Scalable Superscalar Processing

by
Soner Onder
BSc, Middle East Technical University, 1983
MSc, Middle East Technical University, 1988

Submitted to the Graduate Faculty of
Arts and Sciences in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of Pittsburgh
1999

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This dissertation was presented

by

Soner Onder

It was defended on

July 12, 1999

and approved by

Prof. Henry Chuang

Prof. Guang R. Gao

Prof. Mary Lou Soffa

Prof. Rajiv Gupta (Committee Chairperson)

Scalable Superscalar Processing

Soner Onder, PhD
University of Pittsburgh, 1999

In this dissertation, it is demonstrated that there is sufficient parallelism in ordinary programs to scale the issue width of the out-of-order issue superscalar processors provided that processors employ very large instruction windows and near-perfect dynamic memory disambiguation. The state-of-the-art instruction wake-up and dynamic memory disambiguation techniques are thoroughly analyzed and it is demonstrated that they do not scale beyond an issue width of 8. This dissertation proposes alternative techniques for dynamic memory disambiguation and instruction wake-up mechanisms that scale well upto an issue width of 32.

Large instruction windows can be implemented without adversely effecting the processor clock using the concept of dynamically generating a dependence graph which is then used to directly wake-up instructions which are shelved in the reorder buffer. The resulting microarchitecture is called the Direct Wake-up Microarchitecture (DWMA). DWMA implements very large instruction windows with little loss in performance compared to an ideal central window implementation of the same size. For example, The DWMA processor achieves 84 %, 79 % and 67 % of the performance of an ideal central window processor at issue widths of 8, 16 and 32 instructions, respectively.

The solution to scalable dynamic memory disambiguation is based on a novel memory order violation detection mechanism which allows full out-of-order issuing of the store instructions in the instruction window. As a result, memory dependence predictors which rely only on the program counter values to make their predictions can be effectively employed without introducing false memory dependencies. Using this technique together with the store-set memory disambiguator a processor can achieve 100 %, 96 %, and 85 % of the performance of a processor that embodies a "perfect" memory disambiguator at issue widths of 8, 16, and 32 instructions, respectively.

Evaluatation of both the existing techniques as well as the new ones demanded development of many simulators. As a result, a new domain specific language called Architecture Description Language (ADL) has been designed and implemented in a powerful simulation system called the Flexible Architecture Simulation Tool (FAST). FAST has been used to generate all the cycle-level accurate simulators required for this thesis.

Acknowledgments

I am thankful that I had such great parents and two great brothers who have been a constant source of support throughout my life. My mother with her unparalleled love and undiminishing energy to support her children, my father for being such an honest man and hence a great role model for me, my brothers with their unconditional love and understanding, my deceased grandmother for showing me the value of having an absolutely pure heart, my lovely wife Nilufer for finding great many ways to support me while she herself was struggling to finish her Ph.D. and being a great mother to our daughter at the same time, and our daughter Gunseli for just being herself deserves appreciation that no word in existing languages can express. I am eternally indebted to all of them. I hope that God accepts my wishes for their well-being for I always felt myself more than lucky just for having them.

My special thanks are for my supervisor Prof. Rajiv Gupta. I am greatly indebted for his support that extended through many years of work during which I have learned so many things from him. We have carried out an exciting work with its ups and downs together. He was always there, ready to give his time to help sort my convoluted pile of thoughts into an organized structure and make them ideas. These years have been invaluable for me and I will always remember them well.

I would like to thank my committee members, Prof. Henry Chuang, Prof. Mary Lou Soffa and Prof. Guang R. Gao for giving their precious time and helping me develop this dissertation to its final shape. Prof. Soffa was always there to listen to me when I needed her advice. Prof. Chuang was there to answer my questions related to hardware implementations. Prof. Gao's comments about the direction of my work has been invaluable. Their kindness will not be forgotten on my part.

I would like to thank Prof. Robert Daley for arranging my RA support during my first year at Pitt. I also would like to thank my friend Dr. Frederick D. Ullman for finding many many ways to support me from a distance. His friendship and help in guiding me especially during the early years of my Ph.D. studies have been invaluable.

It is also my duty to give due credit to all the good spirited staff of the Department of Computer Science of University of Pittsburgh for helping me achieve things quicker and just for being themselves so that we simply were able to share our feelings about nice things.

As a final word, I would like to wish that my colleagues and friends will soon be writing their own version of this script. I thank them all for being there.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The last decade have witnessed a silent revolution. Today, with few exceptions, every microprocessor that has been produced for the desktop, workstation and server environment is an out-of-order issue superscalar processor. These processors are built with complex instruction fetching, scheduling and issuing mechanisms with each new generation having ever increasing capabilities to exploit instruction level parallelism. There are many reasons behind this trend. Chip manufacturing technologies have made a big leap in their capabilities to put more transistors onto the same chip area. It is already being widely discussed how to best make use of one billion transistors that will soon be possible to put on a single chip [53, 35, 62]. However, the advances in manufacturing technologies is only one side of the coin. Superscalar processors have been successful mainly because of their inherent advantages: (a) they can execute existing code faster without a need for recompilation; (b) they can make use of run time information to extract higher degrees of instruction level parallelism; and (c) they can make effective use of speculative techniques such as aggressive load, branch and value speculation, all of which can be implemented efficiently in such a setting. These advantages make out-of-order superscalar processing the most likely architectural choice for the foreseeable future, well into the next decade.

In order to deliver ever increasing amounts of instruction level parallelism, each successive generation of superscalar processors is being designed with capabilities to issue more instructions every cycle. Today, 4-issue superscalar processors are widely available. Processors that can issue upto 16 instructions are on the horizon, all because of a very simple reason. Delivering higher degrees of instruction level parallelism requires making the machine wider. This is true even when one considers the technique of value prediction [34, 36, 28, 9, 69] which can enable dependent instructions to execute in parallel. Ultimately, it is the number of instructions that a processor can issue simultaneously that puts an upper bound on the performance of the architecture.

This trend however brings in two important questions, namely, the issue of available parallelism and the scalability of existing techniques. It has already been shown by many researchers that ordinary programs have significant degrees of instruction level parallelism that can be exploited by the hardware through a number of techniques [20, 8, 35, 53, 47, 70]. However, we need to know if "typical" programs that are the target domain for superscalar processors have sufficient extractable instruction level parallelism by the out-of-order superscalar processing paradigm. In other words,

we would like to know the limits of instruction level parallelism from a superscalar processing perspective. Once we know that there is sufficient exploitable instruction level parallelism to scale the issue width, we would like to know if existing microarchitectural techniques scale to higher issue widths to exploit it.

As it can be seen, there is a need for changing our approach to evaluating superscalar microarchitecture techniques. We need to ask not only the question of whether or not a new technique increases the performance, but also whether the technique scales as the issue width of the processor is increased. Although some techniques have been evaluated from a scalability perspective in relation to the manufacturing and implementation constraints recently [59], the limitations of existing techniques as the issue width is increased have not been studied before.

1.1 Issues in Superscalar Processing

One of the primary goals of this thesis is to establish the necessary foundation for a change of perspective in superscalar microarchitecture research such that evaluations stress the scalability of the techniques. Doing so, this thesis considers three areas of superscalar processing as performance critical. These are the *instruction fetch*, *memory disambiguation*, and *instruction wake-up and issue* areas. In order to uncover and exploit high levels of parallelism, it is crucial that each of these areas have scalable implementations. Among these, this thesis focuses on the instruction wake-up and issue and memory disambiguation techniques and develops novel alternatives to existing algorithms. Let us now focus on each of these areas, examine the state-of-the-art techniques and briefly discuss the contributions of this thesis.

Instruction Fetch. In order to issue multiple instructions in each cycle, it is essential that the processor possess the capability of fetching a large number of instructions every cycle. This task is made difficult by the presence of frequent branches which disrupt the instruction stream. Branches are problematic for high performance superscalar processors because of two reasons: (a) in a typical program one out of every 4-5 instructions is a branch instruction. In order to issue a large number of instructions every cycle, multiple branches must be predicted correctly and multiple blocks must be fetched and combined; (b) when the instruction stream encounters a taken branch, the remaining instructions in the cache line are discarded. In other words, misalignment is a significant consumer of the instruction fetch bandwidth [48]. In fact, approximately 50 % of the branches on the average in the dynamic instruction stream in Spec95 benchmarks are separated by only one useful instruction (see Figure 1.1).

In addressing the fetch problem, the trace cache mechanism has yielded promising results [61, 17]. Although there is a room for improvement in this area, the trace cache approach is a significant step towards eliminating the fetch bottleneck. Recent work in this area reported that with realistic branch prediction using Spec95 integer benchmarks a 16-issue superscalar can achieve an IPC of 3.95 and with perfect branch prediction an IPC of 7.6. This figure is within close proximity

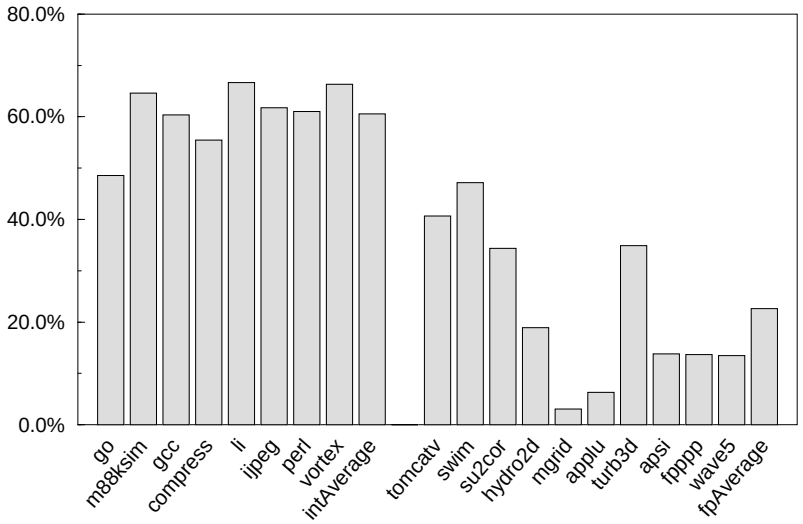


Figure 1.1: Dynamic percentage of branches separated by a single instruction

of a perfect fetcher that can deliver an IPC of 9.5 [6]. Therefore, this thesis does not investigate the fetch problem further.

Instruction Wake-up and Issue. Superscalar out-of-order processors mimic dataflow architectures to exploit large amounts of instruction level parallelism. Doing so, they buffer a large number of instructions which observe true data dependencies to begin their execution. To uncover high degrees of instruction level parallelism, a large number of instructions must be continuously examined for ready instructions. In other words a large instruction window is needed from which ready instructions may be found to sustain a steady flow of instructions to the functional units for execution. In fact, the required window size increases quadratically with increasing issue width [47].

Implementation of a large instruction window in a superscalar processor, without slowing down the processor clock, requires better techniques for identifying ready instructions than what are available today. This is because fetching and buffering a large number of instructions in a processor is by itself not sufficient to derive the benefits of a large instruction window. The waiting instructions must be woken up at the earliest possible time that they become ready to sustain a high degree of instruction level parallelism. In other words, implementation of a large instruction window is made difficult by the need for fanning out the wake-up signal to waiting instructions and selecting for issuing the instructions which are ready. When we assume that instructions are woken up by means of broadcasting, as in a central window implementation, we are faced with significant delays which originate from wire delays, tag matching time as well as the associative logic necessary to implement the wake-up functionality [49]. These delays increase significantly for high issue widths required to exploit high degrees of instruction level parallelism. This is because the delay of the wake-up logic

of an instruction window is a function of the window size and the issue width. These delays increase quadratically for most building blocks of the instruction window [49].

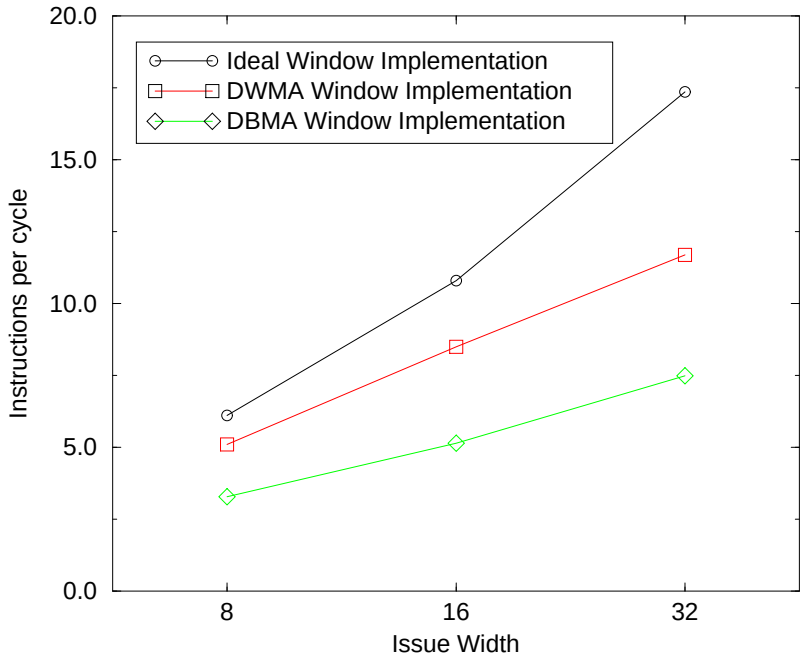


Figure 1.2: Thesis Contribution in Instruction Window Implementation

While a number of alternative architectures have been considered to address the wake-up problem [63, 62, 71, 30, 50], existing solutions within the context of out-of-order superscalar execution paradigm have not produced satisfactory results. Today, efficient implementation of large instruction windows in superscalar processors is an unresolved problem.

This dissertation proposes a novel solution to the implementation of large instruction window problem. The solution is based on the novel idea of generating a special form of dataflow graph called *Direct Data Forwarding Graph* (DDFG). By generating and consuming such a graph dynamically, large instruction windows can be implemented in superscalar processors without slowing down the processor clock. The wake-up process is achieved by associating explicit wake-up lists with executing instructions. The wake-up list of an instruction identifies a small number of instructions that require an operand used and/or the result computed by the instruction for their execution. A design of a microarchitecture, the direct wake-up microarchitecture (DWMA) that implements the wake-up algorithm based upon dynamic construction of wake-up lists has been designed and evaluated fully.

Accomplishments in this area resulted in microarchitecture techniques which out-perform the best non-broadcasting based window implementations such as the *Dependence Based Microarchitecture* (DBMA) by Palacharla et al [50]. Simulation results indicate that with contributions of

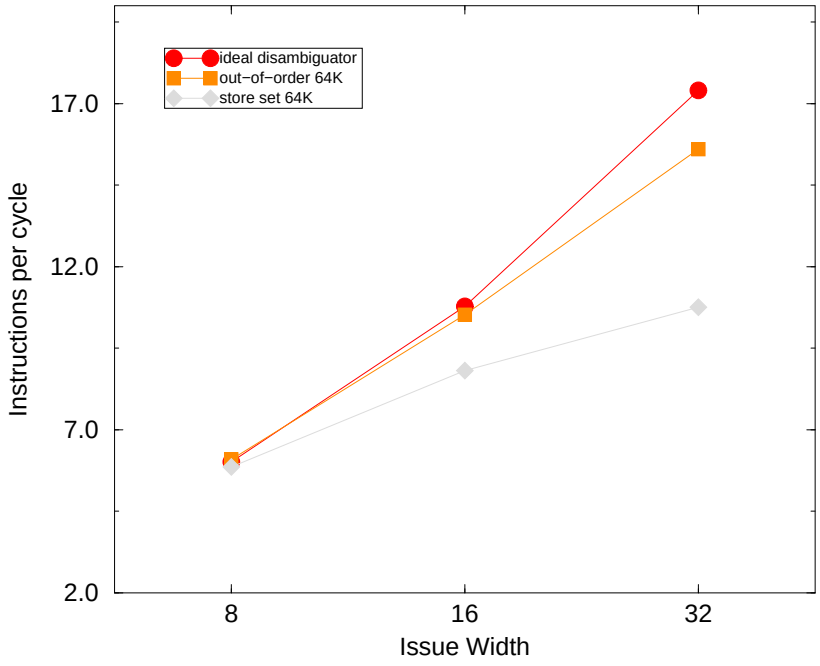


Figure 1.3: Thesis Contribution in the area of Memory Disambiguation

this thesis, we are a step closer to implementing very large instruction windows at the efficiency of ideal window implementations. This performance data is illustrated in Figure 1.2.

Dynamic Memory Disambiguation. In conventional architectures including superscalar out-of-order issue processors data dependencies manifest themselves in two forms. These are dependencies through registers and dependencies through memory. Because of the limited name space (i.e. number of register names), many false dependencies are imposed by the compiler while expressing the semantics of the program. These dependencies can be removed easily by employing *register renaming* techniques since register names are short names which are fully available as soon as the instructions are fetched. On the other hand, dependencies through memory are much more problematic. Memory addresses are not available until the address computation has been performed. In many cases, they are artificially dependent on the completion of other memory operations. Therefore, renaming of the memory operations cannot be done sufficiently early so that performance is not lost. In order to handle this problem, superscalar processors may employ *load speculation*.

Load speculation is a technique that allows a superscalar processor to initiate load instructions before all preceding store instructions perform their address computations so that the effect of artificially imposed dependencies through the memory can be alleviated. However, in order to achieve high performance, load instructions should be held precisely until such time that their issue

will not cause memory dependence violations but not held any longer than necessary [40]. In other words, load instructions should wait for the completion of the store instructions they are dependent on until those store instructions are issued, but not any longer. As a result, for high performance it is essential that an effective dynamic memory disambiguation mechanism be provided.

Recent innovations in the area of memory disambiguation are Moshovos and Sohi's work [39] and the development of a simple and efficient speculative memory disambiguator by Chrysos and Emer [10]. These techniques have shown quite promising results. However, detection of memory order violations when all instructions including the store instructions are allowed to issue out-of-order becomes particularly difficult to handle. As a result, existing algorithms impose an ordering of store instructions in the instruction window. This results in significant loss of performance especially at high issue widths since the processor cannot exploit all the available parallelism because of these artificial dependences.

This thesis introduces a novel memory order violation detection mechanism so that full out-of-order execution can be realized. When applied to the store set algorithm by Chrysos and Emer [10] which is the best performing memory disambiguator to date, the technique out-performs the original technique at all table sizes and in fact produces IPCs which are very close to the values obtained by an ideal memory disambiguator. The performance data for the new technique averaged over Spec95 benchmarks is illustrated in Figure 1.3 where the new algorithm has been labelled *out-of-order* and the original algorithm has been labelled *store set*. These results show that given a high bandwidth instruction fetcher and the improved memory disambiguator, a speculative superscalar processor can uncover significant amounts of instruction level parallelism over a large instruction window.

1.2 Simulation Framework

Computer architecture research is experimental in nature. Within our frame of knowledge, there is no better substitution to analyze the effects of hundreds of parameters and design decisions that may effect performance. As a result, most studies in this area involve significant amount of coding, debugging and simulation activities which are performed repeatedly.

Contrary to other studies, this thesis has followed a largely unexplored approach to the problem of microarchitecture simulation. Observing that many variations of microarchitectural techniques would be needed, instead of hand-coding a simulator and then going through the error-prone process of modifying it many times, a domain specific language called *Architecture Description Language* (ADL) has been designed and its compiler has been implemented [46]. The architecture to be simulated is described in the ADL language, compiled through the ADL compiler to yield an assembler, a disassembler and a cycle level simulator automatically. This is a completely integrated system that provides the desired simulators in a short period of time. The ADL compiler and its host environment have been together named the *Flexible Architecture Simulation Tool* (FAST).

FAST system generates highly efficient detailed execution driven simulators. For a number of simulated architectures, it has been observed that the generated simulators have simulation speeds comparable to those of hand-coded simulators. Typical simulators have been found to be slower than hand-coded simulators by less than a factor of two [46].

1.3 Thesis Organization

As it can be seen, this thesis has made contributions in two related areas of computer science, namely, the microarchitecture research conducted in the sub-field of superscalar processors, and in the programming languages area by developing a domain specific language for microarchitecture simulations in the sub-field of domain specific languages.

The remainder of this thesis is organized as follows. In Chapter 2, a summary of prior work which are related to the topics in this dissertation is presented. Chapter 3 discusses the methodology used for assessing the scalability of existing techniques. In Chapter 4, the DBMA by Palacharla et al. is analyzed in detail and it is illustrated why this solution does not scale well. Next in Chapter 5, the novel idea of *Dynamic Data Forwarding* is presented and its performance is analyzed. Chapter 6, presents the design of the *Direct Wake-up Microarchitecture* and thoroughly analyzes its performance. In Chapter 7, the *Store Set Algorithm* by Chrysos and Emer is presented and its performance is evaluated in detail. It is illustrated that new memory order violation detection techniques are needed to have full out-of-order store instruction issuing. Chapter 8, gives the novel memory order violation detection algorithm developed for this purpose.

Contributions of this thesis in the field of microarchitecture is followed by the contributions in the area of domain specific languages with the presentation of the the *Architecture Description Language* (ADL) in detail in Chapter 9 and its implementation *Flexible Architecture Simulation Tool* (FAST) in Chapter 10. The thesis concludes with a discussion of the accomplishments and future directions in Chapter11.

Chapter 2

Background

In this chapter, a review of the topics that are related to the techniques developed in this dissertation is presented. Since each chapter also includes a brief discussion of the related material, this chapter has rather been organized to be an overview of the related areas. In Section 2.1 techniques that can be used to measure available and exploitable parallelism in programs are discussed. Next, in Section 2.2, a brief history of superscalar processors is presented along with their basic principles of operation. Since dynamic data forwarding and direct instruction wake-up are essentially data-flow techniques, a brief review of dataflow computing is presented in Section 2.3. Although employing these techniques within the context of superscalar processors will not make the resulting processor a dataflow-Von Neumann hybrid, this once quite active research area is relevant and is covered in Section 2.4. Finally, prior work in the area of automatic generation of simulators is discussed in Section 2.5.

2.1 Available Parallelism

Available instruction level parallelism in programs has been the focus of attention since it has been realized that parallel execution of machine instructions is a feasible way to speed-up the execution of sequential programs. Establishing limits of instruction level parallelism is significant as it may yield bounds on attainable performance by instruction parallel machines.

Theoretical modeling of the problem has led to the notion of *program parallelism* (PP) and *machine parallelism* (MP) [27]. According to this definition, machine parallelism is defined as the product of the average degree of superpipelining and the degree of parallel issue. In other words, the machine parallelism is defined to be the maximum number of *in flight* instructions in the execution stages of the processor. Program parallelism is defined to be the average speedup when the program is executed on an infinitely parallel superscalar processor compared to execution on a single issue processor.

According to the model, instruction level parallelism versus machine parallelism curve is divided into two linear regions. In the first region, the machine parallelism is less than the program parallelism and in the second the machine parallelism is greater than the program parallelism. This early model does not explain the rounding of the actual curve in the transition region. Theobald et

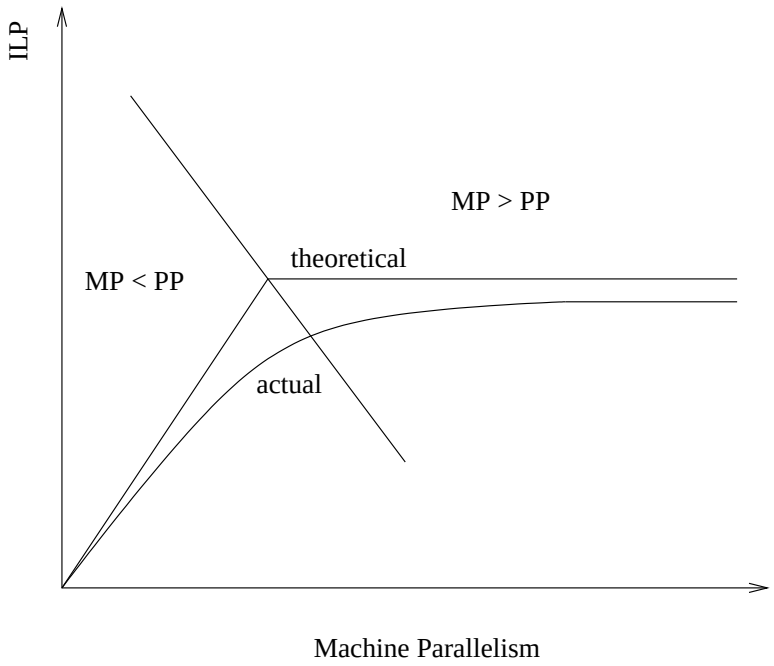


Figure 2.1: Jouppi's Piecewise Linear Superscalar Performance Model

al. introduce the notion of *smoothability* of program parallelism [64] based on the observation that program parallelism is never perfectly smooth. A superscalar processor which has an issue width of n can find more than n instructions ready during some cycles and less than n in others. In this respect, smoothability is defined to be the ratio of the performance with a machine parallelism P to the performance with infinite machine parallelism.

More recently, Noonburg et al. have presented another theoretical model of superscalar performance [44]. In this study, the two techniques, namely, Jouppi's model and the notion of smoothability are combined together by employing a parallelism distribution function which yields better prediction of the actual performance of a given setting.

The theoretical approach is appealing since simulation studies take a long time. It is however difficult to accurately model the machine parallelism that can be obtained using different micro-architectural techniques. Adding to the difficulty are the techniques such as branch prediction, memory dependence prediction and value prediction. Program parallelism that will be found under a infinitely parallel superscalar processor that employs these techniques will be quite different than the processor which does not employ them. Difficulty of modeling these program and data dependent techniques make cycle-accurate simulation still the preferred choice. As a result, most other parallelism studies have been largely experimental. These studies are covered in Chapter 3.

2.2 Superscalar Processors

A superscalar processor is a machine capable of issuing multiple instructions in the same cycle from a single instruction stream. Therefore, superscalar processors fetch and decode several instructions at a time. The outcomes of conditional branch instructions are predicted to supply an uninterrupted instruction stream. Once the data dependencies among instructions are decided, instructions are selected for execution based on the availability of their operands rather than the original program order. An instruction is said to *issue* when it progresses from the fetch stage into the execution stage. By being able to continue issuing instructions even if earlier instructions cannot be issued, a superscalar machine is capable of performing *out-of-order instruction issue*. A generic superscalar that employs separate floating point and integer instruction buffers is shown in Figure 2.2.

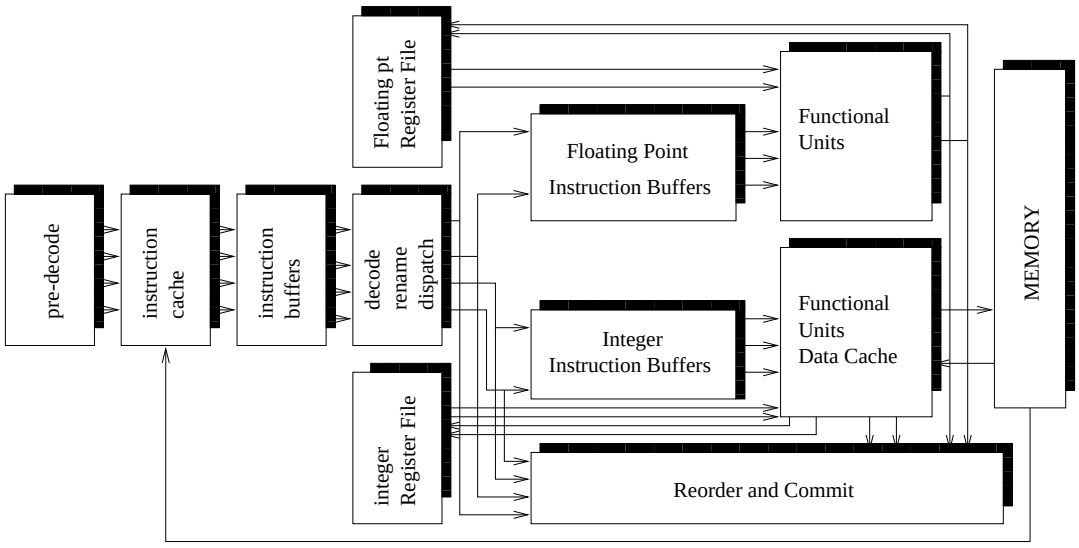


Figure 2.2: A Generic Superscalar Processor

The ability of a superscalar machine to issue multiple instructions is dependent upon the available parallelism in the instruction stream and its ability to look ahead in the instruction stream. The greater the capability of the processor, the better the processor's ability to exploit instruction level parallelism.

Although superscalar processors are thought to be extensions of reduced instruction set processors by many, contrary to the RISC approach of early 1980s, superscalar processors are anything but simple. Unlike dataflow based parallel machines, they rely on complex hardware mechanisms to observe output and anti-dependences. Most widely used technique for this purpose is the *scoreboard* mechanism which was first used in the CDC6600 machine [66]. Another very important processor has been IBM 360/91 which made use of *Tomasulo's Algorithm* [68]. Although both of these machines were limited to issuing a single instruction per cycle, both approaches has played a significant role

in the design of superscalar processors. Tomasulo's algorithm laid out the essentials for out-of-order instruction processing based on data availability in conventional processors.

2.3 Data Flow Architectures

Static dataflow architecture was based on the original design by Dennis and Misuanas [15]. According to Dennis [13], "*the guiding principle has been to design hardware systems that would faithfully implement the semantics of computations expressed as dataflow graphs. The original hardware concept envisioned instruction templates having spaces to receive the operands of the instructions*". In this scheme, a dataflow graph is represented as a collection of activity templates, consisting of an opcode for the instruction, operand slots for receiving operands of the instruction and one or more destination address fields which contain the addresses of other activity templates that should receive the result of the instruction. The machine operates by *firing* an instruction whenever its operands are ready (i.e. tokens are present on all of its input arcs), and there is no token on any of its output arcs. Once the instruction is executed, the machine propagates the computed result to its intended recipients, enabling further instructions for execution.

Being the very first in this field, static dataflow machine has been challenged in many respects. We can group these challenges into two categories, namely, implementability issues and representation related problems. Representation related problems can be listed as the difficulty of representing multiple contexts such as ordinary recursion, and parallel invocation of loop bodies. Moreover, array handling has not been solved satisfactorily. In general, static dataflow machines cannot exploit all the parallelism available in a program. For example, multiple iterations of a loop cannot run in parallel even if there are no loop carried dependences. Implementation related problems were mainly due to the implementation of acknowledgment arcs required by the original design. Acknowledgment arcs limit the parallelism that can be exploited and they double the token traffic.

In order to handle these shortcomings of static dataflow machine, dynamic dataflow machines have been proposed. In order to distinguish between multiple iterations of loops, and different function invocations, activity names are extended to include an iteration count, and the procedure context. A node can be enabled when tokens are present on its input arcs that has the same identifier (i.e. the same iteration count, and the procedure context). This mechanism eliminated the need for acknowledgment arcs, however, brought in other implementation problems. For every token generated, the corresponding node must be found based on the tag of the token. This task requires a search of waiting templates which is achieved by an associative search. Since this storage must both be large and fast, it is not practical to implement this storage using content addressable memories. Therefore, all implemented tagged token dataflow architectures use some form of hashing mechanism to locate the destination slots, thus requiring a very long pipeline. In general, a long pipeline is not preferred for programs that has limited parallelism. In fact, failure of many dataflow machines to execute sequential and vectorizable code as efficiently as conventional architectures has

always formed the basis of criticism for the dataflow architectures. Another significant problem with the tagged token dataflow architectures has been the unpredictability of the amount of parallelism available in the program. If the parallelism is not controlled, a highly parallel program may run until all the resources of the machine are exhausted and then deadlock. Another problem with tagged token dataflow architectures has been the difficulty of deciding a tag size.

The above mentioned problems of tagged token dataflow architectures led to the design of Explicit Token Store architecture (ETS). According to Papadopoulos [52], *the central idea in the ETS model is that storage for tokens is dynamically allocated in sizable blocks, with detailed usage of locations within a block determined at compile time.* Since activation frames are allocated dynamically and explicitly, storage for all tokens will be ready. Mapping of the actual dataflow arcs to these locations is performed by the compiler. A token in ETS consists of an instruction pointer (IP), a frame pointer (FP) and a data value. When a token arrives, the tag bits of the destination is checked. If it is empty, the token's data value is stored in the indicated operand of the instruction. Otherwise, the value is extracted from the location, making it empty and resulting in the instruction being *fired*.

ETS has a very important place in the history of dataflow based computing. Most of the recent work which concentrated on dataflow Von Neumann hybrid solutions uses ideas of ETS. In section 2.4, these hybrid solutions are discussed.

2.4 Data Flow - Von Neumann Hybrids

Current hybrid architectures are usually based on the multi-threading concept and most of them try to eliminate the data fan-out problem by replacing the dataflow aspect of the computation with a Von Neumann style store/fetch mechanism. For example, the *argument fetch* machine proposed by Dennis and Gao [14, 18] is dataflow from the instruction scheduling point of view, while instructions fetch operands from the memory and store computed results into the memory like a Von Neumann machine. This machine later evolved into *super actor machine* (SAM) [23]. Many other hybrids do the sequencing at the thread level based on the availability of operands for the thread but use program counters to execute sequential threads.

One of the first hybrid machines is the Iannucci's hybrid machine [24]. This machine supports a cache memory with synchronization control and a hardware mechanism of processor ready queues for fast context switching. This machine later evolved into IBM's *empire* project, which was later abandoned due to non-technical reasons. ETL in Japan [19] and the Sandia National Laboratories in the United States are working on multi-threaded machines based on the ETS way of handling the storage and the tokens.

A significant step in the introduction of dataflow based multi-threading was the P-RISC idea proposed by Nikhil and Arvind [42]. In a very clear and simple model, they showed how the *two fundamental problems in multiprocessing* [2], namely the latency and the cost of synchronization,

can be handled using a simple model of sequential threads, fork, and join operations. The P-RISC idea continues to be an important one and evolved with the implementation of $*T$ machine [43].

2.5 Automatic Simulator Generation from Specification

To date, the automatic generation of micro architecture simulators from architecture descriptions has been largely unexplored. The notable exceptions are the work of Cook [11] based on a functional programming language called LISAS, *Visualization-based Microarchitecture Workbench* by Diep [16], Larsson's work titled *Generating Efficient Simulators from a Specification Language* [31], the work of Leupers et al. [33] dealing mainly with DSP specific applications, and finally, the work of Ramsey et al. [57] with the *New-Jersey Tool Kit*.

LISAS is used as a specification language for describing instruction set architectures. The language describes instruction formats, and simple register files. Semantics of instructions are implemented through functions. The language is not capable of describing the microarchitecture of the processors.

Visualization-based Microarchitecture Workbench generates an application programs interface (API) for use with the C++ language. The tool generates the API and the programming is done essentially in C++ using the API.

Ramsey et al. have taken the instruction set representation as a general problem. Using the New Jersey Tool Kit, it is possible to rapidly develop system software which deal with the instruction set architecture of the machine such as linkers and assemblers.

Larsson's system can generate a disassembler, and a simulator from a microarchitecture specification. The presented language is capable of describing the instruction set architecture for automatic generation of functional simulators but the language is not capable of describing the microarchitecture. An attempt is also made to generate the assembler, but the automatically generated assembler is a very simple assembler that cannot assemble arbitrary programs. Instead it is used mainly as a debugging tool.

Chapter 3

Exploitable Instruction Level Parallelism and Scalability

There are many studies that have tried to establish limits of instruction level parallelism by either assuming a restrictive processor model or models with unlimited capabilities. Earlier studies in this area mainly assumed restrictive processor models and generally painted a picture that ILP cannot be scaled more than a few instructions per cycle [26, 72]. Later studies incorporated features such as register renaming, perfect branch prediction and perfect caches and reported more optimistic numbers, reaching as high as 60 instructions per cycle [73]. Later, when perfect memory disambiguation mechanisms were considered, IPCs in the order of several thousands were observed [4]. More recently, register renaming, memory renaming, as well as perfect disambiguation and removal of *compiler induced dependencies* through the stack pointer have been considered [55]. All these studies indicate that very large degrees of instruction level parallelism is available in these programs.

These studies however have assumed either a too restrictive processor model or processor models which are not realizable. Examples of restricted processor models include processor models which assume no memory disambiguation capabilities or processors employing the best branch prediction techniques of their time. Examples of unrealizable processors include processors with unlimited issue capability and unlimited look-ahead.

The problem with restrictive processor models is that the established limits of instruction level parallelism through these studies become obsolete quickly. Each successive publication announces higher degrees of *available* instruction level parallelism. Such studies at best provide an indication of the state of the art, and not a limit for the future processors. Unrealizable processor models represent the other extreme. Since these models have never been meant to be realizable, they can only establish the limits on instruction level parallelism imposed by the true dependencies in the benchmark programs. Although it is useful to know that benchmark programs have high degrees of available instruction level parallelism, these studies cannot serve as a baseline for comparing various microarchitecture techniques against each other. As a result, none of the previous studies have been suitable for the goals of this thesis that aims to discover implementable microarchitectural mechanisms which will allow scaling the issue width of the future processors.

In order for a parallelism study to be useful for developing new microarchitectures, the study should be based on a realistic processor model with idealized components. Having a realistic processor model helps innovation on this model to be carried to real implementations. Using idealized components on the other hand establishes performance targets for future innovation in individual components. As newer techniques are developed, idealized components can be replaced with realistic ones and the model can still serve as a performance limit into the future. More significantly, having idealized components allows us to study individual realistic techniques without having side effects. For example, for a configuration which is fetch starved, the claims of good performance of instruction issue logic compared to a central window approach will make sense only until such time that a better fetch mechanism becomes available. On the other hand, a novel instruction issue mechanism evaluated in a machine configuration where all the remaining components such as the fetch and the memory disambiguation are ideal can demonstrate the true potential of the technique and the evaluation of the technique will not become useless with the advances in fetch and memory disambiguation components.

The processor model used in this thesis aims to satisfy the above requirements. It is a *superscalar out-of-order issue processor* with ideal mechanisms. We equip this processor with an ideal fetch unit, an ideal memory disambiguator and an issue mechanism based on the central window model. With continuing advances in microarchitecture research and recent promising results [61, 10] it is only a matter of time before we have realistic techniques for the fetch, memory disambiguation and the instruction selection and issue logic with performances closely following that of ideal implementations. Since this thesis aims to establish such microarchitectural techniques, this model is quite suitable to the goals of the work.

In this chapter, we first present a study of *exploitable instruction level parallelism* in the Spec95 benchmarks by a superscalar out-of-order issue processor. Unlike the prior studies, this study assumes a robust out-of-order issue superscalar processor model with idealized mechanisms. Doing so, the purpose of this study is two fold. First, we would like to demonstrate empirically that there is sufficient exploitable instruction level parallelism in programs such as the Spec95 benchmarks to scale the issue width of superscalar processors. Second, we would like to study the relationship of the processor's issue width to that of the effective window size as prior studies starting with the very early ones [29] have well established that for higher degrees of ILP, processors must posses extensive ability to look forward. Finally, by using the ideal processor model as our reference line we would like to demonstrate that existing techniques for the issue logic and memory disambiguation do not scale well. In the following sections, we *plug-in* each of the best published techniques in these areas into the processor model and leave the rest of the processor ideal. Results obtained indicate that there is a lot of room for improvement in both of these areas.

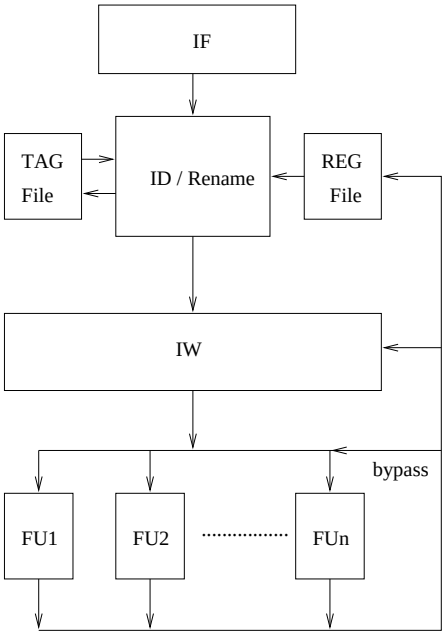
The organization of this chapter is as follows. The realistic idealized processor model used in the study is given in Section 3.1, and its implementation parameters are provided in Section 3.2. The results of the of the experimental evaluation to measure exploitable instruction level parallelism using the Spec95 benchmarks are discussed in Section 3.3. Results obtained in this section serve as

the baseline for the evaluation of the scalability of the instruction issue techniques in Section 3.4 and the scalability of the memory disambiguation techniques in Section 3.5. The chapter is concluded with a brief discussion of the implication of the results of the study.

3.1 Processor Model

A generic superscalar out-of-order processor model employing a central window implementation illustrated in Figure 3.1 forms the basis for the realistic idealized processor model that is used in the studies. The machine model has been derived from Tomasulo's algorithm [68]. By selecting a generic out-of-order superscalar processor model as the basis, we can be fairly confident that the model is implementable with realistic components, but with idealized components it represents the performance limit for the out-of-order superscalar processing technique itself.

For studying the exploitable instruction level parallelism, the processor is equipped with an ideal fetch unit, a central window for scheduling and functional units with realistic latencies. In the following sections properties of each of the main components of the processor are outlined in detail. When a particular technique is to be studied, corresponding ideal component is replaced with the technique being evaluated. The performance of the machine is then compared with the machine that has all the components ideal. This approach exposes the limits of the particular technique that is being studied.



(a) Processor block diagram

Functional Unit	Latency (cycles)
Load	2
Integer division	8
Integer multiply	4
Other integer	1
Float multiply	4
Float addition	3
Float division	8
Other float	2

(b) Functional unit latencies

Figure 3.1: Superscalar Central Window Processor Model (CW)

Instruction Fetch. The fetch unit is a perfect instruction fetcher that fetches a group of instructions upto the processor's issue limit (8, 16, 32 and 64) every cycle and ships them to the ID unit. We provide proper buffering between the fetch unit and decode unit, as well as the decode/rename unit and the instruction window so that any number of instructions can travel from one stage to the next up to the issue limit. The branch prediction is assumed to be perfect, and as long as the ID unit has space to accept instructions, the fetch unit sends the fetched instructions to the ID unit. The ID unit decodes the instructions, renames them and sends them to the instruction window. All result producing instructions are renamed and each instruction is allocated a pair of source registers where the available data can be copied. In this way, all the dependencies except the true dependencies are removed. With this approach, the performance of the machine is dependent solely on the window size, and all other resources are allocated per window entry.

Instruction Window. The instruction window (IW) is implemented as an associative array of reservation stations. If one or more operands for an instruction is missing, the instruction waits for these operands in the IW, until they become available. When all the operands become available or will become available via the by-pass paths by the time the instruction arrives at the functional unit, the instruction proceeds to one of the functional units. There are issue width number of functional units observing the latencies shown in Figure 1(b). At each cycle, upto the issue limit instructions can proceed to functional units for execution. If the number of ready instructions is greater than the issue limit, older instructions are issued before the newer ones, following an oldest-first policy. There are sufficient buses to propagate the results from the functional units to all the destinations without delays. Proper bypassing of results to the functional unit inputs is provided so that a dependent instruction can start execution in the cycle immediately following the cycle in which a result is produced. For example, two integer instructions where one is truly data dependent upon the other will execute in successive cycles.

Functional Units. Functional units are symmetrical fully pipelined units and each can accept a fresh instruction every cycle. The memory subsystem has sufficient number of ports so that port contention is not a problem. An ideal data cache is simulated with unit access time. Load or store instructions may be executed out-of-order as the load store unit does perfect memory disambiguation. We do not however equip this processor with unit latencies. Although there is always the possibility that innovation may reduce functional unit latencies, functional unit latencies have remained relatively constant over the course of last two decades. Assuming unit latencies in the model would shift the processor model towards an unrealizable one, and would adversely effect the results of the studies related to the performance of the issue window techniques.

3.2 Model Implementation

An implementation of the processor has been described for the MIPS ISA using the ADL [46] language and the simulators have been generated from these descriptions automatically. These simulators have then been used to execute the Spec95 integer and floating point benchmarks. Spec95 benchmarks have been compiled using gcc version 2.7.0 with the optimization flags -O3 to generate MIPS code which were then linked with GNU C library version 1.09.1.

Since detailed cycle-level simulations take a considerable amount of time, one common approach is to execute benchmarks until a specified number of instructions are executed. However, in a study employing large windows it is more important to capture the behavior of complete programs. Therefore, the Spec95 test inputs have been used and programs have been executed until completion. However, in a few cases which have intolerably long simulation times, the input data sets have been modified so that a smaller set of data is processed. For example, 099.go plays the game on a 6x9 board, and 104.hydro2d solves a problem of 1/5-th the original size. The instructions per cycle (IPC) figures have been based on the total number of instructions retired and the total number of cycles spent to execute the program (retired/cycles).

3.3 Results of the Parallelism Study

The machine described in the previous section has been simulated across the Spec95 benchmarks for the issue widths of 8, 16, 32 and 64 instructions. For each issue width, the window size has been doubled starting with the issue width until a window size of 8192 instructions, yielding a total of 684 runs. The harmonic means of the Spec95 benchmarks at each issue width/window size combination have been computed and summarized in Figure 3.2. As it can be seen from this graph, very high degrees of instruction level parallelism can be exploited by using an idealized out-of-order issue processor with these programs. More significantly however, at each issue width, the performance gains taper off only after instruction window size reaches to roughly the square of the issue width. It is easy to see that for high performance, the instruction window size must grow quadratically as the issue width is increased. With the provision that the instruction window size is set to at least to the square of the issue width, almost a linear speed-up is possible as the issue width is increased (see Figure 3.3).

We have illustrated that with proper disambiguation, out-of-order issue superscalars can make use of very large instruction windows. For maximal performance, the instruction window size must be at least as large as the square of the issue width of the processor. In other words, the effective window size is a quadratic function of the issue width of the processor. To the best of the author's knowledge, this study is the first such study to establish experimentally that the instruction window of a superscalar processor must grow quadratically in order to provide high performance. Since a central window approach is based on a *broadcast and select* mechanism, it cannot be used

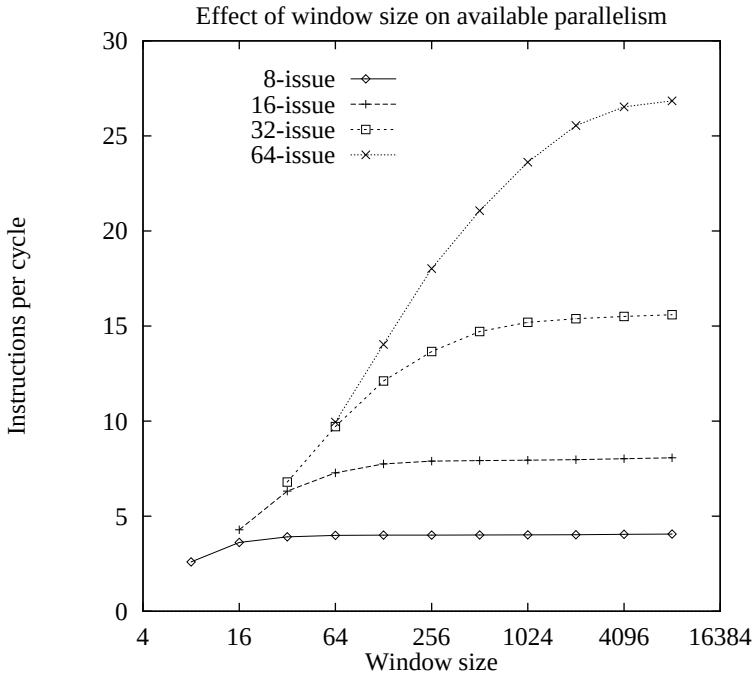


Figure 3.2: Spec95 Performance as a Function of Window Size

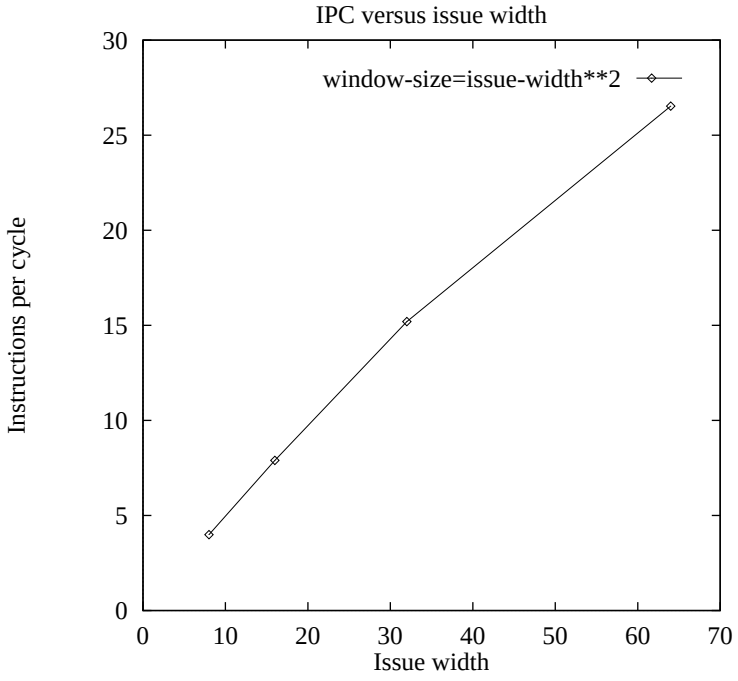


Figure 3.3: Spec95 Performance as a Function of Issue Width

to handle the instruction wake-up requirements of such large instruction windows. We therefore examine alternative means for implementing such large windows.

3.4 Scalability of Instruction Issue Mechanisms

A number of alternative architectures have been considered to address the wake-up problem [63, 62, 71, 30, 50]. Among these, only the approach taken in the design of the dependence based microarchitecture (DBMA) [50] proposes a solution to the wake-up problem in the context of a conventional superscalar architecture that is of reasonable complexity in comparison to a central window processor. DBMA uses a set of FIFOs, equal in number to the issue width, to implement the instruction window. Since the architecture only needs to check the instructions at the heads of the FIFOs, it can buffer a large number of in-flight instructions without increased hardware complexity.

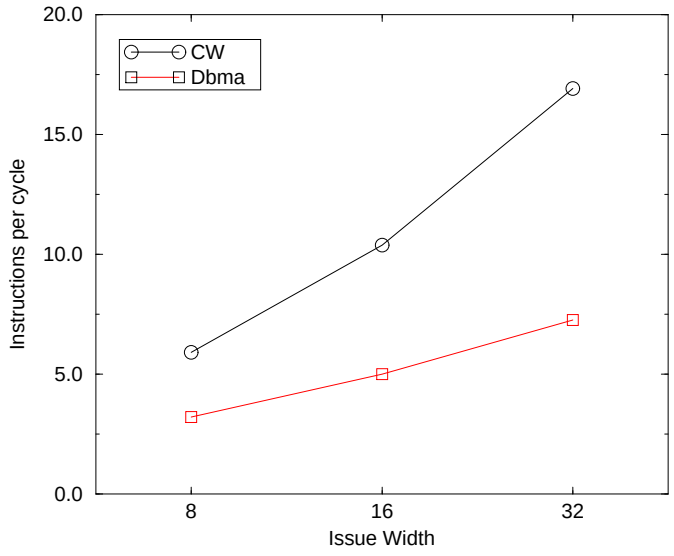


Figure 3.4: Scalability of Dependence Based Microarchitecture

The published results on DBMA indicate that the architecture provides performance very competitive to that of a central window, yet the mechanism has greatly reduced complexity [50]. Unfortunately, the original evaluation used a non-aggressive memory access mechanism that does not perform load speculation. After verifying that published performance of the algorithm can be replicated in our test bed, the algorithm was evaluated using the realistic ideal processor model outlined in the previous sections. In other words, only the central window based issue logic of the processor has been replaced with an implementation of the DBMA. The Spec95 benchmarks were then executed under the DBMA processor and the harmonic mean IPC for the Spec95 suite was computed at issue widths of 8, 16, and 32. It was found that at an issue width of 8, DBMA can

provide about 54 % of the IPC of CW. At an issue width of 16, the performance drops to 48 % and at an issue width of 32 it further diminishes to 43 %. The scalability data for the DBMA processor has been summarized in Figure 3.4. These results clearly indicate that there is a great need for devising a superior instruction wake-up and issue mechanism.

3.5 Scalability of Memory Disambiguation Techniques

In order to get high performance, out-of-order superscalar processors must issue load instructions as early as possible without causing memory order violations. Without the provision of effective memory disambiguation, benefits of providing a large instruction window cannot be harvested. This is because most dependency chains start with a load operation. When the leading load instruction in such a chain is delayed, the whole chain is delayed which results in significant loss of parallelism. To illustrate the point, the performance of three techniques that deal with the scheduling of load instructions were studied. These are, *no load speculation*, *blind speculation* and load speculation based upon the store set memory dependence predictor [10].

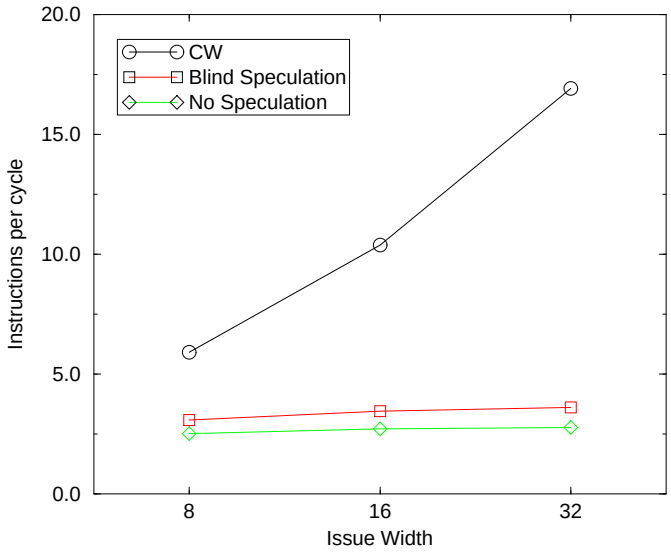


Figure 3.5: Scalability of No Speculation and Blind Speculation Techniques

No Speculation. When a processor employs no load speculation, it checks all prior store addresses against the address of a pending load. When there are no prior stores with a matching address, the load instruction is allowed to issue. On the other hand, if there is an address match and the store data is not ready the issuing of the load is delayed. Depending on the processor implementation, the processor may elect not to issue a load instruction even when the store data is ready in order to reduce by-passing hardware that would be necessary. In these implementations the load instruction

simply waits until the store completes. In this study, it is assumed that when the store instruction has both its address and the data value ready, the load instruction is allowed to issue and obtain its data value directly from the unissued store instruction.

Please note that this scheme requires all prior store addresses to be known before a load instruction can compare its address against the prior store instructions. This is a significant problem with schemes that do not perform load speculation since it results in unnecessary delaying of load instructions because of unrelated store instructions. In a superscalar processor which employs a large instruction window the possibility of having at least a few store instructions with unknown store addresses is quite high. As a result, this scheme can exploit very limited amounts of instruction level parallelism. In most cases, no parallelism across loop iterations is exploited even when the loop iterations are completely independent. The results of simulations for no speculation case is illustrated in Figure 3.5. These results confirm the findings of earlier studies that the load/store parallelism provides little performance gains, and the benchmark programs do not have sufficient instruction level parallelism to issue more than a few instructions per cycle [25].

The performance data shown in Figure 3.5 indicates that no-speculation mechanism can provide only a fraction of the performance of the ideal disambiguator and does not scale as the issue width is increased. At an issue width of 8, no-speculation mechanism achieves about 42 % of the performance of the ideal disambiguator. At an issue width of 16, it can provide only 26 % of the performance of the ideal disambiguator. Finally, at an issue width of 32, a mere 16 % of the performance of the ideal disambiguator is obtained.

Blind Speculation. Observing that too much parallelism is lost because of the strict requirement of prior store addresses to be known, an alternative scheme is to allow issuing of load instructions whenever their register data dependencies are satisfied even when there are prior store instructions with unknown addresses. Since the load instructions are allowed to issue speculatively, store instructions check for *memory order violations* as they are issued. A memory order violation is a violation of a read after write dependency through the memory. When such a violation is detected, the execution is restarted beginning with the load instruction that has obtained the wrong value. Performance of the blind speculation at various issue widths is illustrated together with the performance of no speculation in Figure 3.5. Blind speculation achieves 52 %, 33 % and 21 % of the performance of the ideal disambiguator for 8, 16 and 32 issues respectively.

As it can easily be seen, blind speculation can help boost the performance of the machine, but it also does not scale. As opposed to no speculation technique which looses too much parallelism because of unnecessary delaying of load instructions, the blind speculation looses too much parallelism because of too many memory order violations. It is clear from this experimental data that without the provision of effective load speculation that does not cause frequent memory order violations there is no point in establishing large instruction windows, or increasing the issue width of the machine.

Store Set Memory Disambiguation. Effective load speculation without causing excessive restarts can be accomplished by using a memory dependence predictor to guide instruction scheduling. By caching the previously observed load/store dependencies, a dynamic memory dependence predictor guides the instruction scheduler so that load instructions can be initiated early, even in the presence of a large number of unissued store instructions in the instruction window. A number of such techniques have been developed yielding increasingly better results [21, 40, 39]. More recently, store set algorithm by Chrysos and Emer [10] out-performed all prior mechanisms yielding performance close to that of an ideal disambiguator at an issue width of 8.

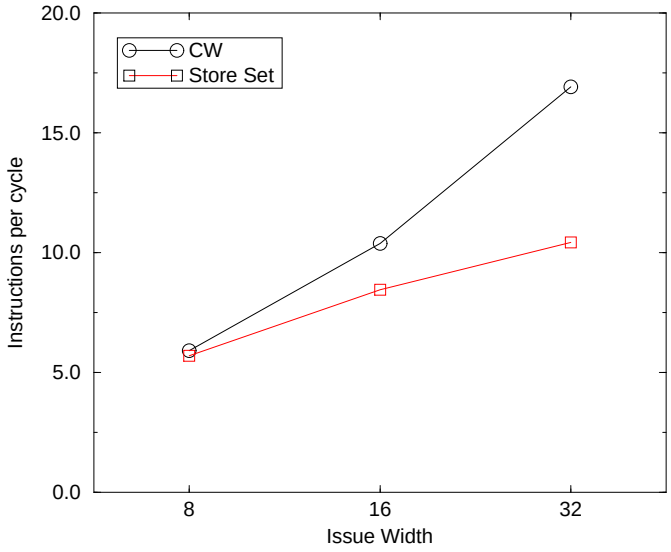


Figure 3.6: Scalability of Store Set Algorithm

This algorithm has been implemented and compared against the baseline processor CW. The performance of the store set algorithm as a function of the issue width is illustrated in Figure 3.6. As it can be seen from this graph, the algorithm closely matches the performance of the ideal disambiguator at an issue width of 8, but starts to loose performance afterwards. The algorithm provides 96 %, 81 % and 62 % of the performance of the ideal disambiguator at issue widths 8, 16 and 32 respectively. As it can be seen, for high performance, an effective memory disambiguation mechanism must be provided. Only with an effective memory disambiguation mechanism, the true potential of out-of-order issue superscalars can be unravelled. In this respect, the store set algorithm provides good memory disambiguation performance upto a certain issue width, but does not scale well afterwards. As a result, there is great room for improvement in this area.

3.6 Concluding Remarks

In this chapter an evaluation of the exploitable instruction level parallelism in the Spec95 benchmarks using a superscalar out-of-order processor model has been presented. Unlike the prior studies, the utilized model is a realistic ideal superscalar model and it is not tied to a particular implementation. This model enabled us to study the true performance potential of each of the evaluated techniques. In the following chapters, the same baseline processor is used to assess the effectiveness of various microarchitectural techniques for the instruction selection and issue, and memory disambiguation techniques.

While it has been illustrated that the existing techniques of memory disambiguation and instruction issue techniques do not scale, the question of why that is the case has not been addressed in this chapter. In Chapter 4, we study the DBMA algorithm in detail and present the reasons for the poor performance of the algorithm. Alternative implementations of the issue window is given in Chapters 5 and 6. Similarly, the store set algorithm is evaluated extensively in Chapter 7 and a greatly improved disambiguator is presented in Chapter 8.

Chapter 4

Evaluation of Dependence Based Microarchitecture

The wake-up algorithm proposed by Palacharla et al. in [50] attempts to reduce the complexity associated with large instruction windows that is demanded by high performance superscalar architectures of the future. The motivation for the design of the wake-up algorithm is the difficulty of scaling a large central window to host many in-flight instructions [49]. The study differs from prior studies in its unique approach to use explicit instruction dependencies for instruction wake-up and scheduling. Nevertheless, as it has been illustrated in Chapter 3, the algorithm's performance is far from ideal with aggressive load speculation using a memory dependence predictor and there is significant room for improvement. In this chapter, a thorough analysis of the underlying reasons for the loss of performance is presented.

The organization of the chapter is as follows. First, in Section 4.1, a detailed summary of the algorithm is presented along with the implementation of the algorithm in the *dependence based microarchitecture* (DBMA). Next in Section 4.2, detailed experimental evaluation of DBMA is presented. In Section 4.3, reasons for the poor performance of the algorithm are analyzed. The performance of the algorithm using state-of-the-art techniques available today is presented in Section 4.4. Finally, a brief discussion of the results is presented in Section 4.5.

4.1 The Wake-up Algorithm

The wake-up algorithm proposed by Palacharla et al. in [50] is based upon the observation that if a set of instructions form a dependence chain, then the wake-up mechanism only needs to examine the first instruction in the chain since the other instructions can never be successfully woken up before the first instruction. Once the first instruction has been woken up, the next instruction in the chain should be considered by the wake-up mechanism. An architecture that exploits this observation, called the *dependence based microarchitecture* (DBMA), was designed and evaluated in [50].

The pipeline of the DBMA microarchitecture is shown in Figure 4.1. This architecture provides a set of FIFOs that decouple the instruction fetch from instruction execution. The dependence

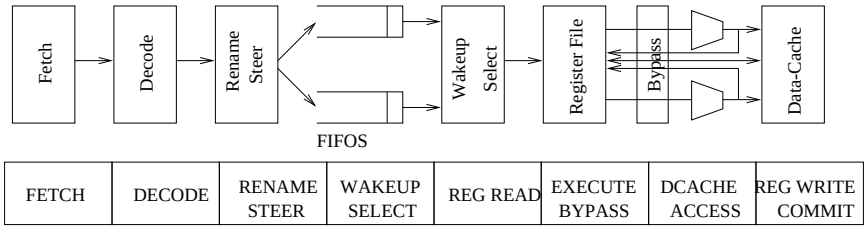


Figure 4.1: Dependence-based microarchitecture

chains are dynamically identified and instructions belonging to a chain are steered into a FIFO queue by the *rename-steer* stage. The number of queues is equal to the issue width. Only the instructions at the queue heads are monitored for operand availability and are thus candidates for being woken up. The availability of an operand is indicated by the setting of a bit in a table called the *reservation table*. The complexity of the wake-up mechanism is proportional to the number of queues, that is, the issue width of the processor.

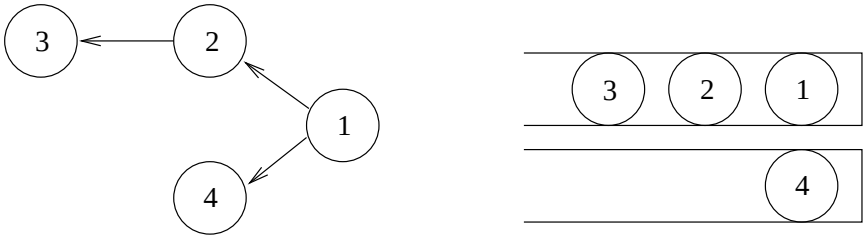


Figure 4.2: Scheduling on DBMA

Let us consider the algorithm for steering instructions into FIFOs in greater detail. Instructions are steered as they are fetched to one of the queues by observing dependencies among instructions. Since dependent instructions cannot execute in parallel, a dependent instruction is steered behind the instruction on which it depends. Multiple instructions may require the same operand value, and thus can be dependent on the same producer instruction. In this case the first dependent instruction is scheduled behind the producer. However, additional dependent instructions are steered to empty FIFOs. By doing so, this heuristic allows all instructions dependent upon the same producer to be initiated simultaneously once the producer instruction makes the operand available. This aspect of the steering heuristic is crucial to its performance. If an instruction cannot be placed in any queue according to the above criteria, the fetching and decoding is stalled till a queue becomes available.

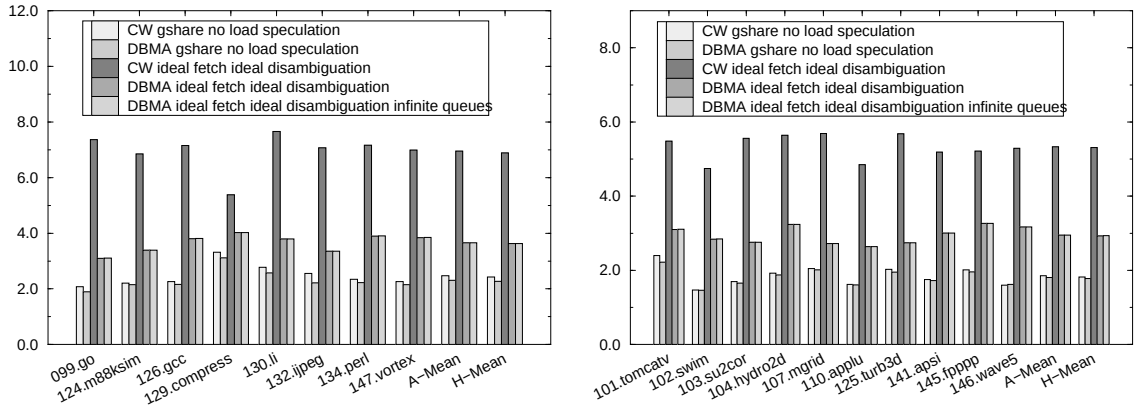
The DBMA scheduling process is illustrated in Figure 4.2. In this example, the processor fetches instructions 1,2,3 and 4 in that order. Dependencies of these instructions are shown in Figure 4.2a. When the first instruction is fetched, it is put into an empty FIFO. When instruction

2 is encountered which is dependent on instruction 1, the instruction is put behind instruction 1. Next, the processor fetches instruction 3. Since instruction 3 is dependent on instruction 2, it is placed behind instruction 2. Finally, when instruction 4 is fetched the processor cannot put it behind instruction 1 as there is already an instruction there. As a result, the new instruction is steered to an empty FIFO.

With the above steering of instructions into the FIFOs, when instruction 1 completes both instruction 2 and 4 will be at the heads of the FIFOs and they can start executing in parallel provided they have only one missing operand. Please note that if a new instruction is fetched that is not dependent on either instruction 3 or instruction 4 the decoding has to stall until an empty FIFO becomes available.

4.2 The Evaluation

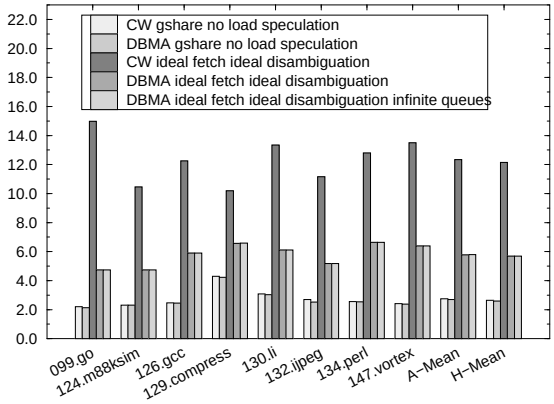
Palacharla et al. carried out an evaluation of the above wake-up mechanism by comparing its performance with a central window implementation. The central window implementation employed the same basic pipeline except for the wake-up and issue stages. They demonstrated that the IPCs obtained by the DBMA microarchitecture were within a few percent of the central window based architecture. The baseline processor used in the above evaluation employed an instruction fetch unit that was based upon McFarling's gshare [37] branch predictor. Furthermore, memory disambiguation was dealt with by issuing a load instruction only after memory addresses of all prior stores were known.



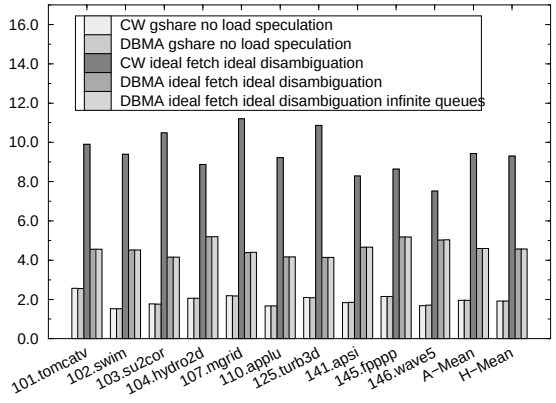
(a) 8 Issue - Integer Benchmarks

(b) 8 Issue - Floating point Benchmarks

Figure 4.3: Performance of DBMA and CW 8-Issue Processors

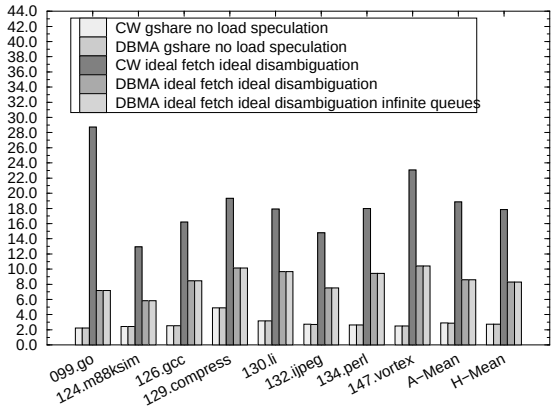


(a) 16 Issue - Integer Benchmarks

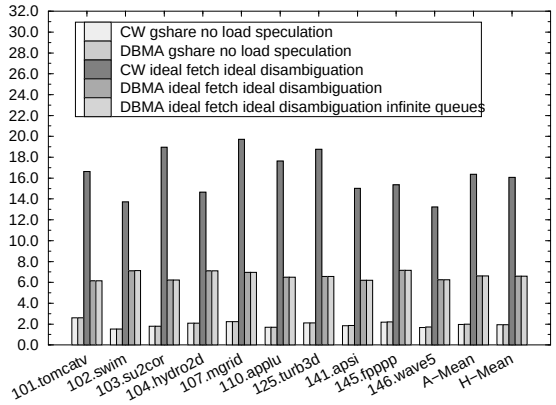


(b) 16 Issue - Floating point Benchmarks

Figure 4.4: Performance of DBMA and CW 16-Issue Processors



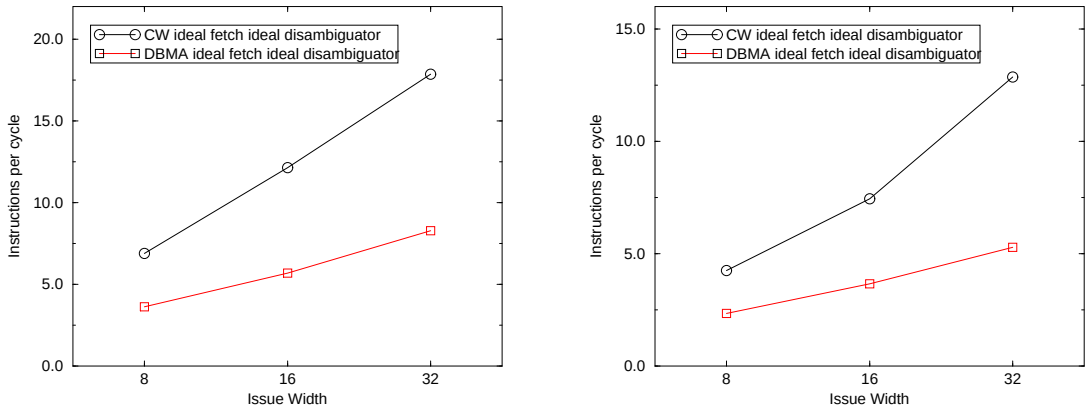
(a) 32 Issue - Integer Benchmarks



(b) 32 Issue - Floating point Benchmarks

Figure 4.5: Performance of DBMA and CW 32-Issue Processors

Using similar processor configurations, the published performance of the DBMA has been successfully replicated. Although not evaluated in [50], the floating point benchmarks have also been evaluated and they showed similar behavior. Also, while the original study has assumed unit latencies, realistic latencies outlined in Figure 1(b) have been utilized in the evaluations. The results of simulations for issue widths 8, 16 and 32 are shown in Figure 4.3, 4.4 and 4.5 respectively. As it can easily be seen, DBMA matches the performance of the central window processor when it is equipped with a regular fetch unit based on the gshare [37] branch predictor and the processor does not employ load speculation (see *CW/DBMA gshare no load speculation* cases). In these figures, IPCs for DBMA are typically within 10% of the central window processor.



(a) Integer Benchmark Harmonic Means

(b) Floating Point Benchmark Harmonic Means

Figure 4.6: Scalability of Central Window and DBMA

However, when dynamic memory disambiguation is employed, the performance of DBMA becomes significantly worse (see *CW/DBMA ideal fetch ideal disambiguation* cases). With most benchmarks, DBMA scores well below CW providing below 50% of the performance of the central window. The overall trend in these cases are further summarized in Figure 4.6(a) and (b). These figures clearly show that as the issue width is increased the performance of DBMA in relation to central window drops even further. The gap between the two architectures is notably larger in case of floating point benchmarks which require larger instruction windows to hide the latency of the floating point operations so that a high IPC can be delivered. Overall, the results of this study show that the DBMA wake-up mechanism typically provides half the IPCs of those achievable by the central window mechanism.

4.3 The Analysis

Next we discuss the reasons for the loss of performance that was observed for the DBMA wake-up algorithm. To uncover high degree of instruction level parallelism, as is needed for wide issue processors, we need to examine an increasingly larger number of instructions [47]. The window of instructions over which parallelism can be detected by the DBMA mechanism can be limited in two situations. If the FIFOs are full, the fetching would be stalled and no more parallelism would be detected. If the instruction steering algorithm requires additional empty FIFOs to proceed, and none are available, it must stall until some FIFOs become empty.

The performance of DBMA assuming that the FIFOs have unlimited lengths was studied. The results of this experiment are indicated by the data point *dbma ideal fetch ideal disambiguator infinite queues* in Figure 4.3, 4.4 and 4.5. As we can see, providing longer queues does not result in any performance improvement. From further analysis it has become clear that the performance of the DBMA architecture was limited by the number of queues. In other words the performance of the design which limits the number of queues to the issue width and uses the proposed steering algorithm, does not scale to high issue widths.

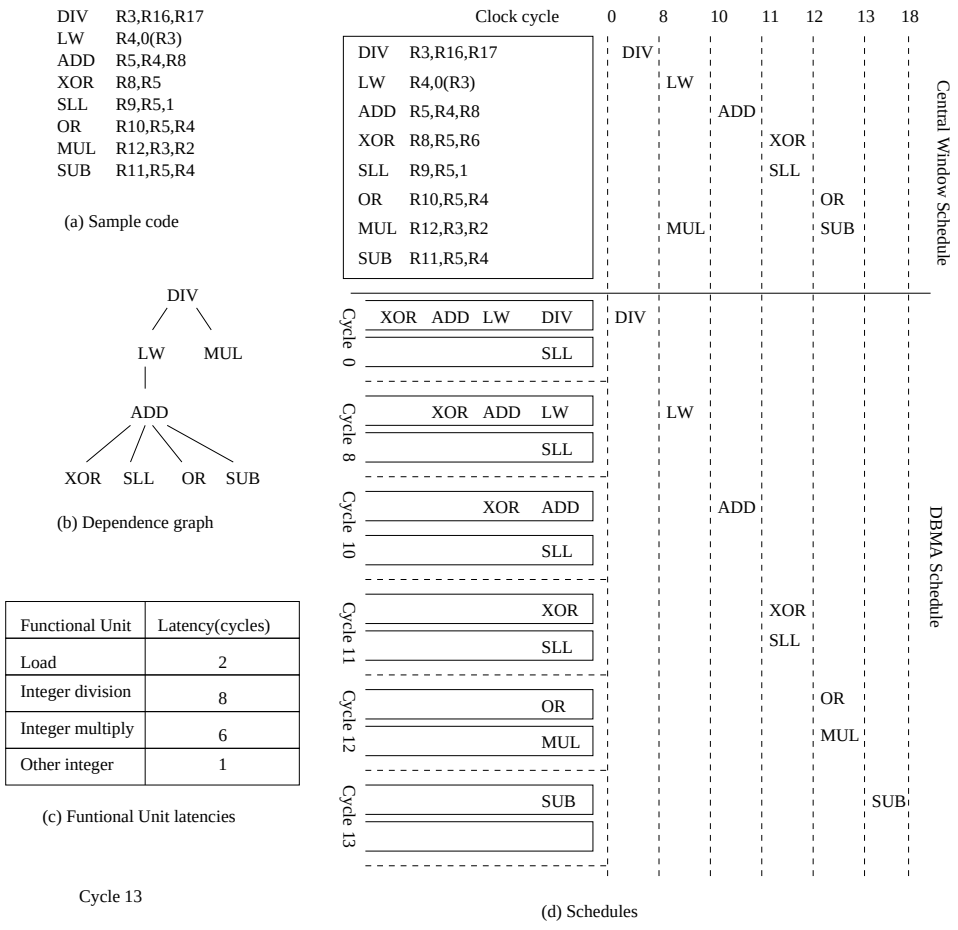


Figure 4.7: Central Window vs DBMA: An Example Schedule.

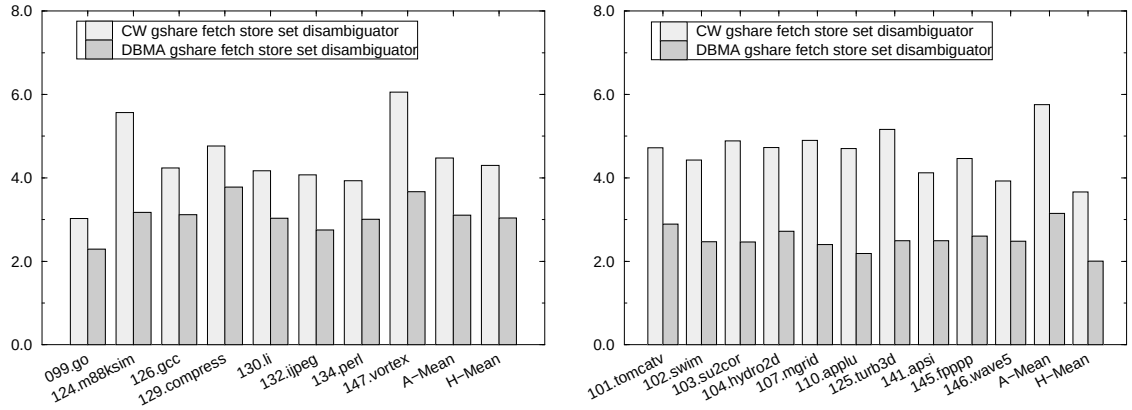
A frequently arising scenario which causes the DBMA to lose performance is illustrated in Figure 4.7. This example also motivates the solution to the wake-up problem which is presented in Chapter 6. Consider the code sequence and its dependence graph as shown in Figure 4.7(a) and Figure 4.7(b). Assuming that no delays are introduced due to fetching of instructions, this code sequence when scheduled on a central window processor of two functional units executes in 13 cycles as shown in Figure 4.7d. The schedule begins with the issuing of the DIV instruction. Upon its completion, at the beginning of cycle 8, the LW and MUL instructions are issued. The completion of the LW enables the issuing of the ADD instruction at cycle 10, and at cycle 11 all the remaining instructions become ready. Assuming that the processor selects and issues the two oldest instructions XOR and SLL first, the execution concludes with the scheduling of OR and SUB at cycle 12.

Now consider the scheduling of the same code sequence on the DBMA which contains two FIFOs corresponding to the two functional units. The first four instructions (DIV, LW, ADD, XOR) are fetched and successfully steered into the first FIFO as they form a dependence chain. The fifth instruction, SLL, is steered to the empty queue since it is dependent upon the ADD instruction in the first queue and the ADD instruction already has a dependent instruction (XOR) behind it. The sixth instruction, OR, is also dependent on the ADD instruction and requires an empty queue. However, since there is no empty queue available, the fetching and steering stalls. At cycle 12 empty queues are available and therefore the OR instruction can be steered to an empty queue and as a consequence now the MUL instruction can also proceed to an empty queue. Even though now the MUL instruction can be issued, its issuing in comparison to the central window schedule has been substantially delayed. Being a long latency operation, the delay in the scheduling of MUL extends the schedule to 18 cycles.

In summary, the above example illustrates that if the DBMA runs out of available queues, fetching of further instructions stalls and thus instruction window over which the DBMA can uncover instruction level parallelism is severely limited. It should be noted that if non-aggressive fetch and memory disambiguation mechanisms had been employed, delays introduced due to them would have slowed down the central window schedule as well and the performance of DBMA and central window would be comparable. This is essentially what led to the observed experimental results presented earlier.

4.4 Performance of DBMA with State-of-the-art Techniques

In Chapter 3, it had been shown that when an effective memory disambiguation is not employed, the set of instructions which can execute in parallel is severely limited. As a result, the instruction issue window cannot extract parallelism regardless of the instruction wake-up and issue technique that is used. In such cases, an inferior scheme may show performance as good as a superior scheme, since the superior scheme cannot utilize its full potential because of other bottlenecks in the processor.



(a) 8 Issue - Integer Benchmarks

(b) 8 Issue - Floating point Benchmarks

Figure 4.8: Performance of DBMA and CW

Since an ideal memory disambiguator cannot be used in a real implementation, in this section we evaluate the performance of the DBMA using the state-of-the-art instruction fetch and memory disambiguation techniques. For this purpose, we utilize an aggressive fetch mechanism based on McFarling's gshare predictor that can fetch multiple blocks every cycle by performing multiple branch predictions. For memory disambiguation, we employ the store set algorithm [10]. Both the CW processor and the DBMA processor with these settings are perfectly realizable today.

The results of the simulations for an 8 issue processor are shown in Figure 4.8. As it can be seen easily, previous results that we obtained using an ideal fetcher and an ideal disambiguator hold, although with some differences. In general, integer benchmarks have lower branch prediction rates than the floating point benchmarks. Because of excessive branch misprediction initiated roll-backs, the central window processor cannot as effectively establish a full large instruction window over which it can exploit high degrees of parallelism. Therefore, DBMA shows better relative performance with the integer benchmarks. On the other hand, floating point benchmarks have both better branch prediction rates and they require exploitation of high degrees of instruction level parallelism in order to yield high IPC values. This is because in order to hide the latency of floating point operations a large number of in-flight instructions is needed. Since DBMA cannot look-ahead as farther as the central window processor if performs significantly worse in case of floating point benchmarks. With these results, it has been demonstrated that the DBMA approach is severely limited with either the contemporary techniques as well as using fetch and memory disambiguation techniques that may be developed in the future.

4.5 Concluding Remarks

Use of the dependence information to guide the instruction scheduling is a significant step towards reduced complexity and increased performance. Unfortunately, because the original evaluation made use of a model which was severely limited, the algorithm's claim of high performance with reduced complexity became false one year after its publication with the advances in memory disambiguation. This event clearly shows the benefits of using an evaluation methodology which makes use of a realistic, yet idealized processor model. These results at the same time show that the so called *window size problem* has not been solved.

In Chapter 5, we develop a novel graph representation for encoding the instruction dependencies that can be dynamically generated from ordinary RISC code. A simple microarchitecture that can make use of such a graph for efficient waking-up of blocked instructions is presented in Chapter 6. It is experimentally demonstrated that use of such explicit wake-up graphs is not prone to the limitations of the DBMA approach, yields high performance and is highly scalable.

Chapter 5

Dynamic Data Forwarding

Contemporary superscalar processors are out-of-order issue processors which do not block instruction issue as long as it is possible. When the processor encounters an instruction whose operands are not yet ready, the blocked instruction is forced to *step aside*, paving the way for the ready instructions to fill-in the pipelines. These blocked instructions are later executed *dataflow style* once the required result becomes available.

As it has already been demonstrated in Chapter 4, the ability of the processor to look ahead as far as possible while communicating the newly produced results to the blocked instructions at the earliest possible time is crucial for extracting high degrees of instruction level parallelism. Ideally, all the instructions waiting for the result should be able to utilize the data value at the beginning of the next cycle. On the other hand, sending the data value to potentially all the instructions in the instruction window requires a *broadcast and select* mechanism. For these reasons, most contemporary superscalar processors implement the instruction window in some form of associative memory where instructions monitor common data buses for the data values they need by matching their tags to those broadcast on the bus [68]. Although this is an efficient mechanism enabling back-to-back execution of dependent instructions, for very large windows the mechanism is not feasible because of the increased hardware complexity [49].

An alternative mechanism to the currently employed instruction wake-up techniques in superscalar processors is to store the blocked instructions in random access memory and to wake-up these instructions through *direct matching* [52]. Since the instructions would reside in ordinary random access memory, it would be possible to implement very large instruction windows efficiently using this technique.

Direct matching has been studied extensively in the context of dataflow architectures by Papadopoulos and Culler [51, 52]. On the other hand, the form of direct matching used in dataflow processors cannot be readily applied in the context of out-of-order superscalar processors. Before direct matching can be used in the context of a superscalar processor, a number of issues need to be addressed:

1. In order to use direct matching, an explicit dataflow graph must be stored and maintained in the processor. In case of dataflow processors this graph is encoded in the instruction stream

whereas superscalar processors utilize instruction sets that make use of implicit communication through register names. Therefore, the dataflow graph must be generated dynamically using register names.

2. Unlike the dataflow processors, most superscalar processors deal with general purpose code that has low to moderate parallelism. This type of code is very sensitive to any delays in the propagation of values.

3. The *data fan-out* problem is a serious problem in a superscalar setting. When the graph is generated dynamically, there is no way of knowing how many destinations a value should be forwarded to in advance. The number of uses of a value is arbitrary and may assume any value between one and the window size. Distributing a result to multiple memory locations rapidly requires many ports which may result in slowing down the clock.

Among the above issues, the solution to the data fan-out problem holds the key to the rest. Without having a fixed number of destinations per instruction, dynamic generation of the dataflow graph from ordinary RISC code is largely unmanageable in the hardware. Similarly, the need to initiate dependent instructions in successive cycles has to be handled as a subproblem of data fan-out.

In order to address these problems, a novel data-fanout mechanism has been developed. This data fan-out mechanism assumes a fixed number of def-use edges and a fixed number of use-use edges per instruction. The fan-out is handled using the def-use edges first. Once the fixed limit is reached, additional uses are satisfied through use-use edges of consumer instructions. In other words, instructions which are being woken-up are used as stepping stones to waking-up further instructions. This form of data forwarding is called *source-to-source* forwarding and a dataflow graph dynamically computed from a conventional instruction stream in this respect is called the *Dynamic Data Forwarding Graph* (DDFG). The semantics of DDFG can easily be implemented using *direct matching* and unlike the prior direct matching techniques [74, 51], the graph does not limit parallelism or introduce additional instructions to carry out data propagation.

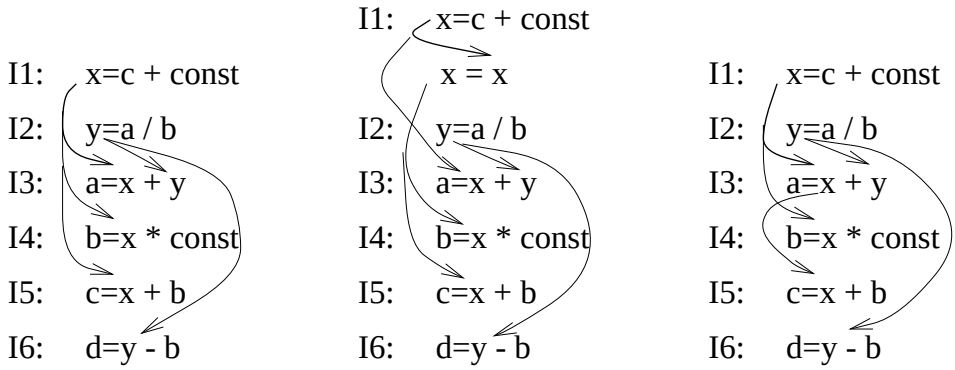
In the remainder of the chapter, in Section 5.1, the existing solutions for the fan-out problem and the reasons they are inadequate for a superscalar setting are discussed and the novel solution of source-to-source forwarding are presented. The dynamic data forwarding graph is introduced formally in Section 5.2, which is followed by the presentation of the execution semantics of the graph in Section 5.3 and the construction algorithm from RISC code in Section 5.4. Finally, in Section 5.5, it is illustrated that the use of a DDFG provides competitive performance to that of a data distribution/wake-up scheme that assumes full fan-out capability such as the central instruction window.

5.1 The Fan-out problem

The fan-out problem is the representation and the implementation of the flow of values from their producers to their consumers. There are mainly three approaches to the problem. These are: (a) Providing varying size destination lists; (b) Assuming a fixed fan-out per instruction and implementing the required fan-out by inserting *identity instructions*; and (c) Assuming a fixed fan-out and blocking the instruction issue if issuing the new instruction will cause the fan-out limit to be exceeded (DTS [74]). Alternative ways of implementing the fan-out has been first demonstrated by Kenneth Todd [67]. Early dataflow machines employed the varying length lists approach (TTDA [3]) whereas later dataflow machines employed the identity instruction approach (ETS [51]).

Using a varying length list is not suitable for superscalar processors since varying length lists are difficult to manage efficiently in the hardware. Similarly, the blocking algorithm (DTS [74]) is not suitable since it severely limits the parallelism that can be exploited. Finally, insertion of identity instructions is not desirable for superscalar processors since the need for a large data fan-out occurs when the instruction window becomes full. Insertion of identity instructions dynamically would fill-up the instruction window, resulting in reduced effective window size and consuming valuable functional unit bandwidth to execute the identity instructions.

The developed solution uses the novel idea of *source operand to source operand forwarding* (SSF). Like ETS, we assume a fixed fan-out, but we give the capability to forward data to the source operands of the instructions as well. When an instruction executes, it sends its source operands to their next uses, as well as its result. Figure 5.1 illustrates the flow of the values of x and y for various techniques.



(a) Varying length (b) Identity (c) SSF

Figure 5.1: Handling of data fan-out

As it can be seen, SSF handles the distribution of data values using a fixed fan-out per operand without introducing additional instructions. However, in its current form, it also restricts the amount of parallelism that can be exploited. This is because the propagation of a data value

is delayed when one operand of an instruction arrives but the other operand of the instruction is missing. Since an instruction is scheduled for execution when both operands are available, the propagation of the available operand does not start until after the other operand is received. For example, in Figure 5.1(c), instruction `I5` cannot start its execution until the long latency instruction `I2` completes and activates `I3`, although `I5`'s other operand would have been available long before the completion of I2. It has been verified experimentally that this case occurs too often and it is detrimental to the performance. Figure 5.2 compares the performance of SSF with a fan-out limit of two and the central window. As it can be seen, the loss of performance is quite large for larger issue widths. In order to eliminate the excessive delay, we need to allow data forwarding to continue when at least one of the operands is available. This approach leads us to the concept of treating the source operands and the instruction itself as independently schedulable entities. We therefore treat the source and result values uniformly and refer to each data value that needs to be propagated an *oplet*.

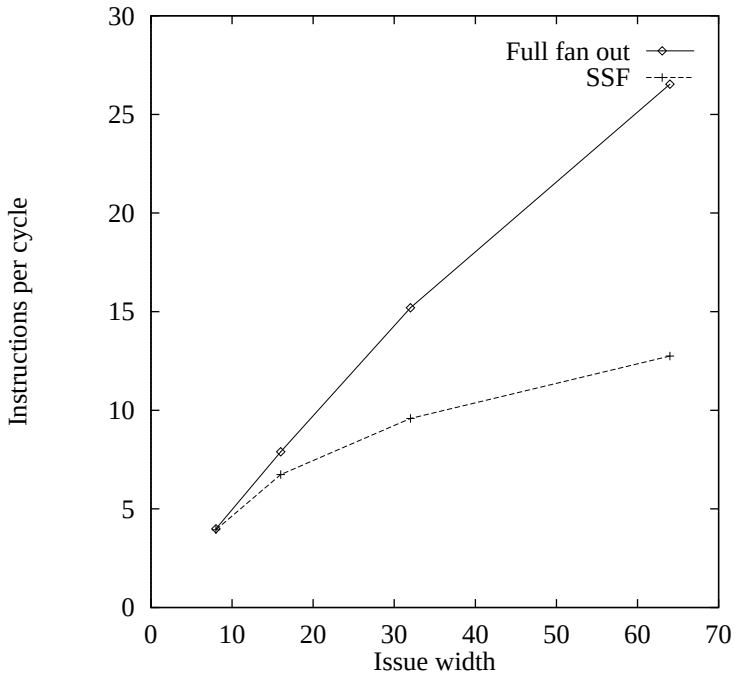


Figure 5.2: SSF-2 versus full fanout

The concept of oplets forms the basis of DDFG. A dyadic instruction has three oplets, while a monadic instruction has only two. An oplet is an executable entity that has a tag indicating the status of the oplet, a value and a number of destinations that need this value. We define the execution of an oplet as the propagation of the value it carries to its destinations whereas execution of an instruction as consuming the input values, performing the operation indicated by the instruction and generating a result oplet. Therefore, the program execution is realized by the execution of the oplets of the graph and its instructions. Like any other dataflow style execution, both oplets and

instructions can only execute when they have the required data values ready. Separation of the instruction operands into oplets has two significant consequences. It allows oplets to have their own *life time*. Therefore, an oplet receiving a data value can propagate it further without delays, given sufficient hardware resources. In other words, data flows freely through the forwarding edges. On the other hand, separate data operands require additional links called *matching links* for joining them later. Revisiting the example in Figure 5.1(c), we observe that the left operand of the instruction `I3` can propagate the value to instruction `I5` although the instruction `I3` itself was still blocked waiting for the value of `y` from the divide instruction. As a result, the execution of `I5` can start as soon as `I4` is completed, while `I2` is still executing.

5.2 Dynamic Data Forwarding Graph

We now define the *Dynamic Data Forwarding Graph* (DDFG) formally.

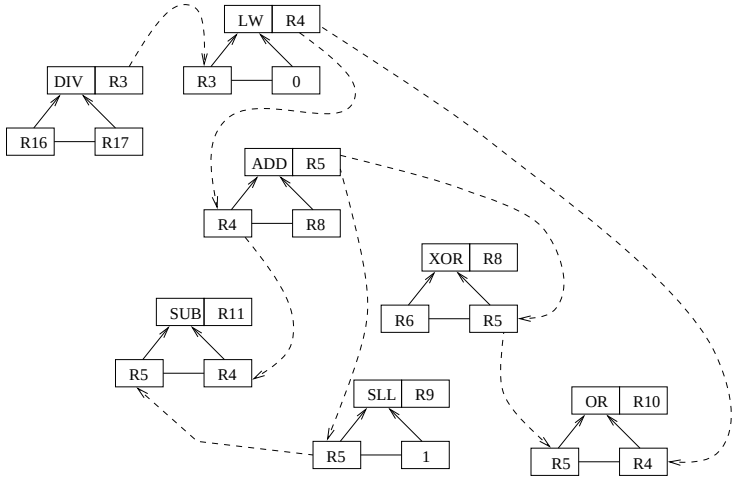
Definition: A DDFG is a directed graph $G = (V_{oplet} \cup V_{operation}, E_f \cup E_{match})$ where $V_{operation}$ is the set of nodes representing program instructions, V_{oplet} is the set of nodes representing instruction oplets, E_f is the set of data forwarding edges, and E_{match} is the set of matching edges between the instructions and their oplets.

Definition: A DDFG has a *forwarding degree* F_d iff $\forall V \epsilon V_{oplet}, outdegree_f(V) \leq F_d$ where $outdegree_f$ is the number of forwarding edges emanating from V.

The DDFG deals with only those instructions which are in the instruction window and fully specifies the data driven execution in the instruction window. An example DDFG which has a forwarding degree of 2 is shown in Figure 5.3. For the purposes of illustration, oplet nodes have been labeled with the register identifiers of the original program code.

```
I1:  DIV   R3,R16,R17
I2:  LW    R4,0(R3)
I3:  ADD   R5,R4,R8
I4:  XOR   R8,R6,R5
I5:  SLL   R9,R5,1
I6:  OR    R10,R5,R4
I7:  SUB   R11,R5,R4
```

(a) Code



(b) DDFG

Figure 5.3: Sample code and its DDFG

There are many possibilities in the assignment of the forwarding edges in a DDFG, each leading to different instruction schedules. For the construction of the forwarding edges, a simple heuristic is used which works well. For each use of a value in the program order, the uses are assigned to producer oplets breadth-first. This approach favors older instructions in the window, and rapidly distributes a value to its destinations since each level in the graph can distribute a value to increasingly more destinations.

5.3 DDFG Execution

The execution of DDFG is data-driven. An operation node can execute when it has both data values (assuming it is a dyadic operation), and an oplet can execute when it has one data value. Arrival of a data value at an oplet triggers the following sequence of events:

1. If the oplet has out-degree greater than zero, a copy of the data value is produced for each link and sent through each link.

2. A copy of the data value is sent through the matching link to the operation node and the oplet is deallocated.

3. Upon having both data values, an operation node performs the operation and sends the result value to the adjoining result oplet.

4. The result oplet sends the value through its forwarding edges and both the operation node and the result oplet are deallocated.

5.4 DDFG Construction

A DDFG is easily constructed as the program executes using an algorithm similar to renaming. For this purpose, an array of queues whose size is equal to the number of architectural registers is needed. Each queue entry holds a descriptor consisting of a counter and a pointer that points at the producer oplet which will have the value of a given register. Fetched instruction's source register identifiers are used to access the queue array to select the set of producer oplets for this register value. If the corresponding queue is empty, there are no pending values for this register and the value is provided to the instruction from the register file. Otherwise the descriptor at the head of the queue identifies the producer oplet that must be used for this consumer. A forwarding edge is set-up from the producer oplet to the current instruction oplet and the counter of the descriptor is incremented. If the value of the counter is greater than the degree of forwarding, the entry is removed from the queue. In any case, a new descriptor is formed which identifies the current instruction's oplet as a new producer and the descriptor is inserted to the tail of the queue. Once all the operands of an instruction are processed, a new descriptor is created for the result oplet,

the queue corresponding to the result register is flushed and the new descriptor is inserted into the queue.

For a DDFG which has a degree of forwarding of one, the queue array becomes unnecessary and a table of size equal to the number of architectural registers is sufficient to generate the graph. When the size of each queue is unlimited, a balanced DDFG is obtained. While a balanced DDFG is ideal for the distribution of the values, in practice generated DDFGs will be unbalanced because of the limited queue sizes. It has been observed that the queues must be as large as the degree of forwarding for achieving high performance but returns diminish rapidly beyond this size.

5.5 DDFG Performance Evaluation

The performance of the DDFG compared to full data fan-out has been studied with degree of forwarding of 2 and degree of forwarding of 4. Data produced using a degree of forwarding of 2 is labelled DDFG-2 and degree of forwarding of 4 is labelled DDFG-4. When we examine the results for 8 and 16 issue processors (see Figure 5.4) we observe that a forwarding degree of 2 captures most of what can be extracted with broadcasting.

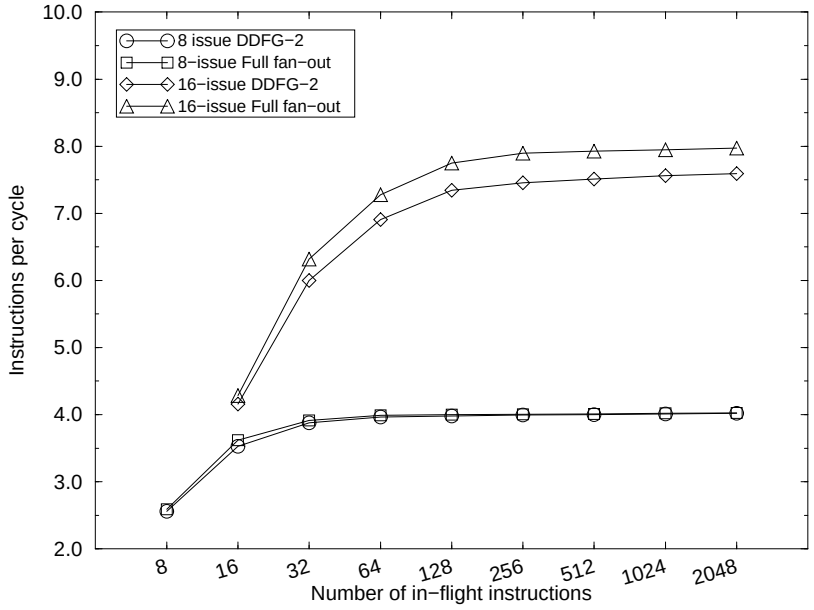


Figure 5.4: DDFG versus Full Fan-Out 8 and 16 Issue Processors

For example, at 256 in-flight instructions, the full data fan-out is only 6 percent better than DDFG-2. Given the cost of increasing the degree of forwarding, the payoff is very little for forwarding degrees greater than 2 at these issue widths. However, at bigger issue rates such as 32, the effective window size rapidly increases. For 32 issue the effective window size is around 1024 entries although a window size of 2048 still provides some measurable performance improvement. For 1024 in-flight

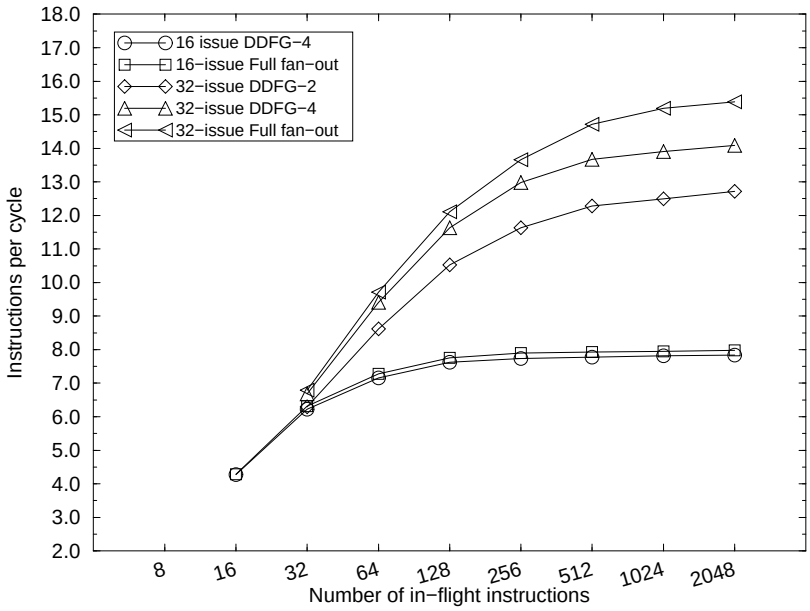


Figure 5.5: DDFG versus Full Fan-Out 16 and 32 Issue Processors

instructions, full data fan-out is faster than DDFG-2 by about 21 percent, and faster than DDFG-4 by about 9 percent. More significantly, the required degree of forwarding increases slowly and a linear increase in the forwarding degree with respect to the issue width always matches the performance of a processor equipped with full data fan-out capability.

5.6 Concluding Remarks

It has been shown that full data fan-out for high performance is not necessary. It has been shown further that direct matching can be feasibly implemented within the context of a superscalar by using the novel approach of generating a DDFG from a conventional instruction stream as part of the renaming process. Using a fixed data fan-out per instruction, this graph can deliver high performance and can be used to implement very large instruction windows.

In Chapter 6, it is illustrated how the DDFG semantics can be feasibly implemented by circulating simple descriptors which represent oplet/instruction pairs. A high performance scalable microarchitecture that utilizes the DDFG is described in full detail and its performance is evaluated.

Chapter 6

Direct Instruction Wake-up

In Chapter 5, it has been illustrated that for high performance full fan-out is not necessary and a Dynamic Data Forwarding Graph DDFG can be used to address the scalability of the issue window in out-of-order issue superscalars using direct matching. Although the DDFG paradigm has been designed around the concept of the flow of values from producer instructions to consumer instructions, an implementation of the concept does not have to rely on explicit movement of data. As it has been demonstrated by Gao et al. [18], instead of propagating data values, signals can be propagated to wake-up waiting instructions and a conventional register file can be used to supply the data values.

In this chapter, a novel instruction wake-up algorithm that dynamically associates explicit wake-up lists with executing instructions according to the dependences between instructions is presented. This technique leads to easy integration of the wake-up mechanism into a conventional superscalar pipeline. The wake-up graph is generated dynamically as part of the renaming process by storing the graph nodes directly in the reorder buffer. It then is utilized to guide the instruction scheduling.

The organization of the chapter is as follows. In Section 6.1, the concept of using a direct wake-up graph DWG for instruction scheduling and the semantics of the graph are discussed. In Section 6.2, a detailed design of a microarchitecture called *Direct Wake-up Microarchitecture*, DWMA which utilizes a DWG for instruction scheduling is presented. In Section 6.3, the algorithm for the generation of the wake-up graph utilized in the DWMA is given. In Section 6.4 the details of the instruction scheduling as it is implemented in the DWMA is presented. In Section 6.5 the performance of the DWMA architecture is analyzed and compared with that of the realistic ideal model. Finally, in Section 6.6 the implications of the approach for future processors is discussed.

6.1 Direct Wake-up Graph

Contrary to the approach taken in a central window implementation where instructions are constantly monitored for readiness, direct wake-up algorithm identifies and considers for wake-up a fresh subset of waiting instructions from the instruction window in each cycle. If in the current cycle an instruction is expected to complete its execution, then another instruction waiting for the result

of the completed instruction may become ready in the next cycle. Thus, such an instruction will be among the set of instructions examined for wake-up and if ready, it would be issued for execution in the next cycle.

In order to guide this wake-up heuristic, a *wake-up* graph is constructed and maintained dynamically. Since the graph is to be maintained in hardware, it is subject to the same constraints that were imposed on the DDFG graph. In particular, the number of edges emanating from a node cannot be unbounded. In the hardware implementation the wake-up graph is represented by associating with each instruction an explicit wake-up list, W-LIST, which identifies the instructions that will be woken up by that instruction (i.e., the wake-up list is the representation of edges in the graph).

Ideally in the W-LIST of an instruction we would like to place all the instructions which use its result (i.e., all instructions to which there is a def-use edge). But since the number of entries in the W-LIST must be a fixed small number in a practical implementation, we cannot include all such instructions in the W-LIST. We therefore follow the same approach that has been used to construct a DDFG. We allow an instruction that computes a result to directly wake-up only a small fixed number of instructions in the instruction stream that use the result while the others are woken up indirectly. The responsibility of waking up future instructions in the instruction stream is delegated to those being woken-up directly through the def-use edges. To carry out the wake-up of further instructions we introduce use-use edges first from the instructions that are being directly woken-up and then from the instructions which are being woken-up indirectly through the use-use edges. This process of introducing use-use edges is repeated till all instructions have been handled. Thus, the wake-up graph contains both def-use and use-use edges. In summary, an instruction can wake-up instructions connected through both def-use and use-use edges. In this implementation, it is assumed that there are at most six edges in the W-LIST of an instruction. These are labelled (d0,d1), (l0,l1) and (r0,r1) corresponding to the result, the left operand and the right operand on an instruction.

It is important to note that the instructions that are linked through a chain of use-use edges may execute in an order different from the order they appear in the chain. Consider a chain of use-use edges that has been created due to an operand value v. When the value v is computed by its producer, the first instruction in the chain is considered for wake-up. Let us assume that the instruction cannot issue because its other operand is unavailable. In this situation, since the value v is available, we will consider the next instruction in the chain for wake-up even though the previous instruction is still waiting. It is possible that this instruction is found to be ready and thus may execute prior to the preceding instruction in the chain. Thus, the introduction of use-use edges does not restrict out-of-order execution. Its only consequence is that all instructions waiting for the same operand are not examined simultaneously by the wake-up algorithm. Rather in each cycle only a subset of instructions waiting for the value are examined.

Clearly the wake-up graph that is constructed dynamically is influenced by the order in which the instructions appear in the code sequence. In addition, the form of the graph is also influenced by the timing of instruction execution. In fact while the initial form of the wake-up graph

is determined by the original instruction ordering, the graph undergoes transformations determined by execution timing. Let us consider the modifications to the graph that may take place. When an instruction is first added to the wake-up graph, it is linked to the producer of its first unavailable operand, that is, it is added to the W-LIST of the producer of this operand. Now let us assume that this operand becomes available. This will cause the instruction waiting for the value to be considered for wake-up. At this point it may be determined that the second operand of the instruction is unavailable. In this case the instruction would be added to the W-LIST of the producer of this missing operand. In other words, while one def-use edge has been removed, another one is added to the wake-up graph.

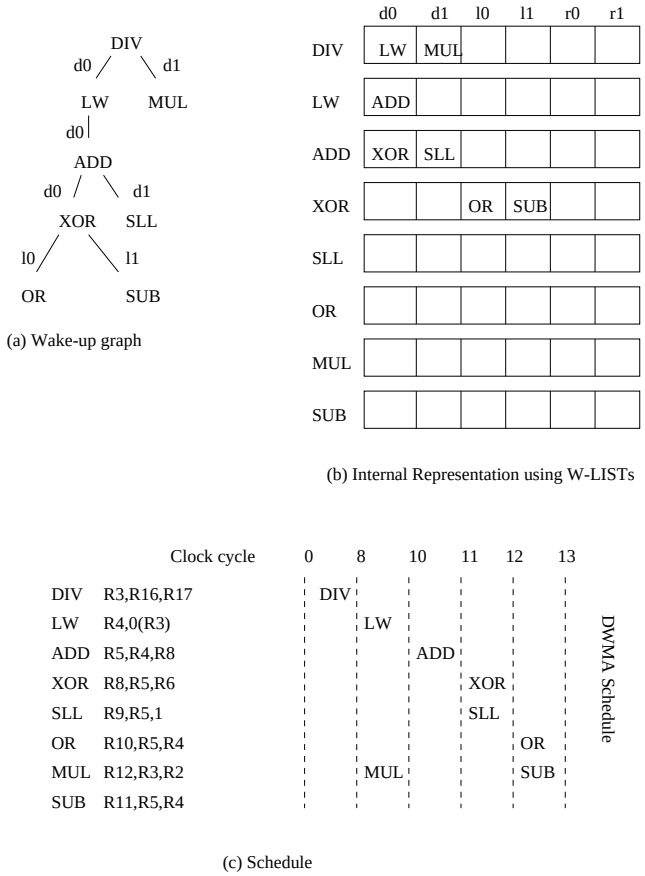


Figure 6.1: Example wake-up graph and its schedule.

Finally we would like to make the observation that an instruction may be woken up 0, 1 or 2 times before it can execute. If the instruction has all of its operands available when it is first fetched, it is issued immediately and thus never woken up. If the instruction has only one operand missing when it is first fetched, then it will be woken up once when that missing operand becomes available (either immediately using a def-use edge or in a delayed fashion through a use-use edge). If both of the operands of an instruction are missing when it is fetched then it may be woken up twice,

once when the first operand becomes available, and then again when the second operand becomes available.

Wake-up Graph Example. Now let us consider the example that has been used in Chapter 4 to demonstrate that the dependence based microarchitecture has limited look-ahead. When we examine the code sequence (see Figure 6.1), we see that the wake-up links for all the instructions except OR and SUB can be established using def-use edges, that is using d0 and d1. This is because the ADD instruction has four children and therefore two result links are not sufficient. We therefore associate XOR and SLL, the first two instructions, directly with the result of ADD through def-use edges, whereas we satisfy the remaining two children OR and SUB using use-use edges emanating from XOR. The resulting wake-up graph is shown in Figure 6.1(a).

The execution schedule is realized as follows. When the DIV instruction completes its execution, it wakes-up the instructions LW and MUL through its def-use edges d0 and d1. When LW completes its execution, it wakes up the ADD instruction through its d0 edge. Once ADD is completed it activates the XOR and SLL instructions also through d0 and d1 edges. When XOR is activated, it can now activate OR and SUB through its l0 and l1 links. The resulting schedule is given in Figure 6.1(c). As we can see, in this case, the schedule generated using the direct wake-up algorithm is of the same length as that generated by the central window algorithm.

6.2 The Direct Wake-up Microarchitecture

Next the direct wake-up microarchitecture (DWMA) that has been designed to dynamically construct the wake-up graph and make use of it for instruction scheduling is presented. As shown in Figure 6.2, DWMA is a highly parallel decoupled superscalar with an eight stage pipeline. The construction of the wake-up graph is begun in the Decode and Graph GEN-1 stage and completed in the Rename and Graph GEN-2 stage. The graph constructed is stored in the form of W-LISTs associated with instructions in their respective reorder buffer entries. Since all instructions that have entered the processor pipeline must have a reorder buffer entry, the size of the instruction window from which instruction level parallelism is extracted is limited by the reorder buffer size. The subset of instructions that are to be considered by the wake-up mechanism are fetched and examined from the reorder buffer by the Fetch W-LIST and Wake-up stage while the updates to the wake-up graph are performed by the Register Read and W-LIST WB stage through write backs to the appropriate entries in the reorder buffer.

Each stage has a width equal to the issue width. The instructions that are executed in parallel are obtained from the heads of the FIFOs and there is a one-to-one correspondence between the FIFOs and the functional units. The instructions are put into the FIFO in the form of instruction descriptors which, in addition to the usual opcode and physical register numbers needed for the instruction's execution, also contain additional information required to carry out the wake-up activities associated with the instruction. Each instruction's descriptor is entered into the queue

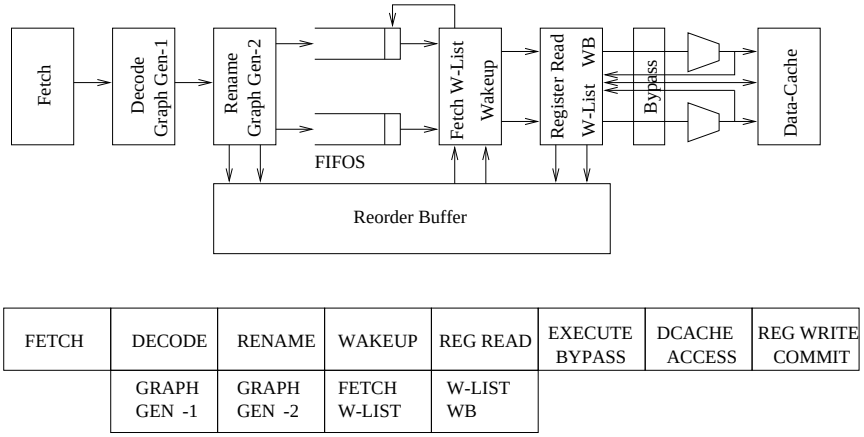


Figure 6.2: The Direct Wake-up Microarchitecture

when it is first fetched and every time it is woken up by another instruction. Thus, the FIFOs contain only a subset of instructions from the instruction window. In particular, these are the instructions that according to our wake-up heuristic may be ready for execution.

Let us consider the operation of the key pipeline stages that construct and process the wake-up graph as well as the reorder buffer that holds the graph in greater detail.

Reorder buffer: The reorder buffer is issue-width interleaved and its address space is divided equally among the individual FIFOs. In other words, the low order bits of the reorder buffer addresses also identify the queue that will process the instruction. The instructions are allocated a reorder buffer entry on a circular basis to distribute the work more or less uniformly to individual queues. In other words, the reorder buffer index allocated to the instruction modulo the number of functional units yields the functional unit that will execute this instruction.

Decode and Graph GEN-1: In this stage, the instruction is decoded and in parallel the producers' W-LISTs, to which the current instruction must be added, are identified from the source register identifiers. A reorder buffer entry is allocated to the instruction by incrementing the tail pointer of the reorder buffer.

Rename and Graph GEN-2: The source register identifiers of the instruction are renamed to physical registers and a result register is allocated if necessary. In parallel, the current instruction and its W-LIST are written to the instruction's assigned reorder buffer entry. The instruction is assigned to a functional unit (or FIFO). The assignment is carried out in a round robin fashion and thus can be computed as the reorder buffer index modulo the issue width. An instruction descriptor is formed and sent to the appropriate FIFO.

Fetch W-LIST and Wake-up: The reservation table is accessed to determine whether the operands of the instruction descriptors at the heads of the FIFOs are ready. In parallel, the reorder buffer is accessed to fetch the W-LISTs of these instructions. The instructions corresponding

to (l0,l1) are woken up if the left operand is available, the instructions corresponding to (r0,r1) are woken up if the right operand is available, and the instructions corresponding to (d0,d1) are woken up if the instruction takes a single cycle to execute. The ready woken up descriptors are pushed back to the heads of the appropriate FIFOs. The current instruction descriptors, that were initially obtained from the heads of the FIFOs, have now been processed. The ones that are found to be ready and have their functional unit free are sent to the next stage. On the other hand if the functional unit is busy, the ready descriptors are pushed back to the head of their respective FIFOs. If the instruction is not ready, then it is simply forwarded to the next stage.

Register Read and W-LIST WB: If the current descriptor is ready, the operand values are read from the register file. If it is not ready the descriptor is written back to the W-LIST of the producer of the missing operand in the reorder buffer.

As we can see from the above description, processing of W-LISTs is done in parallel with the conventional functions of the pipeline stages Decode, Rename, Wake-up and Register Read. Therefore the additional tasks necessary to implement the direct wake-up are juxtapositioned with the conventional pipeline functions in Figure 6.2.

We have discussed the overall operation of DWMA. In the remaining sections, the detailed algorithm for generating the wake-up graph is presented and how the instruction scheduling is carried by DWMA is discussed in greater detail.

6.3 DWG Generation Algorithm

For the purpose of graph generation, an array of queues with each logical register being associated with a single queue is utilized (see Figure 6.3(a)).

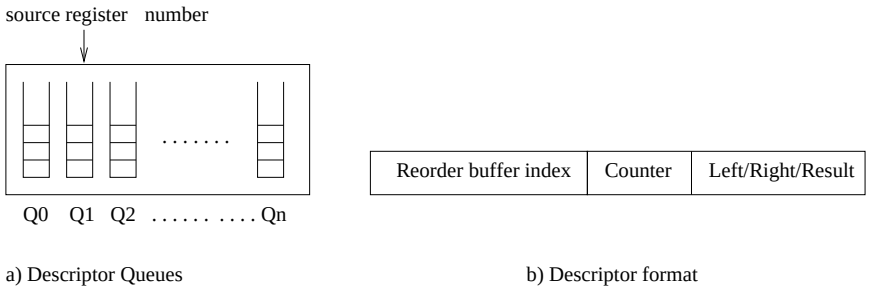


Figure 6.3: Descriptor Queues Used for the Graph Generation

Each entry in a queue is a descriptor identifying a producer for the register's data value. A descriptor contains three fields, namely the reorder buffer index of the producer, the wake-up group of the producer (left operand, right operand, or result) and a one bit counter (see Figure 6.3(b)). When an instruction is being decoded, its source register identifiers are used to access the producer

queue array and the descriptor at the head of the queue is copied to the instruction's corresponding operand. The descriptor's counter is incremented, and if it now overflows, the descriptor is removed from the head of the queue. Otherwise, the updated descriptor is left at the head of the queue. **This process ensures that, for any given counter size, there are at most that many edges emanating from the instruction corresponding to each of its operands and its result**. Two new descriptors are formed, one for each operand which can now serve as new producers of these values and they are inserted at the tail of the corresponding queues. Please note that if there is no space left in a queues, the instruction decode need not stall but can simply discard the new descriptors it just formed since there are already quite a few producers that can be used as wake-up slots by the following instructions that need the same value.

Lets consider the ADD instruction in the code sequence shown in Figure 6.1. When this instruction is fetched, it creates a descriptor for the result register R5, sets its counter to zero and sets its wake-up group to *result*. The queue corresponding to register R5 is flushed and this descriptor is inserted into the queue. When the XOR instruction is fetched, it indexes the queue array for its source register R5 and gets the descriptor. Since its counter is zero, there are available def-use links. The instruction establishes the link by copying the current descriptor, incrementing its counter and re-inserting the descriptor back to the head of the queue. XOR instruction now creates a new descriptor where the wake-up group is set to *left operand*, initializes its counter to zero and inserts the descriptor to the tail of the queue. When the processor fetches the SLL instruction, the above process is repeated. However, since the counter of the descriptor for R5 now overflows, instead of re-inserting the descriptor back to the head of the queue, the descriptor is discarded. At this point, we have consumed all the available def-use edges, but we have two new descriptors in the queue which are of type use-use edges. When the processor fetches the OR and the SUB instructions, the above process is repeated. This process essentially makes these two new instructions children of the XOR instruction through use-use edges.

6.4 Instruction scheduling

DWMA implements the instruction scheduling uniformly by propagating instruction specific information in the form of instruction descriptors which are hardware pointers uniquely identifying the instruction. The format of the instruction descriptor is shown in Figure 6.4. The My-ROB-i field is the index of the reorder buffer entry allocated to this instruction. The Op-bits field indicate operand availability; there is one bit per operand. Left and Right register numbers are physical (renamed) registers for this instruction. Finally the PTR-missing field is a pointer to the W-LIST of a producer instruction. When both of the operands are missing, only one of them has to be recorded as part of the instruction descriptor since the descriptor itself is stored into the W-LIST of the other operand.

The actual processing of the instruction descriptors stored into the FIFOs is handled at a rate of one descriptor per FIFO through the Fetch W-List and Wake-up stage. Each descriptor field is

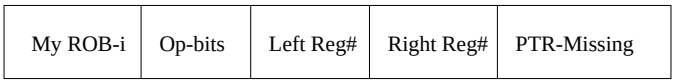


Figure 6.4: Instruction descriptor.

used to handle the wake-up process. The processing of each of the descriptors consists of two main steps. These are determining whether or not the instruction described by the current descriptor is ready and waking up those instructions which are in the wake-up list of the current instruction. The process of determining the status of the current instruction is handled in parallel with the fetching of the wake-up list.

Determining the status of the instruction. The status of the current instruction is determined by accessing the reservation table. For this purpose, the physical register identifiers are sent to the reservation table to fetch the operand availability. Please note that only the instruction described by the descriptor needs to access the reservation table, as the ready status for all the instructions being woken-up can be easily computed from their descriptors and the operand status for the current descriptor.

Fetching of the Wake-up List. Simultaneously with the reservation table access, the stage sends the My-ROB-i field of the instruction descriptor to the reorder buffer to fetch the W-LIST associated with this instruction. Since the descriptor is at the head of the queue, it should now wake-up any instructions which are in its wake-up list.

Once the W-LIST and the operand availability information is obtained, the status of the instructions being woken-up is determined easily by performing the following operations in parallel:

- if the left operand is available, in group l0,l1 one zero bit from the Op-bits field of each descriptor is turned on.
- if the right operand is available, in group r0,r1 one zero bit from the Op-bits field of each descriptor is turned on.
- if the instruction is a single cycle instruction, in group d0,d1 one zero bit from the Op-bits field of each descriptor is turned on.

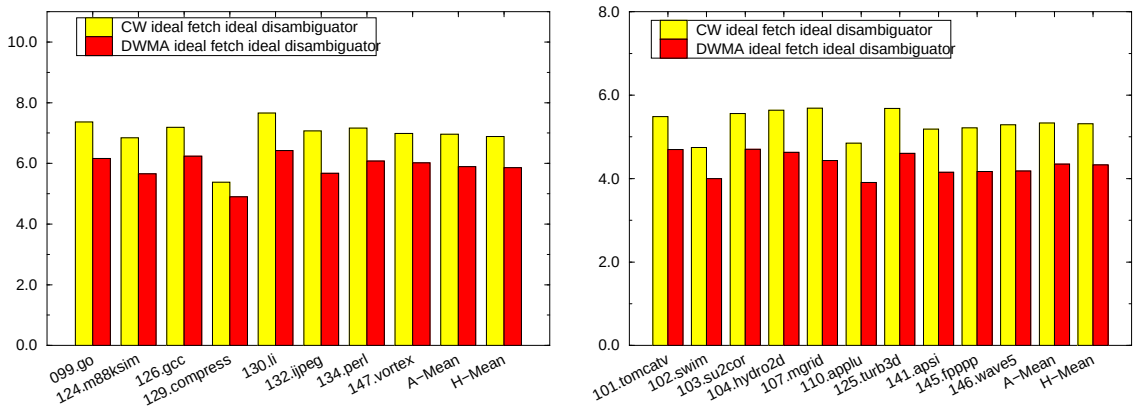
The set of instructions which are ready are easily identified using the operand availability information for the currently processed instruction descriptor. The instruction itself is ready, if both operands are available. The dependent instructions are ready if they are only missing the data of the source operands of the currently processed instruction (i.e., the edges are of use-use type), or if the current instruction is a single cycle integer instruction whose result would make them ready.

Following the above operation, we now have a number of descriptors, namely, the original descriptor and its dependent descriptors. We first process the dependent descriptors. If these

descriptors are ready they are pushed back to the head of the FIFO. In the next cycle, they can now wake-up their dependents and issue. If they are not ready, they are written back to the reorder buffer by the next stage. If the original descriptor is ready, and the corresponding functional unit is free, the instruction is issued for execution through the Read registers and W-LIST WB stage. If the corresponding functional unit is busy, the descriptor is pushed back to the head of the FIFO.

Longer latency operations need a similar treatment with a different timing. One cycle before the completion of the result, they access their reorder buffer entry to fetch the descriptors in their result wake-up group (d0, d1). These descriptors are inserted to the head of the queue they belong. In the next cycle, these instructions can resume operation if they now have all of their missing operands.

6.5 Experimental Evaluation



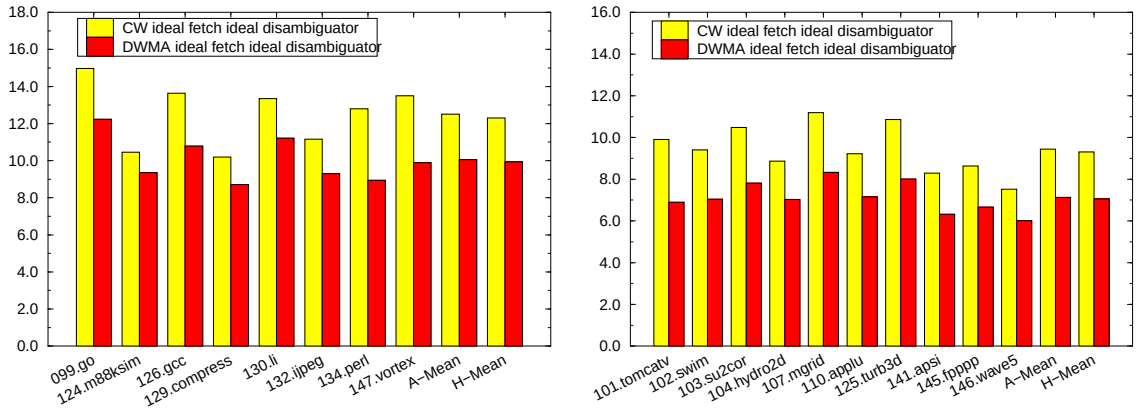
(a) 8 Issue - Integer Benchmarks

(b) 8 Issue - Floating point Benchmarks

Figure 6.5: IPC values for DWMA

DWMA architecture has been evaluated for issue widths 8, 16, and 32 and its performance has been compared with the central window processor. The resulting IPCs for all the benchmarks are shown in Figures 6.5, 6.6 and 6.7. An 8-issue DWMA architecture attains 85% and 82% of the 8-issue central window processor performance for integer and floating point benchmarks respectively. At an issue width of 16, DWMA architecture achieves 81 % of the performance of the ideal central window processor for integer benchmarks and 76 % for floating point benchmarks. The high performance of DWMA is also evident in case of 32 issue processors. With integer benchmarks the same trend is continued at 72 % of the performance of the central window processor. However, with floating point benchmarks DWMA loses some performance achieving about 62 % of the ideal processor. The larger

performance loss for the floating point benchmarks is directly related with the MIPS-I instruction set, which does not have double word load and store instructions. Instead, two load single operations are performed to load a double word. This results in a double precision operand requiring separate wake-up links for each half of the double precision value, consuming valuable wake-up links. Much better performance can be obtained by treating double precision registers as single objects for the wake-up purposes.



(a) 16 Issue - Integer Benchmarks

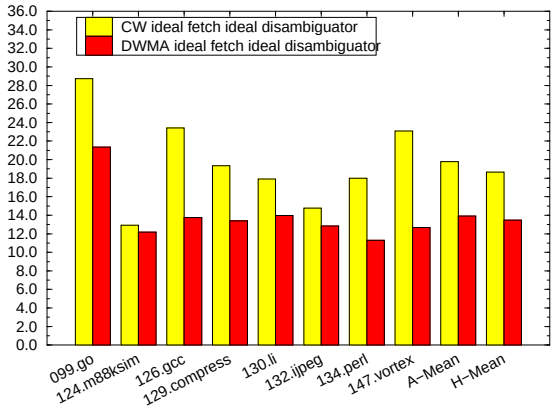
(b) 16 Issue - Floating point Benchmarks

Figure 6.6: IPC values for DWMA

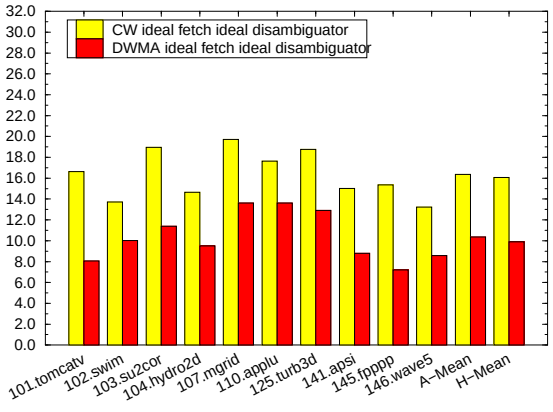
The harmonic means of IPCs obtained by the two algorithms are plotted in Figure 6.8. As it can be seen, DWMA processor closely follows the performance of the central window processor in case of integer benchmarks. In case of floating point benchmarks, performance loss becomes more significant after an issue width of 16. At these high issue widths, available parallelism is being exploited fully by the central window processor. Therefore, the limitation of a degree of forwarding of 2 becomes more significant.

6.6 Concluding Remarks

In this chapter, a superscalar out-of-order processor, namely, DWMA that dynamically generates a wake-up graph and utilizes it has been presented. This processor design is the first such processor in the published literature demonstrating that high performance superscalar processing can significantly benefit from the application of dataflow concepts. The application of dataflow concepts has been made possible by the novel concepts of *source-to-source forwarding*, *direct data forwarding* and *direct instruction wake-up*. This approach makes it feasible to implement very large instruction

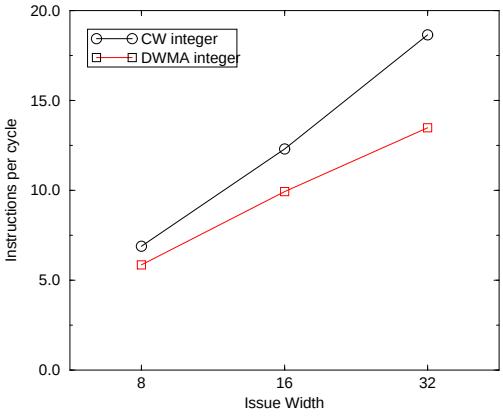


(a) 32 Issue - Integer Benchmarks

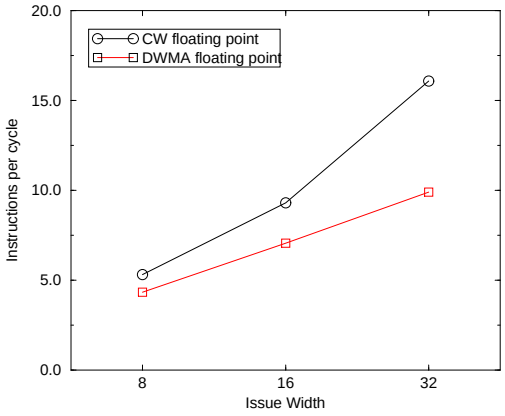


(b) 32 Issue - Floating point Benchmarks

Figure 6.7: IPC values for DWMA



(a) Harmonic Means for Integer Benchmarks



(b) Harmonic Means for Floating point Benchmarks

Figure 6.8: Scalability of CW and DWMA

windows, which otherwise are not possible to implement feasibly with the current implementation technologies.

Approach taken in the design of DWMA exploits instruction level parallelism over a large instruction window. In this respect, it can exploit similar levels of parallelism that can be exploited by dataflow processors such as ETS [51]. In contrast to the dataflow approach of treating memory dataflow as an integral part of the synchronization process by use of the *I-structures*, DWMA has to rely on predictive speculative techniques with explicit load/store instructions being utilized. In the following chapters, it is demonstrated that memory dataflow can be efficiently handled within this framework.

Chapter 7

Analysis of the Store Set Algorithm

The store set algorithm proposed by Chrysos and Emer in [10] is a memory dependence predictor that is typically integrated into the issue window logic. The predictor provides information about whether or not a recently fetched load or store instruction can issue once it satisfies its register data dependencies. If a load instruction is predicted to be dependent on a store instruction, this information is communicated to the scheduler and the scheduler enforces the ordering in the instruction window. In addition to the load/store dependencies, the algorithm may require a store instruction to be ordered with respect to another store instruction. In other words, the predictor may request both load/store dependencies as well as store/store dependencies to be enforced from the scheduler. Since the predictor is accessed early in the pipeline, the implementation does not hold the load instructions longer than it is necessary. As a result, as long as the predictor correctly predicts the dependencies, the algorithm provides very high performance. When the predictor does not correctly predict the dependencies, it may make load instructions dependent on the wrong store instructions which may result in memory order violations or may result in unnecessary delaying of the load instructions.

As it has been illustrated in Chapter 3, the algorithm performs superbly upto an issue width of 8, but algorithm's high performance diminishes at higher issue widths. In this chapter, a thorough analysis of the underlying reasons for the loss of performance is presented.

The organization of the chapter is as follows. First, in Section 7.1, a detailed outline of the algorithm is presented. Next in Section 7.2, detailed experimental evaluation of the algorithm is presented. In Section 7.3, reasons for the loss of performance of the algorithm at high issue widths are analyzed. Finally, a brief discussion of the results is presented in Section 7.4.

7.1 The Store Set Algorithm

The store-set algorithm is a simple and very effective memory disambiguator that relies on the fact that the future memory dependencies can be correctly identified from the history of memory order violations. In this respect, a *store set* is defined to be the set of store instructions a load has ever been dependent upon. The algorithm starts with empty sets, and speculates load instructions around stores blindly. When memory order violations are detected, the offending store and the load

instructions are placed in a store set. Since a load may depend upon multiple stores and multiple loads may depend on a single store, an efficient implementation of the concept may be difficult. In order to use direct mapped structures, Chrysos and Emer propose certain simplifying assumptions in their implementation which limit a store to be in at most one store set at a time as well as the total number of loads that can have their own store set. Furthermore, stores within a store set are constrained to execute in order. With these simplifications, only two directly mapped structures shown in Figure 7.1 are needed to implement the desired functionality.

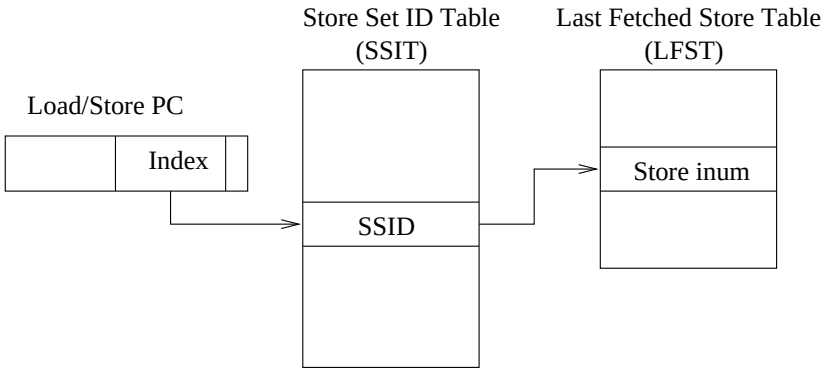


Figure 7.1: Store Set Implementation

When new load and store instructions are fetched, they access the store set id table (SSIT) to fetch their store set identifiers (SSIDs). If the load/store has a valid SSID, it belongs to a store set. Store instructions access the last fetched store table (LFST) to obtain a hardware pointer to the last store instruction that is a member of the same store set which was fetched before the current store instruction. Current store instruction is made dependent on this store. Next, recently fetched store instruction puts its own id, that is, a hardware pointer to itself, into the table. Similarly, load instructions are made dependent upon the store instruction whose id is found in the LFST table. As a result, the algorithm orders stores within a store set in program order, but allows multiple loads to be dependent on a single store.

The assignment of store set identifiers is carried out upon the detection of a memory order violation. As a result, only those loads and stores that need to be synchronized are entered in the tables resulting in efficient utilization of the table space. However, since a store can only be a member of a single store set, it is possible that two different loads each belonging to different store sets compete for a single store and cause additional memory order violations. In order to overcome this problem, Chrysos and Emer propose a set of rules for the creation of store set entries:

1. If neither the load nor the store has been assigned a store set, one is allocated and assigned to both instructions. Although any mechanism could be used to create a store set id, an exclusive or hash of the load instruction's PC works well.

2. If the load has been assigned a store set, but the store has not, the store becomes a member of the load instruction's store set by inheriting load instruction's SSID.

3. If the store has been assigned a store set, but the load has not, the load is assigned the store instruction's store set.

4. If both the load and the store have already been assigned store sets, one of the two store sets is declared the *winner*. The instruction belonging to the loser's store set is assigned the winner's store set.

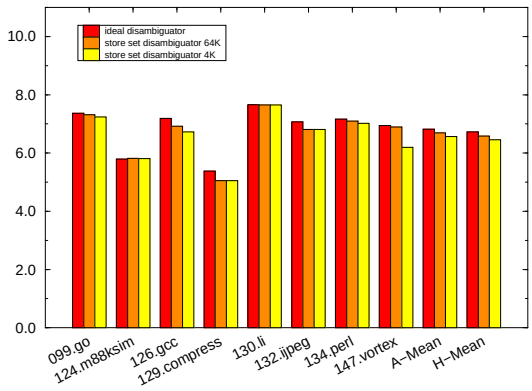
The last rule ensures that no two loads competing for a single store cause thrashing since the rule always declares the same instruction the winner. The algorithm is reported to require modest table sizes. This is attributable to high locality of memory dependencies as well as the algorithm's approach to the store set creation. It has been indicated that the algorithm performs superbly using 4K or more SSITs and 128 or more entries of LFST. At an issue width of 8 instructions per cycle, the performance of the algorithm is within few percentages of what can be accomplished using an ideal memory disambiguator which has perfect knowledge of load and store dependencies.

7.2 The Evaluation

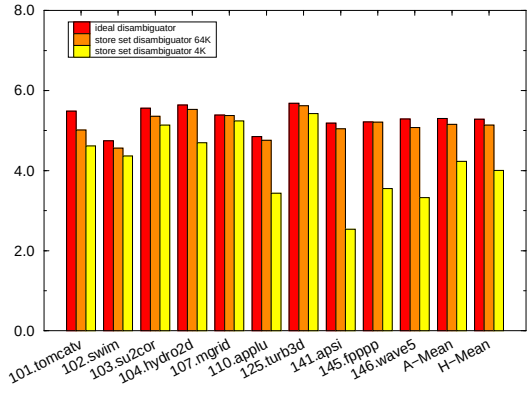
The original store set algorithm has been implemented faithfully using the ADL language [46] and SPEC95 benchmarks have been executed at various issue widths. As a reminder, the ideal memory disambiguator identifies the provider store instruction instance for each of the load instruction instances. Hence, for a given load, the ideal disambiguator indicates whether or not the store instruction on which the load is truly dependent has been issued. The disambiguator uses memory address traces augmented with load/store sequence numbers to identify such dependencies with perfect accuracy.

In order to evaluate the performance of the memory disambiguator, the machines with ideal and the store set disambiguators were kept identical in all other aspects except the memory disambiguator. Both superscalar processors employ an ideal instruction fetcher that has perfect branch prediction and delivers issue width instructions every cycle. Similarly, the issue window is a large central window implementation which can schedule instructions as soon as the data dependencies for an instruction are satisfied. In order to show the effects of the predictor table size on the performance, the performance of the algorithm is reported at both 4K entry tables as well as 64K tables which experience very few destructive aliasing.

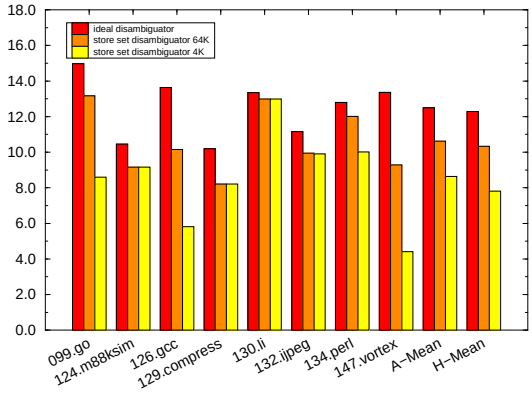
Results for an 8 issue machine are shown in Figures 2(a) and 2(b). These results confirm the published performance of the store set algorithm with some minor differences. Although most benchmarks with the exception of 110.applu have been reported to perform well in the original study, it has been observed that all benchmarks show performance losses compared to the ideal disambiguator with the exception of 107.mgrid and 145.fpppp. With 4K tables, benchmarks 110.applu, 141.apsi, 145.fpppp and 146.wave5 demonstrate significant performance losses. However, with 64K tables the algorithm closely matches the performance of the ideal disambiguator. Differences between these results and the published performance of the store set algorithm are attributable to



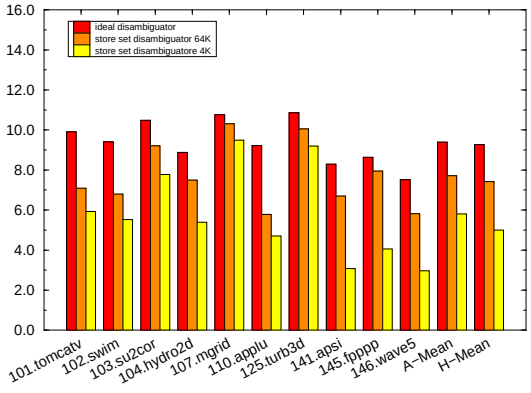
(a) 8 Issue - Integer Benchmarks



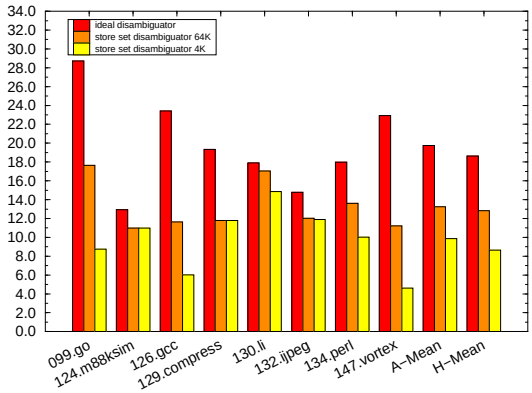
(b) 8 Issue - Floating point Benchmarks



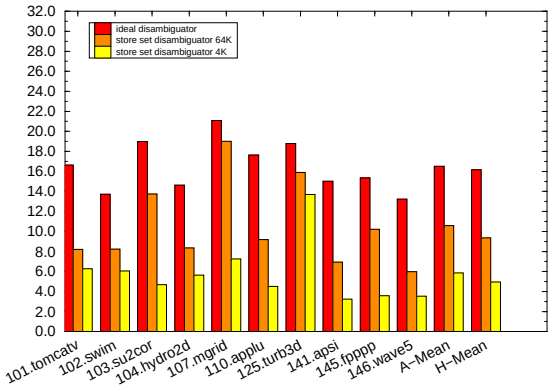
(c) 16 Issue - Integer Benchmarks



(d) 16 Issue - Floating point Benchmarks



(e) 32 Issue - Integer Benchmarks



(f) 32 Issue - Floating point Benchmarks

Figure 7.2: IPC values Store Set and Ideal cases

using an ideal front end as well as using a different ISA (MIPS versus Alpha) and using different compilers (gcc versus DEC cc). On the average, the store set algorithm achieves over 97% of the performance of the ideal disambiguator for floating point benchmarks and over 98% for the integer benchmarks with 64K tables. With 4K tables, the corresponding values drop to 80% for floating point benchmarks and 96% for integer benchmarks.

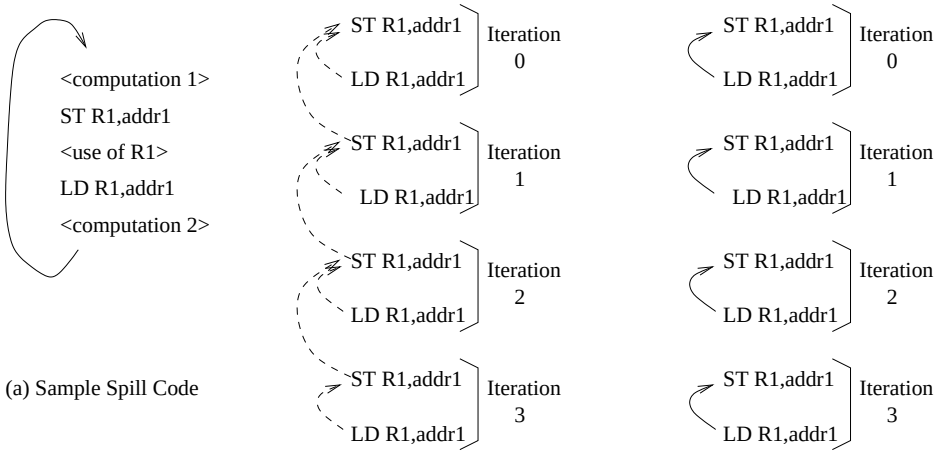
When simulations were carried out using the same machine configurations for the issue widths of 16 and 32, it was observed that performance loss as a result of non-ideal memory disambiguation becomes quite significant (see Figures 2(c), 2(d), 2(e) and 2(f)). Among the integer benchmarks, both 126.gcc and 147.vortex show significant performance losses at an issue width of 16 and above and only 130.li continues to perform well as the issue width is increased. A similar behavior is observed among the floating point benchmarks. With the exception of 107.mgrid and 125.turb3d, all benchmarks indicate significant performance losses compared to an ideal disambiguator. At an issue width of 16 and 64K tables, store set can achieve about 85% and 82% of the performance of the ideal disambiguator for integer and floating point benchmarks respectively. With 4K tables, the algorithm can achieve about 69% of the ideal performance for integer benchmarks and 61% for floating point benchmarks. At an issue width of 32 and 64K tables, performance drops further to 67% and 64%. When harmonic means are used, an additional 3 to 4% performance loss is observed with respect to the ideal disambiguator. With 4K tables, the algorithm can provide only 35% of the performance of the ideal disambiguator for floating point benchmarks and 50% for integer benchmarks. These results indicate that there is a significant room for improvement, especially at issue widths of 16 and above.

7.3 The Analysis

Although the cost of restart increases as the instruction window is enlarged, this is not the main reason behind the poor performance of the algorithm at high issue widths. The algorithm experiences performance losses because it forces the issuing of store instructions within a store set to be in-order. In-order issuing of the stores within a store set in turn causes dependent loads to issue in-order. While this restriction is not significant for a wide range of cases, it creates significant degrees of false memory dependencies with two types of loops.

One of them is the case where certain iterations of a loop occasionally become dependent on another iteration as in the case of 110.applu. The other involves loops with register spill code. In both cases, the algorithm essentially serializes loop iterations once appropriate store set entries are created since the algorithm cannot distinguish between multiple instances of the same load and store instructions. In this case, all the instances of the same store instruction become members of the same set and are forced to issue in-order. Limitations of the algorithm become more pronounced at high issue widths because at small issue widths there is still ample amount of unexploited parallelism to hide the effects of serialization. At large issue widths, the available parallelism in the program is

already being fully exploited. Therefore, the effects of the serialization of the operations cannot be hidden by other operations from the pool of available instructions.



Cycle	Instructions	
0	ST-0	
1	LD-0	ST-1
2	LD-1	ST-2
3	LD-2	ST-3
4	LD-3	...

(d) Store-set schedule

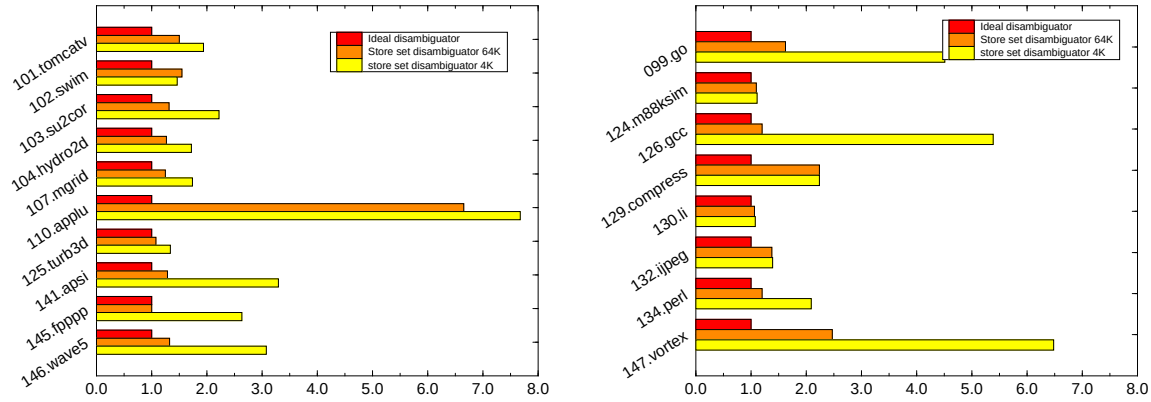
Cycle	Instructions			
0	ST-0	ST-1	ST-2	ST-3
1	LD-0	LD-1	LD-2	LD-3

(e) Ideal schedule

Figure 7.3: Example spill code and its schedule

Let us now examine in detail how the algorithm executes such loops. In Figure 7.3(a), an example loop that contains spill code is illustrated. When such a code sequence is executed using sufficient resources to issue more than one load operation per cycle, it takes only a few iterations to be unrolled before a memory order violation occurs. This is because, when there is no dependency information stored in the tables, any of the loads can issue once they compute their addresses. As a result, any load which is truly dependent on the store at the same iteration may issue before that store. Once this happens, a memory order violation is detected and the store set entries are allocated. From this point on, following instances of these loads and stores share the information stored in SSIT and LFST which yields the dependence graph shown in Figure 7.3(b). The algorithm correctly makes a load dependent on the proper store by means of the LFST table. However, since the algorithm forces stores within the same store set to issue in order, for the given set of loads and stores the generated schedule allows at most one load instruction to execute per cycle (see Figure 7.3(d)). In contrast, an ideal disambiguator would allow fully parallel operation of the multiple instances of the loop body, given sufficient resources (see Figure 7.3(e)).

In order to measure the effect of the serialization on the load latency, additional experiments have been conducted that studied the dynamic load latency and degree of load serialization. The results of these experiments are as follows:



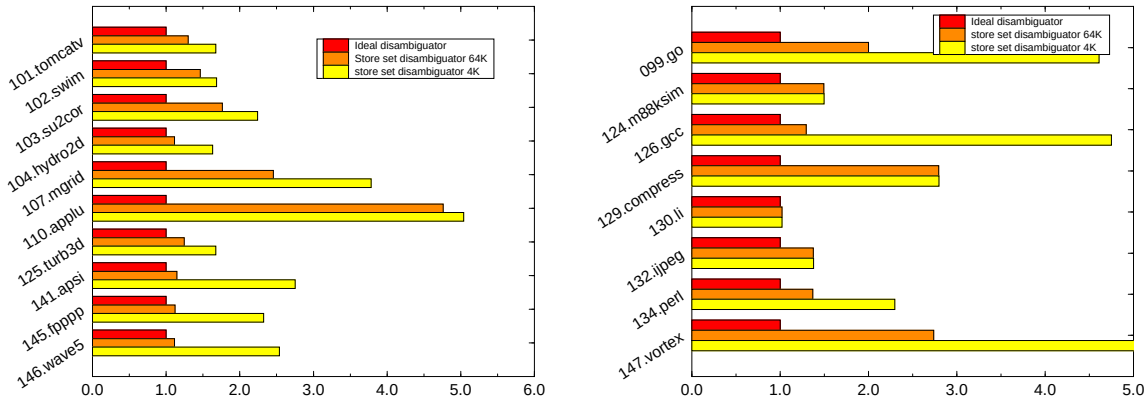
(a) Floating Point Benchmarks

(b) Integer Benchmarks

Figure 7.4: Normalized Average Dynamic Load Latencies

Dynamic load latency. *Dynamic load latency* measurement measures the number of cycles it takes from the moment a load instruction is ready to issue to the moment it has the loaded value. Dynamic load latency is a function of the dependencies imposed upon load instructions. Therefore, it is a cumulative quantity that includes both the true dependencies of the program as well as scheduling/disambiguation imposed dependencies. To obtain the contribution of the falsely imposed dependencies, the dynamic load latency values have been normalized by dividing it with the dynamic load latency value obtained using an ideal memory disambiguator (see Figure 7.4). In addition to the normalized dynamic load latencies, the standard deviation of dynamic load latency across all load instructions executed by the benchmark programs have been computed both for the store set algorithm and the ideal memory disambiguator. The standard deviation values for the store set algorithm were then normalized by dividing it with the corresponding value of the ideal memory disambiguator (see Figure 7.5).

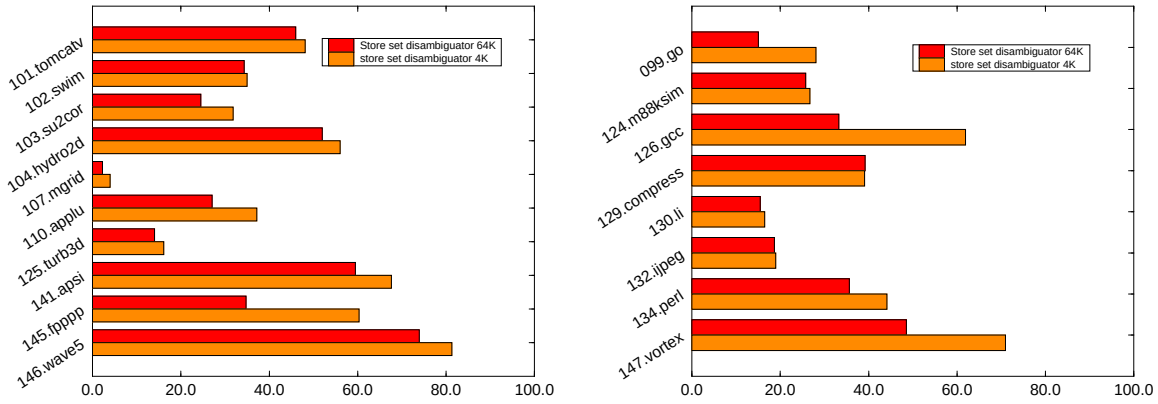
As shown in Figure 7.4, the measurements of the normalized dynamic load latencies exhibit large values for those benchmarks which perform poorly whereas benchmarks which perform well have very small values. Similar behavior is observed in the standard deviation values shown in Figure 7.5. These results are indicative of long chains of dependent instructions that create large fluctuations in the average load latency. It has also been observed that the harmonic mean values of the load latencies are uniform across the benchmark spectrum and quite close to the ideal disambiguator. On the other hand, the arithmetic means show great degrees of fluctuation, and they have the worst values for those benchmarks whose performance does not scale.



(a) Floating Point Benchmarks (b) Integer Benchmarks

Figure 7.5: Normalized Standard Deviation Values for Dynamic Load Latencies

Load serialization. The degree of the serialization of load instructions through load-store-store dependency chains has also been studied. The amount of serialization of load instructions has been measured by identifying the dynamic load instructions which are blocked from issuing for one or more cycles although their predicted provider store instruction is ready to issue. In Figure 7.6, the percentage of total dynamically executed load instructions which have been serialized are shown.



(a) Floating Point Benchmarks (b) Integer Benchmarks

Figure 7.6: Percentage of Serialized Load Instructions

Measurements of the dynamic percentage of serialized load instructions also consistently support the previous observations. As it can be seen from the experimental results shown in Figure 7.6, the three benchmarks, namely, 107.mgrid, 125.turb3d, and 130.li which perform well as the issue width is increased have very low percentages of serialized load instructions. All the remain-

ing benchmarks which perform poorly at high issue widths have significant percentage of the load instructions serialized.

7.4 Concluding Remarks

It has been shown that the store set algorithm as originally proposed by Chrysos and Emer [10] performs well at low issue widths but does not scale to higher issue widths because of store-store induced memory dependencies. Introducing these dependencies is not an oversight, but is a requirement to simplify the memory order violation detection which would simply be unmanageable without this ordering.

In the next chapter, a novel memory order violation detection algorithm is presented which can be utilized to correctly detect the memory order violations even when the store instructions are allowed to issue completely out-of-order. The memory order violation detection algorithm is simple and does not need to be on the processor's critical path. With this algorithm, the original store set algorithm can be modified to remove the requirement that store instructions within a store set execute in-order, yielding very high performance at all issue widths.

Chapter 8

Memory Disambiguation with Out-of-order Stores

In Chapter 7, we have seen through a detailed analysis that for high performance we need to allow full out-of-order issuing of store instructions so that dependent load instructions can also issue fully out-of-order. Unfortunately, without at least a partial ordering of store instructions in the instruction window, the algorithm would have suffered much more significant levels of performance losses because of *false memory order violations*. This is because, a simple mechanism for memory order violation detection checks load addresses of the speculatively issued load instructions when a store instruction is issued and upon an address match raises a memory order violation condition. In this approach false memory order violations occur because this mechanism cannot decisively figure out the set of store instructions which should participate in the memory order violation detection process for a given load. Therefore, we need a mechanism that can detect memory order violations precisely even when the store instructions are allowed to issue fully out-of-order. Once such a mechanism is available, the original store set algorithm can be modified to allow full out-of-order issuing of store instructions.

In this chapter, such a mechanism is developed and its performance is fully analyzed by introducing the necessary changes into the original store set algorithm. In the remainder of this chapter, in Section 8.1, false memory order violations and why they occur with simple memory order violation detection mechanisms are discussed. Next in Section 8.2, requirements for detecting memory order violations correctly are analyzed. The novel solution of delaying exceptions and using values in addition to load/store addresses as opposed to using addresses alone to check for memory order violations is presented in Section 8.3 and Section 8.4. Section 8.5 presents a detailed performance evaluation of the technique. Finally, in Section 8.7 the chapter is concluded with a brief summary of results.

8.1 False Memory Order Violations

To illustrate why false memory order violations occur with simple memory order violation detection mechanisms, when store instructions are allowed to issue fully out-of-order, let us reconsider

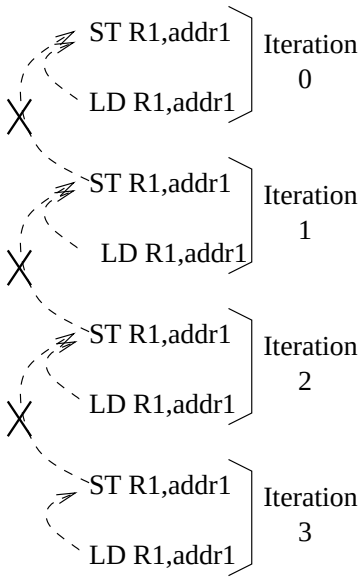


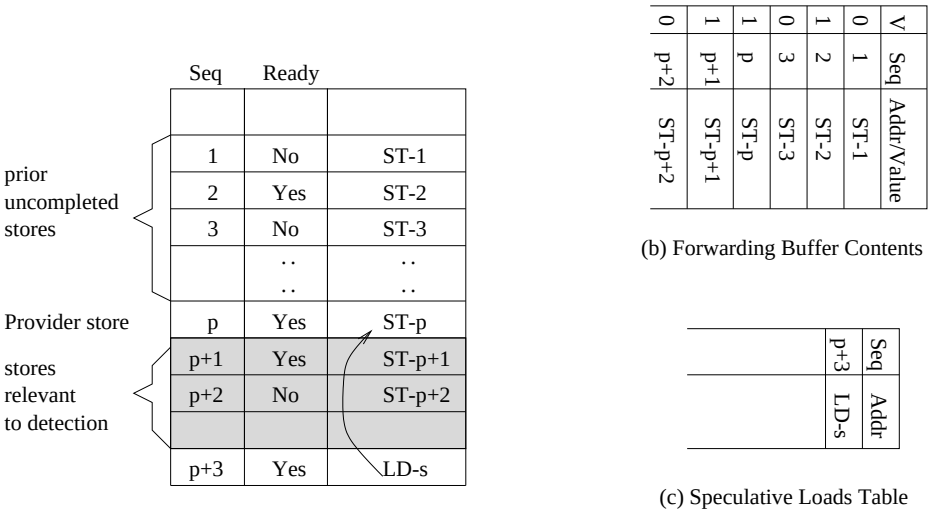
Figure 8.1: Removing the Store-Store Dependencies

the example shown in Figure 8.1. Assume that the dependence edges between the store instructions are absent when the store set disambiguator is being used. As a result, the store instructions would be allowed to issue fully out-of-order. In this case, a store instruction belonging to an earlier iteration may be blocked whereas a store belonging to a later iteration may have a chance to proceed. For example, in Figure 8.1 the store from the second iteration, ST-2, may proceed before ST-1 which belongs to the first iteration. When ST-2 issues it makes its dependent load LD-2 eligible to issue in the next cycle. When LD-2 issues, it gets the *correct value* from the forwarding buffer. The processor however remembers that the load has been issued speculatively and makes an entry regarding this load in the *speculative loads table*. When the store ST-1 issues, it finds that a load with a sequence number greater than its own that computed the same address has issued before the store. In this case, an exception is flagged which is in fact a *false memory order violation*. **In other words, removing the store ordering in the instruction window would convert all store ordering induced false memory dependencies to false memory order violations**. On the other hand, when the memory disambiguator imposes an ordering on those stores which may have the same effective address, false memory order violations will not be observed since a load instruction which is dependent on a later store instruction would not issue before all store instructions preceding the store instruction it is made dependent upon.

8.2 Precisely Detecting Memory Order Violations

In order to detect memory order violations correctly when store instructions are allowed to issue freely, it is necessary to identify precisely the set of store instructions which should participate in the memory order violation detection process. If an issuing store instruction is not the member of

this set with respect to a given speculatively issued load instruction, we should not let this particular store instruction raise a memory order violation for the load instruction in question.



(a) Dynamic code sequence

(b) Forwarding Buffer Contents

(c) Speculative Loads Table

Figure 8.2: Speculative issuing of loads

In order to see how we can identify the set of store instructions which must participate in the decision process, let us consider the sequence of store instructions and the load instruction shown in Figure 8.2(a). Given this dynamic sequence of instructions, assume that the load LD-s is predicted to be dependent on the store ST-p which is indicated through the dependence edge. In this configuration, store instructions which are above the provider store instruction ST-p should not participate in the memory order violation detection process for the speculatively issued load instruction LD-s assuming that the addresses of ST-p and LD-s match. Only store instructions which follow the provider store instruction, namely, ST-p+1 and ST-p+2 should raise an exception if they compute an effective address that is the same as the load LD-s. In other words, the provider store instruction splits the set of uncompleted stores into two sets, and only the ones that follow the provider store instruction should participate in the decision process. In the case that the load instruction obtains its value from the memory, all the prior stores should be involved for checking the memory order violations with respect to the speculatively issued load LD-s.

One possible solution in this case is to include the sequence number of the provider store (or a special identifier if it is memory) in the speculative loads table. In this case, issuing store instructions may check their sequence numbers against the sequence number of the provider as well as the sequence number of the load instruction to determine that if they fall into the shaded region in Figure 8.2(a). If that is the case, and the address generated by the store instruction matches to that of the load, an exception may be raised.

A straightforward implementation of the above mechanism leads to a complex piece of hardware due to the following reasons. First of all, maintaining explicit temporal information through sequence counters is not a trivial task because counters must be of finite size and when they over-

flow, the boundary conditions must be properly handled. Second, for any given store instruction the processor must execute the above algorithm in parallel against all the speculatively issued load instructions, which means that the required hardware must be replicated. Finally, executing the above algorithm on the critical path of the processor is very likely to slow down the processor clock. In the next section, a much simpler yet more effective solution is presented.

8.3 Delayed Exception Handling and Value Matching

The solution to the detection problem builds on the following observations.

(1) The temporal information needed is implicitly available during the retire phase of the instruction execution. In other words, if we can delay the detection of the memory order violations to the retire phase, we do not need to maintain the temporal information explicitly. Since the processor experiences very few exceptions due to memory order violations when equipped with a good memory dependence predictor, the additional penalty of late restart is not high. Once we move the detection logic to the retire phase, memory violation detection entails deciding whether or not the provider store has retired. If that is the case, subsequent store instructions are from the shaded region and they should check for memory exceptions. Of course, if the provider is the memory, the provider store has already retired, and in this case all retiring store instructions will be involved in the checking.

(2) For correctness, we do not need to identify the exact store instruction that provided the value. We only need to verify that given a set of store instructions, the load has obtained the same *value* as the value stored by the last store instruction to the same memory address. This technique eliminates the need for special handling of the case where memory is the provider. Furthermore, as it will be seen shortly that this method can take advantage of the value redundancy available in the programs.

Given the above observations, we can now devise the following scheme that works quite well:

- The checking for exceptions in case of memory references is delayed until the store instruction retires.

- The *speculative loads table* is expanded to contain a value field where the value the load instruction has obtained is stored.

- An exception bit associated with the load instructions stored in the reorder buffer or in speculative loads table is allowed to be set or reset by store instructions as they retire. In other words, each retiring store instruction compares the value it has stored, as well as the address into which the data has been stored, to with that of the speculative loads:

  If the addresses match and values differ, it sets the exception bit associated with the load instruction.

If the addresses match and values match, it resets the exception bit associated with the load instruction.

If the addresses do not match, no action is taken.

- Once the load instruction is ready to retire, it checks its exception bit. If the bit is set, a roll-back is initiated and the fetch starts with the excepting load instruction. Otherwise, the load instruction's entry is deallocated from the speculative loads table.

Please note that setting and canceling of exception bits as store instructions retire in this manner handles the problem of identifying the provider store instruction automatically. When the actual provider store instruction retires, both the address and the values will match, and the exception bit is reset. In other words, this instruction will serve as a sentinel signaling the beginning of the group of store instructions which should participate, nullifying the effects of all unrelated prior store instructions.

Now let us reconsider the example shown in Figure 8.2. Suppose that load LD-s has already been speculatively issued and obtained its value from the store ST-p. Further assume that the store ST-1 has now been issued and has computed the same address as LD-s. Since ST-1 retires first in program order, it will raise the exception bit associated with the load LD-s. Any of the stores between store ST-1 and store ST-p may take the same action upon an address match and a value mismatch. However, when finally store ST-p retires, it will have both an address and a value match and will reset the exception. When the store ST-p+1 retires, if it computes the same address but the value is different, this is a true exception. The exception will be taken when the load instruction retires. Please note that if any of the store instructions ST-p+1 or ST-p+2 in the shaded region have the same address as well as the same value, no exception will be raised and the machine will take advantage of the available data redundancy. The same observation holds for the values coming through memory.

Given the above solution that effectively handles the problem of false memory order violations, the false memory dependencies arising from store-store induced dependencies can now be completely eliminated. To achieve this goal, only a small change to the original store set algorithm is needed. The load instructions are made dependent on the store instruction they find in the LFST table entry, but the store instructions which are members of the same store set are not chained, allowing all the store instructions to issue fully out-of-order constrained only by their own register dependencies. Load instructions however wait for the store instruction that they have been predicted to be dependent on. Thus no load instructions are serialized unnecessarily. Although the memory dependence prediction mechanism still relies only on the load and store program counter values, the above method can effectively handle the problems arising from multiple instances of the same load or store instruction.

8.4 Taking Advantage of Value Redundancy

In the previous section, a simple technique has been presented that correctly indicates if a memory order violation has occurred by matching the value each retiring store instruction stores with that of the speculatively issued loads. Although there is nothing novel about the common-sense technique of determining correctness of speculated instructions by matching actual data values, the use of this approach in the context of the store set disambiguator is unique and yields high performance beyond what is achievable by an ideal disambiguator that faithfully observes the true memory dependencies. A review of the workings of the store set algorithm should explain why.

During the initial start-up, there is no information in the SSIT and LFST tables to guide the scheduling. Because of the blind speculation of loads, loads acquire values either from the forwarding buffer or directly from the memory. When the actual store instruction that the load is truly dependent on retires with a value that is the same as the load instruction's value, the speculation is successful and no new entries are created in the SSIT and LFST tables. In other words, **the load speculates and executes successfully before the store it is actually dependent on**. The machine will continue to speculate the same load instruction until a violation occurs. Once a violation occurs, the load instruction will not be speculated further since it will wait for its producer store. In other words, the machine takes advantage of the value redundancy as long as it is beneficial to do so. By speculating in this manner no performance overhead is incurred in comparison to an address only approach. Instead, those load instructions that can take advantage of the data redundancy are automatically selected by the algorithm. Although this process is not directed *intelligently* as in [9], similar benefits are obtained by not paying the penalties associated with a technique that speculates indiscriminately. The net effect of the technique is reduced load access latency.

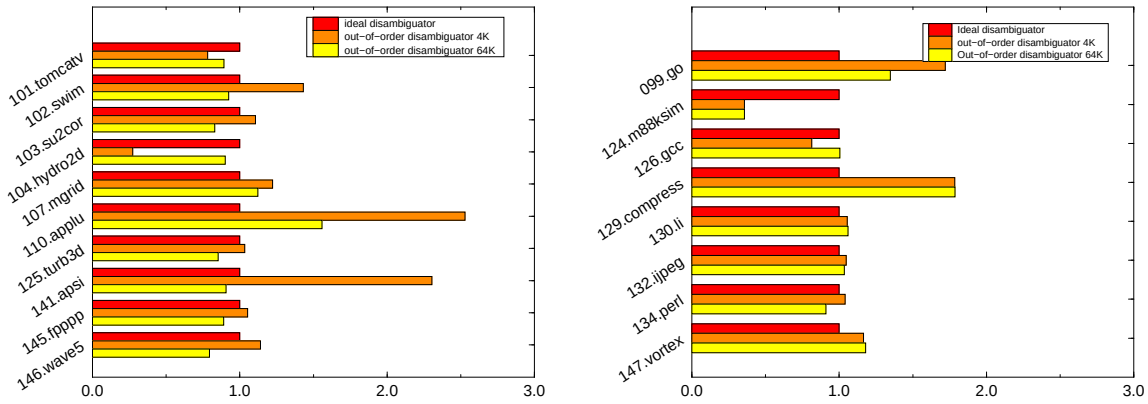
In the next section, it is demonstrated quantitatively that for most benchmarks, the technique indeed reduces the load access latency below what is possible with an ideal disambiguator that faithfully observes the true dependencies.

8.5 Performance Evaluation

In order to assess the performance potential of store set memory disambiguator with the modified back-end series of experiments have been designed. The algorithm has been fully implemented and SPEC95 benchmarks have been executed with their training or test inputs. The processor parameters have been kept as before. In these experiments, the performance of the out-of-order memory disambiguator is compared with both the ideal disambiguator as well as the original store set algorithm when it is appropriate.

8.5.1 Dynamic Load Latencies.

Normalized average dynamic load latencies for out-of-order disambiguator is shown in Figure 8.3. It is interesting to note that with the exception of 110.applu and 107.mgrid, all normalized dynamic load latencies for floating point benchmarks are below 1.0. In other words, out-of-order disambiguator yields better dynamic load latencies than the ideal memory disambiguator. This is expected, especially with those programs that have significant degrees of value redundancy. Such value redundancy occurs when the actual store instruction a load is truly dependent on is value redundant with respect to the stale value in the memory, or other issued but not yet committed store instructions. Since the ideal disambiguator makes a load wait for precisely the exact producer store instruction, in those cases where the store instruction is value redundant it makes the load instruction wait much longer. Although not to the same level of uniformity, a similar behavior is observed also among integer benchmarks.



(a) Floating Point Benchmarks

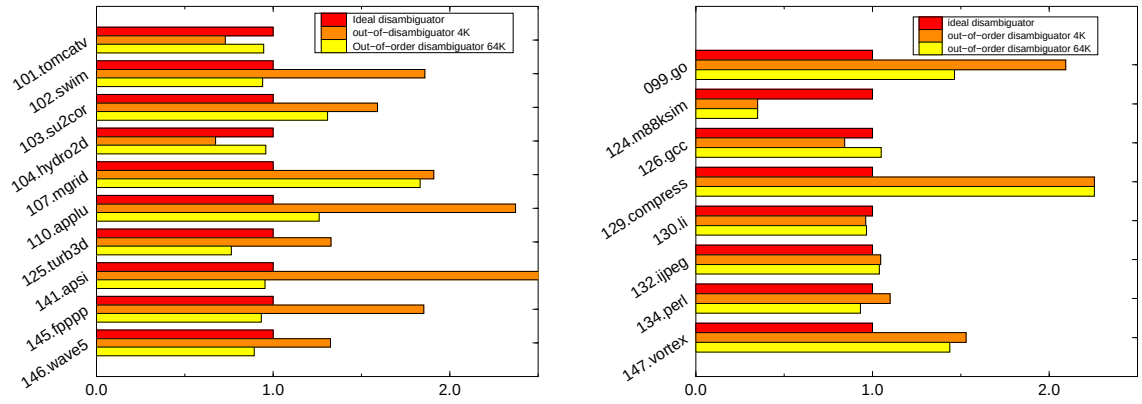
(b) Integer Benchmarks

Figure 8.3: Normalized Average Dynamic Load Latencies

The technique results in a significantly longer dynamic load latency only in case of 129.compress. Nevertheless, the figure is still well below of the original store set algorithm (2.23 versus 1.79). 124.mk88ksim shows an outstanding performance providing a dynamic load latency which is only a fraction of the ideal disambiguator (0.355). Normalized standard deviation values follow a similar trend as shown in Figure 8.4 indicating that the success of the technique is uniform throughout the benchmark's execution.

8.5.2 Instructions Per Cycle.

For the measurement of the instructions per cycle figures, the harmonic means of the IPC values observed for the floating point and integer benchmarks are compared for different algorithms. When the instructions per cycle figures are analyzed, the benefits of using the approach become



(a) Floating Point Benchmarks

(b) Integer Benchmarks

Figure 8.4: Normalized Standard Deviation Values for Dynamic Load Latencies

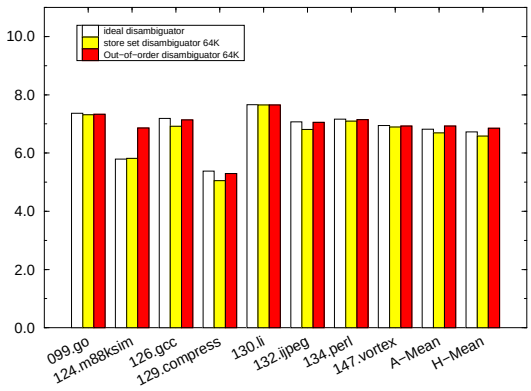
increasingly clear as the issue width is increased. At an issue width of 8, the out-of-order algorithm slightly out-performs the ideal memory disambiguator. With floating point benchmarks, out-of-order disambiguator is better than the ideal memory disambiguator by 0.01% and with integer benchmarks it is better than the ideal disambiguator by about 1.8%. The out-of-order algorithm shows better performance than the original store set algorithm by 4% with both the floating point and integer benchmarks. In case of 124.m88ksim, out-of-order algorithm is better than both techniques by about 18% (see Figures 8.5(a) and 8.5(b)).

When the issue width is increased to 16, the out-of-order algorithm out-performs the original store set algorithm by as much as 18% with integer benchmarks, and 22% with floating point benchmarks (see Figures 8.5(c) and 8.5(d)). The performance difference further widens to 42% and 52% with floating point and integer benchmarks when the issue width is increased to 32 instructions (see Figures 8.5(e) and 8.5(f)). In both cases, highly value redundant 124.m88ksim continues to out-perform the ideal disambiguator and the out-of-order disambiguator closely follows the ideal disambiguator for other benchmarks, even at very high issue widths.

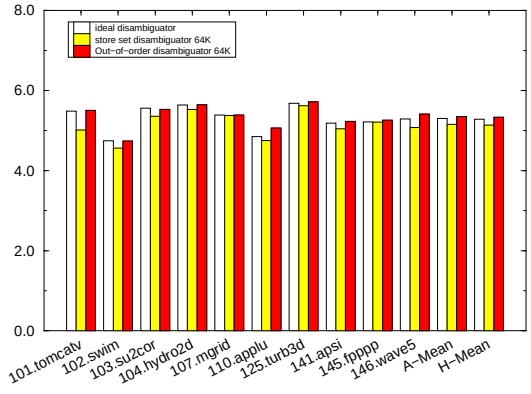
8.5.3 Scalability With Different Table Sizes

In order to further compare the performance of the out-of-order store approach to that of the original algorithm, both algorithms have been executed at predictor table sizes of 4K, 8K, 16K, 32K and 64K entries. It has been observed experimentally that the size of the LFST table is not critical and in these runs it was left to be sufficiently large.

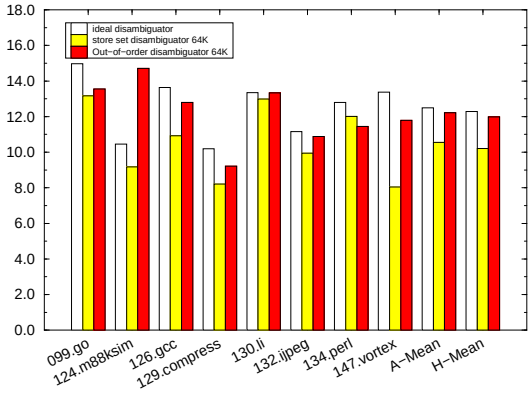
As it can be seen from the graphs in Figure 8.6 and Figure 8.7, the original store set algorithm can out-perform the out-of-order algorithm only when the original store set algorithm has a 32K entry table and the out-of-order disambiguator has a 4K entry table. For both integer and floating point benchmarks, with 8K entries the out-of-order algorithm always out-performs the



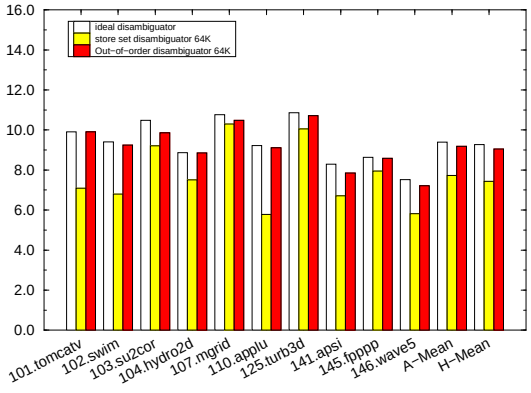
(a) 8 Issue - Integer Benchmarks



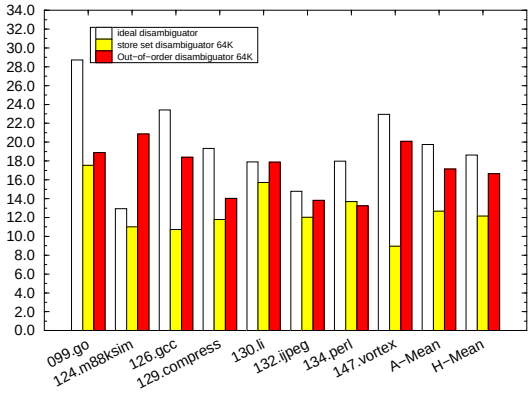
(b) 8 Issue - Floating point Benchmarks



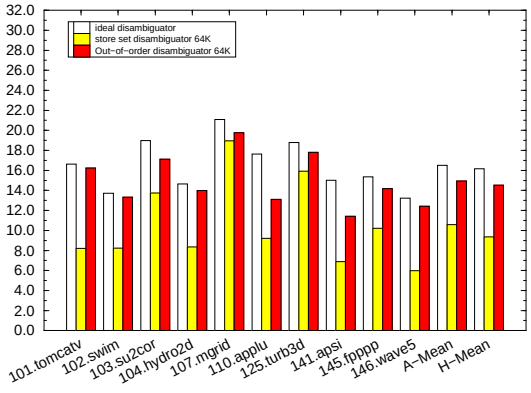
(c) 16 Issue - Integer Benchmarks



(d) 16 Issue - Floating point Benchmarks



(e) 32 Issue - Integer Benchmarks



(f) 32 Issue - Floating point Benchmarks

Figure 8.5: IPC values Out-of-order Store Set, Store Set and Ideal cases

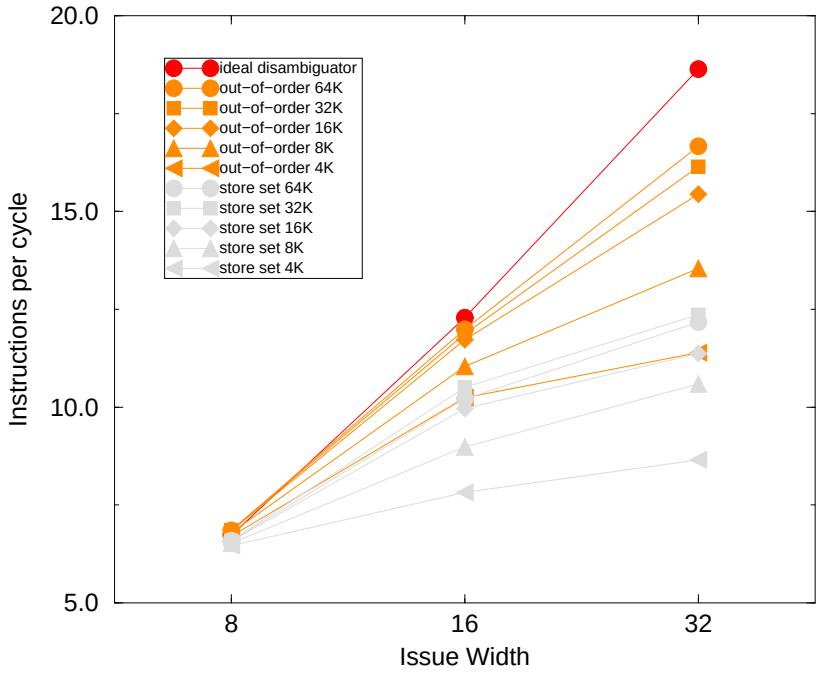


Figure 8.6: Scalability of Out-of-order Algorithm - Integer Benchmarks

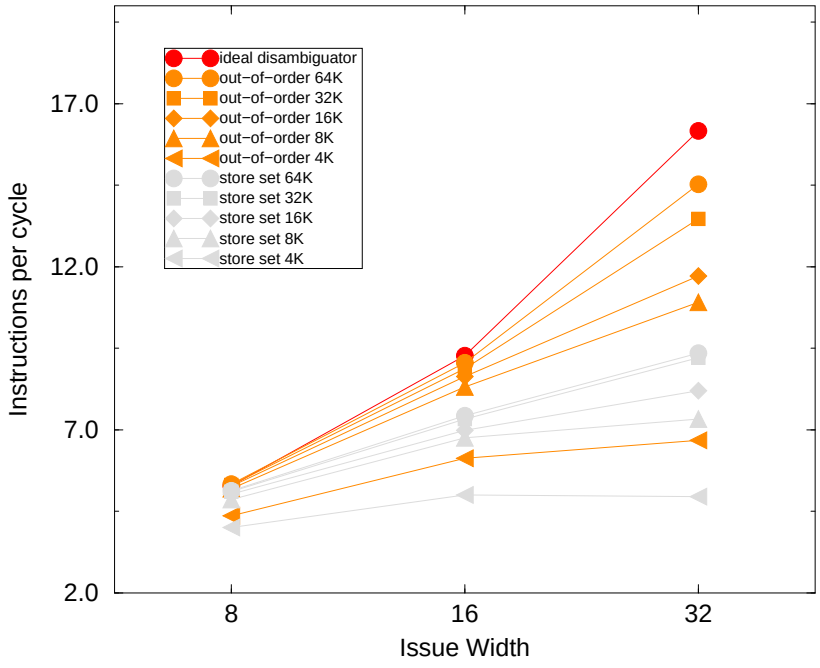
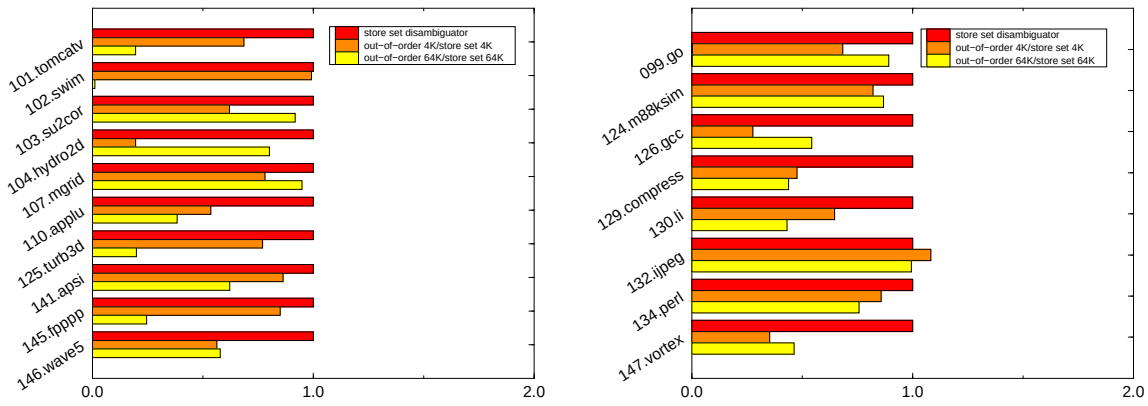


Figure 8.7: Scalability of Out-of-order Algorithm - Integer Benchmarks

original algorithm even when the latter has 64K entry tables. When the out-of-order algorithm is given a large predictor space, it matches the performance of the ideal disambiguator up to the issue width of 16, and very closely follows the ideal curve with slight performance loss. This small performance loss (about 10 % at an issue width of 32) originates from the cost of restarts. The cost of restarts are largely paid for by the gains that are obtained through the exploitation of the value redundancy. Unfortunately, at these high issue widths the exploited value redundancy is not sufficient to compensate for all the restart costs. Nevertheless, it is natural to expect that the algorithm will out-perform the ideal disambiguator at all issue widths if the cost of restarts can be reduced by employing a mechanism which selectively reissues effected instructions instead of squashing a window-full of instructions. Such re-execution recovery is quite feasible with memory order violations. In case of memory order violations, the validity of the instructions are not questioned. Therefore, only a few instructions which uses the wrong value can be reissued instead of throwing away a window-full of instructions.

8.5.4 Reduction of False Memory Dependencies

It has also been observed that when smaller predictor tables are employed, the performance gap between the out-of-order disambiguator and the original store set algorithm widens further, indicating the success of the technique in reducing the false memory dependencies. The false memory dependencies were also measured directly and the comparison of the original store set and our algorithm is given in Figure 8.8.



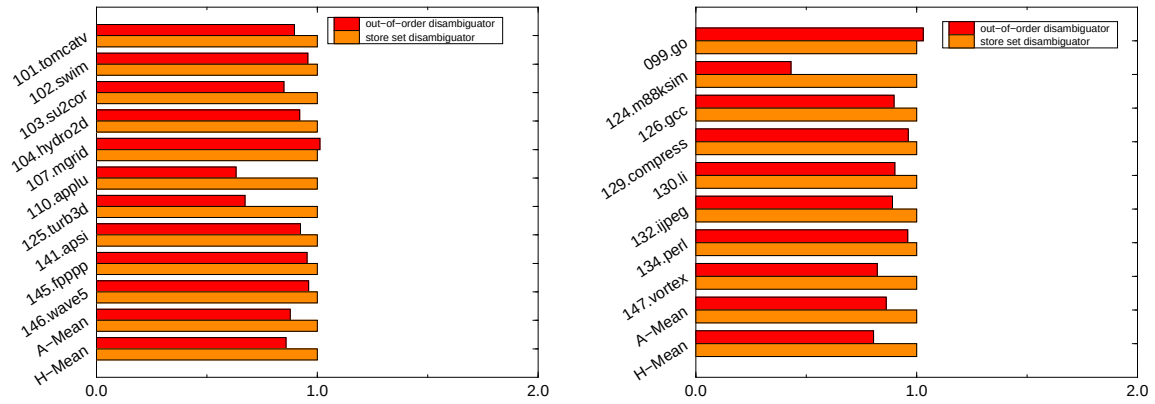
(a) Floating Point Benchmarks

(b) Integer Benchmarks

Figure 8.8: Normalized False Memory Dependencies

The false memory dependencies are reduced because of two reasons. The first reason is the preciseness of the novel memory order violation detection algorithm in creating the store sets. When a conventional approach is used to detect the memory order violations, it may take some number of iterations and a number of false memory order violations before the true store instruction that

a load is dependent on is discovered. In case of the new technique, this happens the first time a violation is encountered. Second, because the approach also takes advantage of value redundancy, it creates fewer number of table entries. In order to verify that this is indeed the case, the number of dynamic instances of load and store instructions for which there is a matching SSIT entry has been measured. This number was normalized by dividing it with the values produced by the original store set algorithm. The results of these experiments are shown in Figure 8.9.



(a) Floating Point Benchmarks

(b) Integer Benchmarks

Figure 8.9: Normalized Counts of Load/Store Instructions Synchronized Through SSIT Table

8.6 Bypassing Memory Operations

Load and store instructions are communication instructions which by themselves do not contribute to computation. They are the means through which values are communicated from the producer instructions to the memory and back to the consumer instructions. A superscalar processor can take advantage of this fact and make the producer of the value communicate directly with the consumer when they co-exist in the instruction window. In this manner, the critical path length is reduced and higher degrees of instruction level parallelism is achieved because of reduced latency of computation involving dependent instructions through memory.

Bypassing of memory operations have been studied by Moshovos and Sohi [39]. It has been indicated that by keeping a history of memory dependencies, producer instructions may communicate directly with the consumers speculatively. Because the critical path through the dependencies is reduced, the technique is called *memory dependency collapsing*. In this technique load instructions are allowed to access the memory and verify that they indeed get the correct value from the memory. This process is illustrated in Figure 8.10. Solution proposed by Moshovos and Sohi can be improved basically in two aspects: (a) proposed solution employs relatively complicated micro-architectural structures. These structures can be replaced with structures that are scalable and capable of pro-

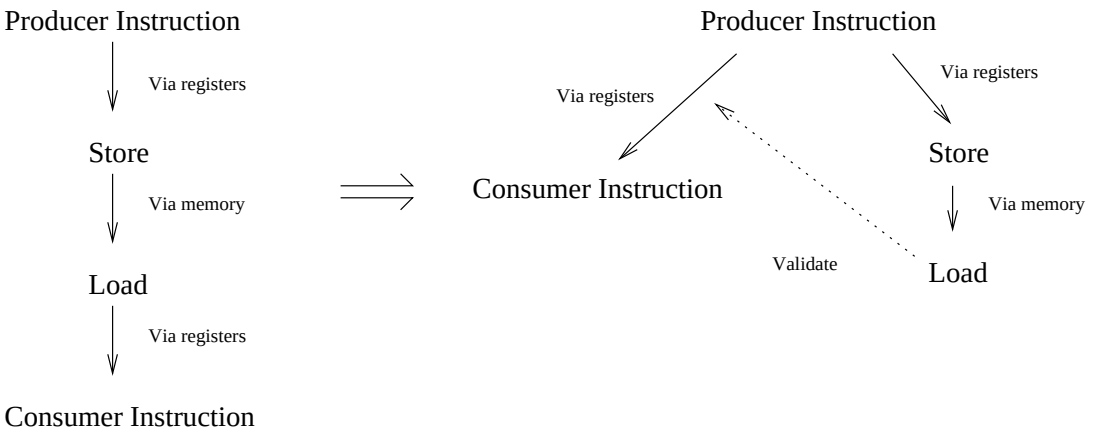


Figure 8.10: Memory Dependency Collapsing [39]

viding the same functionality; (b) access to the memory for verification is indeed unnecessary. It is sufficient to verify that the addresses computed by the load and the store instruction match and there are no intervening stores writing to the same memory location.

Elimination of memory accesses is essential for high performance superscalar processors. Memory ports are expensive and for high performance a large number of memory ports are needed. Furthermore, it is only logical to expect that with larger issue widths and a large instruction window there will be many candidates that can participate in memory dependency collapsing since both the producer and the consumer instruction will co-exist in the instruction window. In fact, more than 20 percent of the load instructions are candidates for memory dependency collapsing in a processor that has a window size of 32. This number rises to more than 45 percent with an instruction window of 256 [39]. Therefore, a mechanism which can provide the verification without a memory access may greatly reduce the memory port requirements and enhance the performance.

Recently, Jourdan et al. presented a modified solution using a *combined renaming scheme* [28]. The new solution essentially addressed the complexity issue related with the original proposal mentioned above. This piece of work illustrated that dependency establishment (i.e., establishing the value flow from the producer to the consumer) can be easily achieved using a modified register renaming scheme. What is left still unresolved is how to implement the verification process that has to ensure all store instructions encountered between the couple access different memory addresses. While for a small number of load and store instructions the process of making sure that no intervening store writes to the same memory location may be manageable, for large instruction windows where hundreds of loads are in-flight at once and a significant percentage of them are involved in memory dependency collapsing process, it is unlikely that a brute force mechanism such as the ones applied in existing processors can be extended to handle this task.

In this section, a new mechanism is presented which extends both the store set algorithm and the developed memory order violation detection mechanism so that memory dependence collapsing can be achieved in a scalable manner without accessing the memory.

Establishing the communication. For establishing the communication, the LFST table is extended to include the physical register number of the store instruction. Upon encountering a load instruction the following steps are performed:

(a) The load instruction accesses the SSIT table and then to the LFST table as usual. If there is an entry, the load is made dependent on the store.

(b) The load instruction is converted to a *sentinel load*. In other words it is marked so that it can be distinguished from other load instructions.

(c) Instead of obtaining a new physical register that will hold the result of the load instruction, the rename table for the current logical register is changed to point to the register number found in the LFST table.

(d) The load instruction is made register dependent on the register that the producer instruction will be writing to and the store instruction will be reading from. Thus, when the load instruction is ready to issue, it can be entered to the speculative loads table as if it has been issued speculatively with an available load value. Please note that the load instruction still has to wait for its address computation register dependencies as usual.

At the end of the process, all subsequent uses of the load instruction will now wait for the result of the producer instruction. In other words, the dependency links from the producer to all the consumers have been established.

Verification Process. As it has been discussed, the verification process involves making sure that the load and the store instruction that were involved in dependency collapsing compute the same address and there is no intervening store to the same memory address between the two. This means both the load and store instruction should have a chance to perform the comparison.

The solution builds on a simple scheme. For the verification process to be successful without accessing the memory, we need to be able precisely identify the store instruction involved in the process. Furthermore, we need to be aware of the fact that before the load has a chance to resolve its register dependencies involving the address computation (i.e., it is entered to the speculative loads table so that the comparison takes place), the store involved in the dependency collapsing may retire. These observations lead to a simple solution which extends the speculative loads table with a *visited flag* and the reorder buffer slot reserved for the load instruction with a *provider store instruction* identifier. This identifier can simply be the reorder buffer index of the store instruction which is stored in the LFST table by the store instruction and retrieved by the load instruction during the communication establishment step discussed above.

When a store instruction is retiring, it checks its address against the addresses in the speculative loads table as before. We have the following possibilities:

(a) The retiring store instruction is the collapsed instruction. In this case, values will always match. However, if addresses are different then the dependency has been incorrectly predicted. The exception bit is set. In either case, the visited bit is also set.

(b) The retiring store instruction is a store between the provider store instruction and the load. In this case, if the addresses match and values are different, the exception bit is set. If addresses match but values are the same, the exception is reset. Although the dependency has been mispredicted, the actual store instruction is value redundant and collapsing of the dependency did not cause harm.

When the load instruction is retiring, it now checks the visited field in the speculative loads table. If it is not set, then the store instruction which has been involved in dependency collapsing has retired before an address comparison could be made. In this case, a memory access has to be performed and the value obtained from the memory has to be compared against the value load is holding. If these values are different the machine state must be repaired.

Verification with Zero Memory Access. Verification with zero memory access can be achieved using a simple mechanism at the expense of increased complexity in the instruction window's dependency checking mechanism. This is achieved by forwarding the store address when it is computed to the load instruction which is involved in the dependency collapsing. In order to implement this mechanism, the store instruction is split into two micro operations, namely, address computation and value storing. The address computation part obtains a physical register and stores the register number in the LFST table. Upon establishing the communication link, the load instruction is made register dependent on: (a) the producer instruction's result register; (b) its own base address register; (c) store instruction's address computation register. With this establishment, there is no need to have the visited flag. Instead, when a sentinel load is ready to issue, it compares the address it has calculated against the address it has received from the store instruction. If there is a mismatch, the load instruction sets its own exception bit. Please note that the exception bit may be reset as usual by the intervening stores if their addresses as well as values match.

8.7 Concluding Remarks

In this chapter, an effective mechanism for reducing false memory dependencies when using a memory dependence predictor has been presented. It has been shown that full out-of-order issuing of store instructions in the instruction window can be allowed using the new memory order violation detection mechanism. In addition to allowing out-of-order issuing of the store instructions, the presented mechanism takes advantage of the value redundancy present in programs. There are two types of value redundancy that the mechanism exploits: (a) Value redundancy through a stale value in memory; (b) Value redundancy that is present among store instructions which co-exist in the instruction window and store a value to the same memory location. The first type of value

redundancy can also be exploited by the value prediction mechanism, whereas the second one can be exploited by load-store pairing mechanisms. In either case, the proposed mechanism does not need additional hardware to take advantage of the value redundancy.

Developed solution is an orthogonal solution that can be utilized with other types of memory dependence predictors as well. For example, the scheme proposed by Moshovos and Sohi based on MDPT/MDST associative structures [39] either forces a load to wait for all dependences predicted, or, MDPT entries are augmented to contain control flow information for each load/store pair. Using the new scheme, there would not be any need to force a load to wait for all dependences predicted or any need for augmenting predictor entries with control flow information. Similarly, the proposed back-end can also be used together with value prediction techniques [34, 36, 28, 9]. Specifically, a machine may employ selective value prediction to a subset of loads, whereas the remaining ones can synchronize through the dependence predictor employing the new back-end. The verification mechanism the scheme uses would work properly with value prediction mechanisms without modifications.

Finally, it is important to note how the performance of the scheme would compare with other predictive techniques. A recent study by Reinman and Calder studies the performance gains that can be obtained using various predictive techniques for load value speculation including the store set as well as value predictors [60]. This study reports that value prediction can out-perform the store set mechanism by about 20 %. Given that in their store-set study store instructions are not allowed to issue out-of-order and out-of-order disambiguator presented in this thesis out-performs both the original store set algorithm as well as an ideal memory dependence predictor by taking advantage of the value redundancy present in programs, further studies are needed to verify their conclusion that the value prediction out-performs all other techniques.

Chapter 9

Architecture Description Language - ADL

The evaluation of previous processor architectures and new microarchitectural techniques described in the previous chapters required detailed implementations of cycle accurate simulators for this purpose. These simulators need to be executed millions of machine cycles that general conclusions can be drawn regarding the performance of the technique being evaluated. As a result, tools are required to enable rapid development of cycle level simulators that are fast enough to carry out extensive simulation studies.

A commonly used approach for developing simulators is their hand coding in a general purpose language such as C. Examples of some popular simulators which were developed using this approach include the SPIM simulator for the MIPS architecture [32], the SimpleScalar simulator [7], and the SuperDLX simulator [41]. The hand coding of simulators is a substantial task which typically takes between 12 to 24 man months. Once developed, such simulators are difficult to *retarget* to a modified microarchitecture or an instruction set architecture without a significant amount of effort. Another problem that one encounters is the difficulty in *porting* these simulators to different platforms. The portability issue arises due to the need for handling of external system calls that are made by the benchmarks being run. Solutions that either disallow such calls or allow external calls but sacrifice portability by allowing the simulator to run only on a specific platform (e.g., SPIM) are undesirable.

An alternative to hand coding a simulator is to generate it automatically from a machine specification written in a domain specific language. Automatic generation not only significantly shortens the development cycle, it also allows retargeting since modifications in the architecture can be made at the specification level and the new simulator can then be automatically generated. Although a number of hardware description languages [1, 38, 54, 65] are available, these languages are not suitable for developing cycle level simulators. These languages are capable of defining the hardware to the smallest detail and result in simulators that are orders of magnitude slower than cycle level simulators. The retargeting of simulators requires significant effort and no solution to the portability problem is offered by these languages.

In order to allow rapid prototyping of required simulators, a domain specific language for specifying processor microarchitectures called the *Architecture Description Language* (ADL) has been designed and its compiler and run-time environment has been implemented in a system called the

Flexible Architecture Simulation Tool (FAST). Required simulators for this dissertation have all been generated automatically using the FAST system from architecture descriptions encoded in ADL.

ADL supports an execution model that is suitable for expressing a broad class of processor architectures. It provides constructs for specifying the microarchitecture elements such as pipelines, control, and the memory hierarchy including instruction and data caches as well as constructs for the specification of the instruction set architecture (ISA), the assembly language syntax and the binary representation. In order to provide portable operation, the language also incorporates a mapping between the calling convention of the simulated architecture and the machine that hosts the simulator. In this way, the simulator can perform external calls on behalf of the simulated program to achieve greater portability.

The language also incorporates built-in constructs for statistics collection as well as a language interface to a debugger so that the debugger can be invoked automatically when error conditions are encountered.

In the remainder of this chapter, in section 9.1, an overview of the Architecture Description Language is presented. In Section 9.2, language constructs which are designed for specifying the microarchitecture of the simulated architecture are described. In Section 9.3, the constructs which are used to define the instruction set architecture, general assembly syntax and the binary representation are presented. Next in Section 9.4, the calling convention specification which allows the automatically generated simulators perform system calls on behalf of the simulated program are outlined. In Section 9.5, examples of statistics collection and debugging related features are presented. Finally, in Section 9.6 the chapter is concluded with a brief discussion of unique characteristics of the ADL language.

9.1 Language Overview

An ADL program primarily consists of the description of a processor architecture which includes the specification of the instruction set architecture as well as the organization of the components of the microarchitecture. Before we discuss ADL in detail, let us first consider the model of execution used by ADL to express the operation of an architecture and highlight some of its design characteristics:

Explicit Instruction Flow and Instruction Context: In ADL the flow of instructions through the architectural components is explicit. The data associated with an instruction under execution is called the *instruction context*. The context is passed from one component to the next and is operated upon by the components till the execution of the instruction is complete. The context is allocated when the instruction enters the pipeline and is deallocated when the instruction retires.

The Machine Clock: The notion of machine clock is built into the language and the operation of the architectural components is described with respect to this clock. The machine clock is viewed as a series of *pulses*. Each discrete pulse is called a *minor cycle*, and a number of minor

cycles are grouped together to form a *machine cycle*. The minor cycles in ADL are represented by a series of labels. The first and the last minor cycles of a machine cycle are labeled as the *prologue* and the *epilogue* and those in between are labeled as *intermissions*. The actions of each component in the system during a machine cycle are divided into the operations that it performs in each of the minor cycles. During the prologue a component receives an instruction context from another component for processing, during the intermissions it operates upon the instruction context, and during the epilogue it sends the modified context to another component. Figure 9.1 shows the clock of a machine in which the major cycle is composed of λ minor cycles.

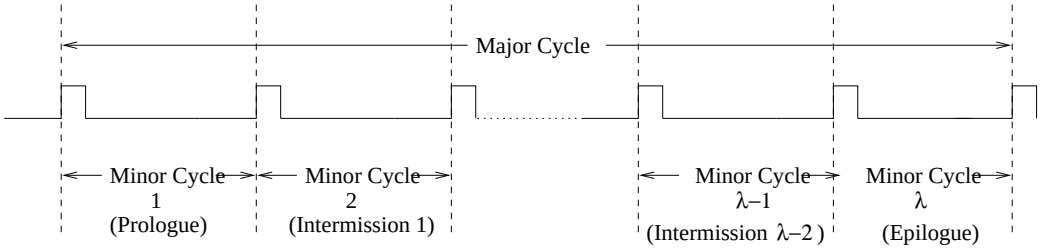


Figure 9.1: ADL Clock Labeling

Artifacts and Processing Stages: The architectural components are divided into two categories: *artifacts* are components with standard well known semantics that are directly supported by the language and *stages* are components whose semantics must be explicitly specified as part of the ADL program.

Examples of artifacts include caches, memory units, and register files. Since they are directly supported by ADL as built-in *types*, the programmer can use them by simply declaring objects of these types in an ADL program. Access to artifacts takes the form of assignments to and from the artifact variables. Different implementations of these components can be used by specifying different attribute values for the artifacts. The interaction of an artifact with the machine clock is also specified as a list of attributes.

Processing stages are architectural components that exhibit a significant functional variety. Their operation is dependent on the microarchitecture as well as the current instruction being processed. Furthermore, the function such an element performs is tightly coupled with the system clock and the status of other components in the system. Thus, it is not feasible to follow a declarative approach for stages but instead the user must explicitly specify their semantics using Register Transfer Level (RTL) statements.

Separation of Instruction Set Architecture and Microarchitecture Specification: The ISA specification is separated from the microarchitecture specification to facilitate the development of different microarchitecture implementations for the same ISA or extend an ISA by adding new instructions without altering the microarchitecture. The above separation has the following consequence on the specification of stage semantics. The RTL statements describing the semantics of stages are divided into two components: the *general component* that is common to all instruc-

tions and the *ISA-component* which depends upon the specific instruction being processed. The former is specified in the microarchitecture description while the latter is included as part of the ISA specification.

Time Annotated Actions and Parallelism in the Microarchitecture: The specification of the actions associated with the execution of specific instructions as well as the actions associated with various architectural components are annotated with timing information so that it can be determined when they are to be performed.

The procedures that implement the *general component* of actions associated with a processing stage carry the name of the stage and the label of the minor clock cycle during which they are to be executed. Such procedures are referred to as *time annotated procedures* (TAPs). Since there are λ minor cycles, there may be up to λ TAPs for a given stage. The *ISA-component* associated with an instruction is labeled with the name of the processing stage and optionally with the label of the minor cycle during which it must be executed. These statements are referred to as *labeled register transfer level* (LRTL) segments.

Parallelism at the architecture level is achieved by executing in each machine cycle the actions associated with each component during that cycle as well as actions associated with an instruction that are annotated with the current cycle. The machine execution is realized by invoking each TAP corresponding to a minor cycle as the clock generates the corresponding label and the parallel operation of individual components is modeled by concurrently executing all TAPs which have the same annotation. During this process, LRTL segments corresponding to the currently processed instruction are *fused* together with the corresponding TAP. The operation of a machine can be described as follows:

```
do forever
   for clock.label := prologue, intermission 1, ... intermission (λ − 2), epilogue do
      ∀ TAP, TAP.annotation = clock.label do
      { process {TAP; TAP.instruction.LRTL } }
end
```

9.2 Microarchitecture Specification

The specification of the microarchitecture consists of describing the artifacts of the architecture, declaring pipelines involved and their stages, specifying instruction contexts, and finally defining TAPs for each of the stages. In the following sections, a simple pipelined architecture shown in Figure 9.2 will be used to discuss each of these steps. In this architecture, the instruction fetch stage (IF) fetches instructions from the instruction cache and ships them to the instruction decode (ID) stage. ID stage decodes the instructions it receives, fetches their operands from the register file, and sends them to the execution unit (EX). The memory access (MEM) stage performs a data memory access for the load and the store instructions, but other instructions pass through this stage

unchanged. Finally, the write back (WB) stage writes the results back to the register file. In order to eliminate pipeline stalls that would otherwise result, data values are forwarded through forwarding paths to the earlier stages.

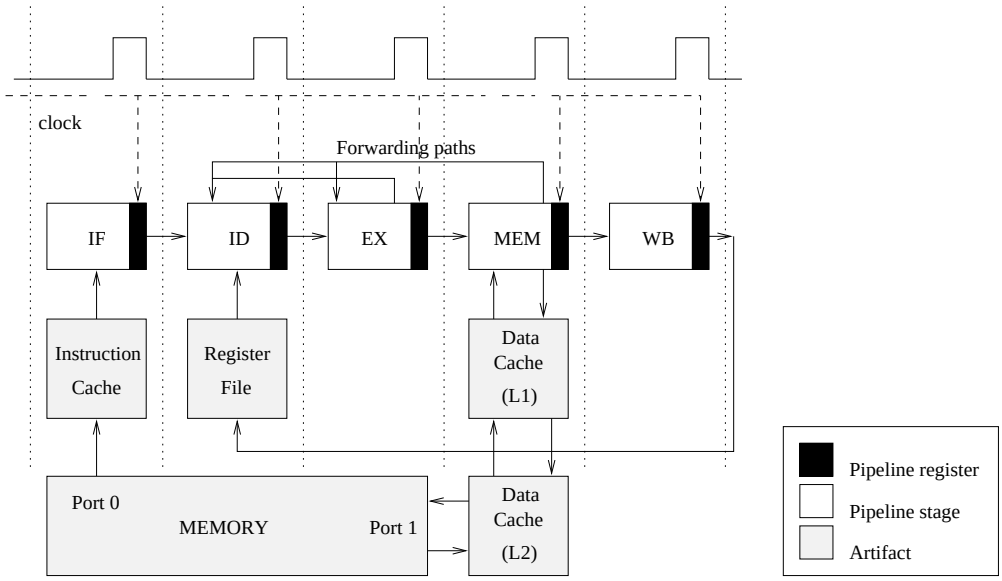


Figure 9.2: A Simple Pipelined Processor

Artifacts: Artifacts are hardware objects with well-established operational semantics and they are supported as built-in *types* by the language. A declaration of an artifact supplies the values of the *attributes* of the artifact to derive a specific implementation of the artifact. For an artifact, it is also specified how long does it take to process a single request in terms of clock cycles (i.e., the *latency*), the rate at which new requests can be issued to the artifact (i.e., the *repeat rate*), and the maximum number of requests that can be outstanding in a clock cycle (i.e., the number of *ports*). The list of the different types of artifacts supported by the language is given below.

$$
\begin{aligned}
\mathit{artifact\text{-}declaration} \quad & \Rightarrow \mathit{register\text{-}declaration} \\
& \mid \mathit{register\text{-}file\text{-}declaration} \\
& \mid \mathit{memory\text{-}port\text{-}declaration} \\
& \mid \mathit{cache\text{-}declaration} \\
& \mid \mathit{buffer\text{-}declaration} \\
& \mid \mathit{token\text{-}declaration}
\end{aligned}
$$

A *register* declaration declares an artifact of type simple register while a *register file* declaration declares an array of registers. Registers and register files may be given the attribute *shadow* which makes them invisible to the instruction set. ADL allows definition of one or more *aliases* for the individual register file entries. A *memory* declaration defines a memory port with a given access latency in units of machine cycles and a data path width in units of bits. For the *cache* artifact,

attribute values include degree of set associativity, the kind of replacement strategy, and whether it is a write-back or write-through cache. Memories, caches and buffers have an important property of being *stackable*. This property is required for building memory hierarchies. When an artifact is declared, the name of the artifact immediately lower in the hierarchy is mentioned using the *of clause*, effectively placing the new artifact higher in the hierarchy.

```
shadow register temp 16;              # A 16 bit temporary register.
register file  gpr [32,32]            # 32 registers,32 bits each.
               $zero 0,               # $zero is another alias for gpr[0]
               $at 1,                 # $at is another alias for gpr[1]
               $v0 2,
               .....
               $sp 29,
               $fp 30,
               $ra 31;
memory mport0 latency 12 width 64, # 64 bit path to memory.
mport1 latency 12 width 64;           # 64 bit path to memory.

instruction cache icache of mport0 directmapped 64 kb 4 wpl;
data cache l2 of mport1 directmapped 64 kb 4 wpl,
            l1 of l2 4 way 8 kb 4 wpl;
```

Figure 9.3: Example artifact declarations

A sequence of artifact declarations for the example pipelined architecture of Figure 9.2 is shown in Figure 9.3. The first declaration declares a temporary register invisible from the instruction set. Next a register file `gpr` is declared and individual registers in the file are assigned aliases. The names `$0`, `$sp`, etc, are ISA visible since the register file itself is ISA visible. RTL statements may use either form of access (i.e., `gpr[31]` or `$ra`). The declaration specifies two memory ports with 12 cycles of access latency and 64 bit data paths. The memory port `mport0` hosts a direct mapped instruction cache of 64 kilobytes with 4 words per cache line. Memory port `mport1` hosts a direct mapped cache of similar attributes and this direct mapped cache in turn services a four way set associative cache of size 8 kilobytes. Thus, the cache L1 is at the highest level in the hierarchy and the memory ports are at the lowest level.

Once declared, artifacts are accessed just like variables by the RTL statements in the specification. For complicated structures, such as data caches, passing of additional parameters may be required. For example, in order to store a single byte to the L1 cache, and retrieve a halfword, the following sequence of RTL statements could be used:

```
l1.(_BYTE) [addr] = data_value;
data_value = l1.(_HALFWORD) [addr];
```

When an artifact is accessed, the status of the result is queried using the `access-complete` statement. This statement returns a true value if the operation has been completed successfully, and a false value otherwise. A false value may be returned because the artifact is slow, such as in the case of memory-ports, or because there is a structural hazard. In these cases the request must be repeated. Further details of why the operation was not successful may be queried using additional statements.

Processing Stages and Instruction Context Declarations: The primary means of declaring stages of the microarchitecture is the *pipeline declaration*. A pipeline declaration specifies an ordering among pipeline stages such that each stage receives an instruction context from the preceding stage and sends the processed context to a later stage. There may be more than one pipeline declaration in an ADL program but the stage names must be unique. Once a stage is declared using a pipeline construct, TAPs may be specified for each of the stages and semantic sections of instruction declarations may utilize the stage names as LRTL labels. The following declaration defines the pipeline for the example architecture:

```
pipeline ipipe (IF, ID, EX, MEM, WB);
```

In ADL, the set of data values carried along with pipeline stages are grouped together in a structure called `controldata`. There is only one such declaration, which means all stages have the same type of context, and the *instruction context* is the *union* of the data required by all the pipeline stages in the system. While in a hardware implementation pipeline stages may carry different types of contexts, definition of instruction context in this way simplifies the transfer and handling of instruction contexts in the simulator. Since there is a uniform single instruction context for all pipeline stages, each pipeline stage name is also an object of type `controldata`. The following is a simple `controldata` declaration for a pipelined machine:

```
controldata register
        my_pc 32,           # Instruction pointer for the instruction.
        simm 32,            # Sign extended immediate.
        ....
        dest 32,            # dest holds the value to be written.
        lop 32,             # lop holds the left operand value.
        rop 32;             # rop holds the right operand value.
```

Elements of the `controldata` structure may be accessed from TAPs and by the semantic parts of instruction declarations (i.e., LRTLs). Access to the elements of the structure may be qualified or unqualified. When they are not qualified, the pipeline stage is the stage of the TAP that

performs the reference or the label associated with the LRTL segment that performs the reference. In its qualified form, the syntax `controldata-element[stage-name]` is used to access the instruction context of another stage. This form is primarily used to implement internal data forwarding by either the source stage writing into the context of the sink stage or the sink stage reading the data from the context of the source stage.

Specifying Control and TAPs: The machine control is responsible for checking the conditions for moving the pipeline forward, forwarding the instruction context from one stage to the next, controlling the flow of data to and from the artifacts, and introducing stalls for resolving data, control, and structural hazards. In ADL, the semantics of the control part of the architecture is specified in a distributed fashion as parts of TAPs by indicating how and when instruction contexts are transferred from one stage to another.

The movement of an instruction context through the pipeline, from one stage to the next, is accomplished through the `send` statement. The send is successful if the destination stage is in the idle state or it is also executing a `send` statement in the same cycle. All pipeline stages execute the `send` statement during the epilogue minor cycle. In the normal pipeline operation, an instruction context is allocated by the first pipeline stage using the ADL statement `new-context`. This context is then filled in with an instruction loaded to the instruction register. When this stage finishes its processing, it executes the `send` statement to send the context to the downstream pipeline stage. When a context reaches the last pipeline stage it is deallocated using the ADL statement `retire`. If any of the pipeline stages does not execute a send, send operations of the preceding stages fail. In this case, they repeat their send operations at the end of next cycle. For decoding the instructions, ADL provides a `decode` statement. The `decode` statement does not take any arguments and establishes a mapping from the current context to an instruction name. This mapping is fully computable from the binary section of instruction declarations. Once decoded, all the attributes of the instruction become read-only `controldata` variables and are accessed accordingly.

The conditions for internal data forwarding can be easily checked by the stage that needs the data. For example, the TAP for the ID stage in the example pipelined machine may check if any of the stages EX and MEM has computed a value that is needed by the current instruction by comparing their destination registers with the source registers of the instruction currently in the ID stage. If that is the case, the stage reads the data from the respective stages instead of the register file.

For the handling of artifact data-flow and the handling of various hazards, ADL provides the `stall` statement through which a stage may stall itself. The `stall` statement terminates the processing of the current TAP and the remaining TAPs that handle the rest of the machine cycle. The net effect of the `stall` statement is that no `send` statement is executed by the stage executing the stall in that machine cycle.

In addition to the `stall` statement, ADL also provides statements to `reserve` a stage, `release` a stage, and `freeze/unfreeze` the whole pipeline. When a stage is reserved, only the *instruction* that reserved it may perform a send operation to that stage, and only this instruction can release it regardless of where in the pipeline the instruction is at. When a stage executes a

```
instruction register ir;
stall category mem_ic,ld_d_dep,pool_full;

(a) procedure ID epilogue
    begin if i_type[EX]== load_type &
              (dest_r[EX]==lop_r | dest_r[EX]==rop_r) then
              stall ld_d_dep;
    end ID;
(b) procedure IF prologue
    begin ir=icache[pc];
          if access_complete then
            begin unfreeze; pc=pc+4 end
          else
            begin freeze; stall mem_icl end;
    end IF;
(c) pipeline RSPOOL(RSTA[64]);
    procedure ID epilogue
    begin reserve_unit RSTA my_pc;
          if ! access_complete then stall pool_full;
    end ID;
```

Figure 9.4: Handling of Hazards.

`freeze`, all stages except the stage that executed the `freeze` statement will stall and only the stage that executed the `freeze` statement may later execute an `unfreeze` statement.

Examples of hazard handling using these statements are shown in Figure 9.4. Figure 9.4(a) indicates the case where the result of a load instruction may be used immediately by the next instruction. Such data hazards cannot be overcome by forwarding alone and therefore require insertion of *pipeline bubbles*. The stage in this case checks for the condition by examining the context of the EX stage and its destination register and stalls appropriately. Because of the stall, the ID stage does not execute a `send` in this cycle. Since the send operations of following stages are not effected by the `stall` of prior stages, the EX stage enters the next cycle in an idle state which is equivalent to introducing a pipeline bubble. An instruction cache miss in a pipelined architecture is usually handled by freezing the machine state. In Figure 9.4(b), the instruction fetch stage executes a `freeze` statement whenever there is a cache miss. A `stall` is also executed so that the epilogue will not attempt to execute the `send` statement. Note that an `unfreeze` is always executed whenever the cache access is successful. Executing an `unfreeze` on a pipeline which is not frozen is a null operation. In this way, the stage code does not have to be history sensitive. Finally in Figure 9.4(c), a structural hazard and its handling is illustrated. The example shows one possible way to implement a unified pool of 64 reservation stations using an array of stages for the Tomasulo's algorithm [68]. The ID stage attempts to reserve a unit from the pool of reservation stations. If the `reserve` statement is unsuccessful, the stage executes the `stall` statement.

9.3 ISA Specification

The ISA is specified by means of instruction declarations which describe the syntax and semantics of both the machine instructions and the macro instructions using a uniform syntax given below:

instruction-declaration ⇒ *machine-instruction-declaration*
| *macro-instruction-declaration*

machine-instruction-declaration ⇒ *syntax-part* `emit`
binary-part *semantic-part*

macro-instruction-declaration ⇒ *syntax-part* `macro`
semantic-part

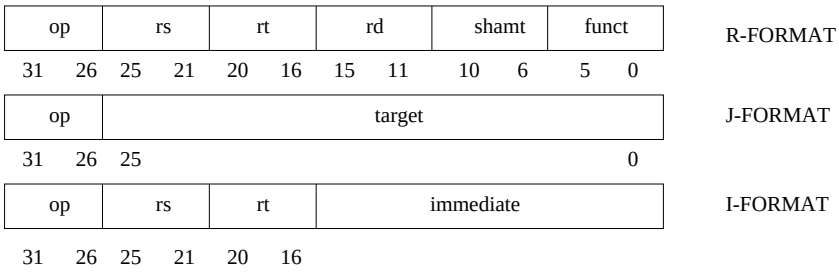
There are three major components of the instruction specification. These are the *syntax-part*, the *binary-part* and the *semantic part*. The syntax part and the binary part together define how the assembler should parse instructions and generate the appropriate binary encoding of them. The binary part is also used to automatically generate the decoder for the implementation of the `decode` statement discussed earlier. The semantic part of a machine instruction description is a list of LRTL segments describing what each stage should compute when the instruction is processed by the stage, whereas the semantic part of a macro instruction description specifies how the assembler should generate machine instructions from the macro specification.

Generation of a binary encoding of an assembly instruction involves three steps. These are the parsing of the assembly instruction, extracting the values of any instruction fields which are derived from the assembly instruction, and packing these values in an instruction format. The instruction format for an instruction is a sequence of fields making up the instruction word. Some of the instruction formats for the MIPS architecture are shown in Figure 9.5(a).

ADL defines instruction fields by associating a *start bit* and *field width* pair with a name. The same pair may be defined multiple times using different names since the same pair may have a different purpose in a different instruction format. If a field has a constant value for all the instructions in the instruction set, it is declared to be a *constant* field. Otherwise, it is declared to be one of the ADL types *register*, *integer* or *signed integer*. Such fields are considered to be *variable* fields. Variable fields typically get their values from the assembly instruction when such an instruction is parsed by the assembler. The instruction fields are specified using the *declare* construct.

declare-construct ⇒ `declare` *declarations*
declarations ⇒ *field-declaration*
| *variable-declaration*
| *temporary-declaration*

field-declaration ⇒ *field-name*
(`constant` | `integer` | `register` | `signed`)
`field` *start-bit* *field-width*



(a) MIPS formats

```
declare   op          constant   field   31   6,
          rs          register   field   25   5,
          rt          register   field   20   5,
          rd          register   field   15   5,
          shamt       integer    field   10   5,
          funct       integer    field   5    6,
          target      integer    field   25   26,
          immediate   integer    field   15   16,
```

(b) Field Declarations

Figure 9.5: Instruction format specification

Examples of field declarations for the MIPS instruction formats are given in Figure 9.5(b). Field declarations alone are not sufficient to describe the binary encoding of an instruction. We also need to define which fields make up the instruction (i.e., the instruction format) as well as how their values are computed. Instead of defining separate instruction formats and then mapping instructions to these formats [12], ADL chooses to specify the instruction format as part of the instruction's binary part. The binary part of each instruction is represented as a sequence of *field expressions*. A field expression is the assignment of a value to a field of the instruction. The value assigned to a field may be a constant, a constant expression, or, it may reference a value to be derived from the assembly instruction by the syntax-part. The ADL syntax for the syntax-part and the binary-parts of an instruction declaration are given below:

syntax-part ⇒ *instruction-mnemonics argument-list*
argument-list ⇒ *argument* | *argument argument-list*
argument ⇒ *label-variable* | *field*

binary-part ⇒ *field-expression*
| *field-expression binary-part*
field-expression ⇒ *field* | *field* = *constant* | *field* = <*fog-list*>

fog-list ⇒ *fog-predeclared* | *fog-list* . *pure-function*
fog-predeclared ⇒ *label-variable* `.base`
| *label-variable* `.offset`
| *label-variable* `.absolute`
| *label-variable* `.delta`
| *label-variable* `.segoffset`

The syntax part of an instruction declaration is a list of *arguments* defined to be either *label variables* or fields. A field in the argument list means that the assembler should expect to find an object of the corresponding type such as a register or an integer constant at the corresponding position of the assembly instruction. A label variable represents an address primary. Examples of address primaries include labels, base/offset pairs, and any constant arithmetic on labels. Field expressions given in the binary-part may query the values of the arguments of the instruction using pre-declared functions such as *base*, *offset*, *absolute*, or *delta*, or substitute them directly. These values may also be transformed by using *pure functions* which are functions which have a single parameter and return a single transformation of this parameter.

Let us now see how the assembler could parse an instruction using the specification shown in Figure 9.6 and generate the appropriate binary. In the example, the argument part consists of a register field (`rt`), and a label variable (`address`). Therefore, the assembler expects to find a register name followed by a sequence of tokens which can be reduced to an address primary when a `lw` mnemonic is detected in the input stream. The field expressions in the binary part indicate that the opcode field must be set to the constant value of 35, `rs` field must be given the base register number representing the address, and the immediate portion must be given the offset representing the address. Since the `rt` field appears in the argument list, it gets a register number from the parsed instruction. ADL representation of binary encoding is a concise representation and is natural. Similar encoding techniques have been employed in the SLED approach [57, 58].

Specifying Instruction Semantics: The semantic-part of an instruction specification serves two purposes. These are the specification of what each stage computes when such an instruction is received and instruction classification so that stages may apply operations specific to a class of instructions. For example, branch instructions may be handled by a specific stage which requires that the type of an instruction be known so that proper instruction steering can be performed.

The instruction specific operations of stages are specified using LRTL segments. A LRTL segment is a program segment that consists of register transfer level statements where each block of such statements are labeled using a stage name. The syntax of the LRTL segment is depicted below.

LRTL-segment ⇒ `begin` *labeled-RTL-list* `end`
labeled-RTL-list ⇒ *labeled-RTL* | ; *labeled-RTL-list*
labeled-RTL ⇒ `case` *stage-name RTL-statement-list* `end`

The classification of instructions is achieved using an optional *instruction attributes* section where the attributes of the instruction are specified. These attributes can be queried by pipeline stages upon receiving the instruction. An instruction attribute is a member of the global enumeration defined by the *attribute declaration* given below:

attribute-declaration ⇒ *identifier* : *attribute-list*
attribute-list ⇒ *name-list* | `integer`
name-list ⇒ *identifier* | *identifier* , *name-list*

Since an attribute of an instruction classifies an instruction, values of attributes must be specified for all the instructions. An example attribute declaration section that classifies instructions according to their operation types is shown below:

```
attributes
    i_type      : alu_type,
                  conditional_direct,
                  conditional_direct_link,
                  unconditional_direct,
                  unconditional_direct_link,
                  unconditional_indirect,
                  unconditional_indirect_link,
                  load_type,
                  store_type;
```

Let us examine the semantic part of the lw instruction declaration shown in Figure 9.6. This instruction has the `i_type` attribute `load_type`, and LRTL segments `ID`, `EX`, and `MEM` define the operations each of the corresponding stages. The LRTL segment ID performs a sign extension using powerful ADL bit operations. The sign extension is achieved by repeating the bit 15 of the immediate field (`|<` operator) for 16 bits and then concatenating (`||` operator) it with the field itself. The result is then stored into the variable `simm`. The LRTL segment EX performs an address computation by adding the contents of the variable *lop* with the sign extended value computed by the ID stage. Similarly, the LRTL segment MEM performs a data cache access using the value computed in the EX stage and stores the returned value into the variable *dest*. Since writing back the results of instructions into the register file is common for all instructions, this task is handled by TAPs.

The address space of a TAP consists of the global address space implemented by the artifacts and the local address space defined by the instruction being currently processed. In Figure 9.6, the variables *simm, dest_r, lop* are part of the local address space or the *instruction context*. When the execution of a TAP is completed, the local address space is *transferred* to another TAP instead of being deallocated. Typically, the next TAP that executes in the same context is the TAP belonging to the same stage that has the next clock label. When the TAP that has the label epilogue is executed, the context is either transferred to the prologue TAP of the same stage or to the prologue TAP of another stage.

```
lw rt address
    emit opcode=_lw rs=<address.base> rt immediate=<address.offset>
    attributes(i_type: load_type, dest_reg: rt)
    begin
        case s_ID
            simm=sign_extend_16(immediate);
        end;
        case s_EX
            lmar=lop + simm;
        end;
        case s_MEM
            dest=ncache [lmar];
        end;
    end,
```

Figure 9.6: MIPS Load Word Instruction

Macro Instructions: Most compilers available today (e.g., gcc) make use of macro instructions in code generation. The task of converting these instructions into actual machine instructions is left up to the assembler. ADL handles macro instructions in a manner similar to machine instructions. The syntax part of the instruction has the same syntax, but no field variables are allowed in the argument part. Therefore, all of the instruction arguments are variables. Since macro instructions themselves do not directly lead to a binary representation, there is no binary generation part. The macro specification can be visualized as a procedure where the procedure arguments correspond to the instruction arguments and the semantic part corresponds to the body of the procedure. The procedure defines what instruction(s) should be generated given a particular instance of arguments. Instructions to be generated are specified using an *instruction call* statement that generates a machine instruction by passing the values of the fields of the instruction as parameters. The syntax for the instruction call statement is shown below.

$$\textit{instruction-call} \quad \Rightarrow \textit{instruction-mnemonics} \ : \ \textit{field-assignment-list}$$

An example macro declaration for the MIPS load immediate instruction is shown in Figure 9.7. This macro generates either a single instruction (`ori`) or a pair of instructions (`lui, ori`) depending on the size of the immediate field.

9.4 Calling Convention Specification

The purpose of the calling convention specification is to enable the simulator to perform external system calls on behalf of the simulated program so that operating system services can be provided through the operating system of the host machine. For this purpose ADL provides a calling convention section where the calling convention of the simulated architecture and the prototypes of

```
declare rdest register variable,
        src2  integer variable,
        tx    integer temporary,
        ty    integer temporary;
instruction li rdest src2 macro
begin tx=src2.[31:16];
      ty=src2.[15:16];
      if (src2.[31:17] == 0x1ffff) | (src2.[31:17] == 0) then
        ori:rt=rdest rs=0 immediate=ty
      else
        begin lui:rt=rdest immediate=tx;
              ori:rt=rdest rs=rdest immediate=ty;
        end;
end;
```

Figure 9.7: Macro Instruction Example.

external references are specified. From this specification, an engine is generated that can execute an external procedure by passing the values of the parameters from the simulated architecture and returning the results back into the simulator. This approach allows the language user to specify external references of a program and treat them as if they are single instructions.

The calling convention specification is based on the formal model and specification language for procedure calling conventions by Bailey and Davidson [5]. Their language has been modified so that it fits the general structure of the ADL language. The specification provides a mapping to a register or a memory location, given an argument's position and type in the procedure call. Since an argument's value may not have been written to the memory cell or to the register file at the time of the call, the mapping has been modified so that each register identifier that may be used to pass arguments to the callee and each stack alignment are associated with a *supplier procedure*. Supplier procedures are microarchitecture specific procedures that return the value of the argument at the time of the call. In a pipelined architecture, the supplier procedure may return the value from an artifact if there are no instructions in the pipeline that are computing the value, or the value may be returned from a stage if the value has been computed, but did not yet reach the write-back phase. If the value is available and is being returned, the procedure sets the built-in variable `access-complete` to true. In the case that more cycles are necessary before the value becomes available, the `access-complete` variable is set to false. An example calling convention specification for the MIPS architecture is given in Figure 9.8.

The calling convention specification consists of two sections, namely a *data transfer* section which describes how arguments are allocated into the registers and the stack locations, and a *prototypes* section, where prototypes of external procedures and names of external data addresses are supplied. The data transfer section consists of *argument declarations*, *set declarations* and a *map declaration*. Argument declarations associate either a register name with a supplier procedure name, or a stack alignment name with a supplier procedure. For example, in Figure 9.8, argument register

```
calling_convention begin
  argument $4:int_p1, $5:int_p2, $6:int_p3, $7:int_p3;
           $f12:flt_p1,$f13:flt_p2,$f14:flt_p3,$f15:flt_p4;
  unbounded stk4: stk_p4, stk8: stk_p8;
  set intregs($4,$5,$6,$7,stk4),
      intfpregs(<$4,$5>,<$6,$7>,<stk8,st4>),
      fpfpregs (<$f12,$f13>,<$f14,$f15>,<stk8,stk4>);
  equivalence ($4,$f12), ($5,$f12), ($6,$f14), ($7,$f14);
  typeset singleword(int, void *, ...), doubleword(double, ...);
  map argument.type begin
    singleword : intregs;
    doubleword : map argument[1].type begin
                    singleword: intfpregs;
                    doubleword: fpfpregs;
                 end map;
  end map;
  prototypes begin
     reference errno, sys_errlist ...
     double cosh(double); int printf(int,...);
  end;
end calling_convention;
procedure int_p1()
begin
   int_p1=gpr[4];
   access_complete=( has_context EX |
            has_context MEM | has_context WB)==0;
end int_p1;
```

Figure 9.8: MIPS Calling Convention Specification

`$4` is associated with the supplier procedure `int_p1`. Stack alignment names are declared using the `unbounded` keyword and correspond to an unlimited pool of argument values starting at a given alignment of the frame pointer for the architecture. Supplier procedures for stack alignment names do the required alignment first and return the first word at the indicated location. The register names and stack alignment names given as part of the argument declarations are called *argument locations*.

`Set` declarations create ordered pools of arguments based on types. In our example, the set `intregs` creates a pool of argument values which consists of four integer registers and an unbounded pool of stack locations. Thus, a call site that requires six integer arguments would find the values of its first four arguments in the registers `$4, $5, $6, $7`, and the remaining two on the stack. In some architectures, if one register is used, some other registers can no longer be used for the following arguments. For example, in the MIPS architecture, if the floating point register `$12` is allocated, integer registers `$4` and `$5` cannot be used to pass the following integer arguments. The specification handles this problem by creating *equivalence* sets given by the *equivalence* declaration.

Register pairs listed in an equivalence declaration are removed together from the respective sets when one of them is allocated.

`Typeset` declarations group variable types that map to the same sized objects. Once the sets and typesets are defined, a `map` declaration creates a mapping from typesets to sets. For each argument type, first the typesets are consulted to find the corresponding typeset. Next the typeset is supplied to the map construct to find the set from which the argument value(s) should be obtained. These sets are consumed one by one for each argument value that is needed. The `map` declaration in the example in Figure 9.8 specifies that any arguments which have a type listed in the singleword typeset will consume the set `intregs` while those which are members of the doubleword set select the set based on the type of the first argument.

The prototypes section is an ADL extension to the calling convention specification which is necessary to call external procedures. This section consists of a list of external procedure prototypes and data reference names used by the benchmark programs. Both procedures and data references can be renamed to match the names of the architecture so that greater portability is achieved.

The calling convention specification when complied, provides an interface that returns a list of supplier procedures given a call site. This interface is used by the simulator to assemble the argument values, perform the external call on behalf of the simulated program and return the values.

9.5 Statistics Collection and Debugging

ADL provides support for assisting the user in collection of statistics that may be required to evaluate the specified architecture. An *instruction category* declaration is supported using which the user can classify instructions into different categories. The counts for the number of retired instructions in each of these categories are provided to the user by the generated simulator. The stall statement may be followed by an optional *stall category name*. In this form, the stall is registered under the mentioned category for the current instruction and the stall statistics for each of the categories are reported to the user. This can be helpful in identifying performance bottlenecks.

More advanced customized statistic collection is also possible. The ADL programmer can insert statements into the ADL program to collect special purpose statistics. For this purpose, ADL provides a *statistics declaration* which accepts a register name and a format specifier string. At the end of execution, the value of the register is printed using the supplied format. The example in Figure 9.9 shows how one could count the number of branch-delay slots which are not filled with useful instructions by the compiler. In this example, the TAP for the EX stage checks if the instruction in EX is a branch instruction and the instruction in the ID stage is a null operation which has an opcode field of zero.

Interaction with the debugger can also be specified in an ADL program. The debugger can be entered through the ISA specification by using the ADL statement *pause*. In general, when an unexpected condition is detected, this statement may be used to enter the debugger. For example, a divide instruction may check for a zero operand and execute *pause* statement as part of an LRTL

```
statistics "Total number of branches %d:",branch_count,
           "Empty slots %d:",empty_slots;
procedure EX epilogue
begin if i_type == branch_type1 | i_type == branch_type0 then
          begin branch_count=branch_count+1;
                if op[ID] == 0 then
                  empty_slots=empty_slots+1;
          end;
          .......
end;
```

Figure 9.9: Language Support for Gathering Statistics

segment. The registers whose contents are desired by the user to be displayed when the debugger is entered can also be specified in the ADL program through the *monitor* declaration.

9.6 Concluding Remarks

In this chapter, an overview of the basic properties of ADL has been presented. ADL is a unique language in many aspects, including the simple domain model it uses as well as the comprehensive solution it presents where a significant portion of the system software is also automatically generated from ADL specifications. The language embodies sufficient information to enable automatic generation of the compiler back-ends as well, although this aspect of the language has not been explored yet. Because of these properties, it is expected that ADL is going to be a very useful language for advancing microarchitecture research in the future as well.

In the next chapter, an implementation of the language called the *Flexible Architecture Simulation Tool* (FAST) is presented along with engineering details that had to be overcome before a running compiler for the language could be implemented.

Chapter 10

FAST - Flexible Architecture Simulation Tool

In this chapter an implementation of ADL which includes a compiler as well as the necessary run-time environment is presented. This integrated system has been named the *Flexible Architecture Simulation Tool* (FAST). The system provides: (a) an implementation of a compiler for ADL through which a cycle level simulator is generated automatically from an ADL processor description; (b) automatic generation of support software tools including the assembler, disassembler and the loader/linker for the architecture; (c) a cycle level assembly language debugger that assists in tracing of program behavior; and (d) support software for displaying statistics and monitored information.

The system has been used extensively for the implementation of the microarchitectures described in this dissertation as well as for other projects by graduate students. Implemented simulators ranged from simple functional simulators and pipelined RISC machines to very sophisticated speculative superscalar processors. Although the language is capable of describing almost any instruction set, the experience with this aspect of the system has been limited to the MIPS ISA [56, 22]. The resulting software can be used to compile and simulate very large benchmark programs. For instance, Spec95 integer and floating point benchmarks on a variety of architecture specifications as well as many other programs ranging from few hundred lines to 10,000 lines of C code have been simulated successfully. Upon the completion of the system, the first three simulators took a short period of 3 months to develop, demonstrating the ability of the system for rapid prototyping. The specifications of the architectures varied from 5000 to 6000 lines of ADL code while the sizes of automatically generated software varied from 20,000 to 30,000 lines of C++ code.

This chapter has been divided into the following sections. In Section 10.1 an overview of the FAST implementation is given. In Section 10.2, basic implementation of the ADL compiler is described. Section 10.3 presents the debugger facility which is an integral part of the system. Experience acquired using the system is discussed in Sections 10.4 and 10.5. Finally, in Section10.6 the chapter is concluded with a brief discussion.

10.1 Overview

The main components of FAST is illustrated in Figure 10.1. The core of the system is the ADL compiler, which is a single pass monolithic compiler that uses program specialization techniques to generate the desired software.

The generated software is synthesized using prototype modules called *templates*. A template is a prototype module of software that consists of only architecture independent components. For example, the assembler template contains a complete assembler with the exception of instruction set specific portions such as the mnemonics tables and specific rules to parse individual instructions and code that converts symbolic addresses to machine addresses. All these portions of an assembler are ISA specific and they are compiled in from the ADL program and filled in by the compiler. Similarly, artifacts have been implemented in another template file. For each instance of artifact declaration, the ADL compiler obtains the corresponding artifact declaration from the template file and generates the desired artifact implementation.

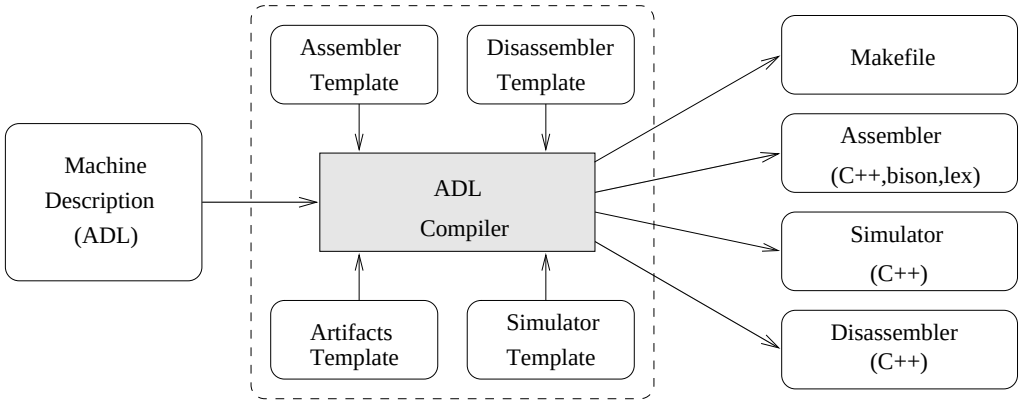


Figure 10.1: FAST Main Components

Generation of the software is accomplished by compiling the architecture description using the ADL compiler. Once the compilation is successful, the resulting software can be compiled by invoking the generated *Makefile* by the user so that the binaries for the simulator, assembler and the disassembler for the architecture are obtained.

Simulation of the benchmark programs is accomplished by first compiling them into the assembly language with the aid of a high level language (HLL)compiler as shown in Figure 10.2(a). The assembly modules are then assembled using the automatically generated assembler to generate the benchmark binaries as shown in Figure 10.2(b). Finally, these binaries are loaded by the automatically generated simulator and interpreted under the simulated architecture (see Figure 10.2(c)).

The simulator can be passed a number of command line arguments to direct its operation. Specifically, the simulations can be carried out until a desired number of simulation cycles. Once the desired cycles are reached the simulation may be terminated, or optionally the integrated debugger can be entered to single step through the simulated program while observing the status change in

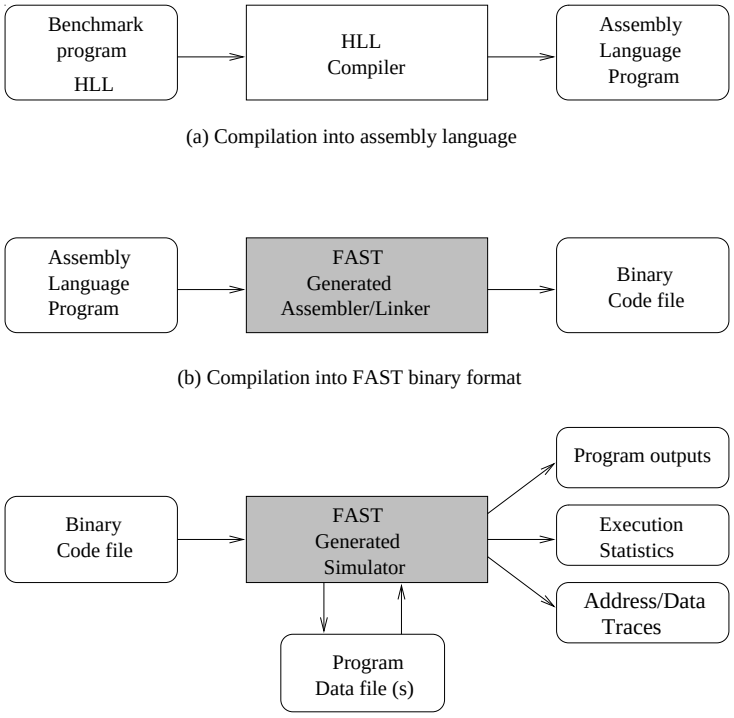


Figure 10.2: Three Steps of Program Simulation

the artifact values. The simulation environment directs the output from the simulated program to the window where the simulator is executed. In this manner, the simulations give the feeling of executing the simulated benchmark program in native mode, albeit slower. This approach also allows executing benchmarks in native mode and then under the simulator and comparing the two outputs by redirecting outputs to disk files.

10.2 The ADL Compiler

The ADL compiler uses separate representations to describe the ISA and the microarchitecture. Imperative code such as TAPs, general procedures and LRTL segments are each represented by a separate syntax tree and these trees emanate from the internal representation of components of the architecture. In case of ISA, LRTL syntax trees emanate from instruction descriptions which represent the assembly syntax, binary representation, and macro implementations. In case of microarchitecture representation, syntax trees emanate from pipeline descriptions.

The generation of the simulator system is accomplished by copying a template until a descriptor marker, indicating the position at which a component should be generated and placed, is encountered. The compiler generates a table, a procedure, or a C++ class described by the marker.

Once the required software element is generated, the scanning continues until another marker is found, or the end of file is reached.

10.2.1 Generating the Assembler

ADL generates a two-pass conventional assembler. During the first pass, the symbolic addresses are resolved. During the second pass, actual binary generation is performed. Generation of the assembler embodies the following steps: (a) generation of the mnemonics tables; (b) generation of the parsing rules; (c) generation of the semantic actions associated with the rules.

```
add,        op_add,        0, 4,
addi,       op_addi,       0, 4,
addiu,      op_addiu,      0, 4,
addu,       op_addu,       0, 4,
and,        op_and,        0, 4,
andi,       op_andi,       0, 4,
bc1f,       op_bc1f,       0, 0,
bc1f__,     op_bc1f__,     0, 4,
bc1t,       op_bc1t,       0, 0,
bc1t__,     op_bc1t__,     0, 4,
beq,        op_beq,        0, 0,
```

Figure 10.3: A Portion of Mnemonics Table

Generation of the Mnemonics Tables. By saving the instruction names encountered as part of the instruction declarations and eliminating the duplicate names, ADL generates simple tables to be implemented by GNU *gperf* tool. This tool generates the necessary hash functions automatically. Mnemonic tables return a simple enumerated value, given an assembler mnemonic. An example table generated by the compiler is illustrated in Figure 10.3. Constructs *op_<mnemonic>* are enumerations generated by the compiler and inserted into the resulting assembler.

Generation of the Parsing Rules and Semantic Actions The parsing rules necessary for parsing the assembly language program is generated in a straight-forward manner from the ADL instruction specifications. For this purpose, the compiler uses the type of the corresponding field declarations to each of the parameters of the instruction specification and generates the appropriate rule entry. For example in Figure 10.4(a), the instruction declaration of the assembly part has three identifiers, namely, *rd*, *rs* and *rt* all of which are declared to be *register fields*. The ADL compiler therefore uses the meta-symbol *register_id* which corresponds to this type to generate each of the *register_id* meta-symbols that appear on the rule. The output is a *yacc* input file that is processed by the parser generator yacc. A sample rule that is generated automatically from the instruction declaration shown in Figure 10.4(a) is illustrated in in Figure 10.4(b). It is easy to see that the values to be bound to the meta symbols are used to generate the corresponding binary.

```
add rd rs rt
    emit opcode=_special rs rt rd shamt=0 funct=_add
    ..........
```

(a) ADL Instruction Declaration

```
.....
enter_rule_add : op_add {any_identifier=true;}
enter_rule_addi : op_addi {any_identifier=true;}
.....

rule_add : enter_rule_add register_id ',' register_id ',' register_id
          { $$=$1;
          invoked_macro=0;
          instruction_type=2;
          if (pass2) then
            begin
              bitsy.emit_LE(0, 6); // w.[31:06] opcode = 0
              bitsy.emit_LE(int_val($4), 5); // w.[25:05] rs = int_val($4)
              bitsy.emit_LE(int_val($6), 5); // w.[20:05] rt = int_val($6)
              bitsy.emit_LE(int_val($2), 5); // w.[15:05] rd = int_val($2)
              bitsy.emit_LE(0, 5); // w.[10:05] shamt = 0
              bitsy.emit_LE(32, 6); // w.[05:06] funct = 32
              if (invoked_macro) then
                bitsy.Boundary(macro_write_instruction,yysline);
              else
                bitsy.Boundary(write_instruction,yysline);
            end
          any_identifier=false;
          }
```

(b) Generated Yacc Rules

Figure 10.4: Sample ADL Instruction Declaration and Generated Rule

Not all parsing rules can be generated in such a straight-forward manner. In those cases which involve *fog* constructs, function applications may result in the generation of additional instructions. For example, an instruction may need a base register/offset form of an address primary, which must be obtained from a label. Depending on the architecture, the process may involve generation of one or more instructions which would place the address into a register at run-time. These instructions must be processed by the assembler before the processing of the current instruction is completed. This is accomplished by saving the generated instructions in a queue as they are generated, and then generating a processed form of the current instruction and placing it at the end of the queue. The scanner is then told to switch its head to this queue and scan the queue instead of the

usual input. This process may be applied recursively (i.e. instructions which were so generated may generate additional instructions) until the scanner may return to scanning to usual input stream.

10.2.2 Generating the Decoder

One of the most important steps in the generation process is the generation of the *decoder*. Decoder generation involves two major steps. The first is the assignment of *opcodes* to individual instructions. The opcode in question is not an opcode assigned by the instruction set architecture, but a unique number that can be used to identify instructions internally both for disassembling and simulation purposes. Assignment of opcodes is carried out in a straight-forward way by assigning integers to instructions as they are encountered. Once the complete set of instructions are compiled, the compiler generates the decoder by following the algorithm outlined below:

(a) Create a set S which contains all the instructions. Assign the set the name *decoder*. Append the set S to the list of sets U.

(b) Remove a set P from U. Create the procedure heading where the procedure name is set to the name of P.

(c) Identify an *unvisited* instruction field which is constant valued among all the members of the set P. There must be at least one such field. Mark the field as *visited*.

(d) Sort the instructions based on the value of this field. For each subset of instructions Q which has the same value for this field, create a procedure name and assign the subset Q this name. Append the subset Q to U.

(e) Generate a switch statement for the field identified in step (c). For each unique value generate a return statement that returns the assigned opcode for that instruction. Values for which a subset Q had been created in step (d), generate a return statement that returns the value of procedure name assigned to Q. Write the procedure closing.

(f) Repeat steps (b) through (e) as long as U has members.

Please note that the actual copying of the generated code does not start until all the sets are processed. Generated switch statements as well as the procedure heading are kept together with the sets until they are completely processed. Once completed, the compiler has successfully generated a decoder which returns the integer value assigned to an instruction, given the binary code for the instruction. The generated decoder is used to implement the disassembler as well as the ADL statement *decode*.

10.2.3 Generating the Disassembler

Generating the disassembler entails the following steps: (a) generating code to load the program binary into the memory; (b) copying the decoder onto the symbol file being generated; (c) generating code to assign the value of the *program counter* to the beginning of the program area; (d) generating code to invoke procedures which disassemble the data segments; (e) generating a procedure per instruction which can disassemble a given instruction; (f) generating a loop that calls decoder, calls appropriate procedure to disassemble the detected instruction and increment the program counter by the size of the instruction.

The generated disassembler tries to match the disassembled assembly program line with that of the original source program so that the disassembled output will have the macro instructions and what is generated from them aligned to increase readability.

10.2.4 Generating the Simulator

Generating the simulator involves the following basic steps: (a) generating code to load the program binary into the memory; (b) inserting the decoder into the appropriate position in the simulator template; (c) generating structures necessary to hold instruction contexts; (d) generating code to assign the value of the *program counter* to the beginning of the program area; (e) generating TAPs; (f) generating the simulator main loop.

The simulator main loop is generated by processing the pipeline declarations in declaration order, and individual pipeline stages in reverse declaration order. For each minor cycle, code is generated so that the corresponding time annotated procedure and the LRTL segment will be invoked. In this way, instruction flow in the pipelines is handled by polling from the sink stages towards the source stages. This mechanism allows modeling of flow of instructions through the pipeline by polling each stage once per minor cycle.

10.3 The Debugger

The debugger is entered through command line arguments or automatically upon detecting an error condition. Command line arguments may specify that the debugger must be entered after a specified number of cycles, or immediately. If a *deadlock* is suspected, that is, no instruction is retired for a large number of cycles, the simulator invokes the debugger automatically. Upon an internal fault in the simulator the system's standard debugger along with the FAST debugger are fired. Finally the debugger may be invoked when the *pause* statement in an ADL program is encountered which is used when an unexpected condition occurs. The debugger when entered fires up two windows. The first window displays a disassembled memory image where the line number of the assembly language program, the memory location, binary encoding of the instruction, the machine instructions and the original assembly language program are shown in that order on each line. The second window displays the contents of registers specified in the *monitor* declarations and

the contents of the pipeline stages of the machine architecture. In addition, the current number of machine cycles, the number of useful cycles and the number of stall cycles are also displayed. A sample output is shown in Figure 10.5.

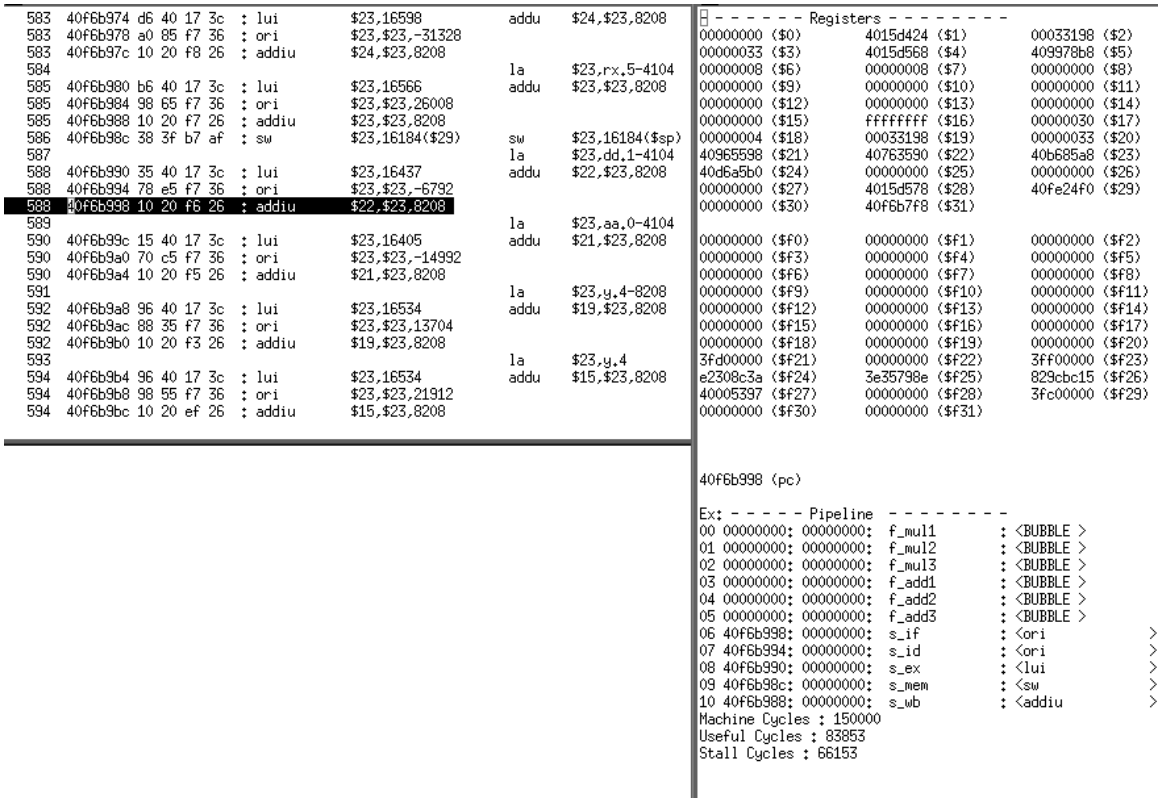


Figure 10.5: Sample Debugger Screens

Once in the debugger, the user can *single-step* the execution, continue the execution until a certain number of additional cycles are executed, or simply resume the execution. In case more powerful debugging is needed, the user may fire the regular system debugger, such as *gdb*, and perform further analysis. Since ADL compiler preserves ADL program names when generating the simulator, the user may inquire the values of variables using the ADL program names.

In some cases, problems surface after large numbers of simulation cycles although the exact cause of the problem may actually be hundreds of cycles prior to the point it is detected. Solving these kinds of problems requires the knowledge of how a specific point in the program execution is reached. For example, a label may be the destination of a number of branch instructions and it is virtually impossible to know which path had been taken to arrive at this point. In order to address these problems, the debugger provides a unique *reverse execution mode*. In order to use this mode, the user specifies a range of cycles during which the simulator saves register and the pipeline contents. When the debugger is entered upon the occurrence of the problem, the program can be traced in reverse using the *backstep* command. In this mode, it is possible to backstep then forward step, within the window of saved cycles. This mode is slow and saves significant amounts of

data. However, it has proven to be very valuable during the development of many ADL architecture descriptions.

10.4 Evaluation of FAST Implementation

The implementation of the FAST system took over 24 months. This is a large software project totaling over 34,000 lines of code (see Table 10.1). Although it took a relatively long period of time to develop, once completed, three simulators have been developed during a course of an additional three months. All the simulators were based on the MIPS ISA consisting of 84 machine instructions and 53 macro instructions. These simulators are: (a) a standard five stage pipelined MIPS architecture (PIPE); (b) an implementation of the Tomasulo's algorithm applied to MIPS ISA (TOM); and finally (c) an initial version of the data forwarding architecture (FWD) that has been used to collect the statistics reported in Chapter 5.

Software Component	Lines
ADL Compiler	19675
Artifacts	1682
Assembler template	4953
Shared modules	535
Linker template	970
Disassembler template	531
Simulator template	4259
Library Support	1200
Debugger	487
Total	34292

Table 10.1: Software Sizes.

For each of these architectures, relative percentages and the sizes of various sections of ADL descriptions are illustrated in Table. 10.2(a). One immediate observation is the larger share of the ISA specification. This is a direct result of the ADL approach to the problem. ADL approach is an instruction oriented approach and in this respect, a significant portion of the semantics of the machine execution is defined as part of the ISA specification. Another important point is the small size of the artifacts section. Although artifacts make up a significant portion of the actual hardware, they can be specified with ease by means of powerful ADL abstractions in a few hundred lines. Finally, while the sizes of the architecture specifications are around 6000 lines of ADL code, the sizes of the simulators vary from approximately 20,000 to 30,000 lines of C++ code. This clearly shows the merit of automatic generation.

Developing the ISA portion has been relatively straight-forward. Few software bugs have been traced to the ISA section. Most of these errors resulted either because of typing errors or ambiguity in the architecture manuals that were used. Although ISA section is fairly large and the microarchitecture section is relatively small, the development times for the ISA component and the

Component	PIPE	%	TOM	%	FWD	%
ISA spec	4549	78.6	4549	73.8	4549	76.6
Artifacts	210	3.6	230	3.7	230	3.9
μ-arch	554	9.6	890	14.4	673	11.3
Other	459	8.2	497	8.1	485	8.2
Total	5782		6166		5937	

(a) ADL lines of code

Component	PIPE	%	TOM	%	FWD	%
Assembler	6775	30.7	6775	21.9	6775	35.8
Disassm.	1508	6.8	1508	4.9	1508	8.0
Simulator	10942	49.6	19834	64.1	7803	41.2
Linker-etc	2838	12.9	2842	9.1	2842	15.0
Total	22063		30959		18928	

(b) Generated C++ lines of code

Table 10.2: ADL programs and generated software

microarchitecture sections were roughly equal. This is expected as the microarchitecture section involves a high degree of parallel operation. These results demonstrate that the separation of ISA from the microarchitecture is a powerful approach since developing three fully functional simulators in three months would not have been possible without this separation.

The size of the ADL generated software for each of the architectures are given in Table 10.2(b). When the size of the ADL generated software is compared to hand coded simulators which implement comparable architectures, surprising similarities are observed. For example, the pipelined MIPS architecture implements essentially the same architecture as SPIM. The automatically generated PIPE simulator consisting of 22,063 lines compares quite well with SPIM that consists of 20,441 lines of C code. Comparison of MIPS-Tomasulo (an out-of-order architecture) implementation with SimpleScalar yields similar results. SimpleScalar package contains a total of 26,500 lines (excluding the library and the provided gcc compiler) and includes three simulators. Considering only the out-of-order simulator would correspond roughly to 25,000 lines, as these simulators are relatively small and share enormous amount of code. Thus, the automatically generated TOM simulator consisting of 30,959 lines compares well the size of SimpleScalar simulator. Finally, the data-forwarding architecture has an intermediate complexity, for which there is no hand coded simulator that it can be compared with.

Simulation speeds are very reasonable and compare well with hand coded simulators. The pipelined version executes at an average speed of 200,000 simulator cycles/second on a 200 MHZ Pentium Pro and the Tomasulo's algorithm executes at an average speed of 100,000 cycles/second. The Tomasulo's algorithm is comparable in complexity to the out-of-order SimpleScalar simulator [7] which reports a simulation speed of 150,000 cycles/second on a 200 MHZ Pentium Pro. Comparing

these figures with the SimpleScalar numbers yields that ADL generated simulators are less than 2 times slower than the hand coded counterparts.

10.5 Advanced Machine Descriptions

Once the initial descriptions have been successfully implemented and used to test and improve the reliability of the implementation, a large number of descriptions have been written for the techniques presented in this dissertation. These implementations includes the scalability study presented in Chapter 3, as well as the DBMA and DWMA. During this process, it has been observed that it is difficult to maintain various processor descriptions coherent as new additions are performed on them. This observation gave rise to the notion of processor descriptions which can host multiple implementations of a given hardware component. Based on this observation, ADL specification has been extended with the notion of tagging various procedures with a compile time evaluatable expression. The compiler would still compile a given component as usual, but the resulting code would not be inserted into the generated simulator if the compile time expression evaluates to false. Since conditional skipping of modules is done by the compiler and not by the preprocessor, the undesirable outcome of a change introduced for a given implementation effecting other implementations is minimized.

Component	ADL Lines of Code
Central window	1000
DBMA	800
DWMA	1300
Branch Predictors	700
Store set algorithm and variants	680
Ideal Memory disambiguator	300
Speculative restart handling	1200
Common code	2050
Total	8030

Table 10.3: Components of the Unified Description

Using the introduced conditional compilation facilities, various processor descriptions have been merged into a single processor description which contains multiple implementations of branch predictors, issue window and memory disambiguation techniques. With this merged description, it became possible to obtain any combination of the implemented techniques by setting the appropriate compile time constants. The break-down of this large processor description which has more than 8000 lines excluding the ISA specification is illustrated in Table 10.3.

Generated simulators for the various combinations ranged between 18,000 to 35,000 lines of C++ code. Depending on the issue width and the window size which is being studied, the performance of the simulator showed a slowdown of a factor 2.5 to 10 compared to the simpler

Technique	Performance
Central window	10,000-23,000 cycles/sec
DBMA	25,000-40,000 cycles/sec
DWMA	13,000-30,000 cycles/sec

Table 10.4: Performance of Various Techniques

implementations discussed before. The performance of various combinations are summarized in Table 10.4 using a 200 MHZ Pentium Pro machine. In these numbers, the lower figures are observed with 32 issue processors and the higher figures are observed with 8 issue processors. This is expected as the speed of the simulator is directly proportional to the hardware complexity that is being simulated.

10.6 Concluding Remarks

In this chapter an overview of the implementation of the ADL language, namely, the FAST system has been presented. Because of the enormous size and complexity of the software as well as irrelevance of many details, only aspects which were considered to be significant have been covered.

Being one the first in the area of automatically generating simulators from specifications, FAST system has proven to be very effective with its unique properties such as the presentation of a complete solution to the microarchitecture simulation problem. FAST does not only generate a cycle accurate simulator from a machine description but it also generates the required system software including the assembler, linker and a debugger. As such, it is expected that it is going to be quite useful for future microarchitecture research.

Given that the development time for SimpleScalar simulator was 18 man-months [7], and the number of simulators developed for this dissertation, it can be comfortably stated that automatic generation using a domain specific language is a cost-effective approach.

Chapter 11

Conclusions

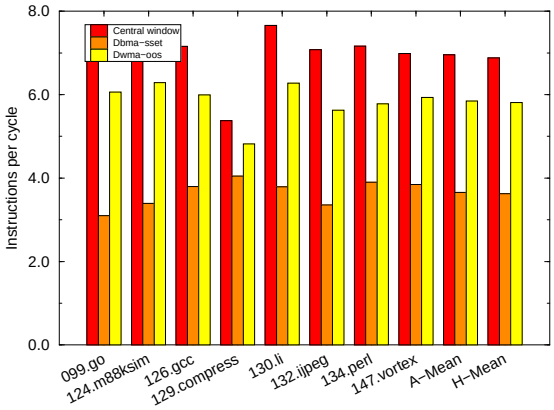
This thesis has been about the scalability of superscalar processing. Each of the evaluated and developed techniques have been examined with an emphasis on their scalability demonstrating that there is a need for techniques that scale better. This observation motivated the development of alternative techniques which were then thoroughly evaluated to verify that the contributed algorithms greatly enhance the scalability of the superscalar paradigm. In the experiments presented throughout the dissertation a robust methodology has been applied which calls for evaluating individual algorithms in settings where only the algorithm being studied is the bottleneck. In this section, the two techniques, namely DWMA architecture and the out-of-order store set algorithm are combined in an architecture to demonstrate that proposed techniques work well together and advance the state-of-the-art.

The first section of this chapter therefore has been reserved for evaluations of the proposed techniques where these techniques are evaluated together. Section 11.2 presents an itemized list of contributions of this thesis. Finally, the dissertation is concluded with a brief discussion of the future research directions in Section 11.3.

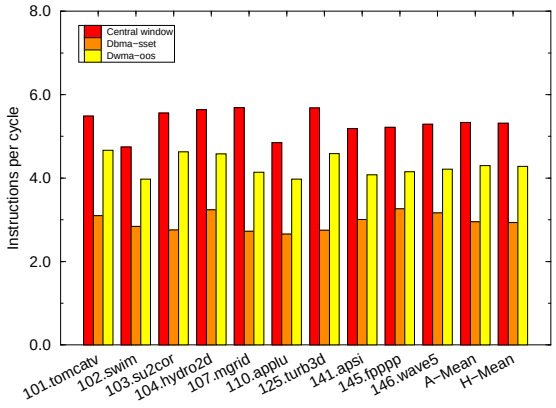
11.1 Improvement in the State-of-the-art

In this section, the performance of the DWMA-OOS processor that employs the best previously known techniques (i.e., store set disambiguator and DBMA) is compared with the DBMA-SSET processor that employs the newly developed techniques (i.e., out-of-order store set and DWMA). The comparison essentially shows the improvement in the state-of-the-art superscalar processing techniques. As a reference line, the ideal central window processor (CW) which employs an ideal memory disambiguator is also plotted. All the processor configurations employ identical fetchers.

The resulting IPCs for all the benchmarks are shown in Figures 11.1, 11.2 and 11.3. An 8-issue DWMA-OOS architecture performs 60% and 46% better than the DBMA-SSET processor for integer and floating point benchmarks respectively. At an issue width of 16, DWMA-OOS architecture achieves 64 % and 50 % better than the DBMA-SSET processor for integer and floating point benchmarks respectively. Finally, for 32-issue DWMA-OOS architecture performs 57% and 45% bet-

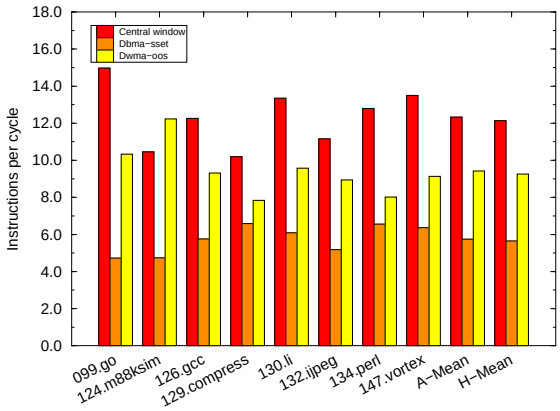


(c) Integer Benchmarks

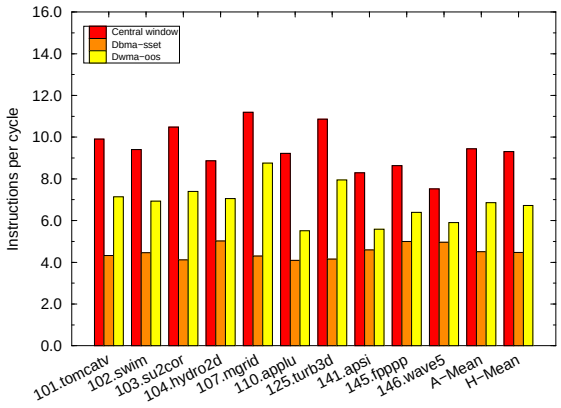


(d) Floating point Benchmarks

Figure 11.1: IPC values for 8-issue CW, DWMA-OOS and DBMA-SSET

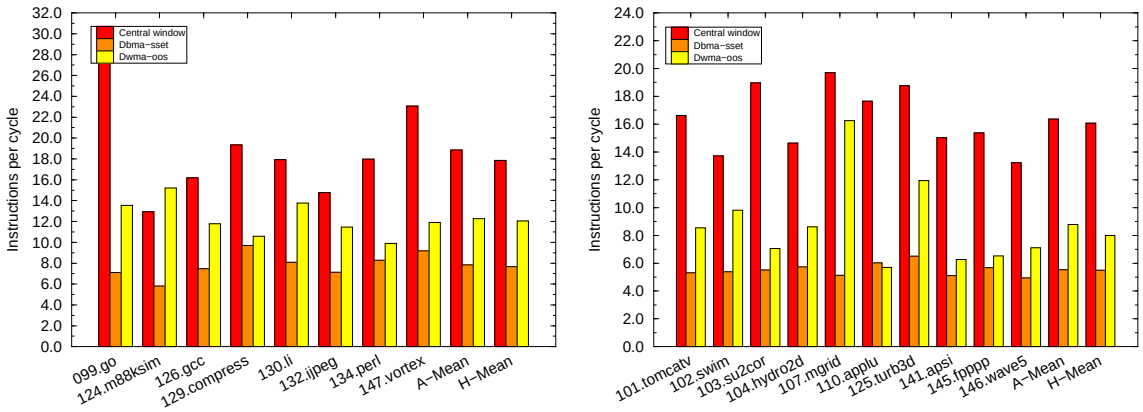


(a) Integer Benchmarks



(b) Floating point Benchmarks

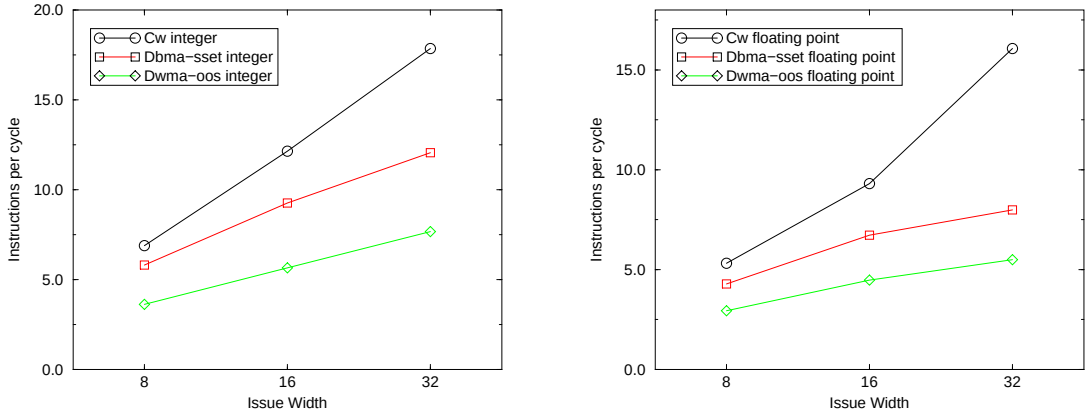
Figure 11.2: IPC values for 16-issue CW, DWMA-OOS and DBMA-SSET



(a) Integer Benchmarks

(b) Floating point Benchmarks

Figure 11.3: IPC values for 32-issue CW, DWMA-OOS and DBMA-SSET



(a) Harmonic Means for Integer Benchmarks

(b) Harmonic Means for Floating point Benchmarks

Figure 11.4: Performance of CW, DBMA-SSET and DWMA-OOS

ter than the DBMA-SSET processor for integer and floating point benchmarks respectively. Therefore it has been demonstrated that the proposed techniques significantly improve the state-of-the-art.

Now let us observe the performance of DWMA-OOS with respect to the ideal central window processor. An 8-issue DWMA-OOS architecture attains 84% and 81% of the 8-issue ideal central window processor performance for integer and floating point benchmarks respectively. At an issue width of 16, DWMA-OOS architecture achieves 76 % of the performance of the ideal central window processor for integer benchmarks and 72 % for floating point benchmarks.

At an issue width of 32, DWMA-OOS achieves 66 % of the performance of the central window processor for integer benchmarks. With floating point benchmarks, DWMA-OOS loses more performance, achieving about 50 % of the performance of the ideal processor. This performance loss has been traced to the sensitivity of the 32 issue processor to instruction schedule ordering as well as the degree of forwarding. Therefore the performance of the DWMA-OOS can be further improved by increasing the degree of forwarding and employing additional heuristics in the descriptor selection.

The average IPCs obtained by the three algorithms are summarized in Figure 11.4.

11.2 Contributions

The contributions of this thesis are:

(a) Development of an effective methodology for the evaluation of superscalar processors with an emphasis on scalability;

(b) Empirical demonstration that for high performance large instruction windows are needed which grow quadratically as a function of the issue width;

(c) A unique superscalar out-of-order processing paradigm which relies on run-time generation of a special dataflow graph called *Direct Data Forwarding Graph* (DDFG) from ordinary RISC code.

(d) The *Direct Wake-up Microarchitecture* (DWMA) which uses principles of DDFG generation to implement large instruction windows, outperforming the best non-broadcasting based instruction window implementations;

(e) A novel memory order violation detection algorithm that greatly enhances the performance of memory dependence prediction based dynamic memory disambiguators by eliminating false memory order violations as well as a significant percentage of false memory dependencies;

(f) A powerful domain specific language for the microarchitecture simulation domain called the *Architecture Description Language* (ADL) which can be used to specify a broad class of instruction set architectures as well as microarchitectures;

(g) A complete implementation of ADL in an integrated system called *Flexible Architecture Simulation Tool* (FAST).

11.3 Future Research Directions

Future research related with the basic principles laid out in this dissertation carries great potential. In the area of micro architecture research for improving the scalability of out-of-order issue processors there are many trails which can be pursued. Especially, in the areas of instruction fetch and instruction issue logic, there is still a lot that can be done to further improve the performance. Techniques which can greatly out-perform even *ideal* mechanisms are possible with a change of mind-set. Contrary to the conventional notion of treating these two areas as two distinct and unrelated areas, we must start thinking about the problem as a whole. Once we remove the artificial barriers on the superscalar processors which are imposed by the application of brute force hardware techniques and start using techniques which are *dependency aware* such as the direct wake-up technique, new opportunities will arise which cannot be seen with a conventional mind-set.

Instruction Fetch and Instruction Wake-up. As it has already been stated in this dissertation, the trace cache approach is quite promising. A processor that implements a trace cache will benefit from its use when it is using the techniques such as direct wake-up and out-of-order store set algorithms developed in this dissertation. However, much greater gains can be obtained by integrating the trace cache with the reorder buffer itself. Because of the reorganization of the wake-up graph as the program executes, the final form of the wake-up graph is a better schedule than the initial one. By caching completed portions of the reorder buffer, it is possible to cache the dependencies as well as the execution trace that the program has followed. Upon a hit in this *cache*, instructions may start firing immediately without going through the initial stages of the program execution. This is a very promising technique since it has the potential to out-perform even a completely ideal conventional superscalar processor setting.

Multi-threaded Program Execution. Direct wake-up and direct data forwarding techniques can be effectively employed for exploiting thread-level parallelism. As it has been illustrated in [45] these techniques can be employed in a multi-threaded setting for direct communication and the synchronization of threads. Such use of the techniques represent an area of the spectrum of dataflow-Von Neumann hybrid architectures which has not been explored fully. Since there are many applications which lend themselves directly to efficient execution on multi-threaded machine models, this avenue of research should be explored as well.

Instruction Issue and Direct Wake-up. Presented design of the direct wake-up architecture is only one design among several which had been tried. The design space is huge and it is only natural to expect that better implementations of the idea may exist. Since Chapter 5 decisively illustrated that the DDFG is not the limiting factor for high performance, future research for better micro architectures that rely on the notion of direct wake-up should also be sought.

Memory Disambiguation. One of the important bottlenecks in the current processor implementations is the number of memory ports. Memory ports are quite expensive to implement and for high performance many memory ports are needed. However, large instruction windows also present better opportunities for bypassing values between load instructions and store instructions as well as the bypassing of values directly between the producer of a value and its consumer without going through the memory. Although existing techniques addressed this question in a number of cases [39], effect of large windows in this manner has not been studied.

Automatic Generation of Simulators. The first design of ADL has been quite successful. However, the language can be enhanced in many ways, the most important aspect being the imperative nature of the microarchitecture specification. The imperative approach has been selected because the current state of knowledge is not mature enough to obtain fully functional and efficient simulators from declarative microarchitecture specifications. Research in this respect may greatly speed-up both the development and the run-time performance of the resulting simulators.

Appendices

Appendix A

Sample ADL Micro Architecture Description

```
processor processor_0 highbit 31
begin

Machineid "simple";
lilliput  little_endian;

shadow register
   code_start 32,
   ex_trace   32,
   _hi        32,          # Division operation HI value.
   _lo        33,          # Division operation LO value.
   linebreak  32,
   #
   dest_r     32,
   check_ex    1,
   check_mem   1,
   check_wb    1,
   ex_has_it   1,
   mem_has_it  1,
   wb_has_it   1,
   target     32,          # Used in the branch target computation.
   data_tmp   32,          # Used in data transfers.
   ptemp      32,          # A temporary value register.
   dummy      32,          # SAA (Same as above)
   which       2,          # For passing parameters to cop branch units.
   equal       1,
   less        1,
   unordered   1;

shadow register file dtemp[2,32];
shadow register file scratch[2,32];

constant generate_trace         0;
constant machine_drained        1;
constant cpc_register_number   32;
constant lo_hi_register_number 32;

# This is the instruction pipeline.
#
pipeline IPIPE       (s_ID);
source s_ID;

latch
   exception       1,
   new_pc         32,
   branch_input    1,
   branch_target  32;
```

```
# Next, each machine register mnemonic is presented together with their
# actual register number. For MIPS $zero and $0 are aliases, so are many
# others. From the perspective of the machine-gen, only the association
# of names with numbers is important. Therefore, as many aliases as
# necessary can be described.

register file fpr [33,32]  # [34 regs,32 bits each].
   $f0       0, $f1       1, $f2       2, $f3       3, $f4       4,
   $f5       5, $f6       6, $f7       7, $f8       8, $f9       9,
   $f10     10, $f11     11, $f12     12, $f13     13, $f14     14,
   $f15     15, $f16     16, $f17     17, $f18     18, $f19     19,
   $f20     20, $f21     21, $f22     22, $f23     23, $f24     24,
   $f25     25, $f26     26, $f27     27, $f28     28, $f29     29,
   $f30     30, $f31     31, $CpC     32;
register file gpr [34,32]  # [34 regs,32 bits each].
   $0       0, $1       1, $2       2, $3       3, $4       4,
   $5       5, $6       6, $7       7, $8       8, $9       9,
   $10     10, $11     11, $12     12, $13     13, $14     14,
   $15     15, $16     16, $17     17, $18     18, $19     19,
   $20     20, $21     21, $22     22, $23     23, $24     24,
   $25     25, $26     26, $27     27, $28     28, $29     29,
   $30     30, $31     31,

   $zero    0, $at      1, $v0      2, $v1      3, $a0      4,
   $a1      5, $a2      6, $a3      7, $t0      8, $t1      9,
   $t2     10, $t3     11, $t4     12, $t5     13, $t6     14,
   $t7     15, $s0     16, $s1     17, $s2     18, $s3     19,
   $s4     20, $s5     21, $s6     22, $s7     23, $t8     24,
   $t9     25, $k0     26, $k1     27, $gp     28, $sp     29,
   $fp     30, $ra     31;
shadow register
       hi_val                32,
       lo_val                32;

# We have to specify the name of the instruction register. The instruction
# register is treated as a special register to allow less typing. That is,
# ir.rt is equivalent to rt iff ir is the instruction register.

instruction register ir 32;
instruction pointer  pc 32;

memory mem_0  latency 0 width 32;
memory ncache latency 0 width 32;

controldata register
   my_pc       32;

shadow register
   ls_bypass    1,
   mem_stat     1,
   access_type 32,
   byte         2,
   lop_r        6,     # lop_r indicates the register number for the lop.
   rop_r        6,     # rop_r indicates the register number for the rop.
   simm        32,     # Sign extended immediate.
   zimm        32,     # Zero extended immediate.
   smdr        32,
   store_v     32,     # Store Memory data register.
   lmar        32,     # load memory address register.
   smar        32,     # store memory address register.
   dest        32,     # dest holds the value to be written.
   dest2       32,     # dest holds the value to be written.
   lop         32,     # lop holds the left operand value.
   lop2        32,     #
   rop         32,     # rop holds the right operand value.
   rop2        32;     #
```

```
bitconstant
   _BYTE        0 0,
   _HALFWORD    0 1,
   _TRIPLEBYTE  1 0,
   _WORD        1 1;

$include mips-instruction-set-simple.adl
$include ../mips-calling-convention.adl

#- Instruction Decode - - - - - - - - - - - - - - - - - - - - - - - - -#
#                                                                       #
#                                                                       #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#

procedure s_ID prologue
begin
   my_pc = pc;                    #- Required by jal.
   ir = mem_0[pc];

   if (branch_input) then
      begin
         branch_input=0;
         pc=branch_target;
      end
   else
      pc=pc + 4;

   #  Fetch input registers. Sign extend the immediate portion.
   #
   decode;

   dest_r = ordinal(dest_reg);

   # Read operands.

   case lop_type of
   begin
      cpc_register:
         lop_r     = cpc_register_number;
         lop       = fpr[lop_r];

      integer_register:
         lop_r=rs;
         lop=gpr[lop_r];

      float_register  :
         lop_r=fs;
         lop =fpr[lop_r];

      double_register :
         lop_r=fs;
         lop =fpr[lop_r];
         lop2=fpr[lop_r+1];

      special_input   :
         lop_r     = 2;
         lop       = gpr[2];

      lo_hi_register:
         lop_r     = lo_hi_register_number;
         lop=gpr[lop_r];
         lop2=gpr[lop_r+1];
   end;

   case rop_type of
```

```
   begin
      cpc_register:
         rop       = tf;

      integer_register:
         rop_r=rt;
         rop=gpr[rop_r];

      float_register :
         rop_r=ft;
         rop =fpr[rop_r];

      double_register:
         rop_r=ft;
         rop =fpr[rop_r];
         rop2=fpr[rop_r+1];
   end;


   if (i_class == branch_class) then
      condition_code(lop,rop);
end s_ID;

procedure s_ID epilogue
begin
   case dest_type of
   begin
      lo_hi_register:
              gpr[dest_r]=dest;
              gpr[dest_r+1]=dest2;

      integer_register :
              gpr[dest_r]=dest;

      cpc_register     :
      float_register   :
              fpr[dest_r]=dest;

      double_register  :
              fpr[dest_r]=dest;
              fpr[dest_r+1]=dest2;

      else : if ordinal(dest_type) then
                builtin printf("skip skip %d\n");
   end;
   builtin sprintf(pointer(scratch),"%6d\n",my_pc - code_start);
   if generate_trace then
   if builtin upfast_write_file
     (
     ex_trace,
     7,
     scratch) ^= 7 then
     begin
        builtin perror("Store ex trace");
        builtin simulation_exit(-1);
     end;
   retire stat;

   newcontext;
end s_ID;

procedure boot_up untyped
begin
   forall gpr = 0;
   code_start = builtin fast_text_begin;

   if generate_trace then
```

```
      ex_trace   = builtin fast_open_file("execution-trace","output");
end boot_up;

initialization boot_up;

# These are the registers we monitor duxecution.
#
#
monitor
   $0,   $1,   $2,   $3,   $4,   $5,   $6,   $7,   $8,   $9,   $10,
   $11,  $12,  $13,  $14,  $15,  $16,  $17,  $18,  $19,  $20,  $21,
   $22,  $23,  $24,  $25,  $26,  $27,  $28,  $29,  $30,  $31,
   linebreak,
   linebreak,
   $f0   , $f1   , $f2   , $f3   , $f4   , $f5   , $f6   , $f7   ,
   $f8   , $f9   , $f10  , $f11  , $f12  , $f13  , $f14  , $f15  ,
   $f16  , $f17  , $f18  , $f19  , $f20  , $f21  , $f22  , $f23  ,
   $f24  , $f25  , $f26  , $f27  , $f28  , $f29  , $f30  , $f31  ,
   linebreak,
   linebreak,
   pc;
end;  # processor

#- C and C++ supplements. Initialize machine must be provided. - - - - - -#
#                                                                          #
#                                                                          #
#  Simulator supplements.                                                  #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#

simulator begin
%%
integer pseudo_procedure_call;
integer pseudo_pipeline_flush;
integer nop_line[4];
integer nop_flush[4];

void procedure initialize_machine
    (code_file_header& H,int arg_count,char **args,integer lim)
begin
   char * p;

   while (lim & 0x7) lim--;
   $sp=(lim-8);

   pseudo_procedure_call=(integer)&nop_line;
   nop_line[0]=0;
   nop_line[1]=0;
   nop_line[2]=0;
   nop_line[3]=0;

   pseudo_pipeline_flush=(integer)&nop_flush;
   nop_flush[0]=0;
   nop_flush[1]=0;
   nop_flush[2]=0;
   nop_flush[3]=0;

   for (integer i=0; i < upfast_ext_count; i++)
   begin
      if (strcmp(externals[i].name,"exit")==0) then
         begin
            $31=e_ref_table_start + ((integer)&externals[i]-(integer)&externals);
            break;
         end
   end

   // cout << "Args are : ";
   // for (integer i=0; i < arg_count; i++)
```

```
   // cout << args[i] << " ";
   // cout << "\n";
   $4=arg_count;
   $5=(integer)args;
   $28=H.sbss_segment_start;
end // initialize_machine //
%%
end simulator;
```

Appendix B

Sample ADL ISA Description

```
# Any instruction fields which are accessed BEFORE the instruction is
# decoded must be declared fixedfields. For example, the values of
# the rs & rt fields from the instruction register are
# available immediately after the instruction register is
# loaded with a new instruction. This information is used
# for 'fixed field decoding' during simulator code generation.

type
   opcode       constant       field          31   6,
   rs           register       fixedfield     25   5,
   rt           register       fixedfield     20   5,
   rd           register       field          15   5,
   shamt        integer        field          10   5,
   funct        constant       field           5   6,
   functco      constant       field           4   5,
   immediate    signed         field          15  16,
   uimmediate   integer        field          15  16,
   code         integer        field          25  20,
   cofun1       integer        field          24  25,
   cofun2       integer        field          24  20,
   cof          integer        field          25   1,
   b_offset     signed         field          15  16,
   j_offset     integer        field          25  26,
   copf1        integer        field          24   4,
   copf2        integer        field          20   5,
   copf3        integer        field          10  11,
   ccf          integer        field          20   3,
   nd           integer        field          17   1,
   tf           integer        field          16   1,

   call_n       integer        field          15  16,
   ft           register       fixedfield     20   5,
   fs           register       fixedfield     15   5,
   fd           register       fixedfield     10   5,
   format       integer        field          24   4,
   uccf         integer        field          10   3,    # unused zero field.
   ufzf         integer        field           7   2,    # unused zero field.
   ufc          integer        field           5   2,    # unused zero field.
   cond         integer        field           3   4,    # unused zero field.

   address      label          variable,

   rdest        register       variable,
   rsrc1        register       variable,
   rsrc2        register       variable,
   rsrc3        register       variable,
   src2         integer        variable,
   fsrc2        float          variable,
   tx           integer        temporary,
   l1           label          temporary,
```

```
   ty           integer        temporary;

attributes
     i_class       : float_class,
                     integer_class,
                     branch_class,
                     long_integer_class;    #- Multi-cycle integer ops.

     i_cycles      : single_cycle,
                     multiple_cycles;

     i_type        : number_of_i_types,
                     alu_type,
                     system_type,
                     conditional_direct,
                     conditional_direct_link,
                     unconditional_direct,
                     unconditional_direct_link,
                     unconditional_indirect,
                     unconditional_indirect_link,
                     load_type,
                     store_type;

     exu           : load_unit,
                     store_unit,
                     integer_unit,
                     call_unit,
                     divide_unit,
                     f_add_unit,
                     f_mul_unit;

     c_what        : condition_equal,
                     condition_gez,
                     condition_gtz,
                     condition_lez,
                     condition_ltz,
                     condition_neq,
                     condition_u,
                     condition_z;

     dest_type     : float_register,integer_register,double_register,
                     special_input,
                     cpc_register,
                     lo_hi_register;

     dest_reg      : integer;

     lop_type      : dest_type;
     rop_type      : dest_type;
     t_count       : general;        # Total instructions of this instruction.
     t_cycles      : general;        # Total cycles for this instruction.
     t_min         : general;        # Minimum cycles for this instruction.
     t_max         : general;        # Maximum cycles for this instruction.
end;


assertion
   #- All single cycle operations are executed by the integer unit.
   #

   1 : i_cycles  == single_cycle      : (exu == integer_unit)     |
                                        (exu == call_unit)        ;


   #- If any operands of the instruction is integer, then it must be
   #  executed by either the integer unit or the load store unit.
```

```
   2 : (lop_type == integer_register) |
       (rop_type == integer_register) : (exu == integer_unit)      |
                                        (exu == call_unit)         |
                                        (exu == divide_unit)       |
                                        (exu == load_unit)         |
                                        (exu == store_unit);
end;


procedure condition_code(x:32,y:32) untyped
begin
   if c_what == condition_equal then
      branch_input=(x == y)
   else
   if c_what == condition_gez then
      branch_input=((x.[31:1])==0)
   else
   if c_what == condition_gtz then
      branch_input=(x > 0)
   else
   if c_what == condition_lez then
      branch_input=(x.[31:1] | (x == 0))
   else
   if c_what == condition_ltz then
      branch_input=(x.[31:1])
   else
   if c_what == condition_neq then
      branch_input=(x ^= y)
   else
   if c_what == condition_u then
      begin
         branch_input=1;
         branch_target=x;
      end
   else
   if c_what == condition_z then
      begin
         branch_input=1;
      end;
end condition_code;

# These procedures are applied by the assembler to compute various
# offsets.
#
procedure add4(x) computation
begin
   add4=(x) + 4 ;
end add4;

procedure add1(x) computation
begin
   add1=(x) + 1;
end add1;

procedure jump_address(x) computation
begin
   jump_address=(x) >> 2;
end jump_address;

procedure upper(x) computation
begin
   upper=x.[31:16];
end upper;

procedure lowerc(x) computation
begin
   lowerc=x.[15:16];
```

```
end lowerc;

# Utility procedures save on typing and eliminate some errors that
# would otherwise may result.
#
procedure sign_extend_24 (x:8)
begin
   sign_extend_24=(x.[7:1] |< 24) || x;
end sign_extend_24;

procedure sign_extend_16 (x:16)
begin
   sign_extend_16=(x.[15:1] |< 16) || x;
end sign_extend_16;

procedure zero_extend_16 (x:16)
begin
   zero_extend_16=x & 0xffff;
end zero_extend_16;

procedure sign_extend_14 (x:16)
begin
   sign_extend_14=(x.[15:1] |< 14) || x || (0 |< 2);
end sign_extend_14;


bitconstant                      # FMT field encodings.
   #
   _single_float    0 0 0 0,     #  s
   _double_float    0 0 0 1,     #  d
   _reserved_1      0 0 1 0,     #
   _reserved_2      0 0 1 1,     #
   _single_fixed    0 1 0 0;     #  w

#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
#  Opcode bits: For simplicity these bits are goruped together and          #
#  they are asssigned as a single constant to the opcode field.             #
#  An alternative strategy could create two opcode fields, opcode1          #
#  and opcode2.                                                             #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #

bitconstant
   #
   #         31..29  28..26              31..29  28..26          31..29  28..26
   #         ------  ------
   _special   0 0 0   0 0 0 , _addi      0 0 1   0 0 0 , _cop0      0 1 0   0 0 0 ,
   #unused    0 1 1   0 0 0 , _lb        1 0 0   0 0 0 , _sb        1 0 1   0 0 0 ,
   _lwc0      1 1 0   0 0 0 , _swc0      1 1 1   0 0 0 , _bcond     0 0 0   0 0 1 ,
   _addiu     0 0 1   0 0 1 , _cop1      0 1 0   0 0 1 , _swc1_set  0 1 1   0 0 1 , # was unused.
   _lh        1 0 0   0 0 1 , _sh        1 0 1   0 0 1 , _lwc1      1 1 0   0 0 1 ,
   _swc1      1 1 1   0 0 1 , _j         0 0 0   0 1 0 , _slti      0 0 1   0 1 0 ,
   _cop2      0 1 0   0 1 0 , #unused    0 1 1   0 1 0 , _lwl       1 0 0   0 1 0 ,
   _swl       1 0 1   0 1 0 , _lwc2      1 1 0   0 1 0 , _swc2      1 1 1   0 1 0 ,
   _jal       0 0 0   0 1 1 , _sltiu     0 0 1   0 1 1 , _cop3      0 1 0   0 1 1 ,
   _sw_set    0 1 1   0 1 1 , _lw        1 0 0   0 1 1 , _sw        1 0 1   0 1 1 ,
   _lwc3      1 1 0   0 1 1 , _swc3      1 1 1   0 1 1 , _beq       0 0 0   1 0 0 ,
   _andi      0 0 1   1 0 0 , #unused    0 1 0   1 0 0 , #unused    0 1 1   1 0 0 ,
   _lbu       1 0 0   1 0 0 , #unused    1 0 1   1 0 0 , #unused    1 1 0   1 0 0 ,
   #unused    1 1 1   1 0 0 , _bne       0 0 0   1 0 1 , _ori       0 0 1   1 0 1 ,
   #unused    0 1 0   1 0 1 , #unused    0 1 1   1 0 1 , _lhu       1 0 0   1 0 1 ,
   #unused    1 0 1   1 0 1 , #unused    1 1 0   1 0 1 , #unused    1 1 1   1 0 1 ,
   _blez      0 0 0   1 1 0 , _xori      0 0 1   1 1 0 , #unused    0 1 0   1 1 0 ,
   #unused    0 1 1   1 1 0 , _lwr       1 0 0   1 1 0 , _swr       1 0 1   1 1 0 ,
   #unused    1 1 0   1 1 0 , #unused    1 1 1   1 1 0 , _bgtz      0 0 0   1 1 1 ,
   _lui       0 0 1   1 1 1 , #unused    0 1 0   1 1 1 , #unused    0 1 1   1 1 1 ,
   #unused    1 0 0   1 1 1 , #unused    1 0 1   1 1 1 , #unused    1 1 0   1 1 1 ,
   #unused    1 1 1   1 1 1
```

```
# Special field.
#
# This field extends 5..0 and is assigned to the funct
# field.
#
# Bits:    5 4 3   2 1 0
#         ------  ------
_sll       0 0 0   0 0 0 , _jr        0 0 1   0 0 0 , _mfhi      0 1 0   0 0 0 ,
_mult      0 1 1   0 0 0 , _add       1 0 0   0 0 0 , #unused    1 0 1   0 0 0 ,
#unused    1 1 0   0 0 0 , #unused    1 1 1   0 0 0 , #unused    0 0 0   0 0 1 ,
_jalr      0 0 1   0 0 1 , _mthi      0 1 0   0 0 1 , _multu     0 1 1   0 0 1 ,
_addu      1 0 0   0 0 1 , #unused    1 0 1   0 0 1 , #unused    1 1 0   0 0 1 ,
#unused    1 1 1   0 0 1 , _srl       0 0 0   0 1 0 , #unused    0 0 1   0 1 0 ,
_mflo      0 1 0   0 1 0 , _div       0 1 1   0 1 0 , _sub       1 0 0   0 1 0 ,
_slt       1 0 1   0 1 0 , #unused    1 1 0   0 1 0 , #unused    1 1 1   0 1 0 ,

_sra       0 0 0   0 1 1 , #unused    0 0 1   0 1 1 , _mtlo      0 1 0   0 1 1 ,
_divu      0 1 1   0 1 1 , _subu      1 0 0   0 1 1 , _sltu      1 0 1   0 1 1 ,
#unused    1 1 0   0 1 1 , #unused    1 1 1   0 1 1 , _sllv      0 0 0   1 0 0 ,
_syscall   0 0 1   1 0 0 , _syscall2  0 1 0   1 0 0 , #unused    0 1 1   1 0 0 ,
_and       1 0 0   1 0 0 , #unused    1 0 1   1 0 0 , #unused    1 1 0   1 0 0 ,
#unused    1 1 1   1 0 0 , #unused    0 0 0   1 0 1 , _break     0 0 1   1 0 1 ,
_call      0 1 0   1 0 1 , #unused    0 1 1   1 0 1 , _or        1 0 0   1 0 1 ,
#unused    1 0 1   1 0 1 , #unused    1 1 0   1 0 1 , #unused    1 1 1   1 0 1 ,

_srlv      0 0 0   1 1 0 , #unused    0 0 1   1 1 0 , #unused    0 1 0   1 1 0 ,
#unused    0 1 1   1 1 0 , _xor       1 0 0   1 1 0 , #unused    1 0 1   1 1 0 ,
#unused    1 1 0   1 1 0 , #unused    1 1 1   1 1 0 , _srav      0 0 0   1 1 1 ,
#unused    0 0 1   1 1 1 , #unused    0 1 0   1 1 1 , #unused    0 1 1   1 1 1 ,
_nor       1 0 0   1 1 1 , #unused    1 0 1   1 1 1 , #unused    1 1 0   1 1 1 ,
#unused    1 1 1   1 1 1

# BCond field.
#
# This field extends 20..16 and is normally the rt field.
#
# Bits:   20..19  18..16
#         ------  ------
_bltz      0 0     0 0 0 , #unused    0 1     0 0 0 , _bltzal    1 0     0 0 0 ,
#unused    1 1     0 0 0 , _bgez      0 0     0 0 1 , #unused    0 1     0 0 1 ,
_bgezal    1 0     0 0 1 , #unused    1 1     0 0 1 ,

# Copf1 field.
#
# This field extends 25..21 and used by coprocessor instructions.
#
# Bits:   24..23  22 21
#         ------  -----
_mf         0 0    0  0 , #unused     0 0    0  1 , _cf         0 0    1  0 ,
#unused     0 0    1  1 , _mt         0 1    0  0 , #unused     0 1    0  1 ,
_ct         0 1    1  0 , #unused     0 1    1  1 , _bc         1 0    0  0 ,
#unused     1 0    0  1 , #unused     1 0    1  0 , #unused     1 0    1  1 ,

# Copf2 field.
#
# This field extends 20..16 and used by coprocessor instructions.
#
# Bits:   20 19 18 17 16
#         --------------
_f        0 0 0  0  0  0 ,
_t        0 0 0  0  0  1 ,

# Cop0
# -----
# Co processor0 operations.
#
```

```
   # Bits :  4 3  2 1 0
   #         ---  -----

   #unused   0 0  0 0 0 ,
   _tlbp     0 1  0 0 0 ,
   _rfe      1 0  0 0 0 ,
   _tlbr     0 0  0 0 1 ,
   _tlbwi    0 0  0 1 0 ,
   _tlbwr    0 0  1 1 0 ,

   _zero     0 0 0  0 0 0 ;

#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
# Floating point compares. Each predicate and its negation
# use the same cond bits.
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#

bitconstant
   _fcond_f     0 0 0 0,  # False / True.
   _fcond_un    0 0 0 1,  # Unordered/ordered (OR)
   _fcond_eq    0 0 1 0,  # Equal / Not equal (NEQ)
   _fcond_ueq   0 0 1 1,  # Unordered or equal /
   _fcond_olt   0 1 0 0,  # Ordered or less than /
                          # Unordered or greater than or equal (UGE)
   _fcond_ult   0 1 0 1,  # Unordered or less than /
                          # Ordered or greater than or equal (OGE)
   _fcond_ole   0 1 1 0,  # Ordered or less than or equal /
                          # Unordered or greater than (UGT)
   _fcond_ule   0 1 1 1,  # Unordered or less than or equal /
                          # Ordered or greater than (OGT)
   _fcond_sf    1 0 0 0,  # Signaling false / Signaling true (ST)
   _fcond_ngle  1 0 0 1,  # Not greater than or less than or equal /
                          # Greater than or less than or equal (GLE)
   _fcond_seq   1 0 1 0,  # Signaling equal / Signaling not equal (SNE)
   _fcond_ngl   1 0 1 1,  # NOt greater or less than /
                          # Greater than or less than (GL)
   _fcond_lt    1 1 0 0,  # Less than / Not less than (NLT)
   _fcond_nge   1 1 0 1,  # Not greater than or equal /
                          # Greater than or equal (GE)
   _fcond_le    1 1 1 0,  # Less than or equal / Not less than or equal (NLE)
   _fcond_ngt   1 1 1 1;  # Not greater than / Greater than (GT)

bitconstant
   _add_fmt     0 0 0 0, _sub_fmt     0 0 0 1, _mul_fmt     0 0 1 0,
   _div_fmt     0 0 1 1, _abs_fmt     0 1 0 1, _mov_fmt     0 1 1 0,
   _neg_fmt     0 1 1 1, _cvt_s_fmt   0 0 0 0, _cvt_d_fmt   0 0 0 1,    # 33,
   _cvt_w_fmt   0 1 0 0, _c_fmt       0 0 0 0, _trunc_w     1 1 0 1;

bitconstant
   fc_arithmetic  0 0,
   fc_cvt         1 0,    # 33,
   fc_c_fmt       1 1;

bitconstant
   _fmul        0 0 1 0,
   _fdiv        1 0 1 0,
   _fadd        1 0 0 0,
   _fsub        0 0 0 1,
   _fneg        0 1 1 1;

instruction

#- SYSCALL : System call - - - - - - - - - - - - - - - - - - - - - - - - - -#
#                                                                           #
#  MIPS I.                                                                  #
#                                                                           #
#  This instruction is used to simulate external calls as if it is a        #
```

```
#  single instruction. This instruction uses $4 as its left operand and       #
#  returns the result in $2.                                                  #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   syscall code
         emit opcode=_special code funct=_syscall
         attributes
             (
              i_class   : integer_class,
              i_cycles  : multiple_cycles,
              exu       : call_unit,
              c_what    : none,
              dest_type : special_input,
              lop_type  : special_input,
              rop_type  : special_input,
              i_type    : alu_type,
              dest_reg  : none
             )
         begin
            case callu
                  dest=builtin do_mips_syscall(lop,lop2,rop,rop2,0);
                  dest2=0;
            end;
         end,


   syscall macro
         begin
            syscall : code=0;
         end,


#- SLL : Shift word left logical - - - - - - - - - - - - - - - - - - - - - -#
#                                                                            #
#  MIPS I.                                                                   #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   sll rd rt shamt
         emit opcode=_special rs=0 rt rd shamt funct=_sll
         attributes
             (
              i_class   : integer_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : none,
              dest_type : integer_register,
              lop_type  : none,
              rop_type  : integer_register,
              i_type    : alu_type,
              dest_reg  : rd
             )
         begin
            case s_EX
                  dest=(+rop) << (+shamt);
            end;
         end,



#- ADD : Add word  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
#                                                                            #
#  MIPS I.                                                                   #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   add rd rs rt
         emit opcode=_special rs rt rd shamt=0 funct=_add
         attributes
             (
              i_class   : integer_class,
              i_cycles  : single_cycle,
```

```
             exu       : integer_unit,
             c_what    : none,
             dest_type : integer_register,
             lop_type  : integer_register,
             rop_type  : integer_register,
             i_type    : alu_type,
             dest_reg  : rd
            )
         begin
            case s_EX
                  dest=lop + rop;
            end;
         end,


#- ADDI : Add immediate word  - - - - - - - - - - - - - - - - - - - - - - - - #
#                                                                            #
#  MIPS I.                                                                   #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   addi rt rs immediate
         emit opcode=_addi rs rt immediate
         attributes
             (
              i_class   : integer_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : none,
              dest_type : integer_register,
              lop_type  : integer_register,
              rop_type  : none,
              i_type    : alu_type,
              dest_reg  : rt
             )
         begin
            case s_ID
                  simm=sign_extend_16(immediate);
            end;

            case s_EX
                  rop=simm;
                  dest=lop + rop;
            end;
         end,


#- ADDIU : Add immediate unsigned word  - - - - - - - - - - - - - - - - - - - #
#                                                                            #
#  MIPS I.                                                                   #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   addiu rt rs uimmediate
         emit opcode=_addiu rs rt uimmediate
         attributes
             (
              i_class   : integer_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : none,
              dest_type : integer_register,
              lop_type  : integer_register,
              rop_type  : none,
              i_type    : alu_type,
              dest_reg  : rt
             )
         begin
            case s_ID
                  simm=sign_extend_16(immediate);
            end;
```

```
           case s_EX
                dest=lop + simm;
           end;
        end,


#- ADDU : add unsigned word  - - - - - - - - - - - - - - - - - - - - - - - - -#
#                                                                             #
#  MIPS I.                                                                    #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   addu rd rs rt
        emit opcode=_special rs rt rd shamt=0 funct=_addu
        attributes
            (
             i_class   : integer_class,
             i_cycles  : single_cycle,
             exu       : integer_unit,
             c_what    : none,
             dest_type : integer_register,
             lop_type  : integer_register,
             rop_type  : integer_register,
             i_type    : alu_type,
             dest_reg  : rd
            )
        begin
           case s_EX
                dest=lop + rop;
           end;
        end,



#- AND : And - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
#                                                                             #
#  MIPS I.                                                                    #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   and   rd rs rt
        emit opcode=_special rs rt rd shamt=0 funct=_and
        attributes
            (
             i_class   : integer_class,
             i_cycles  : single_cycle,
             exu       : integer_unit,
             c_what    : none,
             dest_type : integer_register,
             lop_type  : integer_register,
             rop_type  : integer_register,
             i_type    : alu_type,
             dest_reg  : rd
            )
        begin
           case s_EX
                dest=lop & rop;
           end;
        end,



#- ANDI : And immediate - - - - - - - - - - - - - - - - - - - - - - - - - - #
#                                                                             #
#  MIPS I.                                                                    #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   andi rt rs immediate
        emit opcode=_andi rs rt immediate
        attributes
            (
```

```
             i_class   : integer_class,
             i_cycles  : single_cycle,
             exu       : integer_unit,
             c_what    : none,
             dest_type : integer_register,
             lop_type  : integer_register,
             rop_type  : none,
             i_type    : alu_type,
             dest_reg  : rt
            )
         begin
            case s_ID
                 zimm=zero_extend_16(immediate);
            end;

            case s_EX
                 dest=lop & zimm;
            end;
         end,


#- BEQ : branch on equal - - - - - - - - - - - - - - - - - - - - - - - - - -#
#                                                                           #
#  MIPS I.                                                                  #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   beq__ rs rt immediate
         emit opcode=_beq rs rt  immediate
         attributes
             (
              i_class   : branch_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : condition_equal,
              dest_type : none,
              lop_type  : integer_register,
              rop_type  : integer_register,
              i_type    : conditional_direct,
              dest_reg  : none
             )
         begin
            case s_ID
                 branch_target=my_pc + sign_extend_14(immediate);
            end;
         end,

#- BGEZ : Branch on greater than or equal to zero - - - - - - - - - - - - - #
#                                                                           #
#  MIPS I.                                                                  #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   bgez__ rs  immediate
         emit opcode=_bcond rs rt=_bgez immediate
         attributes
             (
              i_class   : branch_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : condition_gez,
              dest_type : none,
              lop_type  : integer_register,
              rop_type  : none,
              i_type    : conditional_direct,
              dest_reg  : none
             )
         begin
            case s_ID
                 branch_target=my_pc + sign_extend_14(immediate);
            end;
```

```
        end,


#- BGEZAL : Branch on greater than or equal to zero and link  - - - - - - - - - #
#                                                                               #
#  MIPS I.                                                                      #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   bgezal__ rs immediate
         emit opcode=_bcond rs rt=_bgezal immediate
         attributes
             (
              i_class   : branch_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : condition_gez,
              dest_type : integer_register,
              lop_type  : integer_register,
              rop_type  : none,
              i_type    : conditional_direct_link,
              dest_reg  : 31
             )
         begin
            case s_EX
                  dest=my_pc + 8;
            end;
         end,

#- BGTZ : Branch on greater than zero - - - - - - - - - - - - - - - - - - - - - #
#                                                                               #
#  MIPS I.                                                                      #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   bgtz__ rs immediate
         emit opcode=_bgtz rs rt=0 immediate
         attributes
             (
              i_class   : branch_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : condition_gtz,
              dest_type : none,
              lop_type  : integer_register,
              rop_type  : none,
              i_type    : conditional_direct,
              dest_reg  : none
             )
         begin
            case s_ID
                  branch_target=my_pc + sign_extend_14(immediate);
            end;
         end,

#- BLEZ : Branch on less than or equal to zero  - - - - - - - - - - - - - - - - #
#                                                                               #
#  MIPS I.                                                                      #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   blez__ rs immediate
         emit opcode=_blez rs rt=0  immediate
         attributes
             (
              i_class   : branch_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : condition_lez,
              dest_type : none,
              lop_type  : integer_register,
              rop_type  : none,
```

```
             i_type    : conditional_direct,
             dest_reg  : none
            )
         begin
            case s_ID
                 branch_target=my_pc + sign_extend_14(immediate);
            end;
         end,

#- BLTZ : Branch on less zero  - - - - - - - - - - - - - - - - - - - - - - -#
#                                                                           #
#  MIPS I.                                                                  #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   bltz__ rs immediate
         emit opcode=_bcond rs rt=_bltz immediate
         attributes
             (
              i_class   : branch_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : condition_ltz,
              dest_type : none,
              lop_type  : integer_register,
              rop_type  : none,
              i_type    : conditional_direct,
              dest_reg  : none
             )
         begin
            case s_ID
                 branch_target=my_pc + sign_extend_14(immediate);
            end;
         end,

#- BLTZAL : Branch on less zero and link - - - - - - - - - - - - - - - - - -#
#                                                                           #
#  MIPS I.                                                                  #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   bltzal__ rs immediate
         emit opcode=_bcond rs rt=_bltzal immediate
         attributes
             (
              i_class   : branch_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : condition_ltz,
              dest_type : integer_register,
              lop_type  : integer_register,
              rop_type  : none,
              i_type    : conditional_direct_link,
              dest_reg  : 31
             )
         begin
            case s_EX
                 dest=my_pc + 8;
            end;
         end,

#- BNE : Branch on not equal - - - - - - - - - - - - - - - - - - - - - - - -#
#                                                                           #
#  MIPS I.                                                                  #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   bne__ rs rt immediate
         emit opcode=_bne rs rt  immediate
         attributes
             (
              i_class   : branch_class,
              i_cycles  : single_cycle,
```

```
             exu       : integer_unit,
             c_what    : condition_neq,
             dest_type : none,
             lop_type  : integer_register,
             rop_type  : integer_register,
             i_type    : conditional_direct,
             dest_reg  : none
            )
         begin
            case s_ID
                 branch_target=my_pc + sign_extend_14(immediate);
            end;
         end,

   bne__ rs rt address
         emit opcode=_bne rs rt  immediate=<address.delta.jump_address>
         attributes
             (
             i_class   : branch_class,
             i_cycles  : single_cycle,
             exu       : integer_unit,
             c_what    : condition_neq,
             dest_type : none,
             lop_type  : integer_register,
             rop_type  : integer_register,
             i_type    : conditional_direct,
             dest_reg  : none
            )
         begin
            case s_ID
                 branch_target=my_pc + sign_extend_14(immediate);
            end;
         end,

#- Break : Breakpoint - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
#                                                                             #
#  MIPS I.                                                                    #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
#  Description.                                                               #
#                                                                             #
#  We should list any variations of instructions with a constant field        #
#  after the main instruction so that the decoder will correctly print        #
#  out the decoded instruction. For example, the parameterless variant        #
#  of the break instruction sets the code field to zero. By putting           #
#  the one with the parameter first we make sure that its representation      #
#  goes to the decoder.                                                       #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   break code
         emit opcode=_special code funct=_break
         attributes
             (
             i_class   : integer_class,
             i_cycles  : single_cycle,
             exu       : integer_unit,
             c_what    : none,
             dest_type : none,
             lop_type  : none,
             rop_type  : none,
             i_type    : system_type,
             dest_reg  : none
            ),

   break macro
         begin
            break : code=0;
         end,
```

```
#- CTC1  : - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
#                                                                             #
#  MIPS I.                                                                    #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
#  Description.                                                               #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   ctc1 rt fs
         emit opcode=_cop1 cof=0 copf1=_ct rt fs copf3=0
         attributes
             (
              i_class   : float_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : none,
              dest_type : none,
              lop_type  : float_register,
              rop_type  : integer_register,
              i_type    : alu_type,
              dest_reg  : rt
             )
         begin
            case s_EX
                  dest=lop;
            end;
         end,

   cfc1 rt fs
         emit opcode=_cop1 cof=0 copf1=_cf rt fs copf3=0
         attributes
             (
              i_class   : float_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : none,
              dest_type : none,
              lop_type  : float_register,
              rop_type  : integer_register,
              i_type    : alu_type,
              dest_reg  : rt
             )
         begin
            case s_EX
                  dest=lop;
            end;
         end,

#- DIV  : Divide word - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
#                                                                             #
#  MIPS I.                                                                    #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   div  rs rt
         emit opcode=_special rs rt rd=0 shamt=0 funct=_div
         attributes
             (
              i_class   : long_integer_class,
              i_cycles  : multiple_cycles,
              exu       : divide_unit,
              c_what    : none,
              dest_type : lo_hi_register,
              lop_type  : integer_register,
              rop_type  : integer_register,
              i_type    : alu_type,
              dest_reg  : lo_hi_register_number
             )
         begin
            case s_ID
```

```
                latency 7;
           end;

           case s_EXD
                if rop == 0 then
                   exception = 1
                else
                   begin
                      dest2 = lop / rop;
                      dest  = lop % rop;
                   end;
           end;
        end,

#- J : Jump - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
#                                                                              #
#  MIPS I.                                                                     #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   j__ j_offset
        emit opcode=_j j_offset
        attributes
            (
             i_class   : branch_class,
             i_cycles  : single_cycle,
             exu       : integer_unit,
             c_what    : condition_z,
             dest_type : none,
             lop_type  : none,
             rop_type  : none,
             i_type    : unconditional_direct,
             dest_reg  : none
            )
        begin
           case s_ID
                branch_target=my_pc.[31:4] || j_offset || 0 |< 2;
           end;
        end,



#- JAL : Jump and link  - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
#                                                                              #
#  MIPS I.                                                                     #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   jal__ j_offset
        emit opcode=_jal j_offset
        attributes
            (
             i_class   : branch_class,
             i_cycles  : single_cycle,
             exu       : integer_unit,
             c_what    : condition_z,
             dest_type : integer_register,
             lop_type  : none,
             rop_type  : none,
             i_type    : unconditional_direct_link,
             dest_reg  : 31
            )
        begin
           case s_ID
                branch_target=my_pc.[31:4] || j_offset || 0 |< 2;
           end;

           case s_EX
                dest=my_pc + 8;
           end;
        end,
```

```
#- JALR : Jump and link register - - - - - - - - - - - - - - - - - - - - - - - -#
#                                                                               #
#  MIPS I.                                                                      #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   jalr__ rd rs  emit opcode=_special rs rt=0 rd shamt=0 funct=_jalr
         attributes
             (
              i_class   : branch_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : condition_u,
              dest_type : integer_register,
              lop_type  : integer_register,
              rop_type  : none,
              i_type    : unconditional_indirect_link,
              dest_reg  : rd
             )
         begin
            case s_EX
                 dest=my_pc + 8;
            end;
         end,



#- JR : Jump register - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
#                                                                               #
#  MIPS I.                                                                      #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   jr__ rs
         emit opcode=_special rs rt=0 rd=0 shamt=0 funct=_jr
         attributes
             (
              i_class   : branch_class,
              i_cycles  : single_cycle,
              exu       : integer_unit,
              c_what    : condition_u,
              dest_type : none,
              lop_type  : integer_register,
              rop_type  : none,
              i_type    : unconditional_indirect,
              dest_reg  : none
             ),


#- LB : Load byte - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
#                                                                               #
#  MIPS I.                                                                      #
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
   lb rt address
         emit opcode=_lb rs=<address.base> rt immediate=<address.offset>
         attributes
             (
              i_class   : integer_class,
              i_cycles  : multiple_cycles,
              exu       : load_unit,
              c_what    : none,
              dest_type : integer_register,
              lop_type  : integer_register,
              rop_type  : none,
              i_type    : load_type,
              dest_reg  : rt
             )
         begin
```

```
            case s_ID
                  simm=sign_extend_16(immediate);
            end;

            case s_EX
                  lmar=lop + simm;
                  byte=lmar.[1:2];
            end;

            case s_MEM
                  if ls_bypass then
                     dest=dest2
                  else
                     begin
                        dest=ncache [lmar];
                        mem_stat=access_complete;
                     end;
                  lmdr=dest;

                  if mem_stat | ls_bypass then
                     begin
                        if byte == 0 then
                           dest = dest.[07:08]
                        else
                        if byte == 1 then
                           dest = dest.[15:08]
                        else
                        if byte == 2 then
                           dest = dest.[23:08]
                        else
                           dest = dest.[31:08];
                        dest=sign_extend_24(dest);
                     end;
               end;
           end,

 ..................

;


controlflow
   bc1f__, beq__, bgez__, bgezal__, bgtz__, blez__, bltz__, bltzal__,
   bne__, j__, jal__, jalr__, jr__;

instruction category integer_arithmetic
   add, addi, addiu, addu, and, andi, div, divu, lui, mfhi, mflo, mult,
   multu, nor, or, ori, sll, sllv, slt, slti, sltiu, sltu, sra, srav, srl,
   srlv, sub, subu, xor, xori;

instruction category conditional_branch
   beq__, bgez__, bgezal__, bgtz__, blez__, bltz__, bltzal__, bne__;

instruction category other
   break;

instruction category unconditional_branch
   j__, jal__, jalr__, jr__;

instruction category load
   lb, lbu, lh, lhu, lw, lwl, lwr, lwc1__;

instruction category store
   sb, sh, sw, swl, swr, swc1__;

instruction category float_arithmetic
   "cvt.d.w", "cvt.d.s", "cvt.s.w", "cvt.s.d", "div.d", "div.s", "mul.s",
```

```
   "mul.d", "add.s", "add.d", "neg.s", "neg.d", "sub.s", "sub.d", mfc1,
   mtc1, "c.cond.d", "c.cond.s", "abs.s", "abs.d", "mov.s", "mov.d",
   "trunc.w.s", "trunc.w.d";

instruction category float_conditional
    bc1f__,
    bc1t__;
```

Bibliography

Bibliography

[1] Armstrong, J. R. and Gray, F. G. *Structured Logic Design with VHDL.* New Jersey: Prentice Hall, 1993.

[2] Arvind and Iannucci, R. Two fundamental issues in multiprocessing. Computation Structures Group Memo 26, Laboratory for Computer Science, MIT, 1987.

[3] Arvind, Kathail, V., and Pingali, K. A dataflow architecture with tagged tokens. Technical Report 174, MIT, 1980.

[4] Austin, T. M. and Sohi, G. S. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th International Conference on Computer Architecture*, pages 342–351, 1992.

[5] Bailey, M. W. and Davidson, J. W. A formal model and specification language for procedure calling conventions. In *The 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–310, 1995.

[6] Black, B., Rychlik, B., and Shen, J. P. The block-based trace cache. In *Proceedings of the 26th International Conference on Computer Architecture*, pages 196–207, May 1999.

[7] Burger, D. C. and Austin, T. M. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, Computer Science Department, University of Wisconsin Madison, 1997.

[8] Butler, M., Yeh, T., Patt, Y., Alsup, M., Scales, H., and Shebanow, M. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th International Conference on Computer Architecture*, pages 276–286, 1991.

[9] Calder, B., Reinman, G., and Tullsen, D. M. Selective value prediction. In *Proceedings of the 26th International Conference on Computer Architecture*, pages 64–74, May 1999.

[10] Chrysos, G. Z. and Emer, J. S. Memory dependence prediction using store sets. In *Proceedings of the 25th International Conference on Computer Architecture*, pages 142–153, June 1998.

[11] Cook, T. A. *Instruction set architecture specification.* PhD thesis, Norrh Carolina State University, 1993.

[12] Cook, T. A. and Harcourt, E. A. A functional specification language for instruction set architectures. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 11–19, 1994.

[13] Dennis, J. B. The evolution of "static" data-flow architecture. In Gaudiot, J. and Bic, L., editors, *Advanced Topics in Data-Flow Computing*, pages 35–91. New Jersey: Prentice Hall, 1991.

[14] Dennis, J. B. and Gao, G. R. An efficient pipelined dataflow processor architecture. In *Proceedings of the IEEE and ACM SIGARCH Conf. on Supercomputing*, pages 368–373, 1988.

[15] Dennis, J. B. and Misunas, D. P. A preliminary architecture for a basic data flow computer. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, 1975.

[16] Diep, T. A. *A visualization-based microarchitecture workbench.* PhD thesis, Carnegie Mellon University, Pittsburgh, USA, 1995.

[17] Friendly, D. H., Patel, S. J., and Patt, Y. N. Alternative fetch and issue policies for the trace cache fetch mechanism. In *The 30th Annual IEEE-ACM International Symposium on Microarchitecture*, pages 24–33, December 1997.

[18] Gao, G. R., Tio, R., and Hum, H. Design of an efficient dataflow architecture without data flow. Technical Report TR-SOCS-88.14, School of Computer Science, McGill University, 1988.

[19] Grafe, V., Davidson, G., Hoch, J., and Holmes, V. The ϵpsilon dataflow processor. In *Proceedings of the 16th International Conference on Computer Architecture*, pages 36–45, 1989.

[20] Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[21] Hesson, J., LeBlanc, J., and Ciavaglia, S. Apparatus to dynamically control the Out-Of-Order execution of Load-Store instructions. *US. Patent* 5,615,350, Filed Dec. 1995, Issued Mar. 1997.

[22] Huffman, L. and Graves, D. *MIPSpro Assembly Language Programmers Manual.* Silicon Graphics Corporation, Document number 007-2418-002, 1996.

[23] Hum, H. and Gao, G. Efficient support of concurrent threads in a hybrid dataflow/von Neumann architecture. In *Proceedings of the IEEE Symposium and Parallel and Distributed Processing*, pages 190–193, 1991.

[24] Iannucci, R. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th International Conference on Computer Architecture*, pages 131–140, 1988.

[25] Johnson, M. *Superscalar Microprocessor Design.* Prentice Hall, 1991.

[26] Jouppi, N. P. and Wall, D. W. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 272–282, 1989.

[27] Jouppi, N. P. The non-uniform distribution of instruction-level and machine level parallelism and its effect on performance. *IEEE Transactions on Computers*, C-38(12):1645–1658, December 1989.

[28] Jourdan, S., Ronen, R., Bekerman, M., Shomar, B., and Yoaz, A. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *The 31st Annual IEEE-ACM International Symposium on Microarchitecture*, pages 216–225, December 1998.

[29] Keller, R. M. Look-ahead processors. *ACM Computing Surveys*, 7(4):177–195, December 1975.

[30] Kemp, G. A. and Franklin, M. A decentralized dynamic scheduler for ilp processing. In *Proceedings of the International Conference on Parallel Processing*, volume 1, pages 239–246, 1996.

[31] Larsson, F. Generating efficient simulators from a specification language. Technical Report 1997-01-29, Computing Science Department, Uppsala University, Uppsala, Sweden, 1997.

[32] Larus, J. R. SPIM S20: A MIPS R2000 Simulator. Technical Report CS-TR-90-966, Computer Science Department, University of Wisconsin Madison, 1990.

[33] Leupers, R., Elste, J., and Landwehr, B. Generation of interpretive and compiled instruction set simulators. In *ASP-DAC*, January 1999.

[34] Lipasti, M. H. and Shen, J. P. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, 1996.

[35] Lipasti, M. H. and Shen, J. P. Superspeculative microarchitecture for beyond ad 2000. *IEEE Computer*, 30(9):59–66, 1997.

[36] Lipasti, M. H., Wilkerson, C. B., and Shen, J. P. Value locality and load value prediction. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 138–147, October 1996.

[37] McFarling, S. Combining branch predictors. Technical Report WRL-TN-36, Digital Western Research Laboratory, 1993.

[38] Morison, J. and Clarke, A. S. *ELLA2000 A language for Electronic System Design*. McGraw-Hill, Inc., 1993.

[39] Moshovos, A. I. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin - Madison, 1998.

[40] Moshovos, A. I., Breach, S. E., Vijaykumar, T. N., and Sohi, G. S. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 181–193, June 1997.

[41] Moura, C. SuperDLX a generic superscalar simulator. Technical Report ACAPS Technical Memo 64, School of Computer Science, McGill University, 1993.

[42] Nikhil, R. and Arvind. Can dataflow subsume von neumann computing? In *Proceedings of the 16th International Conference on Computer Architecture*, pages 262–272, 1989.

[43] Nikhil, R. S., Papadopoulos, G., and Arvind. *T:a multithreaded massively parallel architecture. In *Proceedings of the 19th International Conference on Computer Architecture*, pages 156–167, 1992.

[44] Noonburg, D. B. and Shen, J. P. Theoretical modeling of superscalar processor performance. In *The 27th Annual IEEE-ACM International Symposium on Microarchitecture*, pages 52–62, November 1994.

[45] Önder, S. and Gupta, R. SINAN: An argument forwarding multithreaded architecture. In *International Conference on High Performance Computing*, pages 347–354, New Delhi, India, December 1995.

[46] Önder, S. and Gupta, R. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, Chicago, May 1998.

[47] Önder, S. and Gupta, R. Superscalar execution with dynamic data forwarding. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 130–135, October 1998.

[48] Önder, S. and Gupta, R. Caching and predicting branch sequences for improved fetch effectiveness. To appear in International Conference on Parallel Architectures and Compilation Techniques, October 1999.

[49] Palacharla, S., Jouppi, N. P., and Smith, J. E. Quantifying the complexity of superscalar processors. Technical Report CS-TR-96-1328, University of Wisconsin Technical Report, 1996.

[50] Palacharla, S., Jouppi, N. P., and Smith, J. Complexity-effective superscalar processors. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 206–218, June 1997.

[51] Papadopoulos, G. M. Implementation of a general purpose dataflow multiprocessor. Technical Report TR–432, MIT, 1988.

[52] Papadopoulos, G. and Culler, D. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th International Conference on Computer Architecture*, pages 82–91, 1990.

[53] Patt, Y. N., Patel, S. J., Evers, M., Friendly, D. H., and Stark, J. One billion transistors, one uniprocessor, one chip. *IEEE Computer*, 30(9):51–57, 1997.

[54] Perry, D. L. *VHDL*. McGraw-Hill, Inc., 1991.

[55] Postiff, M. A., Greene, D., Tyson, G., and Mudge, T. The limits of instruction level parallelism in spec95 benchmarks. In *INTERACT-3 : The third workshop on interaction between compilers and computer architectures, San Jose, CA*, October 1998.

[56] Price, C. *MIPS IV Instruction Set Revision 3.2*. MIPS Technologies Inc., September 1995.

[57] Ramsey, N. and Fernandez, M. F. The new jersey machine-code toolkit. In *Proceedings of the 1995 USENIX Technical Conference, New Orleans, LA*, pages 289–302, January 1995.

[58] Ramsey, N. and Fernandez, M. F. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[59] Reinman, G., Austin, T., and Calder, B. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th International Conference on Computer Architecture*, pages 234–245, May 1999.

[60] Reinman, G. and Calder, B. Predictive techniques for aggressive load speculation. In *The 31st Annual IEEE-ACM International Symposium on Microarchitecture*, pages 127–137, December 1998.

[61] Rotenberg, E., Bennett, S., and Smith, J. E. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, December 1996.

[62] Smith, J. E. and Vajapeyam, S. Trace processors: Moving to fourth generation microarchitectures. *IEEE Computer*, 30(9):68–70, 1997.

[63] Sohi, G. S., Breach, S., and Vijaykumar, T. N. Multiscalar processors. In *Proceedings of the 22th International Conference on Computer Architecture*, pages 414–425, 1995.

[64] Theobald, K. B., Gao, G. R., and Hendren, L. J. On the limits of program parallelism and its smoothability. In *The 25th Annual IEEE-ACM International Symposium on Microarchitecture*, pages 10–19, December 1992.

[65] Thomas, D. E. and Moorby, P. R. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[66] Thornton, J. E. *Design of a computer : the Control Data 6600*. Glenview, Ill.: Scott, Foresman, 1970.

[67] Todd, K. W. High level VAL constructs in a static data flow machine. Technical Report MIT/LCS/TR-262, Massachusetts Institute of Technology, Laboratory for Computer Science, 1981.

[68] Tomasulo, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, 1967.

[69] Tullsen, D. M. and Seng, J. S. Storageless value prediction using prior register values. In *Proceedings of the 26th International Conference on Computer Architecture*, pages 270–279, May 1999.

[70] Vajapeyam, S., Joseph, P. J., and Mitra, T. Dynamic vectorization: a mechanism for exploiting far-flung ilp in ordinary programs. In *Proceedings of the 26th International Conference on Computer Architecture*, pages 16–27, May 1999.

[71] Vajapeyam, S. and Mitra, T. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 1–12, June 1997.

[72] Wall, D. W. Limits of instruction level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 176–189, 1991.

[73] Wall, D. W. Limits of instruction level parallelism. Technical Report WRL-TR-93-6, Digital Equipment Corporation Western Research Laboratory, 1993.

[74] Weiss, S. and Smith, J. E. Instruction issue logic in pipelined supercomputers. *IEEE Transactions on Computers*, C-33:1013–1022, November 1984.