely, as observed in Figure *5.10*, the increase in write-back bus stalls does not decrease the IPC of the PHD scheduler. This is because as seen in Figure *5.10*, the average instruction window stalls that stall instruction dispatch is lower for the PHD scheduler than the RR scheduler. As the instruction window size is 96 instructions, the issue logic is able to issue more instructions while the data dependencies are quickly resolved using the PHD bus scheduler.

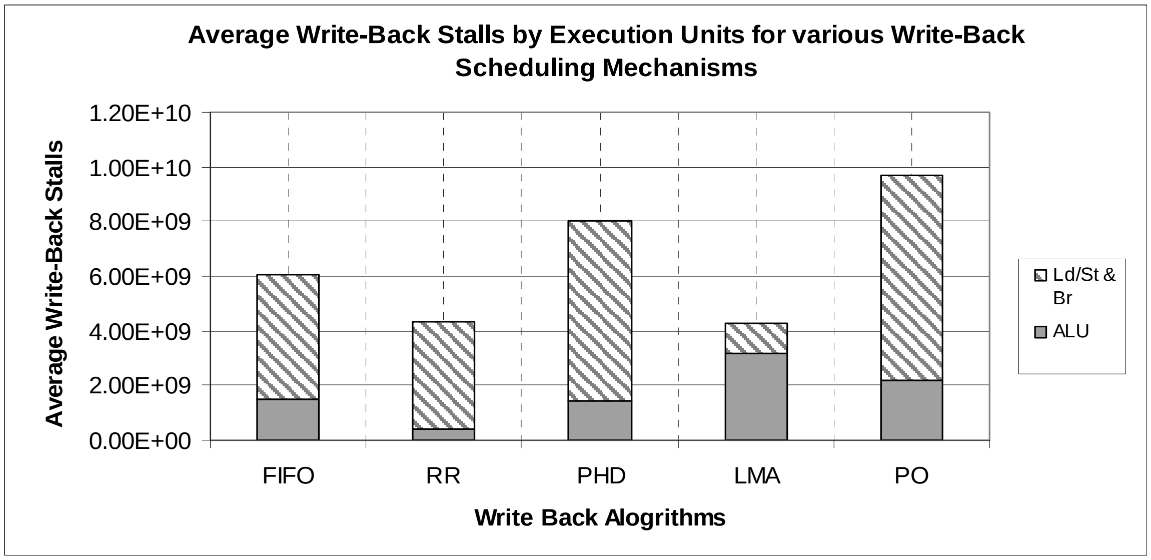


Figure 5.11 Average Write-Back Stalls by Execution Units for various Write-Back Bus Schedulers.

5.4.4. Load-Multiply-ALU (LMA) Write-Back Bus Scheduler:

As shown in Figure *5.11*, priority to load instructions reduces the write-back stalls that are caused by OEU pipelines. However, as shown in Figure *5.12* the average number of instruction window stalls is 14.8% more than the instruction window stalls of the PHD scheduler. As a result the IPC of the simulated architecture with LMA scheduler is ≈ 6% less than the FIFO, PHD and PO bus schedulers. The relatively low IPC by the LMA write-bus scheduler can be attributed to the 2.2 times increase in ALU execution pipeline stall as observed in Figure *5.11*. This is because resolving ALU instructions is critical to the mitigation of the instruction data dependencies. As the LMA gives low priority to ALU instructions, its IPC is less than the FIFO, PHD and PO bus schedulers.

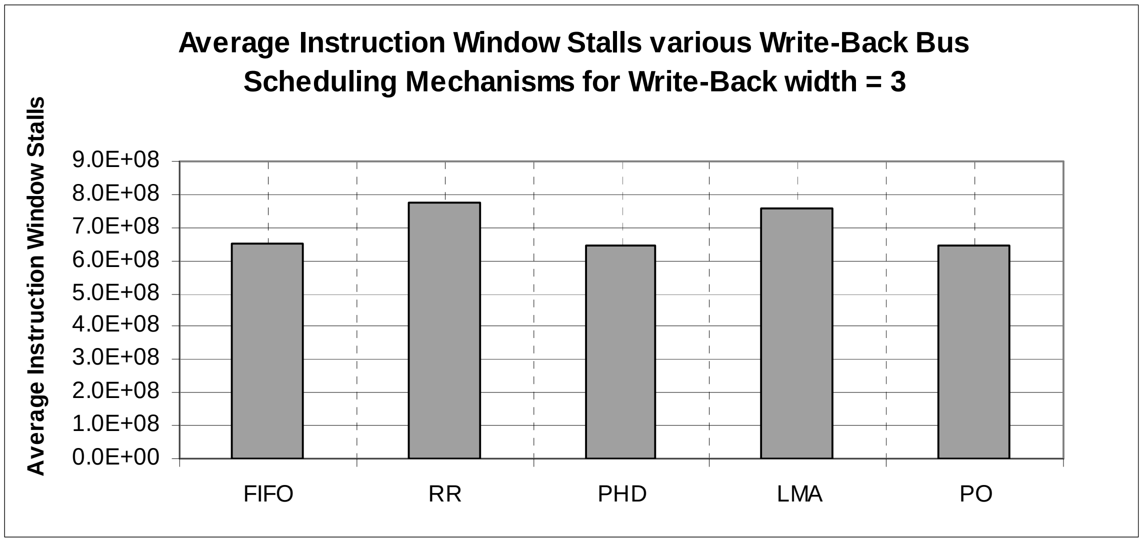


Figure 5.12 Average Instruction Window Stalls for various Write-Back Bus Scheduling Mechanisms

V. Program Order (PO) Write-Back Bus Scheduler:

As shown in Table *5.3*, the characteristics of the PO bus scheduler and PHD bus scheduler mechanisms are similar. Since the PO bus scheduler gives priority to the instruction in the program order, it is likely that later instructions are data dependent on this instruction. Hence, as seen in Figure *5.11*, the PO bus scheduler has more write-back stalls than any other scheduler logic. This indicates that more instructions have finished execution and are waiting for the write-back bus. On the other hand, Figure *5.12* shows less average instruction window stalls than the RR bus scheduler. This indicates that compared to the RR bus scheduler the issue rate is large and the completion time for an instruction is small.

5.5 Summary

The flexibility, simulation speed and closeness to hardware logic design that is emphasized in the design of the *AbaKus* microarchitecture simulator is demonstrated by analyzing various write-back bus strategies. As shown in Figure *5.1*, IPC can be limited by the write-back bus width of the architecture design and can be an important bottle-neck in achieving higher CPU performance, especially in SMT architectures. There is a need to develop an instruction issue policy that corresponds well with write-back bus scheduling policy to maximize the utilization of expensive read and write ports of a register file.

CHAPTER VI

CONTROL DEPENDENT LIMITATIONS ON INSTRUCTIONS PER CYCLE

“Prediction is very difficult, especially about the future”
- *Niels Bohr*

6.1 Program Dependencies

As human brain thinks and reasons out before it makes a decision, it is not clear if this logic flow is conducted in a sequential or parallel manner. However this may be, some degree of sequential and parallel process is involved before the brain arrives at a decision. This argument is necessary because it defines how humans use computer languages to model and describe their logic. Thereby, the nature of these program descriptions introduces dependencies before the logic is computed. These inherent program or instruction dependencies are classified into 2 types – *Data Dependencies* and *Control Dependencies*.

Instruction data dependencies exist in the program due to logic flow and it requires computation time to resolve these dependencies. They can be regarded as the last major bottle-neck of sequential programming model. In many ways, resolving them for a single-threaded process depends on the logic description and physical design limitations. However, solutions have been proposed to hide the latency of the data dependent instructions (IBM 2005 [29][20]) through multiple parallel threads.

On the other hand, *control dependencies* in a program can be related indirectly to *data dependencies*. Nevertheless, the control flows of the program seem to be predicted to a fair degree of accuracy (Nair, 1995 [31], McFarling, 1993 [32]) for machines with small instruction fetch. But, it introduces a limitation for wider instruction fetch machines and is harder to predict the control flow. This is because of lack of sophisticated hardware with small latency to recognize the pattern of the program behavior or in general, due to the innate behavior of the program.

6.1.1 Higher IPC with Superscalars

The goal of the superscalar architecture design is exploit available Instruction Level Parallelism (ILP) in the program code and hence, to achieve maximum IPC. But, to maximize the utilization of ILP, the control flow of the program has to be predicted with accuracy. Branch predictors using 2-bit saturating counters and a branch pattern history table are used to predict a branch instruction with a fair amount of accuracy using gshare branch predictor (McFarling, 1993 [32]).

Maximum possible IPC of a machine is equal to the number of instructions fetched per cycle, denoted by ‘s’- the fetch width, assuming the number of instructions dispatched, issued, finished and completed are all equal or greater than ‘s’. Hence, with the increase in fetch width the IPC is bound to increase. But, this is not found to be true. This is because as the fetch width increases the number of branches in the fetch group also increases. Since, the branch predictor now has to choose among multiple branched paths and predict the correct one. This problem worsens as the machine is super-pipelined and there are more unresolved pending branches due to increase in branch execution latencies.

Let a single branch misprediction error be *Pe* and *k* be the number of unresolved branches in the machine. Then, the probability that all the '*k*' branches are predicted correctly is given by [(1-*Pe*) ^ *k*].

That implies, the probability that at least one out of '*k*' branches is mispredicted is given by,

$$1 - [(1-Pe) \wedge k] \qquad \text{equ(6.1)}$$

TABLE 6.1 PROBABILITY OF MISPREDICTION

Number of Unresolved Branches	P[at least one branch is mispredicted]
0	0
1	0.1
2	0.19
3	0.27
4	0.34
5	0.41
6	0.47
7	0.52
8	0.57
9	0.61
10	0.65

As seen in Table 6.1, simply using branch prediction to predict the control flow is not reliable if there are 6 or more unresolved pending branches in the machine because the branch prediction error is close to 50% or more. In fact, 27% prediction error for 3 pending branches is high enough to deteriorate the IPC. Hence, a better solution other than to simply trust the branch predictor is required to have a high IPC.

Why not simply build multi-cores to solve this problem?

Building parallel core architectures results in speed-up provided there is enough code parallelism to extract in the program. ILP is much more at finer granularity than task or data parallelism that useful for multiple parallel core architectures. In addition, *Agerwala* and *Cocke* (IBM, 1987 [33]) showed that it requires at least *75%* of parallelism in programs for a parallel machine of *100* processors to equal the speed-up of a parallel machine with just *6* processors but with twice the speed-up in its sequential part of the program.

High parallelism is found in programs developed for numerical computations or gaming applications. But, only a few programs have such high degree of parallelism (> *75%*) and hence it is important to address the problem of control dependencies that is present in the non-parallelizable code to boost the performance of modern computing machines.

6.2 Multi-Path Execution Schemes

Streams of instructions from both paths are followed after a branch instruction until the branch gets executed. This strategy seems to be straightforward as there is no influence of branch prediction and most importantly the machine need not recover from the misprediction where many useful CPU cycles are lost. This is because following both the paths of the branch guarantees completion of one path when the branch is executed.

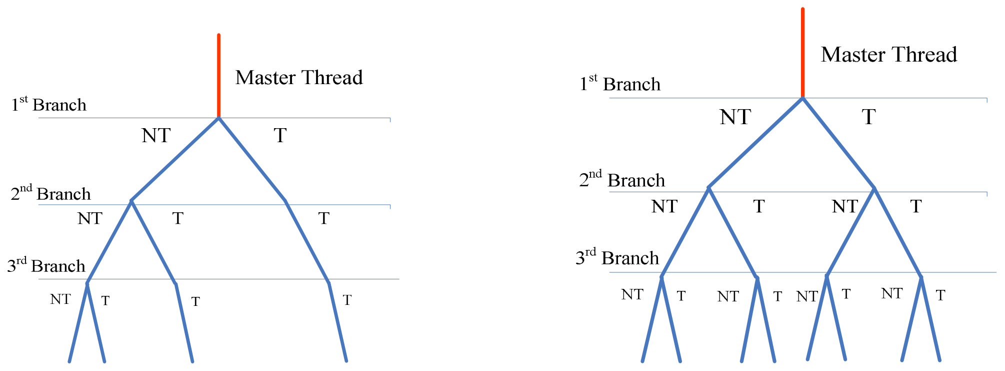
On the other hand, following both paths leads to splitting the machine resources among the paths where one is discarded. Furthermore, if the path has a branch instruction it forks 2 new paths and so on. This results in an increase in the number of paths of the order of (2^n), where ‘n’ is the number of unresolved branches. As each path after the branch instruction maintains its own sub-set of registers and pointers that are then updated at complete, they well fit into the definition to be called as *threads*.

Now, let’s conduct a simple analysis to understand the performance of the branch prediction and multi-path execution schemes. To keep the analysis simple, let’s assume 3 consecutive branches that are executed in parallel and hence all have the same latency at which it is resolved.

Let ‘s’ be the number of instructions fetched per cycle, ‘*Perror*’ be the probability that the 3rd unresolved branch is mispredicted, and ‘*R*’ be the number of recovery cycles, then the IPC of the machine with branch prediction is,

$$\text{IPC}_{\text{(bpred)}} = \frac{1}{\dfrac{1-P_{error}}{(s)} + \dfrac{P_{error}}{(s/R)}} \qquad \text{equ (6.2)}$$

Let’s consider the multi-path case,



a. Minimum Possible Threads (4 Threads) b. Maximum Possible Thread (8 Threads)

Figure 6.1 Multi-Path for 3 Unresolved Branches.

The IPC of the multi-path execution assuming the 3 branches are resolved in the same cycle simply is,

$$IPC_{(multi\text{-}path)} = s/2^m \qquad \text{equ(6.3)}$$

Now, considering 4 to 8 threads (Figure 6.1) the IPC for 8-wide machine (s = 8) is between 2 and 1. If the probability of error is predicting the 1st branch varies from 0.05 to 0.5, then using equ(6.2) and equ(6.3), the *Perror* for the 3rd branch and its corresponding IPC can be calculated as shown in Table [6.2].

TABLE 6.2 CALCULATED IPC USING EQU(6.2) FOR BRANCH PREDICTION

$Pe_{(1st\ Br)}$	$Pe_{(3rd\ Br)}$	$IPC_{(bpred)}$
0.05	0.14	4.347
0.1	0.27	3.053
0.15	0.39	2.395
0.2	0.49	2.030
0.25	0.58	1.785
0.3	0.66	1.612
0.35	0.73	1.486
0.4	0.78	1.408
0.45	0.83	1.337
0.5	0.88	1.273

Note that when the Pe(1st branch) is more than 0.25, the IPC with branch prediction is less than 2, where as in the case of multi-path execution the IPC with 4 threads is 2. On the other hand, the worst-case of multi-path execution with 8 threads the IPC is 1 and the IPC with branch prediction is little more than 1 for its worst-case.

From the above analysis it is not clear if the machine with branch prediction or multi-path execution is better, as it depends on various conditions of path executions in both the schemes. In addition, formulating all possible paths with large number of unresolved branches is a combinatorial problem. Hence, to obtain more deterministic estimate of the processor performance, execution-based simulations have to be conducted and later the results have to be analyzed to determine the machine that has a better average performance.

6.3 Single-Threaded Processor with Branch Prediction

This is the base-line architecture in the present day microprocessor cores. For branch prediction logic, branch target butter (BTB), 2-bit saturating counter and a shift register to maintain global history bits are used to predict the control flow of the single-threaded machine. The 2-bit saturating counters and the BTB are updated non-speculatively in the complete stage. This may result in extra cycles but recovering for a speculative mis-prediction is prevented. Studies have shown there is 1% improvement if branches are updated speculatively, which is insignificant compared to the logic and cost involved in speculative recoveries. The basic architecture of the branch prediction logic in the fetch stage is shown in Figure 6.2.

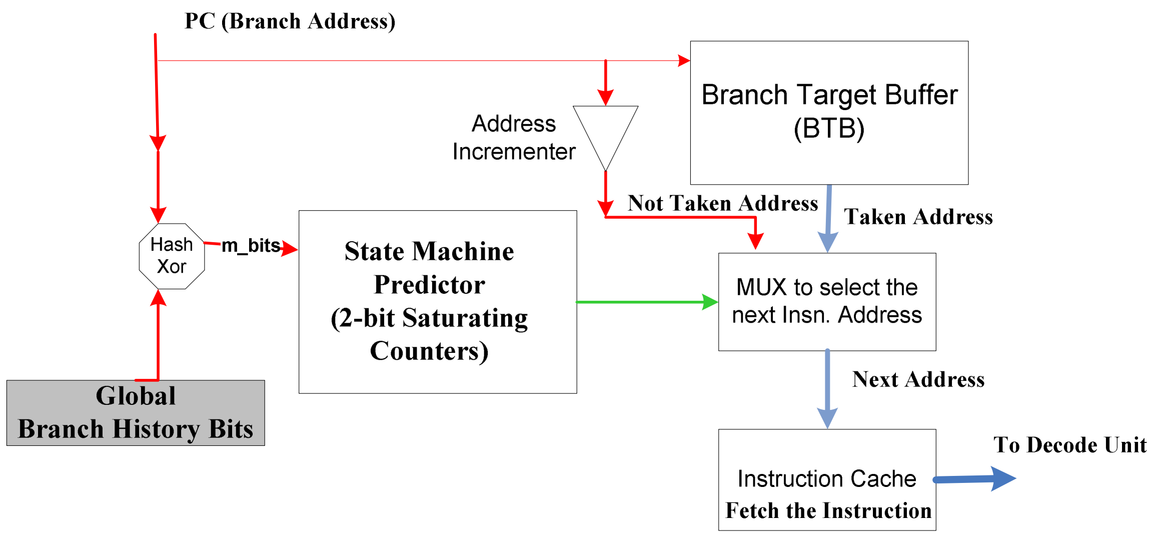


Figure 6.2 Logical Block Diagram of the Branch Prediction in Single-Threaded Processor

Evaluating Branch Prediction Mechanism

Gshare branch predictor is the most commonly used branch prediction because of it reasonably low branch prediction error rate and its simplicity (McFarling, 1993 [32]). It consists of a globally shared history bits (gshare) of a particular size in bits. These history bits are hashed with the branch instruction address to index a column of state predictors. Depending on the State Machine Predictor as shown in Figure 6.2, the next address after the branch instruction is predicted and the corresponding instruction is fetched from the instruction cache.

TABLE 6.3 PROBABILITY OF BRANCH PREDICTION ERROR FOR 3 BILLION COMPLETED INSTRUCTIONS

Benchmarks	> 0.3	0.3 > Pe < 0.7
176.gcc	0.624318	5.21E-01
402.bzip	0.214487	2.09E-01
456.hmmer	0.573799	0.5383107
429.mcf	0.306306	0.3005648
458.sjeng	0.560901	0.48985
255.vortex	0.469276	0.2176803
Average	**0.458181**	**0.3793101**

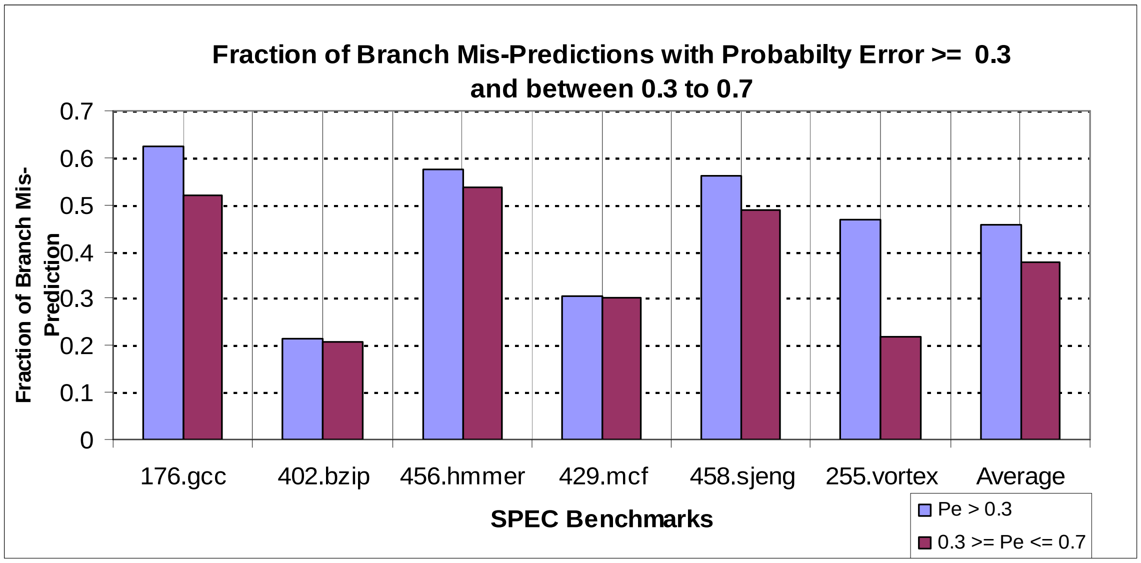


Figure 6.3 Fraction of Branch Misprediction in SPEC benchmarks

gshare: Size: 2048 entries; History Bits: 16; BTB: 512 sets with 4-way associative

Figure 6.3 shows the fraction of branch misprediction that have a probability of error more 0.3 as well as between 0.3 and 0.7. As seen from the plot about 45% of branches are mispredicted. If the predictions of those branches that have a probability of error greater than 0.7 are inverted, since there are wrongly correlated (Klauser, 2001, [34]). Even then there are still about 38% of the branches whose behavior patterns are not correlated with the branch predictor.

Classification of Branches and their Prediction

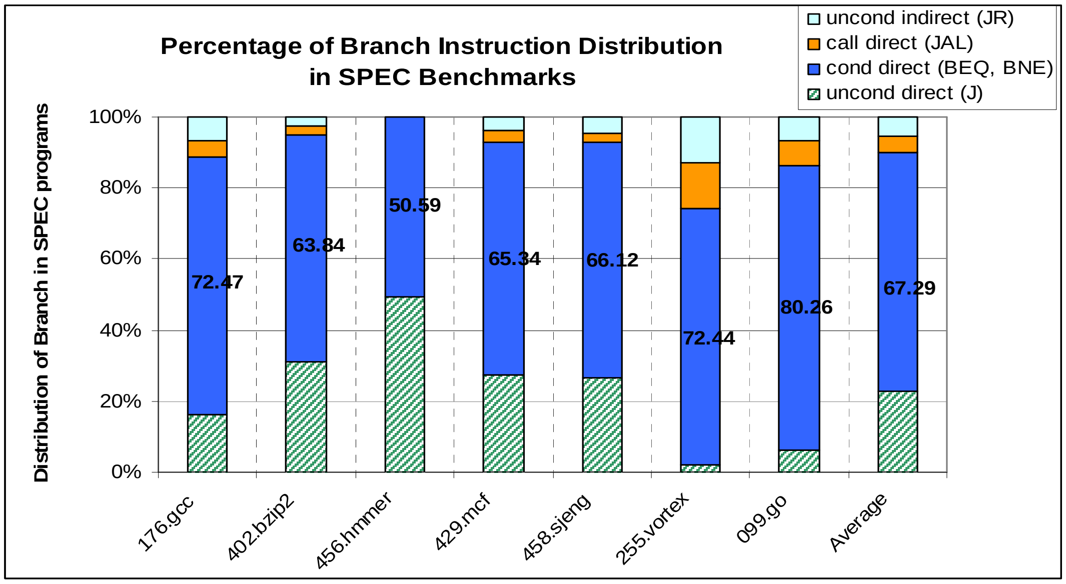


Figure 6.4 Classification of Branch Instructions in SPEC benchmarks

It is important to find the class of branches and to identify the area that needs improvement. In the benchmark programs that are tested, there exist four predominant classes of branch instructions – *Unconditional Jumps*, *Call Direct Jumps*, *Unconditional Indirect Jumps* and *Conditional Direct Jumps*.

Unconditional Jumps and Call Direct jumps are predicted using a BTB, they normally attribute to compulsory misses or aliasing. Although, compulsory misses cannot be avoided, aliasing can be taken care simply by increasing the buffer size. Unconditional Indirect jumps can be predicted using BTB but more advanced techniques using register address stack as in Intel's Nehalem architecture are also used.

From Figure *6.4*, about *67%* of branches fall under the category of conditional branches. Hence with about *38%* of total branches regarded as *hard-to-predict* branches there are approximately about *25.4%* of conditional branches that can be regarded as *hard-to-predict* conditional branches. To maximize the performance of single-threaded execution, it is vital to reduce the misprediction penalties that are incurred due to *hard-to-predict* branches.

To solve the problem of misprediction penalties in single-thread instruction stream, a scheme were multiple paths are followed and executed using *Simultaneous Multi-Threaded (SMT)* architecture designs is adapted. Although, branch prediction can be further improved with confidence estimators, data-value prediction and neural network algorithms, they normally result in diminishing results. Hence, SMT-based architecture design is chosen to solve the *hard-to-predict* conditional branch problem as well as to explore and improve some design techniques in the multi-threaded designs.

6.4 Related Work

Ahuja et al, 1998 [35] show average speedups of 14.4% for multipath architecture with confidence predictor on SPECint95 benchmarks compared to a single path machine. The paper demonstrates that the instruction fetch bandwidth is very important and extra resources to fetch correct execution path can improve performance. However, the study does not indicate how the fetch resources must be allocated and how the confidence values can be used to control the fetch allocation.

JRS confidence estimator by Jacobsen, Rotenberg and Smith, 1996 [36] introduce the concept of confidence estimators. The confidence predictor is implemented similar to a branch predictor. They test the performance of confidence estimator with ones counter (shift registers), saturating and resetting counter. The paper shows that resetting counter tracks ideal curve of misprediction due to dynamic branches closely than other counter methods. Selective Branch Inversion (SBI) is proposed by Klausaur et al., 2001,[34]. An up-down counter is used in the confidence estimator with 0 marked as low confidence and 1 to 3 as high confidence. A relative improvement of 9% reduction in branch misprediction is noted when compared with the McFarling predictor. However, performance improvement in terms of IPC is not indicated in the paper. As an alternative to the SBI scheme, Aragon et al, 2001 [37] use data value prediction and reverse a branch through the Branch Prediction Reversal Unit (BPRU). Over 6% improvement is shown over the SBI scheme in terms of IPC. Manne et al 199 [38] also introduces various useful confidence evaluation metrics such as PVN and Specificity.

Uht et al., 1995 [39] propose a variation in eager execution schemes called the Dis-Joint Eager Execution (DEE). It uses the cumulative path probabilities to determine the highest likelihood path to follow. The differences between single path, eager and disjoint eager execution are illustrated in Figure 6.5.

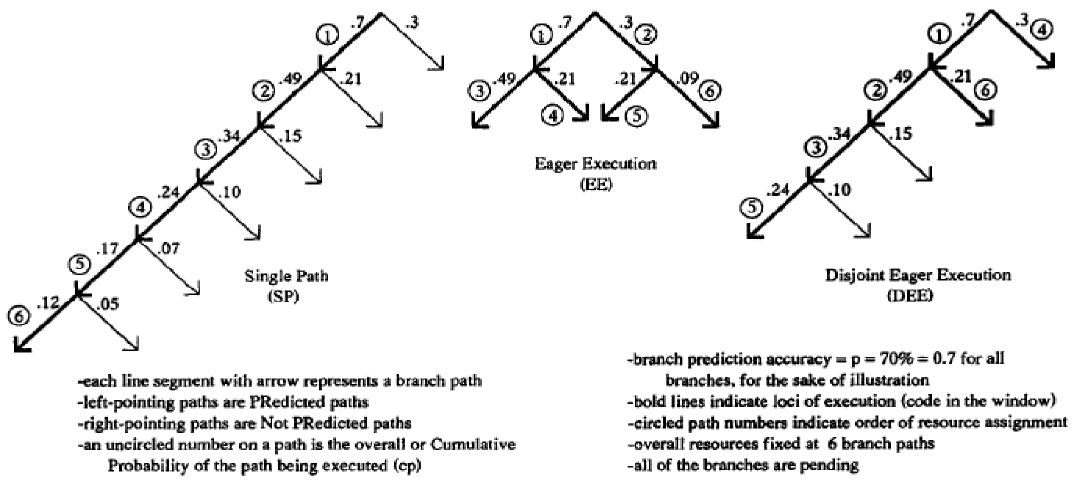

Single Path (SP)
Eager Execution (EE)
Disjoint Eager Execution (DEE)
-each line segment with arrow represents a branch path
-left-pointing paths are PRedicted paths
-right-pointing paths are Not PRedicted paths
-an uncircled number on a path is the overall or Cumulative Probability of the path being executed (cp)
-branch prediction accuracy = p = 70% = 0.7 for all branches, for the sake of illustration
-bold lines indicate loci of execution (code in the window)
-circled path numbers indicate order of resource assignment
-overall resources fixed at 6 branch paths
-all of the branches are pending

Figure 6.5 Comparisons of Execution Strategies (Source: Uht and Sindagi, 1995 [39])

A mean speedup of 4% over single path execution if more than 256 possible paths are followed is recorded. However, the implementation of DEE is simplified by only considering the static branch prediction probabilities and does not consider the dynamic probabilities for each individual branch. As branches in an instruction stream have varying misprediction rates, it is interesting to look into the dynamic prediction probabilities. In addition, the paper also does not propose any realistic hardware design to implement DEE.

Malik et al., 2008 [40] propose a new probability based path confidence predictor and compare them with the standard threshold-and-count predictor. Basically, the threshold-and-counter confidence estimators suffer from a case where low confidence branches are assumed to be mispredicted at the same rate. The probability based predictor calculates the cumulative correct prediction probability in an encoded form. It uses simplified multipliers (log-based circuit) and keeps track of both correct as well incorrect predictions of a branch. It is used in the SMT prioritization of threads and shown to be

5.4% better than the standard JRS confidence estimators. Such predictors can be used on confidence-based eager executions and have to be tested.

Dual Path Instruction Processing is proposed by Aragon et al, 2001 [37] using Branch Prediction Reversal Unit (BPRU). This architecture targets to reduce the pipeline-fill penalty after a misprediction. Through the BPRU if the alternative branch path has low confidence then the instructions from the path are fetched, decoded and renamed but not executed while the other predicted path is executed. If a misprediction occurs then the decoded instructions are allowed to refill the buffer thus reducing the pipeline-fill penalty. An 8% improvement is noted over single path with *gshare* predictors. However, fetching from alternative streams reduces the fetch bandwidth and more than 2 branch paths have to be followed as shown in DEE.

Wallace et al., 1998 [40] propose a method to use the 2-way SMT for multipath execution. They use a fetch policy called the ICOUNT, where the fetch logic gives priority to those threads that have fewest instructions between fetch and issue. The architecture check points at the blocks of branches. Depending on the priority based on confidence values and resource availability the check points are followed until the branch resolves. A 14 % increase in this modified SMT over the baseline architecture is seen.

Selective Dual Path with various fetch polices using confidence values is studied by Heil and Smith, 1997 [42]. The fetch policies are,

Canceled Policy: Ignore subsequent low confidence branches if the earlier branch is followed.

First Delayed Policy: Save the processor state when the 2nd low confidence branch is encountered and follow it when the 1st branch is resolved.

Last Delayed Policy: The processor state of the latest low confidence branch is saved and followed when the 1st branch is resolved. The paper shows that the mispredicted branches occur in clusters. The fetch policies did not provide much improvement and the paper concludes to investigate on machines that can fork multiple branch paths.

Perceptron based branch confidence estimation is discussed by Akkary et al, 2004 [43]. Figure 6.6 shows the block diagram of the perceptron confidence estimator. The delays in calculating (summers and multipliers) and weights training are significant. A 7 % decrease on average in executing wrong instructions is shown using pipeline gating and branch reversal strategies.

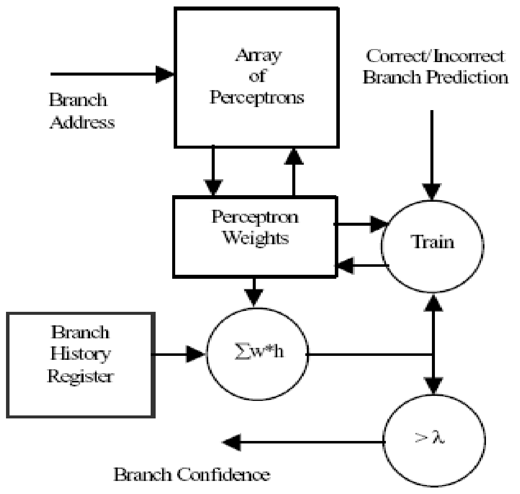


Figure 6.6 Perceptron based branch confidence estimation by Akkary et el. [2004]

Address-Branch correlation for long-latency hard-to-predict branches is investigated by Gao et al, 2008 [44]. It relies on hard-to-predict branches that depend on the address of the memory location rather than its value. Certain memory-intensive benchmarks exhibit this behavior. The concept involves identifying hard-to-predict branches and is based on number of branch penalties. Once the branch is identified then its producer load instruction is tracked. Using the load and branch address relationship the target address of the branch is then predicted. There is less than 10% reduction in misprediction on average and the actual impact on the IPC is not discussed.

6.5 Discussion

The multi-path design using some form confidence estimators has been proposed. Klauser et al., 1998 [45] discuss about Selective Eager Execution using confidence estimator and achieve an average improvement of 14% in IPC for SPECint95 benchmarks. However, schemes such as the DEE (Uht and Sindagi, 1995 [39]) have never been tested even through architecture simulations using dynamic confidence estimators. In addition, dynamic confidence estimators are shown to have problems and the performance of the multi-path design relies to an extent on the performance of the confidence estimators. The performance improvement varies from 4% to 14% in most of the architecture designs that tried to improve the single-threaded program execution. Eager execution techniques like DEE and confidence-based fetch are explored in this dissertation. In the next sections, the important design aspects of the SMT architecture of this dissertation is explained in detail.

6.6 Multi-Threaded Fetch Logic Design

In the case of the multiple threads, a multi-ported BTB and instruction cache are necessary to determine multiple target addresses and to fetch from them. As shown in Figure 6.7, the BTB can now be considered as a Thread Management Buffer due to its increased number of fields. Although, the logical block diagram looks simple, the increase in number of read/write ports increases the cost of the design. For simplicity, the block diagram in Figure 6.7 only shows fetching from 2 threads after a branch.

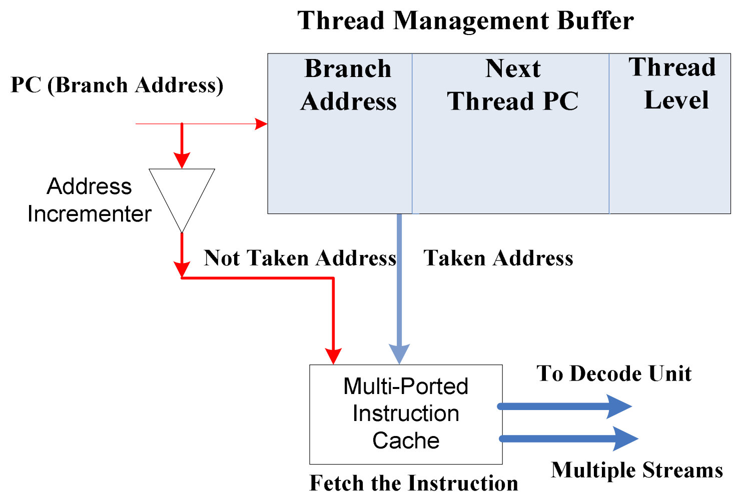


Figure 6.7 Logical Block Diagram of Fetch-Stage in Multi-Threaded Processor.

The challenge in fetching from multiple paths is to make sure the instructions from these streams can be distinguished at any point inside the processor. This is could be done in 2 ways. Structurally the entire processor can be divided for each of these streams or each instruction can be tagged with a path or thread identification tag – Thread ID – to distinguish between various paths. Structurally dividing the entire processor may enforce strict limitation of number of threads and also that these resources can be shared. Hence, to improve resource utilization the hardware functional units and registers must be shared among these paths. Therefore, a unique scheme where the branch history bit is used for Thread IDs is proposed by Chen, 1998 [46]. Through this scheme the taken path is set as 1 and the not taken path is set as 0. Hence, if the instruction path is taken, taken and not taken. The Thread ID would be 110. This scheme not only makes it easier to distinguish between paths but also to find the heritage of the instruction. Determining its heritage or the path's ancestor paths enables to find the correct rename register pointers which is discussed in the following section.

6.7 Register Renaming for Multiple Paths

Register renaming in a single-threaded processor is explained in chapter 4. Although, the mechanism is the same for multi-path architecture, one major difference in this architecture is that the renaming can happen at any level of the forking path. Hence, the challenge is to find the correct ancestor path and also to reference the correct rename pointer. Let's look at the procedure to find the correct ancestor thread ids through an example.

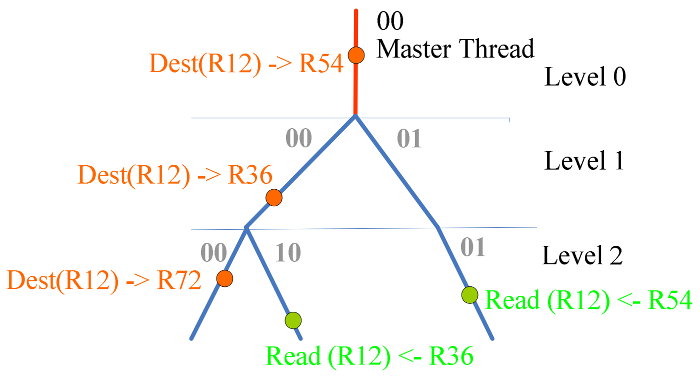
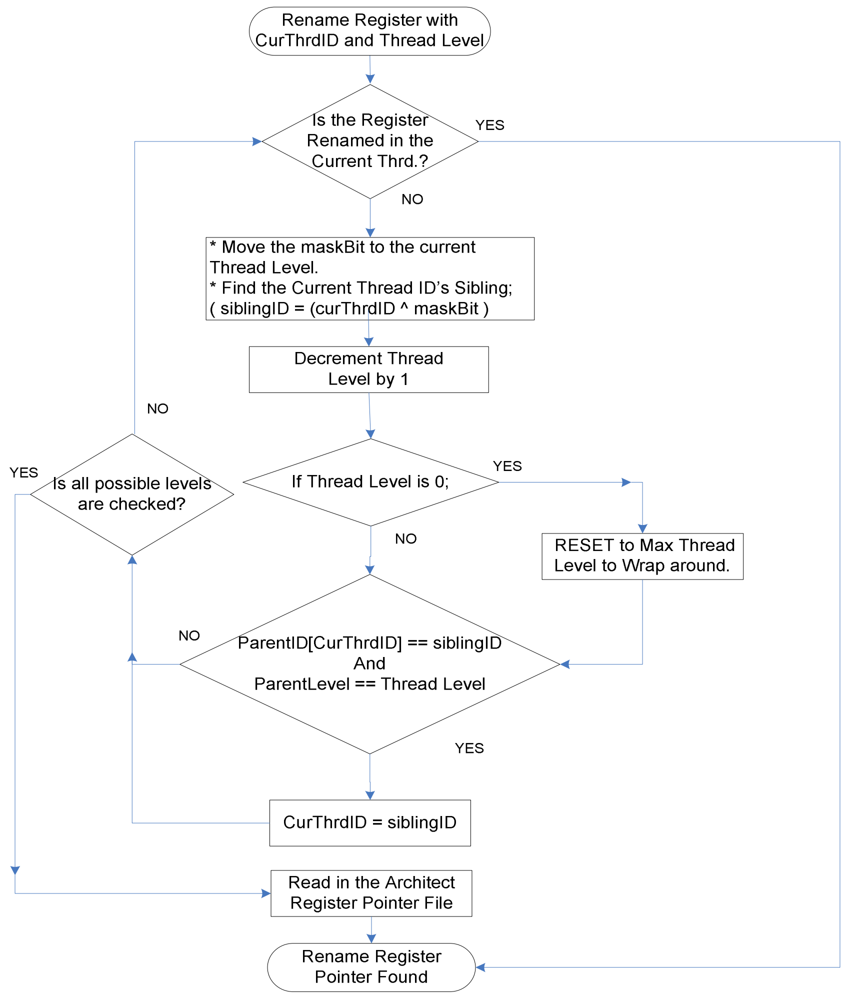


Figure 6.8 Example of Register Renaming in Multi-Path Design

In the example shown in Figure 6.8, register 12 gets renamed once at the master thread as well as twice in Thread ID 00 but at different branch levels. In thread paths 10 and 01, register 12 is being read and the correct register pointers are indicated by arrow symbols in the Figure 6.8. The explanation of how register 12 references correctly to its renamed pointers is given in the Flow Chart 6.1,


Rename Register with CurThrdID and Thread Level
Is the Register Renamed in the Current Thrd.?
YES
NO
* Move the maskBit to the current Thread Level.
* Find the Current Thread ID's Sibling; (siblingID = (curThrdID ^ maskBit)
Decrement Thread Level by 1
NO
YES
Is all possible levels are checked?
If Thread Level is 0;
YES
NO
RESET to Max Thread Level to Wrap around.
NO
ParentID[CurThrdID] == siblingID And ParentLevel == Thread Level
YES
CurThrdID = siblingID
Read in the Architect Register Pointer File
Rename Register Pointer Found

Flow Chart 6.1 Thread Rename Pointer Logic

Rename register logic is one major module that different from that of single-path architecture design. The rest of the units in the pipeline in the multi-path architecture design are similar to single-path.

However, to reduce the number of thread paths that are followed, the thread paths are invalidated at dispatch and complete stages as soon as the branch get executed and its actual path is determined. The reason to keep the number of thread path low in a multi-path scheme is because the more the number of thread paths that are followed the less is the fetch width per thread.

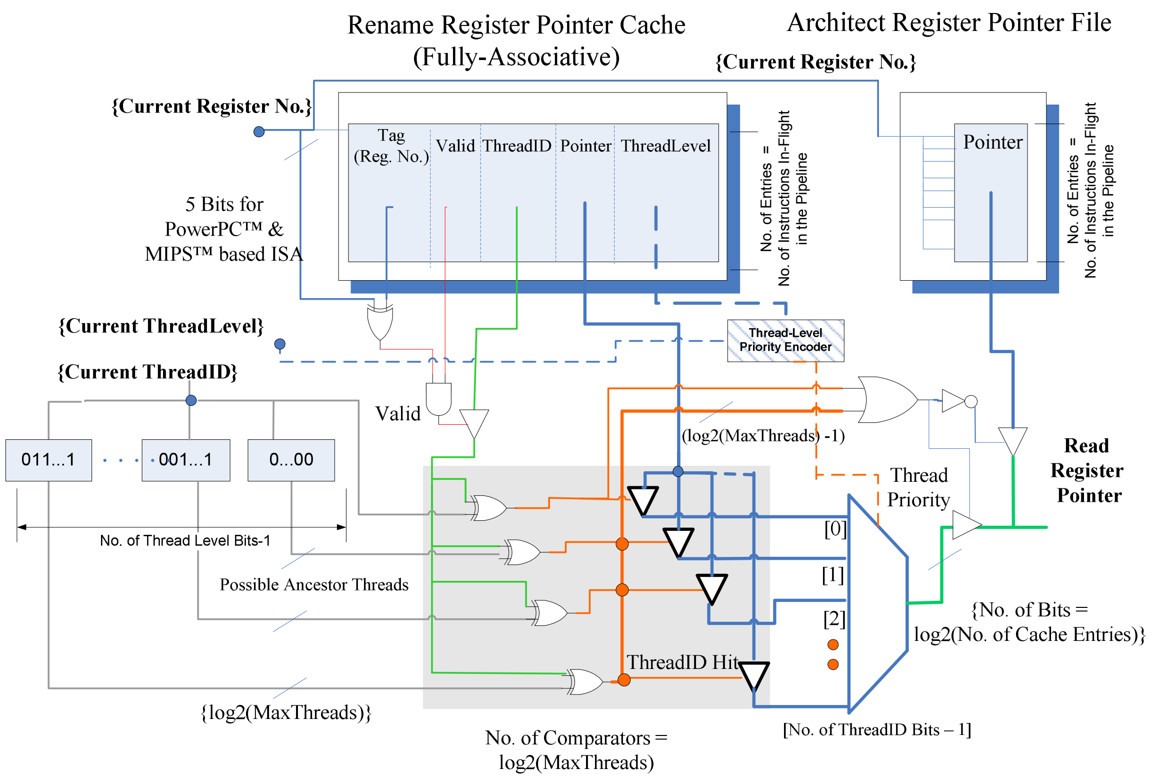


Figure 6.9. Logical Block Diagram of Register Renaming in Multi-Path Design.

6.8 Confidence Estimator

Another approach to reduce the number threads to follow the thread path that has the most likelihood to be executed. This form of execution is called *Dis-Joint Eager Execution* (DEE) and is discussed in detail in the following sections. In this section, the design and performance of the confidence estimator is discussed.

The confidence estimator works similar to the branch prediction except that instead of storing the target addresses, it has a 4-bit saturating counter. The performance of the 4-bit saturating confidence counter and other performance metrics are discussed by Manne et al., 1998 [38]. The following is the *Pseudo-Code* of the confidence update mechanism when the branch executes:

Prediction Correct:

if (Low Confidence): confidence < 8: set confidence value = 8

if (High Confidence): confidence value >= 8: Increment

Prediction Incorrect:

if (Low Confidence): confidence < 8: decrement

if (High Confidence): confidence value >= 8: set value = 7 (low confidence)

6.8.1 Fetch Logic using Confidence Estimates

The major difference with fetching instructions based on confidence estimates is that instead of a branch predictor a table of saturating counters is used by the fetch scheduler to determine the path of the next instruction fetch. The fetch scheduler may use different policies and are list in Table 6.4.

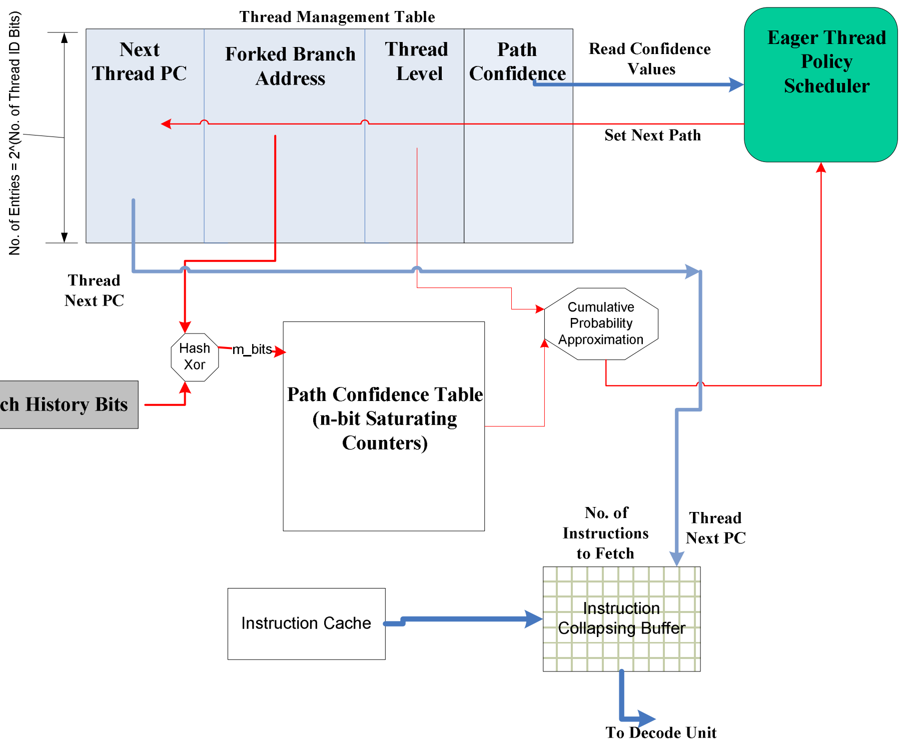

Thread Management Table
No. of Entries = 2^(No. of Thread ID Bits)
Next Thread PC
Forked Branch Address
Thread Level
Path Confidence
Read Confidence Values
Eager Thread Policy Scheduler
Set Next Path
Thread Next PC
Cumulative Probability Approximation
Hash Xor
m_bits
Branch History Bits
Path Confidence Table (n-bit Saturating Counters)
No. of Instructions to Fetch
Thread Next PC
Instruction Cache
Instruction Collapsing Buffer
To Decode Unit

Figure 6.10 Logical Block Diagram of Fetch Policy using Confidence Estimator

As shown in Figure 6.10, the BTB is now augmented by *Thread Management Table* which has the following fields, the next Thread PC, the forked branch address, thread level and path confidence. These fields are explained below,

Next Thread PC: Stores the next program counter of each active path.

Forked Branch Address: This is the branch address where the path is forked.

Thread Level: It indicates the level of the thread path and it changes as the path traverses down.

Path Confidence: It stores the confidence value of the path and changes as the path forks new paths.

In addition, to provide continuous fetch stream after switching different paths in the same clock cycle, an *instruction collapsing buffer* has to be modeled. This buffer stores the starting instruction address of a block and the length of the block. By using these fields, different sequences of instruction streams are combined to form a wide fetch group in the next cycle. Hence, with the help of the instruction collapsing buffer the fetch group in the cycle is not broken because of multi-path switching and it maximizes the fetch resource utilization.

Although, the structure of the instruction collapsing buffer is not modeled in the simulation, its functional behavior is implemented to ensure the entire bandwidth of the fetch is utilized. The cumulative probability approximation is a small multiplier unit that multiplies the current path confidence and the confidence of the forked path during the thread creation process. The branch predictor is used to determine if the confidence value should be associated with the taken or the not taken path.

6.8.2 Thread Path Creation Logic

A new path is created only if there is a hit in the BTB. If there is no hit in the BTB, the path continues in the not taken path (BTB only stores the taken addresses). At complete stage, if the completion logic detects the branch instruction did not spawn a thread, then it recovers the machine state if the branch is taken.

If there is a hit in the BTB, then a new path is forked in the new thread path level with complemented bits in the respective thread path level by the *Spawn New Thread* module. At the same time, the confidence value from the *Path Confidence Table* is read and a new entry is recorded in the *Thread Management Table* as illustrated in Figure 6.11. Depending on the *Thread Policy Scheduler*, the new thread may be followed or not.

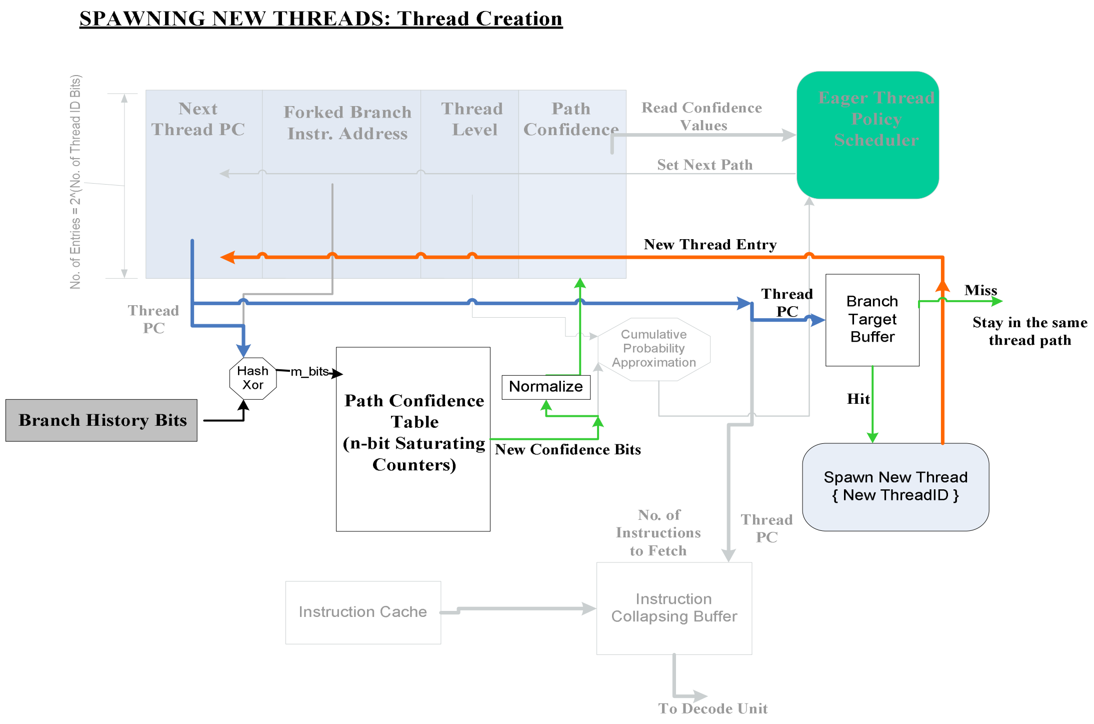

SPAWNING NEW THREADS: Thread Creation
No. of Entries = 2^(No. of Thread ID Bits)
Next Thread PC
Forked Branch Instr. Address
Thread Level
Path Confidence
Read Confidence Values
Eager Thread Policy Scheduler
Set Next Path
New Thread Entry
Thread PC
Thread PC
Branch Target Buffer
Miss
Stay in the same thread path
Hash Xor
m_bits
Branch History Bits
Path Confidence Table (n-bit Saturating Counters)
Normalize
Cumulative Probability Approximation
New Confidence Bits
Hit
Spawn New Thread { New ThreadID }
No. of Instructions to Fetch
Thread PC
Instruction Cache
Instruction Collapsing Buffer
To Decode Unit

Figure 6.11 Thread Creation Process

TABLE 6.4 COMPARISON OF FETCH POLICY SCHEMES THAT ARE EVALUATED IN THIS STUDY

Policy	**Perfect**	**Single-Thread**	**Eager Execution**		**Dis-Joint Eager Execution**		**Dis-Joint Eager Execution with selective threads**
			50 % allocation	*Confidence Based Fetch*	*Static Confidence*	*Dynamic confidence*	*Dynamic Confidence*
Fetch Group	Depends on BTB and perfect predictor	Depends on BTB and Branch Predictor	Split equally among all active paths	Allocated *propor-tionally* among all paths based on Confidence Values	Same as Dynamic	A path with high confidence values is chosen	A set of paths with high confidence values are chosen
Reason to study this scheme	Perfect case	To prove branch predic-tion for high fetch band-width is poor.	To illustrate the machine perfor-mance without any kind of branch prediction	To show how confidence values can be utilized.	To com-pare with the dyna-mic case	To limit the number of threads with confidence values	To minimize of dependence on confidence values as they can be misleading
Max. Possible Number of Threads	1	1	2^n, where 'n' is no. of branch levels	2^n, where 'n' is no. of branch levels	2^n, where 'n' is no. of branch levels	2^n, where 'n' is no. of branch levels	Depends on the Target IPC limit
Unconditional Branches	With BTB	With BTB	With BTB	With BTB	With BTB	With BTB	With BTB
Conditional Branch Prediction	Perfect	2-Bit State Predictor	Used after maximum thread level	Used after maximum thread level	Used after maximum thread level	Used after maximum thread level	Used after maximum thread level
Confidence Estimator	No	No	No	Yes	Yes	Yes	Yes
Updates	No Updates	Predictor updated at Com-plete	No Updates	Confidence Values updated at Finish Stage	No Updates as it is Static	Confidence Values updated at Finish Stage	Confidence Values updated at Finish Stage
Additional Hardware	-	Counters and BTB	Multi-Path machine	Multi-Path and Multipliers for propor-tional allocation at Fetch	Same as dis-joint	Multi-Path & Confi-dence multipliers	Multi-Path Confidence multipliers & priority encoder.

6.9 Simulation Environment

AbaKus simulation framework is used to explore the architectural features of the processors with both the branch prediction and multi-path execution schemes. This framework with module and port-structures gives a fair degree of accuracy in the simulations with reasonable speed. The details of AbaKus framework and superscalar models are discussed in Chapters 3, 4 and 5.

To focus the study on conditional branch effects on the processor, the component designs of simulated architecture are widened to minimize any structural design hazards. Perfect memory is assumed as conditional branches only have indirect effect on memory. The summary of architecture details are described in Table 6.5. The simulation is executed using Intel Xeon CPU 3.2 GHz (128-node cluster) with 4GB RAM. In the next section, the architecture descriptions of the single-threaded and multi-threaded designs are discussed.

To test the architecture design, benchmarks from Standard Performance Evaluation Corporation (SPEC) are used. In addition, the benchmarks are cross-compiled for Simplescalar MIPS IV instruction format. Due to the library compatibility problems only few of SPEC benchmarks were successfully compiled and are used in this study. The benchmarks are run up to *500* million and then the architecture designs are tested for the next *100* million instructions. This is done get past the start-up code in the benchmarks. This set of *100* million instructions, however, does not represent the entire benchmark that typically has more than 1 trillion instructions.

TABLE 6.5. SIMULATION DETAILS OF THE MULTI-PATH SMT ARCHITECTURE

Design Parameters	**Multi-Path SMT**
Maximum No. of Threads	2^{25} possible threads. Exclusively depends on Fetch Policy
Instruction Fetch Width per Thread	8 or 32 insts/cycle but depends on fetch policy
Instruction Window Size	4096 entries
Physical Registers	32
Issue Width	64
Commit Width	128
BTB & Branch Predictor (if used)	BTB: 8192 16-way Gshare: 16384 entries; 16 History Bits
Confidence Estimator (if used)	8132 entries
Confidence Counters (if used)	4-bit Saturating Counters
Integer ALU units (Latency =1)	40
Branch Units (Latency = 1)	40
Load/Store (Latency = 2)	40
Mul/Div (Latency = 5)	20
Float/Special Units (Latency = 3)	40
Write Back Bus Width	128
Complete Width	128

6.10 Implications

To understand the performance limitations of the conditional branches, a processor with perfect conditional branches is evaluated. This is done by gathering the target address traces of the conditional branches in a single-threaded processor and then, allowing the simulation to read from this trace when a conditional branch is encountered. In this way all the architecture parameters are the same between the perfect and the single-thread processor except the conditional branch prediction.

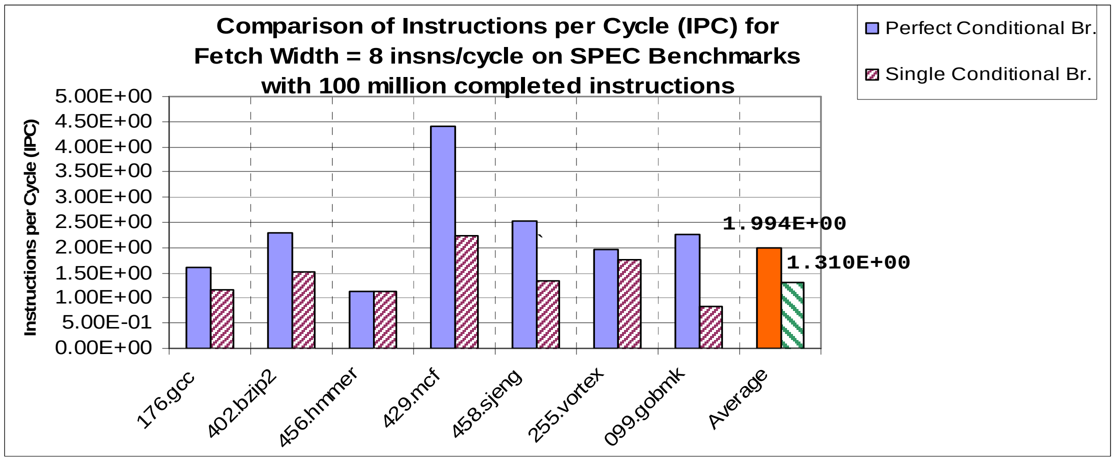


Figure 6.12 Performance Comparison between Perfect and Single-Threaded Processor

The average IPC in Figure 6.12 is calculated by finding the average CPI and then taking its inverse. The margin of improvement required on average is about *0.684 IPC*. Although, this may look small, there are some benchmarks that suffer more conditional branch mispredictions penalties than other benchmarks. From Figure 6.12, the IPCs of *429.mcf*, *458.sjeng* and *099.go* are likely to have more conditional branch misprediction penalties.

6.10.1 Increasing Fetch Width

Increasing fetch width to feed on more Instruction Level Parallelism (ILP) is not effective as seen in Figure 6.13. There are 2 factors that affect this, data dependency and, fetch width partition and penalties due to indirect jump mispredictions and exceptions. It also results in increase recovery cycles because more instructions from the window have to be cancelled during recovery.

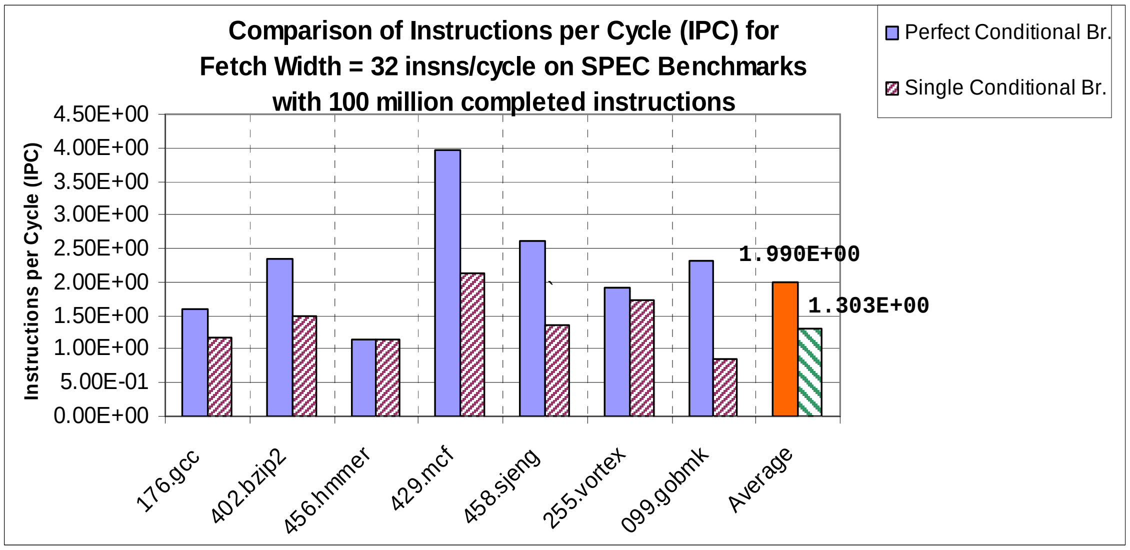


Figure 6.13 IPC for Fetch Width of 32. IPC for 32-wide fetch is slightly less than 8-wide fetch because of increased latency in recovery.

6.10.2 Reducing Conditional Branch Mispredictions

As shown in Figure 6.14, a single-threaded processor suffers from conditional branch error rate of *10 %* on average. Figure 6.14 also shows the number of conditional branch error for the set of *100* million completed instructions.

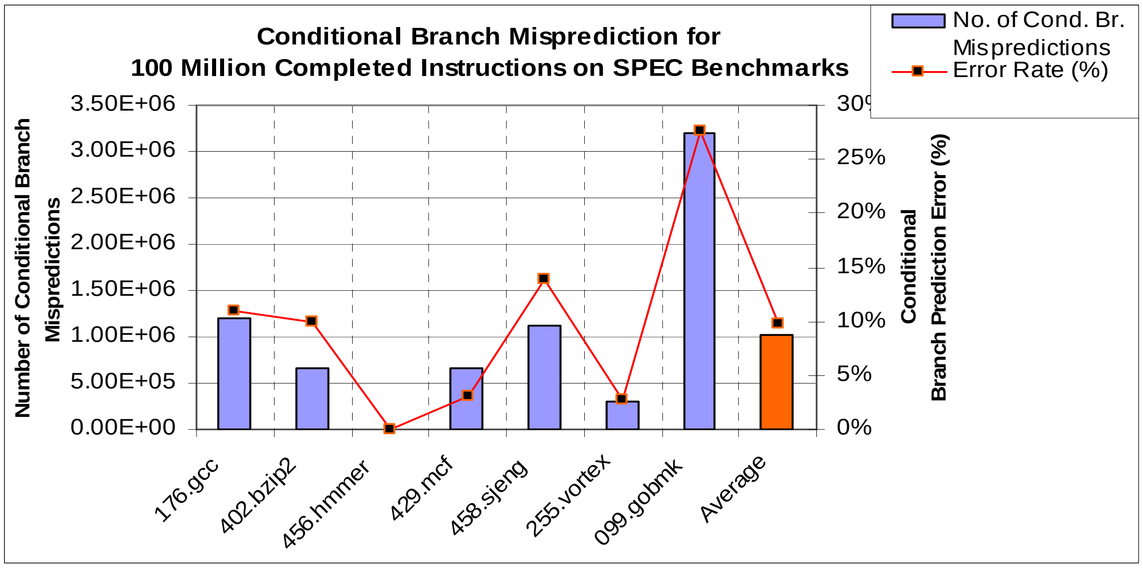


Figure 6.14 Conditional Branch Error Rate. The plot represents the number of Recoveries due Conditional Branch Misprediction

Figure 6.14 show that '*456.hmmer*' at an extremely low error of just 1 conditional branch error. This set of *100* million instructions happens to be the best case for this benchmark. Because no improvement can further be made on this phase of the benchmark, '*456.hmmer*' will not be tested with the eager-based architectures for this set of *100* million instructions.

6.10.3 Eager-Based Execution Schemes

The eager-based fetch policy schemes are detailed in Table 6.4 through a comparison with single-threaded fetch policy. Figure 6.15 shows the percentage of mispredictions due to conditional branches for eager execution policy.

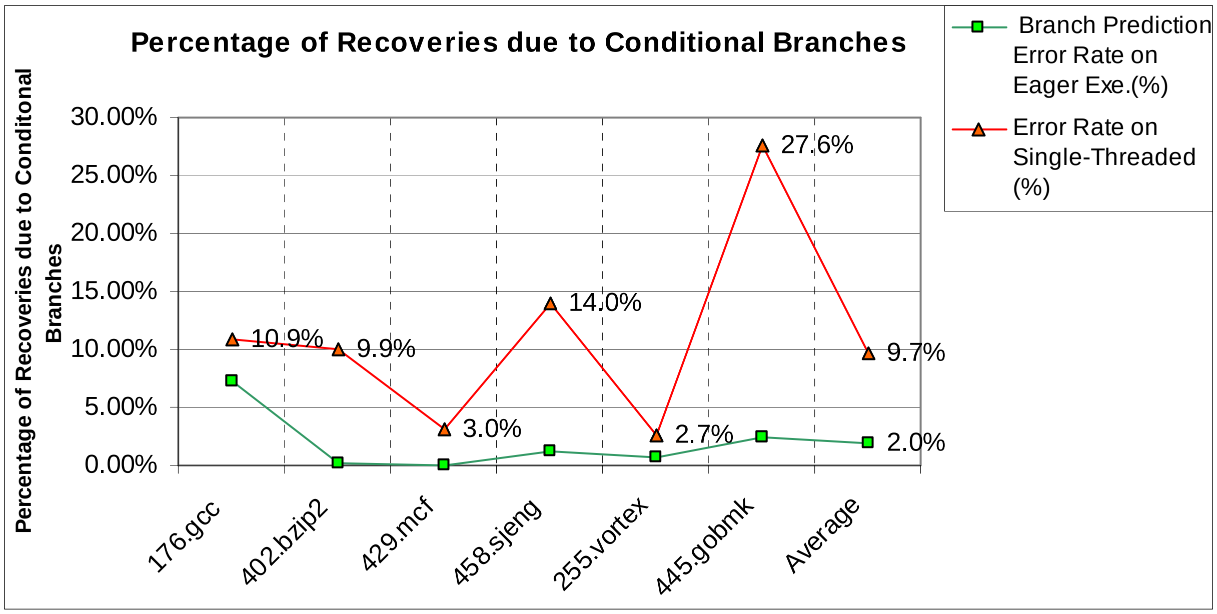


Figure 6.15 Percentage of Recoveries due to conditional branch misprediction. The figure shows that eager execution has reduced the number of recoveries. Mispredictions in eager based executions are due to compulsory BTB misses and if the number of unresolved branches reaches the maximum number of branch levels possible in the processor.

Branch prediction is used in the eager-based execution only if the maximum possible unresolved branch level is reached in the processor. If branch prediction is used then it leads to a possibility of misprediction. Hence, it is important for eager-based executions to use branch prediction rarely by increasing the number of maximum possible branch levels in the machine. This results in increase in more possible threads to handle in the processor. For example, if 3 unresolved branches exist in the processor then it leads to a maximum possibility of 2^3 or 8 threads.

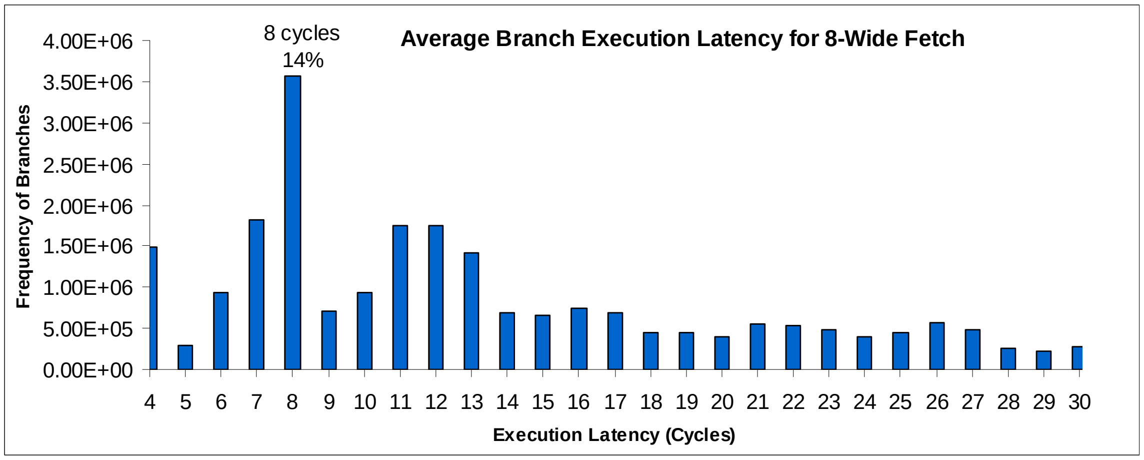


Figure 6.16 Average Branch Execution Latency for 8-Wide Fetch in the SPEC benchmarks.

As seen in Figure 6.16, the average branch execution latency involves more than 8 cycles. For an instruction window of size *4096*, the worst-case latency can be even little more than *4096* cycles implying a highly dependent instruction chain. But, as seen in Figure 6.16 one half of the pie-chart rotation is about *15* cycles. Hence, in order to make sure that the branch prediction is not used very often, the simulated eager-based processors can handle up to *25* unresolved branch levels or up to a maximum possible of 2^{25} short threads. The results of the simulations with IPC as the measure of performance for both 8 and 32-wide fetch are shown in Figure 6.17 and 6.18.

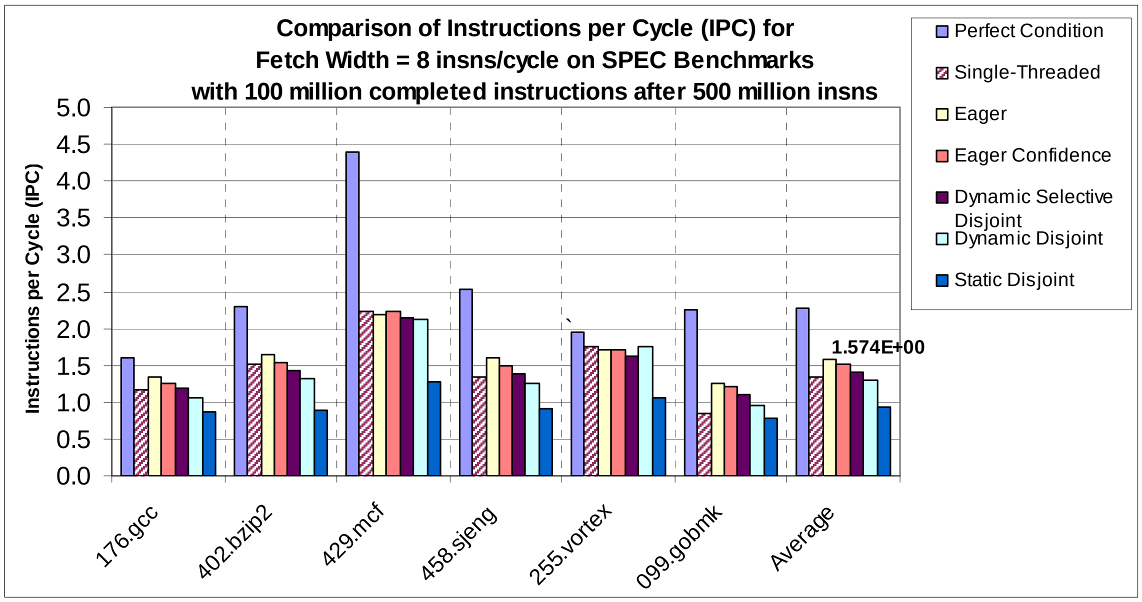


Figure 6.17 Comparison of IPC for different eager-based polices with single-threaded processor for 8-wide fetch.

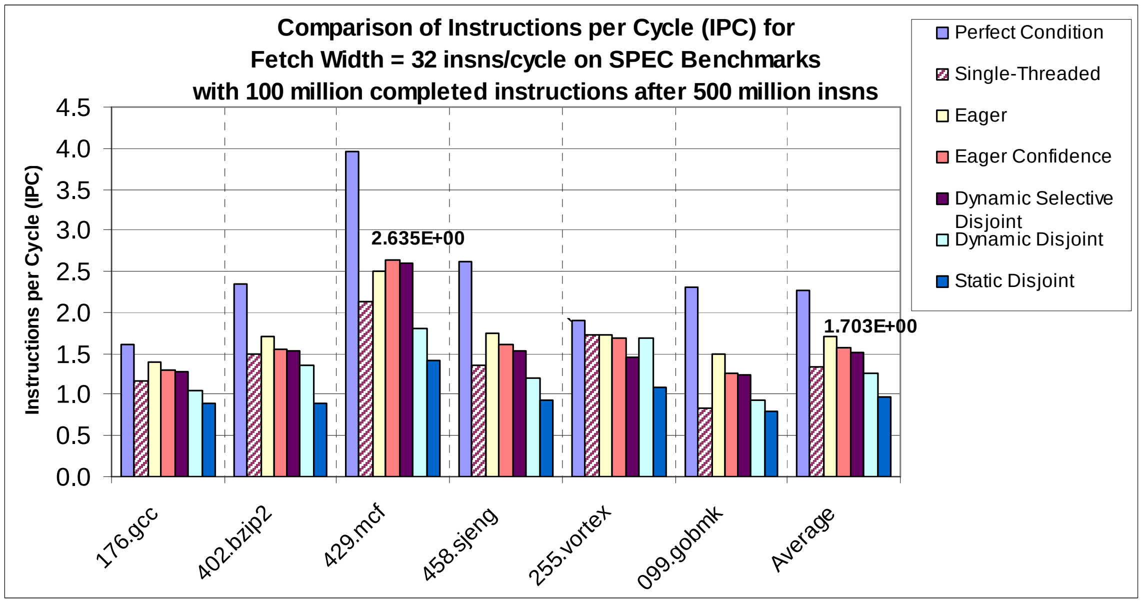


Figure 6.18 Comparison of IPC for different eager-based polices with single-threaded processor for 32-wide fetch.

From the Figure 6.17 and 6.18, one subtle but important observation is that the IPC for 32-wide has increased for eager-based execution while it did not for a single-threaded processor with branch prediction. For 8-wide and 32-wide fetch the eager execution with

50 % allocation (Table 6.4) has *17.21%* and *27.60%* improvement, respectively. The maximum possible improvement between the processor with perfect conditional branch prediction and the single-threaded processor with gshare branch prediction is about *70%* on average. *0.99.go* has the best improvement on IPC with about *77.26%* for the 32-wide fetch with eager execution. The low IPC value of static confidence-based disjoint execution signifies the importance of dynamic confidence estimator in the design.

6.11 Discussion on Confidence-Based Eager Execution Schemes

There are 3 important factors that need to be considered to attain the IPC of the perfect conditional branch prediction – *confidence estimates, branch prediction* and *fetch width*.

Using the confidence estimator described by Manne et al, 1998 [38] only supplements branch prediction. Eager polices that depend on confidence values such as disjoint, disjoint selective and confidence-based eager execution assumes that branch prediction error can be corrected by confidence estimates correctly. On the other hand, the dynamic nature of code execution proves to be far more complex than the confidence estimator can handle. This is illustrated in Figure 6.19 that shows the values of PVN, PVP, Specificity and Sensitivity of the confidence estimator. It is important that PVN – probability that low confidence is mispredicted correctly and Specificity – fraction of mispredictions that are low confidence are close to 1.

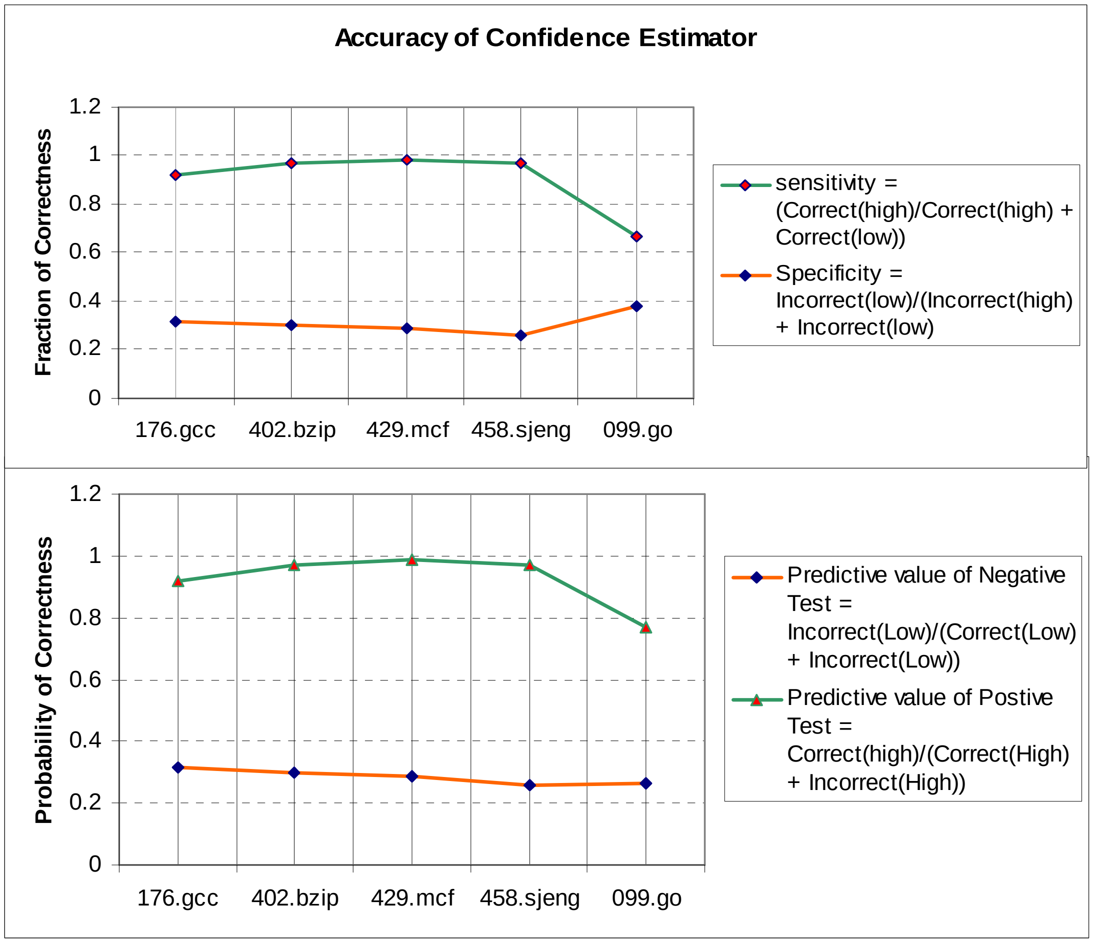


Figure 6.19 Accuracy of the Confidence Estimator with 4-bit saturating counters. The low values of PVN and Specificity highly affects the performance of the confidence-based eager executions.

In addition to confidence estimators, branch prediction and fetch width have a direct effect on IPC. The use of branch prediction is dependent on the maximum number of branch levels available in eager execution schemes. As seen in Figure 6.20, as the number of available branch levels decrease the processor relies more on the branch predictor and tend to make more branch mispredictions. This directly results in decrease in IPC. On the other hand, as seen in Figure 6.21, if the eager schemes have more number of branch levels, then the number of active-threads increase resulting in dividing of fetch resources. The way in which the fetch resources are divided depends on the imposed fetch policy of processor. However, as a result of dividing the fetch resources the number of instructions supplied to each thread is reduced impacting the IPC. This can be seen in Figure 6.20. The eager and disjoint-eager based executions of 25 and 16 levels have more or less a similar IPC (about 2.9 from Figure 6.20) where as the disjoint-eager with 8-levels have less number of threads but falters as it relies more on the branch predictor.

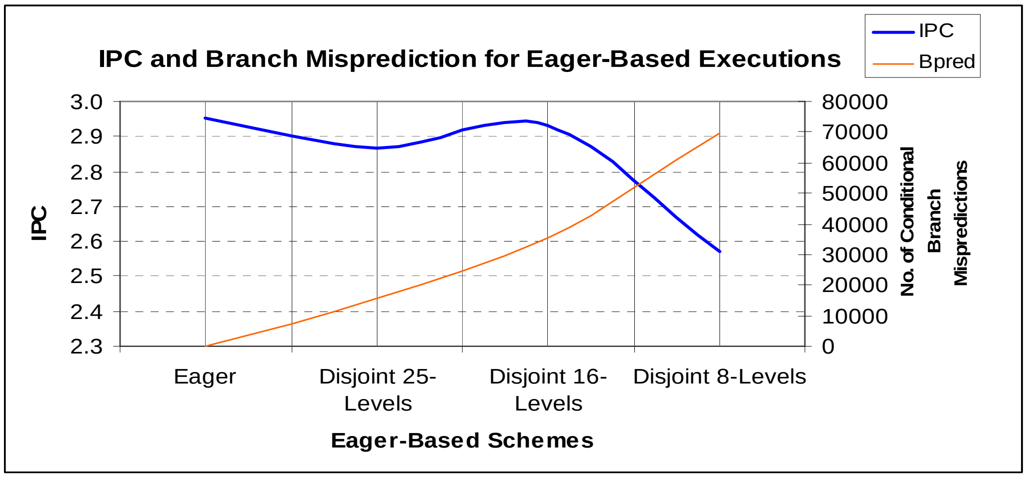


Figure 6.20 Relationship showing how different eager schemes rely on branch prediction and its effect on IPC

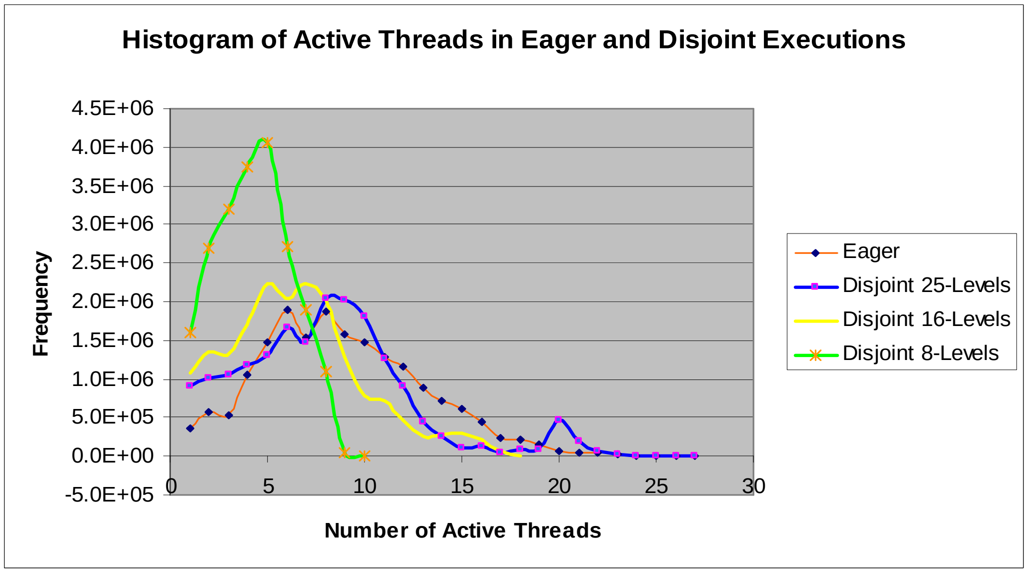


Figure 6.21 Histogram of Active Threads. The Disjoint with 8-levels has less number of active threads but relies more on branch prediction.

6.12 Summary

The effect on conditional branch misprediction on IPC of the processor is clearly seen in Figure 6.17 and Figure 6.18. There is about *70%* performance loss due to such mispredictions. Two distinctly different approaches of eager-based execution schemes are considered. The schemes directly affect the fetch bandwidth. In the first approach of simple eager-based execution with *50% allocation*, the branches spawn multiple paths and divided the fetch and pipeline resources. Although, the dependence on branch predictions is reduced, it also reduces the number of instructions being processed in each thread path. The second approach is the disjoint eager execution where the thread path that has the high confidence gets the priority to utilize the fetch resources. Although, this scheme allocates the fetch resources to high confidence paths, the confidence values tend to be error prone as shown in Figure 6.19. The confidence estimator is poorly identifies the paths that mispredict. This results in the thread to be discarded and hence wasting the fetch resources. A more judicious confidence estimator using advanced schemes such as data value prediction or neural network-based predictors would benefit the disjoint eager execution scheme.

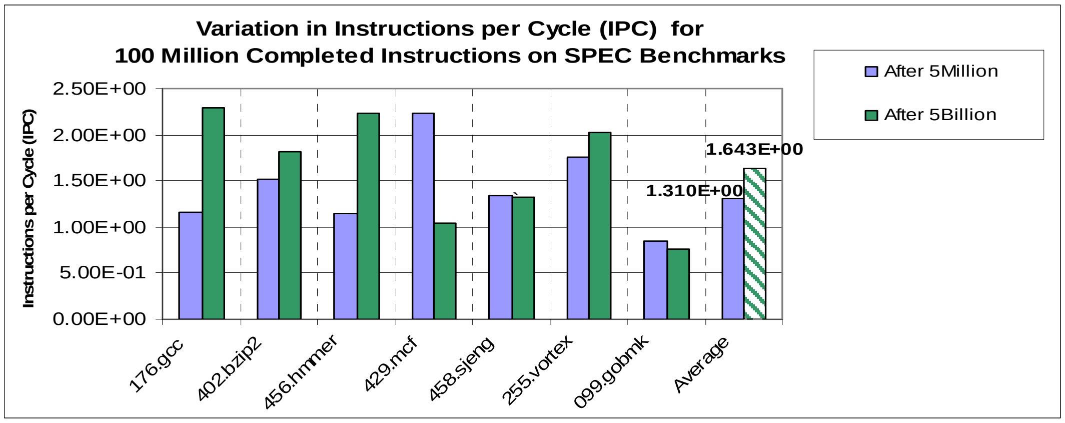


Figure 6.22 Code Phase Variations in SPEC benchmark

The *27%* average increase in IPC for eager-based execution is relatively significant considering the benchmarks that are chosen for performance evaluation. The SPEC benchmarks have code variations depending on the instruction group that is evaluated as seen in Figure 6.22. The size of each benchmark (more than 1 trillion instructions) and such code variation makes it hard to understand the true performance of the architecture design. However, by using *sim-points* (Lau et al., 2004 [46]) where statistical and other clustering techniques are used to determine subsets of code that represents the entire benchmark can help in finding the regions of code subsets for evaluating the architecture design.

CHAPTER VII

CONLCUDING REMARKS

Several of computer architecture simulation tools are available for architecture design space explorations. However, each of these simulation tools is developed to model certain specific aspects of the architecture. Hence, it is the task of the designer to make proper tool selection considering *accuracy, speed* and *flexibility* of the simulator. In addition, the simulator should also have cross-compiler features, if required, for extensive hardware design verification.

7.1 AbaKus Simulation Framework

AbaKus simulation framework is developed to model hardware functionality with simple behavior-level details but also with cycle-accurate timing. The timing information is described through port interfaces and is implicitly incorporated in the simulation for module communication. This is ideal for CPU core simulation because instruction flow is pipelined on a cycle-by-cycle basis. Moreover, there is one aspect where the simulation speed can be increased, which is by simply multi-threading the simulator as modules are task independent.

Although the simulation has sufficient task-level parallelism, the modules must communicate and hence, must synchronize every simulated cycle making it as a set of tightly-coupled threads. However, existing computers do not facilitate in speeding up of such multi-threaded codes as they synchronize much slower at second-level cache memory. AbaKus simulation framework can be extended to simulate multiple cores to study memory hierarchy designs as well as memory coherency problems. This dissertation has showed the usefulness of AbaKus framework by conducting performance studies in CPU core designs.

Evaluating architecture designs extensively with large benchmarks is essential for validating the design and measuring the performance. In the study of register write-back bus width discussed in Chapter 5, about *six billion* instructions are evaluated in the wide superscalar design. This shows both the capability of AbaKus as well as the extent to which the designs can be evaluated.

7.2 Instruction-Level Parallelism

Instruction-Level Parallelism may seem to have hit the brick-wall and has been extremely hard to even go beyond IPC of 2.5 in the evaluated SPEC benchmarks. Although this may be a limiting case to increase the speed-up of sequential programs, these programs are compiled with compilers that takes no account of the different hardware architectural features. This is a major problem as compilers could also aid in finding the ILP necessary for wide superscalar processors. Many new compilers such as OpenMP (Chapman et al., 2008 [48]), NVIDIA CUDA™ compiler [49] and Intel® C++

Compiler for Itanium architectures [50] take this into account to extract the parallelism available at all levels in the program.

7.3 Conclusion and Future Work

This dissertation has demonstrated the successful design and development of an open-source computer architecture simulator – AbaKus - and also in identifying the key aspects of design limitations in wide superscalar processors. The following are some of the contributions made in this research,

- **AbaKus Computer Architecture Simulator**

  AbaKus incorporates a simple timing structure in its framework that enables the tool to be adapted to other existing hardware description languages. This timing structure based on *Moore State Machine* also provides cycle-time accuracy that is the baseline for all pipelined architecture designs. In addition, the AbaKus superscalar models can be reused for future design evaluations and as shown in the case studies, it can be extended to simulate complex multi-threaded and multi-core architectures.

- **Designed and verified architecture designs for Eager-Based Executions**

  Confidence-Based fetch polices are proposed and evaluated. It optimizes the use of the fetch bandwidth by dynamically varying the fetch rate of eager-threads based on the path confidence values. Since the confidence estimator is very important for the design, future work on eager execution would be to increase PVN and Specificity of this estimator. The design and performance of the disjoint eager execution using dynamic confidence estimators is also evaluated architecturally.

REFERENCES

[1] D Burger and T. M. Austin, "*The SimpleScalar Tool Set*, Ver. 2.0, University of Wisconsin Computer Sciences Technical Report #1342, (June 1997).

[2] M. Vacchharajani, N. Vachharajani, D. A. Penry, J. A. Blome, D. I. August, "*Microarchitectural Exploration with Liberty*", Proceedings of 35th Internl. Symp. on Microarchitecture, (2002).

[3] M. Vacchharajani, N. Vachharajani, D. A. Penry, S Malik, D. I. August, "*The Liberty Simulation Environment: A deliberate approach to high-level system modeling*", ACM Transaction on Computer Systems., .vol. 24, No. 3, (August 2006), pp. 211-249.

[4] J. Emer et el, "*ASIM: A Performance Model Framework*", IEEE Computer, (Feb. 2002), pp. 68-76.

[5] Pellauer, M., Vijayaraghavan, M., Adler, M., Arvind, and Emer, J. 2008. A-Ports: an efficient abstraction for cycle-accurate performance models on FPGAs. In *Proceedings of the 16th international ACM/SIGDA Symposium on Field Programmable Gate Arrays* (2008).

[6] Open SystemC Initiative, 2008, http://www.systemc.org/

[7] D. Perez, G. Mouchard, and O. Temam, "*MicroLib: A Case for Quantitative Comparison of Microarchitecture Mechanisms*," Proc. International Symposium of Microarchitecture, 2004.

[8] Virtutech® Simics™, 2008, http://www.simics.net/

[9] Sandro Rigo, Guido Araujo, Marcus Bartholomeu and Rodolfo Azevedo, "*ArchC: A SystemC-Based Architecture Description Language*" In proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC'04). Foz do Iguacu - Brazil, October 2004.

[10] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Teslyar, Daxia Ge, Christos Kozyrakis, Kunle Olukotun, *A Practical FPGA-based Framework for Novel CMP Research*, Proceedings of the 15th ACM SIGDA Intl. Symposium on Field Programmable Gate Arrays, Montery, CA, February 2007

[11] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, S. K. Reinhardt, "The *M5* Simulator: Modeling Networked Systems," IEEE MICRO, pp. 52-60, July/August, 2006.

[12] SESC: SuperESCalar Simulator. http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/, 2002.

[13] Yourst, M.T., "*PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator*," Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on , vol., no., pp.23-34, 25-27 April 2007

[14] E. Larson, S Chatterjee and T. Austin, "*The MASE Microarchitecture Simulation Environment*", IEEE Internl. Symp. on Perf. Analysis of Systems and Software, June 2001.

[15] D. G. Perez, G. Mouchard and O. Temam, "A New Optimized Implementation of the SystemC Engine Using Acyclic Scheduling", Proceeding of the Design, Automation and Test in Europe Conf., 2004, pp. 1530-1591.

[16] J. P. Shen and M. H. Lipasti, "*Modern Processor Design, Fundamentals of Superscalar Processor", Tata McGraw-Hill Edition,* ISBN 0-07-059033-8

[17] Nam Sung Kim, Trevor N. Mudge, "Reducing Register Ports Using Delayed Write-Back Queues and Operand Pre-fetch", ICS 2003, pp: 172-182.

[18] José-Lorenzo Cruz, Antonio González, Mateo Valero, Nigel P. Topham,"Multiple-banked register file architectures", ISCA 2000, pp 316-325

[19] Rajeev Balasubramonian, Sandhya Dwarkadas, David H. Albonesi,"Reducing the complexity of the register file in dynamic superscalar processors", MICRO 2001, pp 237-248

[20] Tetsuya Sueyoshi, Hiroshi Uchida, Hans Jürgen Mattausch, Tetsushi Koide, Yosuke Mitani, Tetsuo Hironaka,"Compact 12-port multi-bank register file test-chip in 0.35µm CMOS for highly parallel processors", ASP-DAC 2004, pp 551-552.

[21] Nam Sung Kim, T. N. Mudge: The microarchitecture of a low power register file. ISLPED 2003: 384-389

[22] Mark Smotherman, Shuchi Chawla II, Stan Cox, Brian A. Malloy: Instruction scheduling for the Motorola 88110, MICRO 1993: 257-262

[23] D. A. Penry and D I. August, "Optimizations for a Simulator Construction System Supporting Reusable Components,"40th IEEE Design Automation Conf., 2003, pp. 926-931.

[24] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. MICRO 07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pages 249--261, 2007.

[25] Lee, J., Kim, J., Jang, C., Kim, S., Egger, B., Kim, K., and Han, S. 2008. FaCSim: a fast and cycle-accurate architecture simulator for embedded systems. SIGPLAN Not. 43, 7, 89-100, 2008.

[26] M. M.K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset,Computer Architecture News (CAN), 2005.

[27] Patt, Y. N., Patel, S. J., Evers, M., Friendly, D. H., and Stark, J., "One Billion Transistors", One Uniprocessor, One Chip. Computer 30, 9, 51-57, 1997.

[28] Aashish Phansalkar, Ajay Joshi, and Lizy K. John, Analysis of Redundency and Application Balance in the SPEC CPU2006 Benchmark Suite, ACM SIGARCH Computer Architecture News, Vol. 35 , Issue 2, May 2007, pp: 412 – 423.

[29] J.A. Kahle et al.,"*Introduction to the Cell Multiprocessor*", IBM Journal of Research and Development, Vol 49, No. 4/5 2005

[30] H. M. Mathis et al., "Characterization of Simulatneous Mulithreading (SMT) efficieny in POWER5", IBM Journal of Research and Development, Vol 49, No. 4/5, 2005.

[31] Ravi Navir, IEEE Transactions on Computers Staff 1995. Optimal 2-Bit Branch Predictors. IEEE Trans. Comput. 44, 5 (May. 1995), 698-702.

[32] S McFarling, “Combining Branch Predictors”, Techincal Report TN-36, Digital Equipment Corp, 1993.

[33] Agerwala, T., and J. Cocke, “High performance reduced instruction set processors”, Technical Report, IBM Computer Science, 1987.

[34] Artur Klauser , Srilatha Manne , Dirk Grunwald, Selective Branch Inversion: Confidence Estimation for Branch Predictors, International Journal of Parallel Programming, v.29 n.1, p.81-110, February 2001

[35] Ahuja, P. S., Skadron, K., Martonosi, M., and Clark, D. W. 1998. Multipath execution: opportunities and limits. In Proceedings of the 12th international Conference on Supercomputing (Melbourne, Australia). ICS '98.

[36] Erik Jacobsen , Eric Rotenberg , J. E. Smith, Assigning confidence to conditional branch predictions, Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, p.142-152, December 02-04, 1996

[37] Aragon, J.L.; Gonzalez, J.; Garcia, J.M.; Gonzalez, A., "Selective branch prediction reversal by correlating with data values and control flow," Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on , vol., no., pp.228-233, 2001.

[38] Manne, S., Klauser, A., and Grunwald, D. 1999. Branch Prediction Using Selective Branch Inversion. In Proceedings of the 1999 international Conference on Parallel Architectures and Compilation Techniques (October 12 - 16, 1999).

[39] Uht, A. K., Sindagi, V., and Hall, K. 1995. Disjoint eager execution: an optimal form of speculative execution. In Proceedings of the 28th Annual international Symposium on Microarchitecture (Ann Arbor, Michigan, United States, November 29 - December 01, 1995).

[40] Kshitiz Malik, Mayank Agarwal, Vikram Dhar, Matthew Frank. "PaCo: Probability-based Path Confidence Prediction". International Symposium on High-Performance Computer Architecture, (HPCA-14), February, 2008.

[41] Wallace, S., Calder, B., and Tullsen, D. M. 1998. Threaded multiple path execution. SIGARCH Comput. Archit. News 26, 3 (Jun. 1998), 238-249.

[42] T.H. Heil and J.E. Smith. "Selective Dual Path Execution". Technical Report, University of Wisconsin-Madison, ECE, 1997.

[43] Akkary, H., Srinivasan, S. T., Koltur, R., Patil, Y., and Refaai, W. 2004. Perceptron-Based Branch Confidence Estimation. In Proceedings of the 10th international Symposium on High Performance Computer Architecture (February 14 - 18, 2004).

[44] H. Gao, Y. Ma, M. Dimitrov, and H. Zhou, “Address-Branch Correlation: A Novel Locality for Long-Latency Hard-to-Predict Branches”, The 14th International Symposium on High Performance Computer Architecture (HPCA-14), pp. 74-85, Feb., 2008.

[45] Artur Klauser and Abhijit Paithankar and Dirk Grunwald, "Selective eager execution on the polypath architecture", In 25th Annual International Symposium on Computer Architecture, 1998, 250-259.

[46] Tien-Fu Chen. Supporting Highly Speculative Execution via Adaptive Branch Trees. In Fourth Intl. Symp. on High-Performance Computer Architecture, February 1998.

[47] Jeremy Lau, Stefan Schoenmackers, and Brad Calder, Structures for Phase Classification, 2004 IEEE International Symposium on Performance Analysis of Systems and Software, March 2004

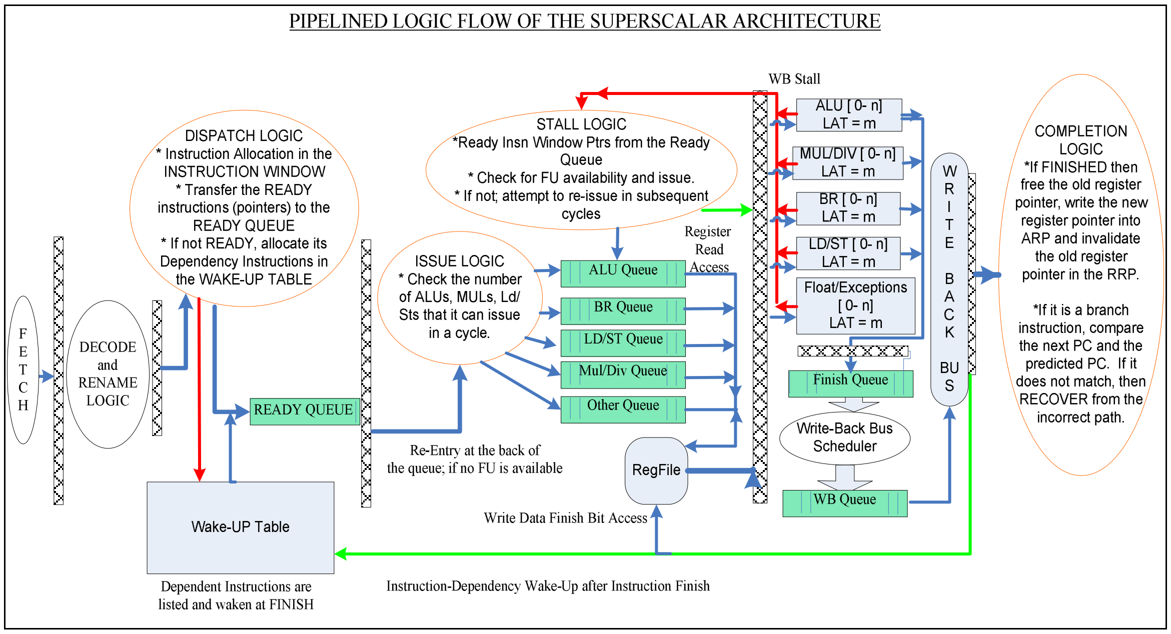
[48] Barbara Chapman, Gabriele Jost and Ruud van der Pas, “Using OpenMP: Portable Shared Memory Parallel Programming”, MIT Press, 2008, ISBN 978-0-262-53302-7.

[49] NVIDIA CUDA™ Education, 2008, http://www.nvidia.com/object/cuda_education.html

[50] Intel® C++ Compiler, 2008, http://software.intel.com/en-us/

APPENDIX

Software Design of the Simulated Superscalar Architecture


PIPELINED LOGIC FLOW OF THE SUPERSCALAR ARCHITECTURE
FETCH
DECODE and RENAME LOGIC
DISPATCH LOGIC
* Instruction Allocation in the INSTRUCTION WINDOW
* Transfer the READY instructions (pointers) to the READY QUEUE
* If not READY, allocate its Dependency Instructions in the WAKE-UP TABLE
READY QUEUE
Wake-UP Table
Dependent Instructions are listed and waken at FINISH
ISSUE LOGIC
* Check the number of ALUs, MULs, Ld/Sts that it can issue in a cycle.
Re-Entry at the back of the queue; if no FU is available
STALL LOGIC
*Ready Insn Window Ptrs from the Ready Queue
* Check for FU availability and issue.
* If not; attempt to re-issue in subsequent cycles
ALU Queue
BR Queue
LD/ST Queue
Mul/Div Queue
Other Queue
Register Read Access
RegFile
Write Data Finish Bit Access
WB Stall
ALU [0- n] LAT = m
MUL/DIV [0- n] LAT = m
BR [0- n] LAT = m
LD/ST [0- n] LAT = m
Float/Exceptions [0- n] LAT = m
Finish Queue
Write-Back Bus Scheduler
WB Queue
WRITE BACK BUS
COMPLETION LOGIC
*If FINISHED then free the old register pointer, write the new register pointer into ARP and invalidate the old register pointer in the RRP.
*If it is a branch instruction, compare the next PC and the predicted PC. If it does not match, then RECOVER from the incorrect path.
Instruction-Dependency Wake-Up after Instruction Finish

VITA

Aswin Ramachandran

Candidate for the Degree of

Doctor of Philosophy

Dissertation: PERFORMANCE LIMITATIONS IN WIDE SUPERSCALAR PROCESSORS

Major Field: Electrical Engineering

Biographical:

Personal Data: Born 20th August 1980

Education:

Graduated from University of Madras, Madras, India earning a Bachelor's degree in Electronics and Communication Engineering during May 2001;
Graduated from Oklahoma State University, Stillwater, India earning a Master’s degree in Electrical Engineering during December 2003;
Completed the requirements for the Doctor of Philosophy in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2008.

Experience:
School of Electrical Engineering
Graduate Research Assistant (2004-2008)
Graduate Teaching Assistant (2004-2008)

Professional Memberships:
IEEE Student Member

Name: Aswin Ramachandran Date of Degree: December, 2008

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: PERFORMANCE LIMITATIONS IN WIDE SUPERSCALAR PROCESSORS

Pages in Study: 119 Candidate for the Degree of Doctor of Philosophy

Major Field: Electrical Engineering

Scope and Method of Study:

Superscalar processors with wide instruction fetch only results in diminishing performance returns. The aim of this research to find what causes these limitations. In addition, a new cycle-accurate computer architecture simulator – AbaKus - is developed to study and evaluate the performance of the architecture designs.

Findings and Conclusions:

Eager-Based executions and their designs are tested to overcome the effects of low-accuracy of branch prediction on 38% of the conditional branch instructions. An improvement IPC of 27% on average is shown. However, confidence estimators need improvement on its design logic as they prove critical on the performance of eager-based executions. In addition, the limitation of compilers to extract ILP from the benchmark programs leads to a severe restriction on performance of Superscalar architectures due to data dependencies.

ADVISER’S APPROVAL: Dr. Louis G. Johnson