Performance Evaluation of Improved Superscalar Issue Mechanisms

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

in

Faculty of Engineering

by

Tulika Mitra

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

January 1997

Acknowledgements

It is my pleasure to express the deepest sense of gratitude to my project supervisor, Dr. V. Sriram, for his constant guidance, support, and encouragement during the course of this work. He has introduced me to the interesting field of Computer Architecture, and I thank him for motivating me and building up my confidence in this field. Working with him has been a pleasure.

I thank the Chairman, members of faculty, student, and staff of the department of Computer Science and Automation for providing a congenial environment. I wish to express my appreciation to the authorities of SERC for providing the computing resources.

My colleagues, in particular, KVS, Gomu, and Muthu were ever ready to help me, and have always been a source of constant inspiration and amusement. I wish to thank them wholeheartedly.

I extend my heartfelt thanks to my friends, Ananya and Manaswita, who shared with me, my joys and sorrows, helped me to smile in difficulties, and did all they could do whenever I was in need. They have made my hostel life a memorable one.

A big thanks to my friends, Ambica, Anasua, Anupama, Bhaswati, Debashis, Indranath, Jagadish, Modhumita, Sarla, Shibashis, Soumi, Sridevi, Subhankar, and all my batchmates, whose association I shall ever cherish.

Last, but not the least, I acknowledge my parents for years of encouragement, love, support, and faith in me.

Contents

List of Tables

List of Figures

Abstract

Current superscalar processors have the potential to establish a large and accurate dynamic instruction window from which independent instructions can be extracted and scheduled for parallel execution. Recently proposed improved instruction fetch mechanisms help to achieve a large instruction window. Although dynamic scheduling with a large centralized window has the potential to exploit instruction level parallelism, the complicated hardware requirement for this mechanism imposes a cycle time penalty. Also, register renaming latency is bound to increase with instruction fetch bandwidth. A recently proposed microarchitecture addresses this tradeoff between cycle time and clocks per instruction by partitioning the instruction window as well as the register file, each part taking care of a dynamic code sequence. Register renaming bandwidth is increased by reusing the renaming information for a dynamic code sequence. This thesis presents a performance evaluation of this architecture. The results indicate that the proposed mechanisms for improving issue logic can achieve a speedup of 1.5 to around 3 for the Spec integer benchmarks over traditional superscalar architecture. A dynamic loop vectorization method for ordinary programs, to be built on top of the above mechanisms, has been proposed recently. We find that this mechanism can capture up to 60% of dynamic instructions, though the average number of iterations per loop is quite small. The preliminary findings encourage further investigation of dynamic vectorization.

Chapter 1

Introduction

1.1 Motivation

In von Neumann machines, programs are written with the tacit assumption that instructions will be executed in the same order as they appear in the program. To achieve high performance, however, it is essential to execute multiple instructions simultaneously, and perhaps in an order different from the original program sequence. Superscalar processors belong to the class of architectures that uses aggressive hardware techniques to perform this reordering and extract large quantities of instruction level parallelism(ILP) from ordinary high level language programs while preserving the sequential semantics assumed in the program.

An inspection of many sequential programs reveals that there exists a large amount of theoretically exploitable ILP [1] [26]. However, most programs are written in an imperative language for a sequential machine with a limited number of architectural registers for holding temporary values. Hence, it is quite likely that instructions of close proximity are data dependent. This implies most of the parallelism can be found only further down in the instruction stream. The current trend in superscalar processors is to exploit this parallelism in an obvious way, i.e., to establish a large and accurate dynamic window of instructions from which independent instructions can be extracted and scheduled for

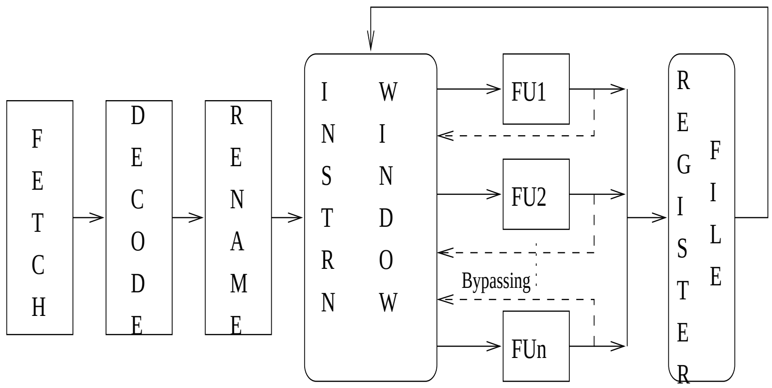

Figure 1.1: Typical Pipeline Stages of a Superscalar Processor

parallel execution. Although dynamic scheduling with a large centralized window has the potential to reduce the number of cycles for program execution, the complicated hardware required for the mechanism imposes a cycle time penalty.

1.2 Complexity

Current superscalar processors are capable of fetching multiple basic blocks per cycle so that the dynamic scheduler can exploit ILP beyond basic block boundaries [14] [3] [8]. However, this increased fetch bandwidth introduces complexity in the later pipeline stages. Figure 1.1 shows an aggressive superscalar processor capable of out-of-order issue. The fetch unit fetches multiple instructions every cycle. Following instruction fetch, the instructions are decoded and the register operands are renamed (i.e. architectural registers are mapped to appropriate physical registers to eliminate write-after-read and write-after-write dependencies). Renamed instructions wait in the instruction window for their source operands and appropriate functional unit to become available and then get issued and executed. The result values are written to the register file and/or bypassed to all the waiting instructions. In this section, the sources of complexity of this out-of-order engine are discussed.

1.2.1 Register Renaming

The register renaming bandwidth must match the instruction fetch bandwidth. However, increasing this bandwidth beyond a few instructions per cycle is not straightforward. Traditional renaming requires $n(n\text{-}1)$ comparators to determine dependencies between the n instructions getting renamed simultaneously, (each source register should be compared with the destination registers of instructions earlier in the instruction stream, but being renamed simultaneously), and $2n$ register rename table read ports to handle potentially 2 source operands for each instruction. Also, this renaming process is sequential in nature [27]. This is because a source register might be identical with a number of previous destination register in the current rename group, but it should pick up the most recent mapping of the architectural register. Weiss [27] suggested a method to rename n instructions per cycle at the cost of maintaining $n/2$ extra rename tables. The hardware cost for these auxiliary structures might be prohibitive with increased fetch bandwidth. The partially parallel method of dependency checking [13] requires at least $2n$ read ports and n write ports for the register map table. Increased number of ports for the map table increases its silicon area requirement, and thereby slows down the map table lookup logic. A recent study [13] shows that the register renaming latency increases linearly with the number of instructions being renamed every cycle. In essence, achieving high rename bandwidth is quite difficult.

1.2.2 Dynamic Scheduling

As fetch bandwidth increases, the instruction window size grows. Austin *et. al.* [1] observed that higher degree of parallelism can be achieved with increased window size. Their study using SPEC benchmarks indicated that a very extensive look ahead, of the order of 100,000 instructions, will be required to exploit all the available parallelism. However, they did not have a well defined fetch bandwidth. Our preliminary study with finite fetch bandwidth (refer Table 1.1) shows increased demand on the window size as

Benchmark	Fetch Bandwidth(Instructions/Cycle)							
	8		16		32		64	
	Avg. Win Size	ILP	Avg. Win Size	ILP	Avg. Win Size	ILP	Avg. Win Size	ILP
Bubsort (100 rand. numbers)	18	8.0	149	16.0	424	31.8	989	63.2
Eqntott (short in)	28	8.0	1095	15.6	5714	27.8	6496	51.2
Espresso (short in)	52	8.0	2698	9.0	4421	9.0	5283	9.0

Table 1.1: Impact of fetch bandwidth on instruction window size and performance

fetch bandwidth increases to achieve higher ILP [1], even though we assumed unlimited functional units and unlimited instruction issue bandwidth.(This study assumed perfect branch prediction and perfect memory disambiguation.) Dynamic instruction scheduling for this large instruction window is quite complex. It involves determining ready instructions from the window and selecting n instructions for execution from the pool of ready instructions. These two tasks must be sequentialized and typically involves wide associative search of the window which is non-trivial. Recent cycle time evaluations [13] [9] indicate increased delay of this logic as window size increases.

1.2.3 Register File

As window size increases, the physical register file size should increase since each instruction (except for branch and store instructions) requires a unique destination physical register. Also, large window requires multiple read ports for the register file to provide operands for parallel execution. This increases the register file die area and makes the register read slow thereby imposing cycle time penalty.

[1]Throughout the report ILP implies IPC i.e. Instructions Per Cycle

1.2.4 Operand Bypassing

The complexity of bypassing grows rapidly as the degree of parallelism increases. As the window size increases, a large number of functional units are required to exploit the parallelism. The full bypassing of these results to all the waiting instructions in the window imposes a significantly large penalty on superscalar machines.

Palacharla *et. al.* [13] indicated that instruction scheduling determines the critical path delay followed by register renaming and full bypassing for superscalar processors as we go for higher fetch rates. Hence, new microarchitectures should address the tradeoff between the clock speed (cycle time) and instructions per cycle.

1.3 Overview

The performance of a superscalar processor is critically dependent on the complexity and performance of issue logic, register renaming, and the broadcasting of results to waiting instructions. This thesis evaluates the potential of novel, improved superscalar processor mechanisms that alleviate the described bottlenecks, and is organized as follows. Chapter 2 presents a discussion of the design and implementation of various previously proposed superscalar processors that simplifies the hardware logic to achieve a faster clock. A new architecture incorporating improved instruction dispatch, rename, and issue mechanisms was recently proposed in [25]. A description of this architecture is provided in Chapter 3 to facilitate the subsequent description of its performance evaluation. The effectiveness of the new architecture over a traditional superscalar processor is shown in Chapter 4 through trace-driven simulation. A dynamic vectorization technique, built on top of the new architecture to reduce the demand on instruction window and register file, has been proposed in [25]. A description of the proposed mechanism is included in Chapter 5, and a preliminary performance evaluation is reported. Finally Chapter 6 offers some concluding remarks and future directions for work.

Chapter 2

Background

In this chapter various designed or implemented microarchitectures that aim to simplify the issue logic, register file read or register renaming are discussed.

2.1 Decoupled Access/Execute Architecture

Decoupled access/execute architectures [17] separate the memory access and execution components of a program into independent instruction streams, and execute them on separate processors. These processors are loosely coupled through architectural queues, which provides an asynchronous message passing channel, and allows the two instruction streams to *slip* dynamically with respect to each other i.e.. the access processor can run ahead of the execute processor, effectively migrating data fetching to an earlier time. Architectural queues in addition to out-of-order execution, also provides means to register renaming. Branch queues are used in order to allow the two processors to synchronize at branch points. This design is significantly less complex and costly in terms of issue logic and register renaming compared to superscalar processors. However, it relies on an intelligent compiler to separate, schedule, and coordinate the two independent instruction streams operating on a task.

2.2 Semi-Unified Instruction Window

Several current superscalar processors divide the instruction window into several parts, each part being devoted to a set of functional units. For example, integer, floating point, and memory reference instructions can be decoupled using one queue for integer, one for floating point, and other for load/store instructions. This matches the features of MIPS R10000 [28]. Another alternative is to separate integer, floating point, and branch instructions as done in IBM RISC System/6000 [7]. These processors naturally have separate floating and integer register files that can potentially improve register access time by a factor of two. Since the instructions of this two types are independent, mostly there is no need for dependency checking between the different queues and instruction issue and result bypass logic become quite simplified. However, across-queue dependence checking is still necessary for load/store instructions and is quite complicated [27]. Also, when the application program is either integer intensive or floating point intensive this splitting will not result in any performance gain.

2.3 Tomasulo's Algorithm

An alternative design was provided by Tomasulo's algorithm [22] that partitions the instruction window into the well-known *reservation stations*, each of them being devoted to one functional unit. Concerning the issue procedure, the select logic is extremely simple because they select only on entry. However, when an instruction completes, its result and tag should be broadcasted to all the waiting instructions and hence is slow. Also, it induces a performance loss since a functional unit may be waiting for issuable instructions while another of the same type may be requested by several issuable instructions [16].

2.4 Multiscalar Architecture

Multiscalar architecture [11] [19] splits a large operation window into small windows, and executes these small windows in parallel on simple, sequential processors. The register and memory values are passed effectively across the multiple windows in execution. The independent processors conceptually form a circular queue, with the top-level control unit performing the task of assigning basic windows and enforcing dependencies between them, and each processor identifying independent instructions within the small window. Each processing element has a local register file called *future file*, and only the last updates to the registers in a processing element is passed on to the next element. Since a significant number of register instances are used up, and eventually overwritten soon after they are created, much of the register traffic can be handled by the local register files [6]. Task partitioning and scheduling for multiscalar processors are quite important. If there are too many inter-task dependencies, the consuming task will stall the producing task. If tasks are not of proper granularity, a processing unit that completes a comparatively short task performs no computation while it waits for all the predecessor tasks to complete.

2.5 Dependence-Based Microarchitecture

Palacharla *et. al.* [13] proposed a dependence-based microarchitecture that partitions instruction window into a number of FIFO queues, each containing a dependent chain of instructions determined at runtime. Thus result broadcasting is required only to head of the queues, and select logic also needs to examine only the instruction at the head of the queues. [1]

2.6 Fill-Unit Approach

Franklin and Smotherman [5] used a structure called *fill unit* to dynamically group instructions that can be issued together. The fill unit accepts decoded instructions from the normal instruction stream generated by program order, simultaneously compacts them into a VLIW-like instruction based on data-dependency and resource requirements, and stores this instruction in a shadow instruction cache. Later, if there is a match in the

[1] Trace-Window Architecture and Dependence-Based Microarchitecture have been proposed independently at the same time

shadow cache, the instructions in the shadow cache line are issued and executed together preempting the scalar instruction stream. This eliminates the need for decoding and hardware dependence checking. The drawback of this approach lies in the fact that the VLIW-like instruction length is very small on general-purpose code due to data dependencies among the instructions. Also, there is a tradeoff with the large on-chip shadow cache structure and the logic for fill unit.

Smotherman and Franklin [18] later extended the fill unit approach to CISC instructions, where, the fill unit captures a set of decoded microoperations into a decoded instruction cache(DIC), thereby bypassing the complicated CISC decode logic when the instruction hits in the DIC. Also, they partition the register file into *fill unit register file* that stores the microarchitectural registers, and general register file. When a line is fetched from the DIC, the microarchitectural registers are renamed to fill unit register file, whereas architectural registers are renamed to general register file. For instruction stream from instruction cache both are sent through general renaming hardware.

2.7 Combined Static/Dynamic Register Renaming

Sprangle and Patt [20] partitioned the register file into a number of internal register files, one for each basic block, that handles the register instances created and needed within the basic block, and an external register file that keeps track of the instances needed beyond the basic block. An external bit associated with each operand indicates whether the operand refers to internal or external register file. If the compiler is forced to generate unique destination registers within a basic block, the renaming tag for the internal registers can be easily generated by combining the dynamically defined unique basic block number with the static register number. This eliminates the dependence check logic within a basic block. Also, mapping for source instruction is requested from the map table only when the instance is generated by an instruction in a previous basic block, and thereby reducing the number of read ports required for map table considerably. However, for general purpose programs, average basic block size is quite small, and most of the register instances might be needed outside the basic block. This will result in extremely poor utilization of the internal register files.

Chapter 3

Trace Window Architecture

The instruction window operations, register renaming, and full bypassing of results are the primary contributors of complexity and bottlenecks in typical out-of-order microarchitecture. A new microarchitecture called trace window architecture has been proposed [25] that replaces the instruction window with a simpler structure that facilitates a faster clock and thereby exploits better instruction level parallelism over traditional architecture. In this chapter, this architecture is described.

3.1 Trace Window

The main idea is to partition the centralized instruction window into a fixed number of small sub-windows so that instruction selection logic becomes simplified. Selection and issue of the ready instructions can be done independently for each sub-window and hence, in parallel. Each sub-window is logically equivalent to a *trace line* of the trace cache [14].

The trace cache is a special instruction cache that captures dynamic instruction sequence. A trace line is a sequence of at most n successive instructions and at most m successive basic blocks starting at any point in the dynamic instruction stream [14]. n is determined by cache line size and m is the branch predictor throughput. A trace can

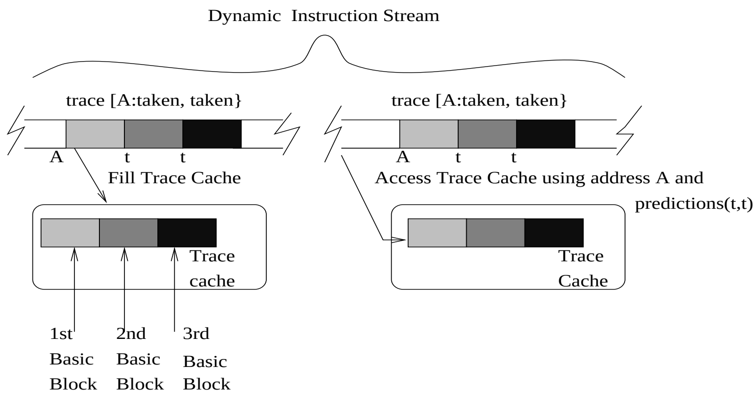

Figure 3.1: Trace Cache Approach

be uniquely identified by the starting address and the sequence of up to m-1 branch outcomes. The first time the trace is encountered in the dynamic instruction stream it is captured in the trace cache. On each fetch cycle the trace cache is looked up using the next address and the branch outcomes. If there is a match, the trace line can be directly fed to the decoder. Otherwise, fetching proceeds normally from the instruction cache. Figure 3.1 shows the trace cache approach. [1]

In the instruction window, each sub-window holds a trace line, and the instruction window is termed trace window. A trace line, decoded and renamed is dispatched to the sub-window and the sub-window can be freed only when all the instructions of that sub-window have been issued. Figure 3.2 shows the organization of trace window architecture.

3.2 Trace Lines

After a trace line is fetched from the trace cache, it is decoded, renamed, and then dispatched to a single sub-window in the trace window. To simplify the issue logic, each trace line has its own copy of functional units. Duplicating functional units is not costly considering the increased number of transistor density per chip. However, if required, only

[1] Refer [14]

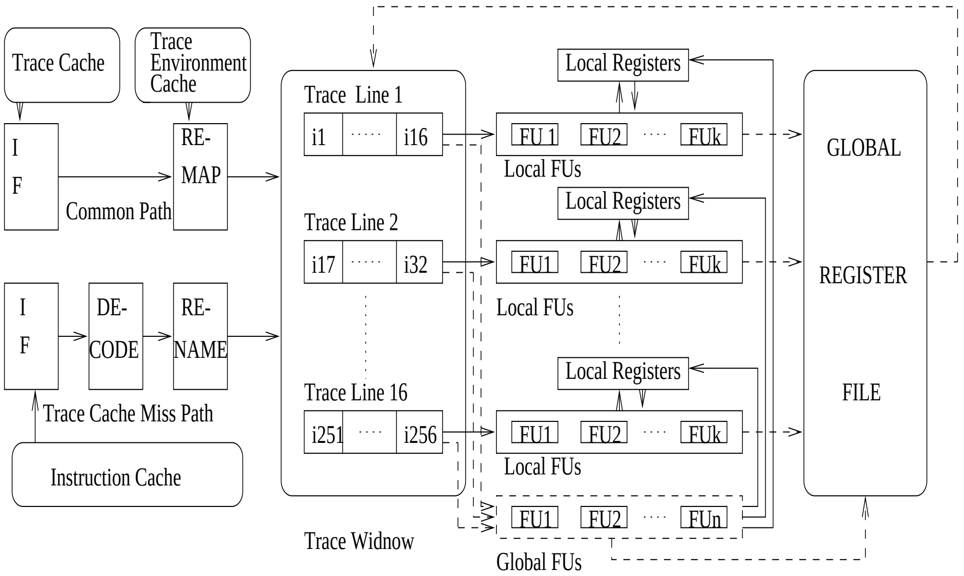

Figure 3.2: Trace Window Architecture

the frequently used functional units can be duplicated, and the costlier, rarely used units can be shared across the trace lines. Result bypassing is now feasible from each functional unit group to the associated trace line in single clock cycle. But, broadcasting results to other trace lines can not be done in a single clock and will require a separate register write-back stage. The key point to note here, however, is that most results get consumed within the next 32 instructions in the dynamic instruction stream [6]. This implies local result bypassing within the trace line is sufficient in most of the cases, and broadcasting results to other trace lines will be required rarely for trace line of 32 instructions or more. It can be noted here that multiscalar architecture is also based on similar philosophy as far as register traffic is concerned [11] [19]. Also, trace window architecture will scale well as processor technology goes for higher fetch bandwidths since the trace line size will increase and restricted result bypassing will achieve the same effect as full bypassing.

As discussed in Section 1.2.2, instruction issue logic has two components:

1. *wakeup* that determines the ready instructions, and

2. *select* that selects instructions for execution from the pool of ready instructions.

Now, we observe that both wakeup and select logic can be distributed, one for each trace line, and can be done independently and in parallel decreasing the latency of that logic considerably. Centralized issue logic is required to take care of result broadcasting across trace lines and for the instructions that require the shared functional units. However, the pressure on the centralized issue logic will be much less, decreasing its latency and the latency of the issue stage as a whole.

In order to simplify the issue logic further, the issue bandwidth is restricted to a small number of instructions per cycle per trace line. This reduces the time and number of wires required to ship the opcode and operands of the selected instructions to the respective functional units, thereby decreasing the window die area. It is known that the average ILP within a basic block is very small (around 2-3) and it is not much within a set of dynamically successive basic blocks as well. Hence, one can expect not to lose much by restricting the issue bandwidth as long as the trace line holds 3-4 basic blocks. (Table 4.1 shows a trace line of 16 instructions will typically hold 4 basic blocks.) For larger trace line size, the compiler can perform scheduling for the basic blocks so that the dependent instructions appear in order, and the trace line can be treated as a queue with instructions being examined and issued from the queue in order.

3.3 Register File Decomposition

As the instruction window size increases, the number of physical registers required also increases. Each instruction in the window (except for branch and store instructions) needs a physical register as destination, increasing register file size and making register file access a bottleneck (Section 1.2.3). Table 4.2 shows branch and store instructions constitute 30% of the total dynamic instructions i.e. a window of 256 instructions will require around 180 physical registers on an average.

However, most results are expected to be consumed locally within the trace line. Tables 3.1 through 3.9 indicate the number of live-on-entry, live-on-exit and local registers for different choices of trace line size for the SpecInt92 benchmarks and IBM RS/6000 instruction set architecture. (The floating point registers were not considered since compress, eqntott, and espresso do not use floating point registers and cc1 and xlisp use them rarely.) The live-on-entry and live-on-exit registers contain a single reference to an architectural register, whereas, local registers may contain more than one reference to the same architectural register. For a trace line of 24 instruction, the number of local registers is half of the total destination registers and this proportion increases considerably for increased trace line size. [2] [3] So, the physical register file can be partitioned into a global register file that takes care of the live-on-entry and live-on-exit registers, and local register files, one for each trace line that contains the local registers. Two bits are tagged with each operand to indicate whether the physical register is local, live-on-entry or live-on-exit. Figure 3.3 shows a renamed instruction augmented by local/live-on-entry/live-on-exit bit for each operand. [4] This partitioning decreases the global physical register file size by half for a trace line of 32 instruction. The local physical register file is small, being equal to the number of instructions in a trace line at most, and is cleared when the corresponding trace line is removed from the window.

---

[2] The numbers reported in [25] for trace line size of 8, 16 and 32 instructions are different because of an averaging error.

[3] Maximum number of live-on-exit registers can be greater than the trace line size because of load update instruction that has two destinations.

[4] The description in [25] used only local/global bit. But it should be augmented with live-on-entry/live-on-exit information to maintain correct data dependency during reuse of register renaming.

Benchmark	Trace-Line Size(Instructions)									
	8		16		24		32		64	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
cc1	3.79	13	5.19	18	5.99	20	6.55	21	7.83	22
compress	4.95	11	6.80	12	7.90	14	8.62	16	10.21	20
eqntott	3.43	12	4.10	13	4.38	16	4.65	17	5.54	19
espresso	4.14	10	5.68	13	6.90	15	7.72	16	8.39	19
xlisp	3.65	9	4.81	12	5.47	14	5.89	14	6.86	17

Table 3.1: Live-on-entry integer registers for different trace line sizes

Benchmark	Trace-Line Size(Instructions)									
	8		16		24		32		64	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
cc1	3.52	9	5.56	16	6.95	20	8.08	25	11.07	29
compress	3.55	8	5.66	14	7.05	16	7.97	18	10.09	24
eqntott	3.87	8	4.73	15	5.03	19	5.28	18	6.15	24
espresso	4.22	9	5.77	14	6.64	16	7.24	22	8.01	27
xlisp	3.46	7	5.38	11	6.66	14	7.60	15	9.59	17

Table 3.2: Live-on-exit integer registers for different trace line sizes

Benchmark	Trace-Line Size(Instructions)									
	8		16		24		32		64	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
cc1	1.09	7	3.66	14	6.87	21	10.35	30	25.80	62
compress	1.06	4	3.56	7	6.79	10	10.47	16	26.79	33
eqntott	1.26	5	5.52	13	10.34	21	15.22	29	34.86	61
espresso	1.79	6	6.24	15	11.37	24	16.78	34	40.04	62
xlisp	0.87	5	3.29	9	6.34	15	9.74	20	25.08	43

Table 3.3: Local integer registers for different trace line sizes

Benchmark	Trace-Line Size(Instructions)									
	8		16		24		32		64	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
cc1	0.68	2	0.82	2	0.87	2	0.88	2	0.91	2
compress	0.64	2	0.74	2	0.74	2	0.74	2	0.74	2
eqntott	0.63	2	0.67	2	0.67	2	0.68	2	0.70	2
espresso	1.10	2	1.30	2	1.33	2	1.33	2	1.34	2
xlisp	0.65	2	0.80	2	0.86	2	0.88	2	0.89	2

Table 3.4: One bit conditional live-on-entry registers for different trace line sizes

Benchmark	Trace-Line Size(Instructions)									
	8		16		24		32		64	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
cc1	0.61	2	0.98	2	1.23	2	1.39	2	1.74	2
compress	0.92	2	1.40	2	1.61	2	1.74	2	1.85	2
eqntott	1.13	2	1.23	2	1.38	2	1.44	2	1.59	2
espresso	0.55	2	1.01	2	1.42	2	1.73	2	1.89	2
xlisp	0.61	2	0.98	2	1.23	2	1.39	2	1.74	2

Table 3.5: One bit conditional live-on-exit registers for different trace line sizes

Benchmark	Trace-Line Size(Instructions)									
	8		16		24		32		64	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
cc1	0.29	6	1.02	9	1.95	15	2.95	19	7.20	34
compress	0.03	1	0.50	3	1.25	5	2.06	7	5.77	15
eqntott	0.40	4	1.77	7	3.22	9	4.69	10	10.68	19
espresso	0.02	3	0.13	6	0.29	6	0.56	8	2.68	15
xlisp	0.10	4	0.43	6	0.90	7	1.45	9	3.94	13

Table 3.6: One bit conditional local registers for different trace line sizes

Benchmark	Trace-Line Size(Instructions)									
	8		16		24		32		64	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
cc1	4.48	13	6.02	18	6.87	20	7.45	22	8.78	24
compress	5.59	12	7.54	13	8.64	15	9.36	17	10.95	21
eqntott	4.06	12	4.77	13	5.06	16	5.32	17	6.25	19
espresso	5.24	11	6.98	13	8.23	16	9.05	16	9.73	20
xlisp	4.30	9	5.62	17	6.34	22	6.79	25	7.79	27

Table 3.7: Live-on-entry registers(integer and one bit conditional) for different trace line sizes

Benchmark	Trace-Line Size(Instructions)									
	8		16		24		32		64	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
cc1	4.35	9	6.77	16	8.36	21	9.59	25	12.79	31
compress	4.47	8	7.06	15	8.66	18	9.72	20	11.94	26
eqntott	5.00	9	6.03	16	6.41	20	6.73	20	7.74	26
espresso	4.77	9	6.78	14	8.07	17	8.97	23	9.90	29
xlisp	4.08	8	6.38	16	7.91	22	9.02	26	11.39	31

Table 3.8: Live-on-exit registers (integer and one bit conditional) for different trace line sizes

Benchmark	Trace-Line Size(Instructions)									
	8		16		24		32		64	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
cc1	1.38	7	4.68	14	8.82	21	13.31	30	33.01	62
compress	1.09	4	4.06	9	8.03	12	12.53	18	32.56	38
eqntott	1.66	5	7.29	13	13.57	21	19.91	29	45.53	61
espresso	1.80	6	6.38	15	11.67	24	17.34	34	42.72	62
xlisp	0.97	5	3.72	9	7.24	15	11.18	20	29.01	43

Table 3.9: Local registers (integer and one bit conditional) for different trace line sizes

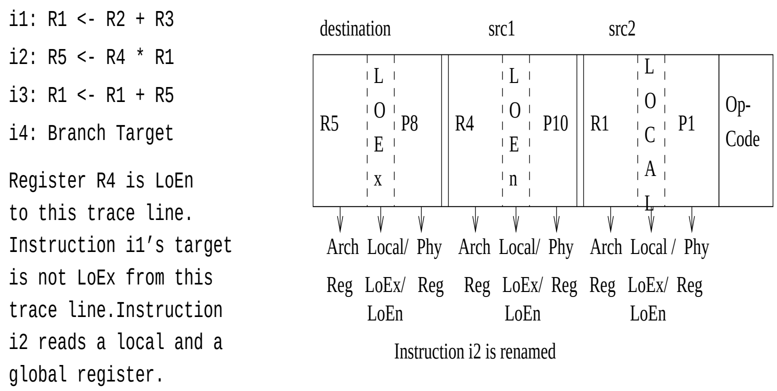

Figure 3.3: Renaming of Registers into Local and Global Register File

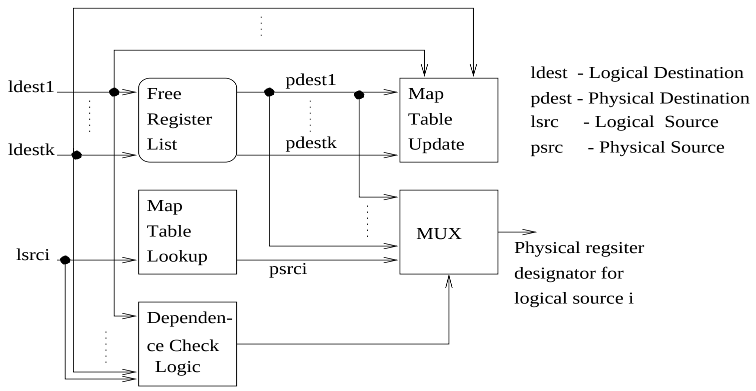

Figure 3.4: Traditional Register Rename Logic

3.3.1 Register Renaming

Register renaming mechanism has to be modified to take care of the partitioning of the register file. In traditional register renaming, each source operand accesses a map table with the architectural register as the index. The map table holds the current architectural to physical register mapping. In addition, each destination register picks up a physical register from the free list. In parallel, the dependence check logic detects true dependencies amongst the instructions in the current rename group and its output controls the MUX so that appropriate physical register is chosen (see Figure 3.4). [5] At the end, the map table is modified to reflect the recent mappings as produced by the current instructions.

[5] Refer [13]

To partition the register file, it is necessary to tag each operand with local/live-on-entry/live-on-exit bit. This is done by the output dependence check logic that operates in parallel with the true dependence check logic. Each architectural destination picks up a local physical register identifier (which can be equal to the instruction position in the trace line) as well as a global register. Depending on the output of the output-dependence check logic, either the local or the global register is selected using a 2-to-1 multiplexer. (see Figure 3.5) If the register is not live-on-exit, the global physical register is returned to the free list. The mapping of the architectural source registers now depends on the previous map table, true-dependence check logic as well as output-dependence check logic. In parallel, the operands are tagged with local/live-on-entry/live-on-exit bit using the output of true-dependence check and ouput-dependence check logic. This modification introduces the delay of the 2-to-1 multiplexer. However, that will not reside on the critical path, since the delay of looking up the map table is much larger than the dependence check logic delay [13].

3.3.2 Register Renaming Bandwidth

Currently, processor technology can achieve a register rename bandwidth of 4 instructions per cycle which implies a trace line containing 16 instructions can be renamed and dispatched at the rate of 4 instructions per cycle [28] [13]. However, register rename logic can not be pipelined. A destination register which is live-on-exit within a 4-instruction chunk might become local to the 16-instruction chunk. So, all the register rename logic outputs have to be buffered till the last chunk of instructions for that trace line gets renamed. This involves determining true dependence as done in traditional machines, as well as output dependence checking of the previously renamed instructions with the instructions currently being renamed within the same rename group. If there is an output dependence, the mapping of the previous instruction has to be changed, i.e., it should return the global physical register and use the local physical register instead. Now that register renaming stage can not be pipelined, poor register renaming bandwidth might

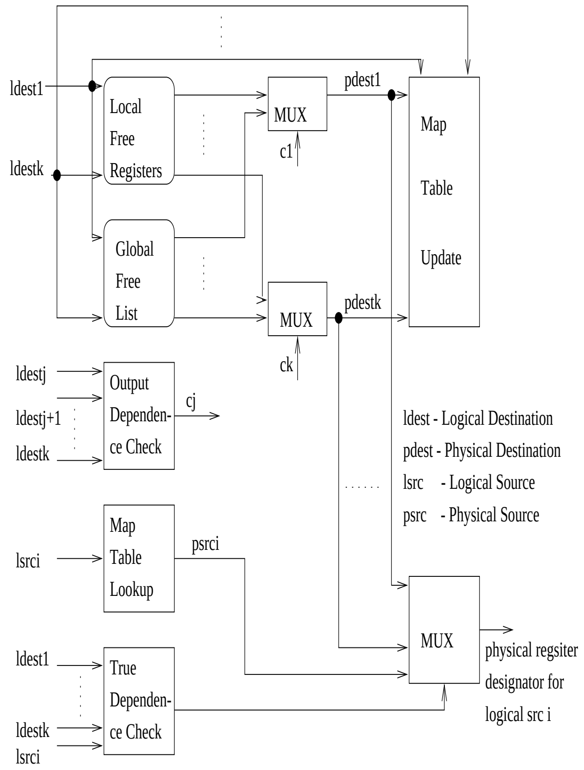

Figure 3.5: Modified Register Rename Logic

cause serious performance impediment. In the next section, a mechanism to alleviate this problem is discussed.

3.4 Reusing Register Renaming

In the trace window architecture, it is not necessary to perform full dependence checking and renaming of registers if the register renaming can be remembered. In this section, a mechanism to reuse register renaming is discussed. The idea is quite simple. After a trace has been renamed, the renaming is remembered in the trace cache. When this trace line is encountered again, the previous renaming can be used to considerably decrease the burden on the register rename logic. The trace line also stores the live-on-entry and live-on-exit registers for that line. In addition, the decoded opcode is also stored in the trace cache so that the decoding logic can be bypassed when the trace line is reused. Figure 3.6 illustrates a trace line of the trace cache with the decoded opcodes and the

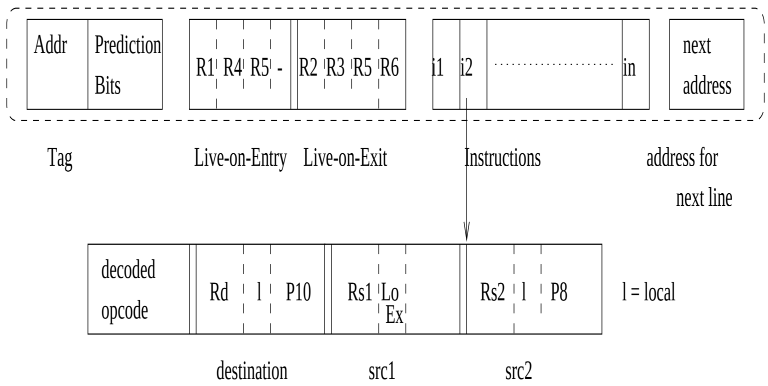

Figure 3.6: A Single Trace Line of the Modified Trace Cache

register renaming information.

Mapping is required only for the live-on-exit and live-on-entry registers. Each live-on-entry register picks up the current mapping from the map table. All operands with the live-on-entry bit set copy the old mappings in parallel. Each live-on-exit register picks up a physical register from the free register list. The operands with the live-on-exit bit set copy the new mapping in parallel. The rest of the operands with local bit set can reuse the previous mapping as stored in trace cache (refer Figure 3.7).

This strategy completely eliminates the dependence check logic. Map table lookup is also simplified with number of read ports required for the map table equal to the live-on-entry registers and number of write ports equal to the live-on-exit registers. From Tables 3.7 and 3.8, a trace line with 16 instructions will require around 7 table look-ups and 7 table updates, and this becomes around 12 table look-ups and 12 table updates for a trace line of 64 instructions. To rename 4 instructions, we require around 8 table lookups and 4 table updates. So, 16 instructions can be renamed in a single cycle using this approach. Also, it is clear that it will scale well with increased fetch bandwidth.

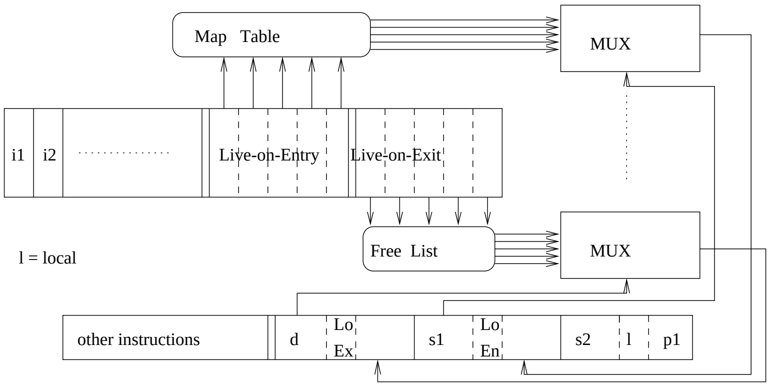

Figure 3.7: Single Cycle Renaming of a Trace Line

3.5 Trace-Environment Cache

Apart from the instructions, the trace cache should now hold the decoded opcodes, live-on-entry/live-on-exit registers and the register mapping for each opcode (Figure 3.6). This increases the size of the trace cache, and given the poor hit rate for the trace cache, it might have some impact on performance. This problem is alleviated to a great extent by storing the live-on-exit and live-on-entry information, and the next address for the trace line in a separate cache, called the trace-environment cache. Both the trace cache and the trace-environment cache are looked up during the fetch stage. For an architecture with 32 integer registers and 32 floating point registers, 8 byte is sufficient to hold the live-on-exit/live-on-entry information (1 bit for each architectural register). This requires 8 byte trace line as opposed to 64 byte trace line for the trace cache with 16 instructions per trace line. So, we can expect close to 100% hit rate for the trace-environment cache.

3.6 Pipeline Stages of Trace Window Architecture

In the following discussion, it is assumed that a trace line that hits in the trace cache will also hit in the trace-environment cache. Figure 3.8 illustrates the different pipeline stages when a trace hits in the trace cache. The decoding stage can be bypassed, and all the instructions of the trace line can be renamed in a single cycle.

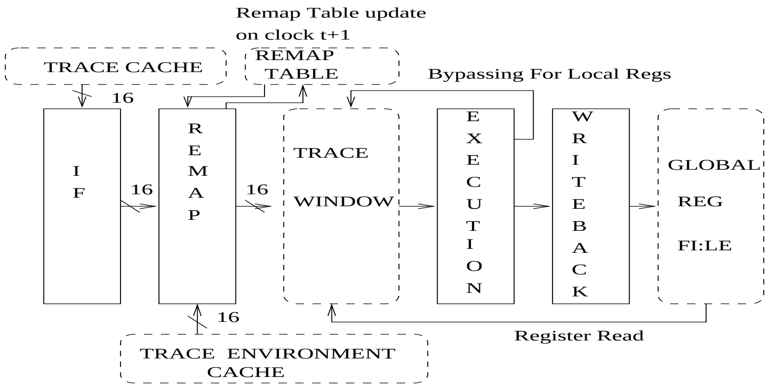

Figure 3.8: Pipeline Stages under Trace Cache Hit

A trace line can miss in the trace cache, but hit in the trace-environment cache. This involves fetching instructions from the instruction cache, followed by decoding, register renaming, and dispatching to the trace line, and parallel trace cache filling. However, the next address field of the trace-environment cache can be used to fetch the next trace line from the trace cache when the previous miss is being serviced. The live-on-exit and live-on-entry tags are used to update the map table in a single clock and the new mappings are remembered in a shadow local structure so that the global operands of the trace line after decode pick up the mapping from this shadow structure, instead of the global map table. This facilitates the renaming of the instructions from subsequent trace lines while this line is filled up from the instruction cache, since the dependencies across the trace lines are maintained with renaming the live-on-exit registers. Figure 3.9 illustrates the above scenario with 16-instruction trace line and rename bandwidth of 4-instruction per cycle.

When a trace line misses in the trace-environment cache as well, the fetching should be stalled till the previous trace line is fully fetched from the instruction cache (refer Figure 3.10).

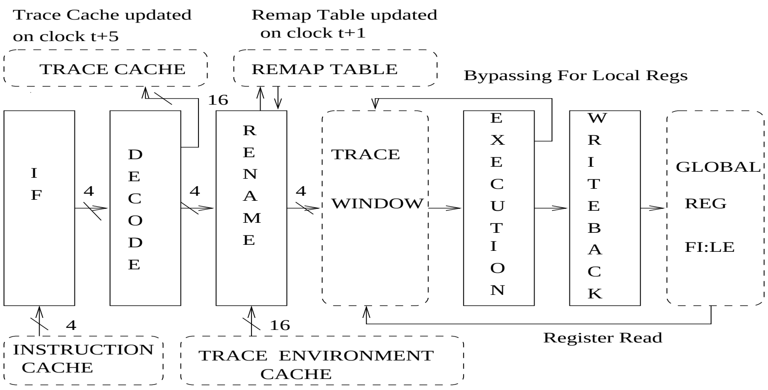

Figure 3.9: Pipeline Stages under Trace-Environment Cache Hit and Trace Cache Miss

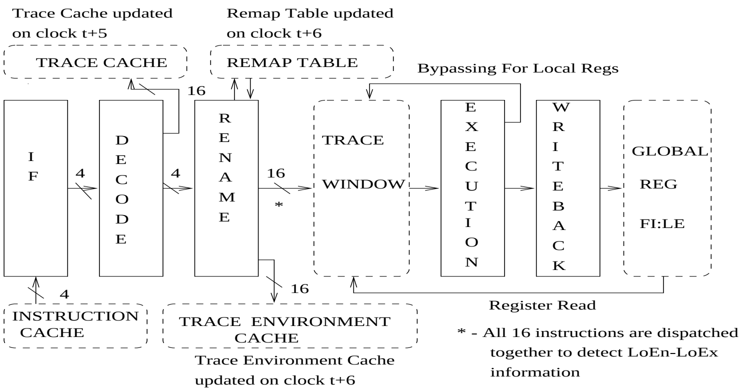

Figure 3.10: Pipeline Stages under Trace Cache as well as Trace-Environment Cache Miss

3.7 Branch Prediction Bandwidth

Several extensions to 2-level adaptive branch predictors have been proposed to allow the prediction of more than one branch per cycle to achieve high prediction bandwidth. The 2-level adaptive block predictor [4] can be used for the trace window architecture to obtain multiple predictions quickly. A trace line fetched from the trace cache provides the address of the next trace line in dynamic program order. This address is used to index into a pattern history table, which provides the branch predictions to be tagged on to the next address prior to looking up the trace cache in the next cycle. Each entry in the pattern history table is a simple fixed-size bit pattern holding prediction bits, plus a mask indicating the number of valid prediction bits. Since the trace cache provides the next address for each trace line, the pattern history table can be augmented to provide an alternate address usable when the last prediction bit returned by the table resolves as incorrect. This can speedup recovery from misprediction of loop terminating branches and will be useful for dynamic vectorization.

Chapter 4

Performance Evaluation

We evaluated the performance of trace window architecture using trace driven simulation. In this chapter, we first describe the tools and benchmarks used by simulation, followed by an overview of the machine models and machine parameters. Finally, we present the results of performance evaluation.

4.1 Tools and Benchmarks

Five integer intensive benchmarks from SPEC92 suite were used to evaluate the performance of trace window architecture. This is because we want to investigate performance for general-purpose codes. The benchmarks were compiled on IBM RISC System/6000 workstation running operating system AIX version 4.1 using XL C compiler (version 3.1) with optimization flags as distributed in the SPEC benchmark makefiles (-O2). Compiler effects are discussed in section 4.5. RS/6000 instruction traces were generated using the XPROF utility [12] and then fed into the trace-driven simulator that we have developed. Data and trace caches were simulated using Dinero [10] as a procedure call from the simulator.

Table 4.1 shows the benchmarks, their inputs, dynamic instruction count and average

Benchmark	Input	Total Instructions	Simulated Instructions	Simulated Basic Blocks	Avg. Basic Block Size
cc1	stmt.i	105,097,984	105,097,984	25,174,274	4.17
compress	in	73,304,795	73,304,795	18,205,655	4.03
eqntott	int-pri-3.eqn	489,618,166	100,000,001	46,191,044	2.16
espresso	bca.in	201,373,798	100,000,010	32,121,551	3.11
xlisp	8 queens	581,274,087	100,000,003	23,262,451	4.30

Table 4.1: Benchmark characteristics

Benchmark	Add/ Sub	Mult/ Div	Logical	Load	Store	Cond. Branch	Uncondn Branch
cc1	36.9	0.1	9.1	22.0	10.7	18.0	3.3
compress	36.8	0.0	11.0	21.8	10.3	16.6	3.6
eqntott	33.9	0.0	2.7	28.4	3.7	30.5	3.3
espresso	32.7	0.0	13.8	23.2	2.4	22.6	5.3
xlisp	35.2	0.0	2.4	25.7	15.6	16.0	5.1

Table 4.2: Percentage of different instruction classes

basic block size. Except for compress, which had only 73 million instructions, all benchmarks were simulated for 100 million instructions. The instruction mix for the different benchmarks is shown in Table 4.2.

4.2 Machine Models

To perform a thorough evaluation of the trace window architecture, the following three broad categories of machine models were simulated:

1. TW (Trace Window model) - This is a representative model of the trace window architecture.

2. IF (Ideal Fetch model) - This is similar to TW model except for the fact that the fetch stage is ideal, i.e., branch prediction is perfect, and there is no trace cache miss or trace environment cache miss. This model gives an upper bound for the performance of TW model.

3. SUPER (SUPERscalar model) - This is representative of traditional superscalar processor with a single, large instruction window.

Apart from this, different variations of each model were simulated to evaluate the performance gain/loss due to each of the factors that can have some impact on the instruction level parallelism (for example limited issue from each trace line, reuse of register renaming etc.).

4.2.1 TW Model

The trace window for the TW model contains 16 trace lines each containing 16 instructions. This gives an instruction window of 256 instructions. A trace line can contain unlimited number of branches. On a trace cache hit, the entire trace line can be dispatched to the window in one clock. On a trace cache miss, we assume an instruction fetch and register renaming bandwidth of 4 instructions per clock. However, if the trace line hits in the trace environment cache, the subsequent trace line can be fetched from the trace cache while the previous miss is being entertained (see Section 3.6). If the trace line is a miss in the trace environment cache as well, then the instruction fetch from the trace cache should be stalled till the last instruction of the trace line is fetched (refer Section 3.6). An instruction can be considered for issue only in the clock subsequent to its entry in the instruction window. Issue is limited to 2 instructions per clock from each trace line i.e. peak issue bandwidth is 32, which is double the fetch bandwidth to avoid saturation of the trace window. Result bypassing is limited to within each trace line. Instructions with global destination register need an additional write back stage to write to the global register file, i.e., an instruction dependent on this result can be issued only one clock after the instruction completes execution.

4.2.2 IF Model

The IF model is similar to the TW model except that it assumes 100% hit rate for the trace cache and 100% branch prediction accuracy. This will give an upper bound on the performance of the TW model under ideal fetch condition.

4.2.3 SUPER Model

The SUPER model is the traditional superscalar processor with a large instruction window. This will compare the performance of TW model with the traditional approaches. The cost of large instruction window and register file is captured by the fact that no result bypassing is possible, i.e., an instruction dependent on a previous instruction can start execution only 1 clock after that instruction has completed. We assume only 32 instructions can be issued on each clock from the window of instructions. This limit has been imposed to compare the performance of TW model with the traditional superscalar model on a level ground. However, it is difficult to select and issue 32 instructions from a window of 256 instructions in one clock. This will definitely slow down the clock [13] [9]. This delay of the issue logic cannot be modeled by our SUPER model. Hence the performance of the SUPER model is optimistic i.e. it will be difficult to achieve in practice. We assume fetch bandwidth of 16 instructions per clock, and register renaming bandwidth of 4 instructions per clock. For a trace cache miss instructions can be fetched from the instruction cache at the rate of 4 per cycle.

4.3 Machine Parameters

In this section only the machine parameters that are common across the different machine models are discussed.

All the machine models use a 4-way associative trace cache with 256 entries and LRU replacement policy. Each trace line contains 16 instructions. The way we lookup the trace cache is somewhat different from what was originally proposed by Rotenberg *et. al.* [14]. They indexed the trace cache with the starting address and included branch predictions in the tag match. However, we concatenate the branch predictions with the starting address in the following manner:

The first branch prediction forms the least significant bit of the index. The least significant 5 bits of the starting address are interchanged with the 5 higher order bits of the address,(For RS/6000 and the benchmarks we have simulated 20 least significant bits are enough to distinguish all instruction addresses) and this forms the next higher order bits for the index. This interchange is done to take care of possibly long sequential paths through the control flow graph till a branch is predicted taken. In this scenario, each starting address will have the same 4 least significant bits (addresses are differing by 16), and will conflict in the trace cache. Finally, the other branch predictions are concatenated(up to a maximum of 7 branch predictions). When the number of branch predictions exceeds 7, we simply ignore the other predictions. This will have negligible impact on the performance as well as correctness of trace cache lookup since only a few lines will have more than 7 branches, given the average basic block size of around 4 instructions. The trace environment cache is assumed to be infinite, i.e., there can be only compulsory misses for this cache. We assume a perfect instruction cache.

There are unlimited execution units and memory ports, i.e., there are no structural hazards. This is done to remove the effect of limited hardware resources on instruction window size and thus highlight the impact of pipeline stage latencies. We also assume sufficient result buses and register read/write port for each model. Table 4.3 shows the functional unit latencies assumed for all the models. The data cache is 64KB 4-way set associative (except for espresso that uses a 16KB cache) with 32 byte line size and LRU replacement policy. Espresso has a much smaller working set[2] compared to other benchmarks. We assumed perfect memory disambiguation for all the models.

A 2-level adaptive GAg(18) [29] [30] branch predictor with adequate prediction bandwidth is simulated instead of the prediction scheme described in Section 3.7. The block-level predictor [4] is known to be as accurate as the GAg predictor. Recovery from a mispredicted branch requires one clock i.e. instruction fetch should be stalled for one clock after the branch resolution.

4.4 Results

Table 4.4 shows the performance of different machine models. TW model achieves a

Integer Functional Unit	Latency (cycles)	Floating Point Functional Unit	Latency (cycles)
Integer Add	1	FP Add	3
Integer Mult	4	FP Mult	4
Integer Div	8	FP Div	8
Load (Cache hit)	2		
Load (Cache miss)	12		
Store	1		
Conditional & Uncnd. Branch	1		
Logic	1		

Table 4.3: Functional unit and data cache latencies

Benchmark	Machine Model		
	IF (Peak ILP 16) ILP(Speedup)	TW (Peak ILP 16) ILP(Speedup)	SUPER (Peak ILP 4) ILP(Speedup)
cc1	10.80(4.91)	4.40(2.00)	2.20(1.00)
compress	7.93(4.45)	2.25(1.26)	1.78(1.00)
eqntott	11.31(4.73)	3.65(1.53)	2.39(1.00)
espresso	14.96(5.91)	7.42(2.93)	2.53(1.00)
xlisp	11.91(4.88)	6.17(2.53)	2.44(1.00)

Table 4.4: Performance of different machine models

Benchmark	Conditional Branches	Prediction Accuracy	Avg. Prediction Bandwidth (pred./cycle)	Max. Prediction Bandwidth (pred./cycle)
cc1	18881119	94.00	2.67	11
compress	12170752	90.61	2.39	7
eqntott	30509524	96.47	4.57	8
espresso	22562867	97.57	3.47	12
xlisp	15979557	96.82	2.47	7

Table 4.5: Branch prediction accuracy & bandwidth(TW model)

speedup of 1.26 to 2.93 for different benchmarks over the SUPER model i.e. the traditional superscalar processor. The speedup is quite impressive for cc1, espresso and xlisp, while there is around 25% to 50% improvement for compress and eqntott respectively. There is a significant gap between the performance of the IF model and the TW model. In particular, for compress, IF model achieves a speedup of 4.45 over SUPER, whereas for TW model the speedup is just 1.26. The ILP sustained for compress for IF model is significantly smaller than other benchmarks. Compress spends almost all of its time in a single big loop with complex control flow. This loop is bound by recurrences (getting the index into the hash table) that results in a long critical path through the entire program. In the rest of this section, we look at the different machine models carefully.

4.4.1 Instruction Fetch Efficiency

The significant difference in ILP between the IF and the TW model indicates the major impact of instruction fetch mechanism on performance. There are two factors that cause this difference in ILP: (i) branch prediction accuracy, (ii) trace cache hit rate

The branch prediction accuracy is quite high for all the benchmarks except for compress which has a prediction accuracy of only 90.6 percent (refer Table 4.5). This might explain the significant gap in speedup between TW and IF model for compress. Table 4.5 also shows that the average branch prediction bandwidth required for the benchmarks

Benchmark	Perfech Branch Pred.		Realistic Branch Pred.	
	Total Reference	Miss Rate (%)	Total Reference	Miss Rate (%)
cc1	6568609	68.34	20927106	54.93
compress	4581550	47.06	31285668	56.19
eqntott	6250001	50.53	25527152	21.92
espresso	6250001	51.77	12445956	46.70
xlisp	6250001	62.19	14584104	32.42

Table 4.6: Trace cache performance

per clock is 3 to 4. This prediction bandwidth can be easily achieved using the currently available multiple branch predictors [4]. The maximum prediction bandwidth required for three of the benchmarks is at most 8 which justifies our assumption that ignoring the last few predictions (when prediction bandwidth exceeds 7) for trace cache lookup will have negligible impact on the correctness of the trace cache performance.

Table 4.6 shows the trace cache performance for perfect and realistic branch predictions. It can be noted here that the trace cache miss rate for perfect branch prediction is much more than realistic branch prediction for all the benchmarks except compress. This is because when a branch is mispredicted, instructions are fetched along the wrong path. However, for the branches along the mispredicted path the outcome is never known. To simplify the job of the two level branch predictor under these conditions, we predict all the branches along the mispredicted path as not-taken. Hence the trace cache miss rate for realistic branch prediction cannot perfectly model the interference from the wrong path. This implies that the effective miss rate for the trace cache for TW model will actually be close to that of perfect branch prediction. This high trace cache miss rate might translate into the significant performance difference of IF and TW model.

To isolate the impact of branch prediction accuracy and trace cache hit rate on performance, we simulated two more machine models:

Benchmark	Machine Model			
	IF (Peak ILP 16) ILP(Speedup)	Oracle TW (Peak ILP 16) ILP(Speedup)	Perfect Trace Cache TW (Peak ILP 16) ILP(Speedup)	TW (Peak ILP 16) ILP(Speedup)
cc1	10.80(2.45)	9.74(2.21)	4.86(1.10)	4.40(1.00)
compress	7.93(3.52)	7.52(3.34)	2.52(1.12)	2.25(1.00)
eqntott	11.31(3.10)	10.44(2.86)	3.79(1.04)	3.65(1.00)
espresso	14.96(2.02)	13.97(1.88)	7.98(1.07)	7.42(1.00)
xlisp	11.91(1.93)	11.12(1.80)	6.50(1.05)	6.17(1.00)

Table 4.7: Effect of branch prediction and trace cache hit rate on performance

1. Oracle TW - This is same as TW model, but with 100% branch prediction accuracy,
2. Perfect Trace Cache TW - This is TW model with a perfect trace cache but with dynamic branch prediction.

Table 4.7 shows the ILP sustained with these models. It is clear from the table that trace cache performance hardly has any impact on ILP. This is because of the sustained fetch bandwidth made possible by the trace-environment cache under a trace cache miss. Branch prediction accuracy is causing all the performance loss of the TW model compared to IF model. This might seem to be surprising given the prediction accuracy as high as 97.5% for espresso and around 96-97% for all the other benchmarks except compress. The limited issue column of Table 4.10 shows the average lifetime of mispredicted branches for TW model. This lifetime is quite high for compress and eqntott, and these are the two benchmarks with the smallest speedup for TW model. It shows the significant penalty for the few mispredicted branches. This penalty can be reduced by software or hardware scheduling for the trace line that gives higher priority for the conditional branches over other instructions. Also, some mechanism might be developed that will squash only the control and data dependent instruction following the resolution of a mispredicted conditional branch.

Table 4.8 shows the contribution of different factors that cause loss of fetch cycles.

Benchmark	Percentage of Fetch Cycles Wasted		
	Branch Misprediction	Unavailability of Sub-Window	Trace-environment Cache Miss
cc1	63.00	6.00	2.00
compress	84.00	0.00	0.00
eqntott	73.00	3.00	0.00
espresso	48.00	3.00	0.00
xlisp	53.00	7.00	0.00

Table 4.8: Fetch engine efficiency(TW model)

Branch misprediction has the highest penalty in terms of fetch clock. More than 50% of the instruction fetch cycles are wasted because of misprediction for all the benchmarks. For compress and eqntott it is as high as 84% and 73% respectively. The instruction fetch cycle is stalled rarely because of trace window saturation, and trace-environment cache compulsory miss rate is too low to cause any significant performance impact.

4.4.2 Limited Issue Bandwidth

In this section we carefully look at the performance impact of limited issue bandwidth of each trace line. For this we evaluate the performance of two more machine models:

1. Super IF - This is similar to IF model, but with unlimited issue bandwidth of each trace line(i.e. at most 16) as opposed to 2 per trace line,

2. Super TW - This is same as TW model, but with unlimited issue bandwidth for each trace line.

Table 4.9 shows the speedup obtained with the unlimited issue bandwidth models. The speedup is not significant for super IF model over IF model. The maximum speedup is 1.31 for xlisp. Similar trend is followed for TW model. However, there is a speedup of 1.51 for eqntott. This indicates that eqntott has more ILP within a trace line compared to other benchmarks, and hence, restricting the issue bandwidth to 2 per trace line causes

Benchmark	Machine Model			
	Super IF ILP(Speedup)	IF ILP(Speedup)	Super TW ILP(Speedup)	TW ILP(Speedup)
cc1	13.21(1.22)	10.80(1.00)	5.20(1.18)	4.40(1.00)
compress	9.08(1.14)	7.93(1.00)	2.44(1.08)	2.25(1.00)
eqntott	14.04(1.24)	11.31(1.00)	5.58(1.53)	3.65(1.00)
espresso	15.81(1.06)	14.96(1.00)	8.77(1.18)	7.42(1.00)
xlisp	15.67(1.31)	11.91(1.00)	7.63(1.24)	6.17(1.00)

Table 4.9: Effect of limited issue from each trace line on performance

Benchmark	Unlimited Issue			Limited Issue		
	Average speculation depth (cond. br.)	Average lifetime of mis-predicted branch (cycles)	cycles lost for branch mis-prediction (%)	Average speculation depth (cond. br.)	Average lifetime of mis-predicted branch (cycles)	cycles lost for branch mis-prediction (%)
cc1	12.9	5.4	59.9	13.1	7.9	63.0
compress	9.3	17.4	82.7	10.0	19.6	84.0
eqntott	18.5	5.6	59.4	21.6	14.2	73.0
espresso	20.7	3.7	41.8	23.5	6.8	48.0
xlisp	15.9	7.8	49.2	16.8	12.1	53.0

Table 4.10: Effect of limited issue on branch prediction and resolution

a significant performance loss for this program. Overall the speedup is not impressive enough to justify the unlimited issue at the cost of cycle time penalty.

Table 4.10 shows the effect of unlimited issue on branch prediction and resolution. The average speculation depth of conditional branches (i.e. the number of outstanding conditional branches per clock) does not change significantly from one model to the other. Similarly, the average lifetime of mispredicted branches also remains almost the same except for eqntott where the lifetime increases from 5.6 to 14.2, and thus results in a large misprediction penalty and poor ILP for TW model. For other benchmarks the branch misprediction penalty increases slightly from one model to the other.

Benchmark	Unlimited Issue			Limited Issue		
	Avg. number of sub windows	Avg. lifetime of sub window (cycles)	Window saturation stall/useful clocks	Avg. number of sub windows	Avg. lifetime of sub window (cycles)	Window saturation stall/useful clocks
cc1	5.0	14.3	0.08	5.9	19.8	0.20
compress	2.9	17.2	0.02	3.2	20.6	0.03
eqntott	5.1	13.5	0.08	5.4	21.7	0.12
espresso	6.1	10.8	0.01	7.7	15.9	0.07
xlisp	6.1	12.3	0.02	7.6	18.9	0.17

Table 4.11: Effect of limited issue on trace window

Next, we look at the effect of unlimited issue on the trace window since unlimited issue might result in better outflow from the instruction window resulting in less saturation of the window. However, Table 4.11 indicates that the average number of trace lines in the trace window remains almost same across the two models. As expected, average lifetime of a trace line in the trace window also increases insignificantly for all the benchmarks except eqntott. The table also shows the percentage of useful cycles lost because of trace window saturation, i.e., unavailability of trace line. With unlimited issue, this percentage is very small, while with limited issue, it is significant for at least three of the benchmarks. However, as indicated by the ILP, this window saturation has not been translated to have an overall effect on performance for most of the benchmarks (eqntott is an exception).

4.4.3 Reusing Register Renaming

In this section, we investigate the effect of reuse of register renaming on performance. To perform a thorough evaluation, we simulate two machine models:

1. TW-noTEC(TW with no Trace-Environment Cache) - This is TW model, but the trace-environment cache(TEC) is absent. The live-on-entry and live-on-exit information is stored in the trace cache itself. Hence, when a trace line misses in the

Benchmark	Machine Model			
	TW (Peak ILP 16) ILP(Speedup)	TW-noTEC (Peak ILP 16) ILP(Speedup)	TW-noRRR (Peak ILP 4) ILP(Speedup)	SUPER (Peak ILP 4) ILP(Speedup)
cc1	4.40(1.00)	2.58(0.59)	2.00(0.45)	2.20(0.50)
compress	2.25(1.00)	1.98(0.88)	1.59(0.71)	1.78(0.79)
eqntott	3.65(1.00)	2.88(0.79)	2.11(0.58)	2.39(0.65)
espresso	7.42(1.00)	3.72(0.50)	2.37(0.32)	2.53(0.34)
xlisp	6.17(1.00)	3.50(0.57)	2.32(0.38)	2.44(0.39)

Table 4.12: Effect of reuse of register renaming on performance

trace cache, instruction fetching should be stalled till the last instruction of the trace line is fetched and renamed. Thus it loses the advantage of continuing instruction fetch from the trace cache when the trace line hits in the TEC. This model thus serves to isolate the benefit of a separate trace-environment cache as opposed to a combined trace cache.

2. TW-noRRR(TW with no Reuse of Register Renaming) - This model assumes that the the TEC is not present and the trace cache does not record this information as well. Effectively this is TW model without the reuse of register renaming. The instructions should be renamed at the rate of 4 instructions per clock. This model thus brings out the effect of reuse of register renaming.

Table 4.12 shows the performance of these models. The first observation is that TW-noRRR model performs worse than the SUPER model for all the benchmarks. TW-noRRR model has the advantage of local bypassing. But it seems that this advantage is totally offset by the freedom to choose any 32 instructions from a large window of 256 instructions for the SUPER model. However, as discussed before(Section 4.2.3), it might not be possible to include the complicated issue logic of the SUPER model without increasing the clock cycle time. TW-noTEC model performs better than the TW-noRRR model, but it sustains an ILP which is almost same as the SUPER model except for espresso and xlisp. This is because TW-noTEC model has the advantage of local bypassing and reusing register renaming under trace cache hit. But under trace cache miss, instruction fetch from trace cache has to be stalled to perform the register renaming. This situation occurs quite often given the poor trace cache hit rate. SUPER model has the freedom to choose 32 instructions from a pool of 256 instructions that offsets the advantage of the TW-noTEC model for most of the benchmarks. It is clear from the table that the separate trace-environment cache is quite important to sustain the fetch bandwidth under trace cache miss, and combined trace cache is not sufficient to achieve a high ILP.

4.4.4 Performance of Trace Cache and Trace-Environment Cache

The hit rate of trace cache as well as the trace-environment cache(TEC) has significant impact on the performance of the TW model. In this section we look at the performance of these two structures. We generate the references for the two caches using the IF model. But the cache lookup policy is different from the one followed (Section 4.3) for the simulation of the different machine models. This has been done to have a trace cache lookup policy almost similar to the indexing policy used in [14] for performance evaluation using MIPS instruction set architecture, and evaluate the performance of similar indexing policy for IBM RS/6000 instruction set architecture. The least significant 20 bits of the starting address concatenated with maximum of 12 branch predictions is used to lookup the trace cache as well as the TEC. Hence, the cache is indexed using the starting address, and the branch predictions form the tag. Next, we simulate various configurations of trace cache and TEC using Cheetah [21], a cache simulator that can simulate multiple configurations in one pass.

Tables 4.13 through 4.17 show the miss rate of the trace cache under different configurations. In general, the hit rate of trace cache is very poor. It is worst for cc1 and xlisp, and somewhat better for eqntott, espresso, and compress. Also, it is clear that for small number of entries, increasing the associativity of the cache is not giving a better hit rate. For large number of entries, the hit rate improves slowly with associativity. Hence,

Total Entries	Associativity			
	1	2	3	4
64	87.31	86.33	86.10	-
128	83.85	83.48	83.23	83.24
256	80.24	79.32	79.06	78.96
512	75.57	73.39	72.39	71.95
1024	69.25	65.92	63.92	63.07
2048	62.00	56.44	53.63	52.37

Table 4.13: Trace cache miss rate for cc1

Total Entries	Associativity			
	1	2	3	4
64	88.51	87.09	87.29	-
128	81.84	79.66	76.82	76.67
256	66.64	63.81	70.41	59.13
512	47.95	40.43	36.73	34.32
1024	46.45	21.67	14.54	7.96
2048	32.64	21.04	7.58	2.47

Table 4.14: Trace cache miss rate for compress

it might be better to have a direct mapped trace cache than an associative one.

We simulated an infinite TEC with only compulsory misses on the assumption that a very small fully associative cache will be sufficient to hold all the unique trace lines. The miss rate for various configurations of the TEC is shown in Table 4.18. The unique reference column gives the compulsory miss rate. The goal is to achieve this miss rate with a small fully associative cache. Assuming each line in the trace-environment cache is 8 bytes, a 4K fully associative TEC achieves a miss rate that is close to the performance of infinite TEC. For cc1, however, number of unique references is too big to fit into a small fully associative cache.

Total Entries	Associativity			
	1	2	3	4
64	59.29	53.60	52.49	-
128	52.05	40.48	36.30	34.80
256	43.18	32.29	24.27	22.58
512	39.82	24.65	16.91	12.14
1024	37.96	22.34	12.59	7.45
2048	36.83	20.53	11.10	4.84

Table 4.15: Trace cache miss rate for eqntott

Total Entries	Associativity			
	1	2	3	4
64	68.22	70.85	75.41	-
128	51.64	51.60	48.28	49.10
256	47.93	40.02	26.95	20.17
512	45.63	37.09	20.21	14.54
1024	44.27	34.67	17.84	11.18
2048	43.19	33.10	15.37	9.00

Table 4.16: Trace cache miss rate for espresso

Total Entries	Associativity			
	1	2	3	4
64	94.60	93.89	95.33	-
128	88.70	88.62	89.10	89.39
256	83.02	81.79	81.40	81.48
512	73.26	71.15	70.58	70.28
1024	60.69	54.96	52.86	51.54
2048	51.80	37.02	30.61	29.06

Table 4.17: Trace cache miss rate for xlisp

Benchmark	Total Reference	Unique Reference(%)	Fully Associative Cache Size(Number of Entries) 1K (128)	2K (256)	4K (512)	8K (1024)	16K (2048)	32K (4096)
cc1	6568625	90478(1.38)	71.47	62.63	51.74	38.23	26.73	21.12
compress	4581550	634(0.01)	30.95	3.23	0.01	0.01	0.01	0.01
eqntott	6250001	4111(0.06)	10.37	2.41	1.18	0.35	0.06	0.06
espresso	6250001	9834(0.16)	13.44	10.34	8.14	5.64	3.02	0.70
xlisp	6250001	4954(0.08)	70.63	49.79	27.27	9.29	1.76	0.08

Table 4.18: Trace-environment cache miss rates

Benchmark	Machine Model 8SUPER (Peak ILP 8) ILP(Speedup)	bypassSUPER (Peak ILP 4) ILP(Speedup)	SUPER (Peak ILP 4) ILP(Speedup)
cc1	3.01(1.37)	2.24(1.02)	2.20(1.00)
compress	2.17(1.22)	1.86(1.04)	1.78(1.00)
eqntott	3.35(1.40)	2.43(1.02)	2.39(1.00)
espresso	3.66(1.45)	2.54(1.00)	2.53(1.00)
xlisp	3.54(1.44)	2.50(1.02)	2.44(1.00)

Table 4.19: Performance of different superscalar models

4.4.5 Effect of Result Bypassing on SUPER Model

In the SUPER model, we assume that results can not be bypassed i.e. one full clock is required to write the result to the register file. Table 4.19 shows that the absence of result bypassing does not have significant impact on performance. We simulated a model called bypassSUPER which is a superscalar processor similar to SUPER, but we assume that results can be bypassed without a separate register write stage. The ILP achieved by bypassSUPER is almost same as SUPER for all the benchmarks. This indicates that the considerable performance gap between TW and SUPER model comes from the extra register renaming bandwidth. We also consider an impossibly good superscalar model called 8SUPER to isolate the importance of register renaming bandwidth. This model is similar to SUPER in every respect except that it has twice the register renaming bandwidth of SUPER. 8SUPER consistently performs better than SUPER highlighting

Benchmark	Total References	Loads	Stores	Miss Rate
cc1	31224675	20027219	11197456	0.53
compress	20025081	12509431	7515650	12.10
eqntott	18454740	14799640	3655100	4.23
espresso	17205018	14824593	2380425	0.52
xlisp	36276871	20648341	15628530	0.01

Table 4.20: Data cache performance(TW model)

the effect of register renaming bandwidth. It performs almost as well as TW model for compress and eqntott, but performs well below TW for the other three programs.

4.4.6 Performance of Data Cache

Table 4.20 shows the miss rate of data cache for different benchmarks. The data cache miss rate is quite small for all the benchmarks except compress and eqntott. It can be recalled that compress and eqntott are the two benchmarks with poor ILP for the TW model.

Overall, the results show the benefits of the improved instruction rename and dispatch bandwidth provided by the proposed trace window scheme. Also, this architecture shifts the pipeline bottlenecks from instruction issue logic and register rename stage to the other sources of pipeline bubbles, like branch misprediction and data cache miss penalty.

4.5 Effect of Compilers and Instruction Set

So far we have ignored the compiler issue. Whereas our results are solely applicable to the IBM RS/6000 with XL C compiler, we expect to see the similar trend for other architecture and compiler since we believe, the results manifest the inherent nature of the benchmarks. However, depending on the specific instruction set architecture, specially the number of architectural registers and the register allocation method used by the compiler (for example, the number of live registers and the lifetime of a register), the number of live-on-exit, live-on-entry and local registers could change. But, this can not invalidate our results. For example, previous study [6] also indicated

1. a significant number of register instances are used up in the same basic block in which they were created, and

2. most of the remaining registers are used up in the subsequent basic block.

This implies the increase in local register number as the trace line size increases. We leave this as future work to evaluate the performance of trace window architecture for different instruction sets and compilers.

Chapter 5

Dynamic Vectorization

Vector instructions [15] simplify instruction fetch and issue logic, and reduce the demands on the instruction window and register files. This is because a vector instruction is fetched only once and multiple iterations of the same instruction are issued from that instruction using simple issue logic. Vector instructions also allow the overlapping of sequential code subsequent to a loop body with the execution of instructions from the loop body.

The limitation of vector instructions is that only regular loops can be vectorized by the compiler. Ordinary programs typically follow complex control flow paths within the loop body that might be difficult to predict. Also, there might be subroutine calls and unpredictable memory dependencies across loop iterations. These factors prevent the vectorization of the loops of ordinary programs. However, we have often found that even for statically complex loops, there are only a few simple control paths followed during the dynamic program execution. Often only one or two paths are frequently followed. If these control flow paths do not contain subroutine calls, a dynamic mechanism can detect and vectorize such loops, and thereby provide the performance benefit of static vectorization. A simple mechanism to perform dynamic vectorization in the framework of the trace window architecture has been simultaneously proposed in [25]. In this chapter this mechanism is described and the fraction of dynamic instructions that can be vectorized by this mechanism is presented.

5.1 Trace Line Vectorization

To perform dynamic vectorization, status bits are associated with each trace line, as well as each instruction (refer Figure 5.1). The default values of these status bits correspond to normal operation of the trace window, without any dynamic vectorization capability. The main idea is to detect a trace line that ends in a (predicted) taken branch to the beginning of itself. To facilitate this detection, the occurrence of a (predicted) taken backward branch terminates the current trace line, i.e., all the instruction slots of that trace line beyond the branch are marked invalid. This helps the target instruction of the backward branch to be placed as the first instruction of a new trace line. Each trace line in the trace window is tagged with its address (recall that the address consists of the starting address as well as the branch predictions) and the address of the immediate predecessor trace line in dynamic execution order. The instruction fetch stage is now modified as follows:

The trace window is looked up in parallel with the trace cache to detect whether the current trace line corresponds to the start of a dynamic loop. A hit occurs in the trace window if the address of the current trace line matches both with the address of a previous trace line in the trace window and its predecessor tag. The predecessor tag matching is required to ensure that the current trace line is a copy of a loop iteration. When a match occurs in the trace window, a copy of the trace line from the trace window is renamed (renaming is necessary only for the live-on-entry source operands) and then dispatched to the trace window with its VECTOR status bit set. (If the trace line is a miss in the trace window the trace line will be renamed and dispatched in the normal fashion.) This is to inform the issue logic that multiple iterations of this trace line have to be issued. The above mechanism achieves benefits similar to that of a vector instruction since the entire loop is dispatched in a single clock. It is not required to fetch, rename and dispatch subsequent iterations of the trace line. Hence, further instruction fetch can proceed along the fall-through path of the loop-terminating backward branch. If the loop really terminates with the loop-terminating backward branch being not taken (i.e. the loop is not exited from within the loop body), and the fall-through instructions are data

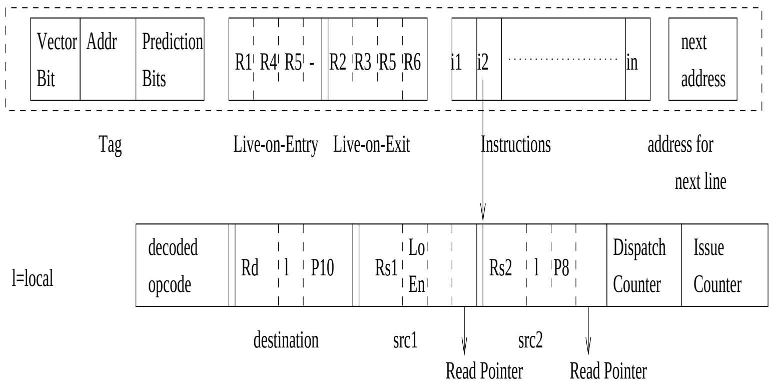

Figure 5.1: Status Bits Associated with a Trace Line to Perform Dynamic Vectorization

independent of the instructions constituting the loop body, then this overlapping of two portions of the code gives more instruction level parallelism.

5.2 Vector Renaming and Dispatch

There is a DISPATCH COUNTER associated with each vectorized trace line. This counter indicates the number of iterations of the loop to be executed and is initialized to a default maximum value (e.g. infinity). The misprediction of a branch instruction of the trace line terminates the loop by setting the DISPATCH COUNTER to a proper value. Renaming the registers for a trace line from the trace window is quite simple. Only the live-on-entry registers are to be renamed since all the iterations of the loop except the first one should read the values generated by the previous iteration. All the other operands use the same register renaming as that of the first iteration. All the instances of the vectorized instructions use the same physical register identifier by associating queue semantics with each physical register, as discussed later.

5.3 Vector Issue

Multiple instances of vectorized instructions are issued, one per clock. To keep track of the number of instances of an instruction being issued, a counter called ISSUE COUNTER is maintained with each instruction. An instance of an instruction can be issued only if the value of ISSUE COUNTER is less than the DISPATCH COUNTER, and the instruction is ready to execute. ISSUE COUNTER is incremented on each issue. An instruction issued from the trace window is always tagged with this ISSUE COUNTER to facilitate squashing after branch misprediction. Branch misprediction terminates the issue of further loop iterations. If the loop-terminating branch is resolved to be incorrect, the ISSUE COUNTER of the branch instruction will indicate the total number of iterations the loop is executed. The DISPATCH COUNTER is then set to that ISSUE COUNTER value preventing further iteration of loop from being executed. All the instructions in the pipeline with the ISSUE COUNTER tag value greater than the loop limit are squashed. If an internal branch is mispredicted, the DISPATCH COUNTER of the instructions preceding this branch is set to ISSUE COUNTER value and the instructions succeeding the branch are set to ISSUE COUNTER - 1.

5.4 Register Semantics for Vectorized Trace

Multiple instances of a vectorized instruction refer to the same physical register identifier. Correct data dependencies are maintained for these instructions by dynamically associating a register queue with each register used by a vectorized trace line. Hardware register queues are associated with the physical registers using a queue map table that maps physical register to a register queue. If the VECTOR status bit for a trace line is set, the physical register identifier is used to lookup the queue map table and find the register queue number. For the first instance of a vectorized instruction this mapping will be absent for a live-on-entry source operand. This maintains correct data dependency since the first instance of the live-on-entry source operand will read from the physical register file itself. The mapping is also absent for all the destination operands. The first time an instruction is encountered, a register queue will be associated with the destination physical register with proper update of the queue map table. Each source operand thus maps to the proper register queue. Note that for an operand that is live-on-entry but not modified in the loop body, no queue will ever get associated with the physical register, and hence all instances of the operand will read from the physical register file instead of the register queue. A source operand that is both live-on-entry and modified in the loop body will read from the physical register file for the first instance of the instruction (since no queue is associated at that point), and from the register queue for subsequent instances. Physical registers that are modified in the loop body are assumed live-on-exit; they are marked busy on trace line dispatch and are freed only when the last instance of the corresponding instruction completes execution. A register queue associated with a destination operand is freed only when all the instances of the associated instruction completes execution.

5.5 Register Queue Operations

Following are the basic queue operations involved in maintaining proper register queue semantics and correctness of the program.

1. Queue Write - Queue write involves writing the new value at the tail of the queue and setting the ready bit associated with that queue element. An instance of an instruction can be issued only if the register queue associated with its destination is not full.

2. Queue Read - An instance of an instruction must read from the proper register queue element. To implement this, a read pointer is associated with each operand, and a current pointer is associated with each register queue. Both the current pointer and the read pointer are initialized to 1. An instruction can not read its source operand until its read pointer matches the current pointer of the register queue and the ready bit of the element at the head of the queue is set. On a successful read, the operand's read pointer is incremented.

3. Queue Shift - When all the readers of the element at the head of the queue have accessed the value, the queue head can be deleted. This is detected by maintaining a MAX READERS field with each queue. All the readers of the queue head belong to the same vectorized trace that generated the value (except possibly for the last loop

iteration) and this number remains same across all the iterations. This count can be obtained while renaming a trace line and is maintained in the MAX READERS field. Each queue also maintains NUM READERS field. This is initialized to MAX READERS when a queue element comes to the head of the queue and is decremented on each queue read. When NUM READERS reaches 0, the queue head is deleted and the current pointer is incremented.

Instructions that succeed a vectorized trace line should access the last queue element produced by the live-on-exit registers of the trace line. When the last instance of a vectorized instruction completes execution, the tail of the register queue is copied to the corresponding physical register file and the busy bit of the physical register is unset.

5.6 Fraction of Instructions Vectorized

The TW machine model was enhanced to incorporate the dynamic vectorization mechanism. In this section, we present the fraction of dynamic instructions vectorized by the proposed mechanism. Table 5.1 shows this fraction for five of the SpecInt92 benchmarks. This study assumes unlimited trace line size for vectorized traces. We make this assumption to study the potential of dynamic vectorization when not limited by trace line alignment problems. This assumption is also reasonable as the average size of the loops in terms of number of dynamic instruction is quite small to fit into a single trace line (Refer Table 5.1). It is interesting to note that except for xlisp, a significant fraction of dynamic instructions can be captured using this dynamic vectorization scheme. However, the average number of iterations executed for each loop is quite small, (median loop limit of around 3-4 except for eqntott whose median loop limit is 10 iterations) whereas the maximum loop limit is quite large. The following conclusions can be drawn from this study:

1. The loops whose loop limit is quite large will benefit from the proposed scheme.

2. Since the median loop limit is very small, a small auxiliary storage can be used to capture the loops so that subsequent visit of the same loop does not incur the loop

detection overhead. This information can be stored in the trace cache as well.

3. Some loops follow only a few control flow paths (for example compress follows only two control flow paths in its main loop). Some simple enhancements of the mechanism proposed can capture this kind of loops (for example some simple pattern matching scheme can be used).

5.7 Previous Work

The CONDEL architecture [23] [24] proposed by Uht tries to achieve run time vectorization by capturing a single copy of the static loop body in the instruction window. This architecture uses a number of status-bit matrices stored with the static instruction stream to enforce correct data and control dependencies. The key differences between the CONDEL architecture and the proposed dynamic vectorization scheme are the following:

1. The CONDEL architecture captures the static loop body in the instruction window. For some loops, we have observed that the static size of the loop body is quite large, whereas the dynamic loop body is small for most of the iterations. This might help our architecture to capture the loop, whereas CONDEL model will fail since the size of the loop exceeds the implementation window size.

2. For a loop with a chain of procedure calls, CONDEL model will expand each call and store the instructions from the procedure body in the instruction window. So it is not possible to handle nested procedure calls in this architecture. Our scheme does not take care of procedure calls. But we observe that for some loops, even though there are a number of procedure calls in the static loop body, for most of the iterations those procedures are not executed. That is precisely the reason why the dynamic vectorization scheme can capture a large fraction of dynamic instruction stream. Also, some simple modification of the proposed scheme can take care of procedure calls as well.

Benchmark	Fraction of Insts Vectorized	Avg. Loop Size (Dyn. Instrns)	Loop Limit		
			Avg.	Median	Max
cc1	18.15	15.12	9.75	4	1031
compress	33.38	39.95	4.46	3	4311
eqntott	61.73	7.11	12.38	10	2131
espresso	61.30	14.29	6.69	3	1075
xlisp	6.20	8.32	2.81	2	999

Table 5.1: Fraction of dynamic instructions vectorized by the proposed scheme

3. There is no overlapping of the post-loop code with the instructions from the loop body in case of CONDEL architecture. The dynamic vectorization scheme will benefit from this overlapping.

Chapter 6

Conclusions

Improved instruction fetch bandwidth in superscalar processors can lead to later pipeline stages like register renaming, dynamic scheduling and result bypassing a bottleneck to performance. Performance evaluation of a set of recently proposed mechanisms for improving instruction issue and dispatch logic indicate a speedup of 1.5 to around 3 for Spec integer benchmarks over a traditional architecture. The dynamic vectorization method for loops of ordinary programs with complex control flow paths can capture a significant fraction of dynamic instructions, up to 60% for two of the benchmarks. The overheads of the above mechanisms lie in the auxiliary storage requirement to store the decoding and register renaming information. However, with increased transistor density, this might not be a problem.

Several issues still remain to be exploited. The hit rate for the trace cache is very poor for the benchmarks and machine platform (IBM RS6000) studied. Some simple modifications of trace lookup technique, for example, using a number of branch prediction bits for trace cache indexing might result in improved hit rate. With the improved scheme for instruction fetch, issue, and dispatch, the clock cycle penalty for branch misprediction and data cache miss increases. New mechanisms can be proposed to reduce this cost of this pipeline disruption. Also, the performance of the new architecture should be evaluated for other instruction set architectures and compliers. Several simple modifications of the proposed dynamic vectorization mechanism can be implemented to capture more than one control path within the loop body and loops with procedure calls. We leave it as future work to perform modifications and subsequent thorough evaluation of the dynamic vectorization mechanism.

Bibliography

[1] T. M. Austin and G. S. Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *Proceedings of the 19th Annual Insternational Symposium on Computer Architecture*, 1992.

[2] D. Burger, J.R. Goodman, and A. Kagi. Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[3] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *Proceedings of 22nd Annual International Symposium on Computer Architecture*, 1995.

[4] M. Franklin and S. Dutta. Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors. In *Proceedings of the 28th International Symposium on Microarchitecture*, 1992.

[5] M. Franklin and M. Smotherman. A Fill-Unit Approach to Multiple Instruction Issue. In *Proceedings of the 27th International Symposium on Microarchitecture*, 1994.

[6] M. Franklin and G. S. Sohi. Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Procesors. In *Proceedings of the 25th Annual Internations Symposium on Microarchitecture*, 1992.

[7] G. F. Grohoski. Machine Organization of the IBM RISC System/6000 Processor. *IBM Journal of Research and Developement*, 1990.

[8] E. Hao, P-Y. Chang, M. Evers, and Y. Patt. Increasing the Instruction Fetch Rate via Block-Structured Instruction Set Architecture. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[9] T. Hara, H. Ando, C. Nakanishi, and M. Nakaya. Performance Comparison of ILP machines with Cycle Time Evaluation. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[10] M. D. Hill and A. J. Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984.

[11] M.Franklin and G. S. Sohi. The Expandable Split Window Architecture for Exploiting Fine-Grain Parallelism. In *Proceedings of the 19th Annual International Symposium on computer Architecture*, 1992.

[12] Ravi Nair. IBM T. J. Watson Research Centre, Personal Communication.

[13] S. Palacharla, N. Jouppi, and J. E. Smith. Quantifying the Complexity of Superscalar Processors. Technical Report CS-TR-96-1328, University of Wisconsin Madison, November 1996.

[14] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996.

[15] R. M. Russel. The Cray-1 Computer System. *Communications of the ACM*, 21, January 1978.

[16] Jourdan S, P. Sainrat, and D. Litaize. An investigation of the Performance of Various Instruction-Issue Buffer Topologies. In *Proceedings of the 28th International Symposium on Microarchitecture*, 1995.

[17] J. E. Smith. Decoupled Access/Execute Architecture. *ACM Transactions on Computer Systems*, November 1984.

[18] M. Smotherman and M. Franklin. Improving CISC Instruction Decoding Performance using:w a Fill Unit. In *Proceedings of the 28th Insternational Symposium on Microarchitecture*, 1995.

[19] G. S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22rd Annual International Symposium on Computer Architecture*, 1995.

[20] E. Sprangler and Y. N. Patt. Faciliating Superscalar Processing via a Combined Static/ Dynamic Register Renaming Scheme. In *Proceedings of the 27th International Symposium on Microarchitecture*, 1994.

[21] R. A. Sugumar and S. G. Abraham. Efficient Simulation of Caches using Binomial Trees. Technical Report CSE TR-111-91, University of Michigan, 1991.

[22] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, January 1967.

[23] A. K. Uht. A Theory of Reduced and Minimal Procedural Dependencies. *IEEE Transaction on Computers*, 40(6), June 1991.

[24] A. K. Uht. Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream. *IEEE Transaction on Computers*, 41(7), July 1992.

[25] S. Vajapeyam and T. Mitra. Improving Instruction Dispatch and Issue for Superscalar Processors running Ordinary Programs. Technical Report IISc-CSA-96-08, Indian Institute of Science, November 1996.

[26] D. Wall. Limits of Instruction Level Parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[27] S. Weiss. Implementing Register Interlocks in Parallel-Pipeline, Multiple Instruction Queue, Superscalar Processors. In *Proceedings of the 1st International Symposium on High Peformance Computer Architecture*, 1995.

[28] K. C. Yeager. The MIPS R10000 Superscalar Processor. *IEEE Micro*, April 1996.

[29] T-Y. Yeh and Y. N. Patt. Alternative Implementation of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[30] T-Y. Yeh and Y. N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.