ABSTRACT OF THE THESIS OF

Tong Liu for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on March 2, 2001.

Title: PERFORMANCE IMPROVEMENT WITH LOGIC-LEVEL SPECULATION

Redacted for Privacy

Abstract approved: _____

Shih-Lien Lu

Current superscalar microprocessors' performance depends on its frequency and the number of useful instructions that can be processed per cycle (IPC). Higher frequency is achieved with process advancement, new circuit techniques, and microarchitectural improvement. Number of instructions processed per cycle depends mainly on microarchitecture techniques that exploit parallelism both spatially and temporally. Most techniques employed to exploit parallelism spatially tend to increase circuit complexity and may affect the frequency thus offset the performance gain intended. Finer pipeline stages exploit parallelism temporally but may suffer reduced efficiency when there are dependencies and hazards in the long pipeline. Careful balancing between frequency and useful number of instructions processed per cycle is one of the important microprocessor design tradeoffs. In this thesis we propose a method called *approximation* to reduce the logic delay of a pipe-stage. The basic idea of approximation is to implement the logic function partially instead of fully. Most of the time the partial implementation gives the correct result as if the function is implemented fully but with fewer gates delay allowing a higher pipeline frequency.

We apply this method on three logic blocks. Simulation results show that this method provides some performance improvement for a wide-issue superscalar if these stages are finely pipelined.

PERFORMANCE IMPROVEMENT WITH LOGIC-LEVEL SPECULATION

by

Tong Liu

A THESIS

Submitted to

Oregon State University

In partial fulfillment of

the requirements for the

degree of

Doctor of Philosophy

Presented March 2, 2001

Commencement June 2001

Doctor of Philosophy thesis of Tong Liu presented on March 2, 2001

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Tong Liu, Author

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor Professor Shih-Lien Lu. His contribution to this thesis has been not only through his advice, patience, many hours of detailed discussion, but also through his continuous encouragement and support during the course of this research.

I wish to thank Professor Lung-Kee Chen, Andreas Weisshaar, Alexandre Tenca and Vivek De for been my committee members. I appreciate the entire electrical engineering faculty for their wisdom.

Finally, I would like to give special thanks to my parents, my brother Jiang and my wife Xinyue for their love and sacrifice in all aspect of my life in USA.

TABLE OF CONTENTS

PAGE

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

FIGURE **PAGE**

LIST OF TABLES

TABLE **PAGE**

PERFORMANCE IMPROVEMENT WITH LOGIC-LEVEL SPECULATION

1. PROBLEM DEFINITION AND CONTRIBUTION

Microprocessors have gone through lots of changes during last decades, however, the basic computational model has not changed much. A program consists of instructions and data. The instructions are encoded in a specific Instruction Set Architecture (ISA). The computational model is still a single stream sequential model operating on the architecture states (memory and registers). The metrics to characterize a microprocessor includes:

. Frequency: the rate in which the internal clock ticks.

. Performance: the time it takes to complete a certain piece of work.

. Power: how much energy per time-unit it consumes

. Area (cost): the size of the chip and its manufacturing cost

. Complexity: a qualitative measurement indicating the time/effort to develop a processor and to verify it produces correct result.

In this thesis, we discuss only the performance of a microprocessor. The performance of microprocessor has been accelerating rapidly in recent years. This gain has been achieved through two fronts. On one front, microarchitecture innovations have been able to take advantage of the increase number of devices to process more useful instructions per cycle (IPC). Superscalar is the predominant scheme used. A superscalar processor issues multiple instructions and execution them with multiple

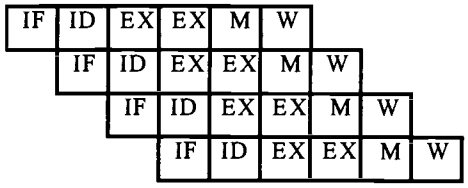
identical function unit. It employs dynamic scheduling techniques and executes instructions out of the original program order. The main goal of superscalar is to exploit as much instruction level parallelism as possible in a program. On the other front, the miniaturization of devices improves layout density and makes the circuits run faster since electrons and holes need only to travel shorter distance. Clever circuit techniques have also been invented to further speed up the logic. Together with finer pipestages, modern microprocessor has accelerated its clock frequency greatly in recent years.

However, it is believed more complexity is necessary to continue the exploitation of larger instruction level parallelism. This complexity increase tends to cause more circuit delay in the critical path of the pipeline, thus limiting the clock frequency to go up further. The current approach is to allow logic structures with long delays to spread over multiple pipe-stages resulting in logic structures that complete the computation in single pipe-stage previously to take more than one cycle time. The employment of finer pipeline stages increases pipeline latencies and imposes higher penalty due to branch miss-prediction and other miss-speculation. Moreover, other instructions that depend on the results of these multi-staged functional blocks will have to wait until they finish in order to move forward in the pipeline. In [1], the impact of data dependencies and branch penalty on pipeline performance is discussed. Since these two factors draw back the performance gain of increasing pipeline stages, there is an optimal number of pipe stages that a microprocessor will achieve maximum performance. If increase or decrease the pipe stage number from the optimal one, the performance will be degraded. This means that only increasing number of pipe stages

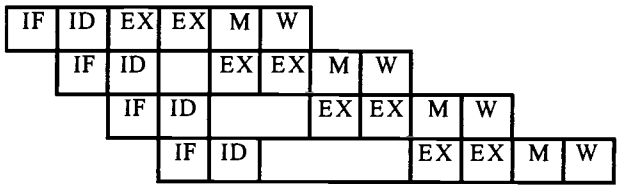
and clock frequency doesn't necessarily improve performance. Figure 1.1 illustrates the effect of executing consecutive dependent instructions. Suppose the execution delay is long enough so that it has to be expanded into two consecutive pipestages. In (a), all four instructions are independent, so they can be fully pipelined regardless of the delay of previous instruction. In (b), any instruction is dependent on it previous instruction. Bubbles have to be asserted in the pipeline while waiting for resolving the dependencies. The instruction per cycle (IPC) in (b) is only half of that in (a), so as the performance. Therefore, these long delay logic structures may become the performance bottleneck of microprocessors as clock frequency continues to rise in the future. Thus, one of the essential challenges in achieving higher performance in future microprocessors is the ability to increase IPC without compromising the ever-increasing clock frequency.

Much work has been devoted to finding methods to increase IPC. One possible approach is to increase the width of the superscalar processor [2-7]. Another approach considered by many researchers is multi-threading [8-13]. Both methods tend to increase the size of the structures used internally such as instruction window and re-order-buffer. Larger size means longer delay and may affect the growth in clock frequency. Work done by Cotofana and Vassiliadis [14] identified the delay complexity of issue logic in a superscalar processor to be a function of issue width. Work by Palacharla et. al. [15, 16] concluded that possible clock limiting structures in a superscalar processor include, register rename logic and issue logic. Also as the machine data and address width increases (currently moving from 32 to 64 bits), we believe adder may also become a bottleneck limiting the increase in frequency because

many groups reporting the design of high performance microprocessors include their adder circuits in their papers [17-19]. In one of the work, adder circuit is specified as having the second longest delay path in the microprocessor [20]. This suggests that adders may limit the frequency of a microprocessor if we want to have finer pipeline stages in the future.



(a) Pipeline with Independent Instructions



(b) Pipeline with Dependent Instructions

Figure 1.1 Dependent and independent instructions pipeline execution

In this thesis, we propose to use logic level "prediction" to "speculate" the output of critical logic blocks. The approach calls for a simpler and faster circuit implementation to approximate the original complex function. We termed this technique *approximation*. An approximation circuit should be designed so that it produces the correct result most of the time. Since it is not 100% correct all the time it does require a way to verify the correctness of the approximation. A duplicated logic block, which implements the true function and samples the output at the original worst case delay frequency is used for verification. Results from the approximation block and verification block are compared to determine if the approximated result used to advance the pipeline is correct or not. When the comparison result is negative we kill that instruction and use the correct result to continue. The recovery mechanism is similar to what is reported in [21].

In the following section, we review the background of microprocessor and current trend. In chapter 3 we describe the baseline microarchitecture and circuit design of the critical logic block we want to speed up. Chapter 4 describes the application of approximation method on three potential speed path and the recovery method. In chapter 5 some theoretical analysis is done to evaluate the important factors that impact the performance. We then show the experimental result performed through a modified SimpleScalar [22] tool set in section chapter 6. The last chapter draws conclusion.

2. BACKGROUND OF MICROPROCESSOR AND CURRENT TREND

2.1 PERFORMANCE MEASUREMENT

Microprocessor's performance continue to be a key factor for it success in the market place. Even though software compatibility drives the users to favor certain ISA, without adequate performance at first place, user will tend to switch quickly. It is the job of the microarchitecture, the logic and the circuit to process the instruction stream quickly to achieve the best performance. The performance of a microprocessor is measured by the time it takes to complete a program. It includes the following three factors: (a) Number of instructions in a program (b) Number of clock cycles taken by one instruction (c) Clock cycle time. It can be described by the following equation:

Time (Seconds) to execute a program = Instructions/Program x

Clock cycle/Instruction x Seconds/Clock cycle. (2.1)

In order to improve performance, we need to decrease the product of the three items at right hand side of the above equation. The number of instructions in a program depends on the instruction set architecture. For a specific instruction set on a specific program, the performance can be expressed by re-writing the above equation in a reverse way:

Performance = Instructions/Cycle (IPC) x Clock Frequency (2.2)

Since different processors have some advantages of higher performance when running its favorite program, Benchmark programs are the ones to make fair comparisons. One of the popular benchmarks is the SPEC processor benchmark [23], which uses primarily real applications. The SPECint and SPECfp, which targets on

integer and floating point performance, are now becoming standard of measuring today's microprocessors.

2.2 TECHNOLOGY DESCRIPTION

The microprocessor revolution owes its fast growth to a combination of several technologies.

. *Process technology.* The key to smaller and faster devices that consume less and less power.

. *Circuit.* Faster and more efficient building blocks.

. *Microarchitecture.* Logic to execute more instructions per cycle at an increasing frequency.

. *Architecture and compilers.* More efficient ways to translate a task into machine instructions.

. *CAD tools.* Allow designers to study design trade-offs quickly.

Process Technology is the fuel that has moved the entire VLSI industry and the key to its growth. A new process generation is released every 2-3 years. A process technology is usually identified by the length of MOS gate, measured in microns (10^{-6} meters, denoted as u or μ).

Every new process generation brings huge relative improvement in all relevant vectors. Process technology scales by a factor of around 0.7 all physical dimensions of device and wire (including those vertical to the surface) and all voltages pertaining the devices. With such scaling, typical improvement figures are:

. 1.4-1.5X faster transistors.

. 2X smaller transistors.

. 1.35X lower operating voltage.

. 3X lower switching power

Theoretically, with the above figures, one could expect potential improvements like:

. "Ideal Shrink": use the same # of transistors to gain 1.5X performance, 2X smaller die and 2X less power.

. "Ideal New Generation": use 2X the # of transistors to gain 3X performance with no increase in die size and power.

Process technology is the single most important technology that drives the processor industry. Growing 1000X in frequency (from 1 MHz to 1 GHz) and integration (from ~10K to ~10M devices) in 25 years was not possible without process technology improvements.

New and useful circuits constructed with a small number of devices are still being invented. These circuits either provide a better performance or operate with less power while implementing a logic function. New logic families are occasionally invented and provide a new methodology to realize logic functions more effectively.

Microarchitecture attempts to increase both IPC and frequency. A simple frequency boost applied to an existing microarchitecture can reduce IPC, and thus does not achieve the expected performance increase. Microarchitecture techniques, such as caches, branch prediction and out of order execution, increases IPC. Other microarchitecture ideas, most notably pipelining, help to increase frequency beyond the increase provided by process technology.

Modern Instruction Set Architecture (ISA) and good optimizing compiler can reduce significantly the number of dynamic instructions needed to execute a given program. Furthermore, being aware of the underlying microarchitecture they lead to higher IPC for code generated for the target microarchitecture.

Design tools help designers to tune circuits for performance. It also helps designers to explore much more design space than possible by hand. It enables the designer to manage the complexity growth required for performance boost.

2.3 MICROARCHITECTURE EVOLUTION

Microprocessor performance depends on its frequency and IPC. Frequency increase is achieved with process, circuit, and microarchitectural improvements. New process technology reduces gate delay time, thus frequency, by ~1.5X. Microarchtitecture affects frequency by reducing the length of work done at each clock cycle.

The early general-purpose microprocessor was non-pipeline, single issue and in-order architecture. It basically takes one instruction at a time, and won't start the next instruction until the previous one finishes. Later, with higher integration capability of VLSI technology, latches and flip-flops can be placed to separate different steps of the microprocessor logic. There are five basic steps for each instruction to go through a microprocessor: Fetch, Decode, Execution, Memory access, Write back. For a pipeline processor, each step of one instruction can be overlapped with other steps of its neighboring instructions.

Pipelining is a very effective technique. We see a clear trend of increasing the number of pipe stages and reducing the amount of work per stage. Employing many pipe stages is sometimes termed deep pipelining or super-pipelining. However, there are problems with indefinitely increasing pipe stages:

. Latch/Flip-flop timing overhead. The latch or flip-flop themselves consume time, and also with setup/hold time and clock skew, the overhead could be large enough that there is no design space left for useful logic.

. Performance of a pipelined machine depends on the slowest stage of the pipe. Good balancing of the overall work is becoming more difficult as the number of pipe stages increase.

. Interdependencies among instructions result in wasted cycles, causing performance to scale less than linearly with number of pipe stages.

For a given partition of pipeline stages, the frequency of the processor depends on the time it takes the logic to perform the longest stage. Logic and circuit optimizations as well as new process technology help to accelerate the execution of the logic within each stage, thus reducing the cycle time and increasing frequency without increasing the number of pipe stages.

Although the pipeline structure allows frequency to scale linearly with the number of stages, the performance does not. With longer pipes, the portion of wasted cycles, termed pipe-stalls, becomes bigger. Main reasons for stalls are resource constraints, data dependencies, memory dependencies and control dependencies.

. Resource constrains happens when an instruction needs a resource (e.g., execution unit) that is currently used by another instruction in the same cycle.

. Data dependency occurs when a result of one instruction is needed as a source to another instruction. The new instruction has to wait unit it sources are available, even if there is a free execution unit. Data dependencies occur frequently in multiple-cycle operations such as complex integer and floating point instructions.

. Memory delay is a special case of data dependencies, sometimes termed as load-to-use delay. At very short cycle time, even accessing fast memory takes several cycles. Accessing slower memory may take tens of hundreds of cycles. Memory delays are mentioned specifically due to their adverse impact on performance.

. Changing the control flow of the program may cause a pipe stall as well. A branch instruction changes the address from which the next instructions are fetched. This address is known only at later stages of the pipeline, causing a control flow stall.

The performance can be further improved by allowing multiple non-dependent instruction to execute by multiple execution units. This is the idea of superscalar processor. Widening the pipeline makes it possible to execute more than one instruction per cycle, but there is no guarantee that any given sequence of instructions can take advantage of this capability. Instructions are not independent of one another, but are interrelated; these interrelationships prevent some instructions from occupying the same pipeline stage. Furthermore, the processor's mechanisms for decoding and executing instructions can make a big difference in its ability to discover instructions that can be executed at the same time.

Overall, the reasons for stalling a pipeline (resource conflict, data dependencies, memory dependencies and control dependencies), also apply to blocking the performance of superscalar machine. The processors described so far execute

instructions in-order. That is, instructions are executed in the program order. In an in-order processing, if an instruction cannot continue, the entire machine stalls. For example, a cache miss delays all following instructions even if they do not need the results of the stalled load instruction. A major breakthrough in boosting IPC is the introduction of the Out-of-Order execution, where instruction execution order depends on data flow, not on program order. That is, an instruction can execute if its sources are available, even if previous instructions are still waiting. The effect of super scalar and out-of-order execution is shown in the following example:

Out-of-order processing hides stalls. For example, while waiting for a cache miss the processor can execute newer instructions, as long as they are independent on the load instructions. A super-scalar out-of-order processor can achieve much higher IPC than in-order one. Out-of-order execution involves dependency analysis and instruction scheduling. Therefore, it takes longer time (more pipe stages) to process an instruction. Out-of-order processor can overcome the performance loss by instruction interdependencies and resource conflict by re-arranging the execution order. However, with longer pipe, an out-of-order machine suffers more from branch mis-predictions and the hardware is more complex.

2.4 CURRENT TREND AND CHALLENGES

2.4.1 IMPROVEING THE FREQUENCY – THE PIPELINE APPROACH

Process technology and microarchitecture innovations enabled 2X frequency increase every process generation. As the process improves, the speed increases and

the average amount of work being executed between pipeline stages decreases. Reducing stage length is achieved by improving design techniques and by increasing the number of stages in the pipe. While in-order processors used 4-5 pipe stages, a modern out-of-order processor use over 10 pipestages, and with frequencies over 1 GHz, we can expect close to 100 pipeline stages. Improvement in the frequency does not always improve the performance of the processor. Performance increase rate is less than linear mainly because deep pipelining does not reduce the time wasted due to each cache misses and branch mis-prediction flushes. In order to keep the performance growth, our main challenge is to increase the frequency much faster than the reduction in the IPC.

2.4.2 THE INSTRUCTION SUPPLY CHALLENGES

The instruction supply is responsible for feeding the instruction into the parallel execution pipes. The rate of instructions which are entered the execution pipe, depends on average number of bytes fetched from memory, and the rate of useful instructions in that stream. The fetch rate depends on quality of the memory subsystem. The number of useful instruction in the instruction stream depends on the ISA and the Branches. The ISA determines the average length of a single instruction, the branch instructions determine how many of them are useful. Unused instructions result from (1) Control flow change within a block of fetched instructions, making the rest of the block unused, and (2) Branch predictor provides a wrong prediction discarding all the instructions in the wrong path. On average, a branch occurs every 4-5 instructions, limiting the instruction fetch bandwidth to basic block at a time. In order to increase

the effective fetch bandwidth, the compiler can optimize the code to produce larger basic blocks, special structure of caches can be used and in the future, maybe "non-sequential" fetching techniques need to be developed.

The decoder is the next stage in the front-end. RISC architectures, using fixed length instructions, can easily decode instructions in parallel. Parallel decoding is a major challenge for CISC architecture, such as IA32, that use variable length instruction. Some implementations use speculative decoders to decode from several potential instructions addresses, later discarding the wrong one, other store additional information in the instruction cache to ease decoding 2nd time around. Some IA32 implementations translate the IA32 instructions into an internal representation, allowing the internal part of the processor to work on simple instructions at high frequency, similarly to RISC processors [24].

2.4.3 EFFICIENT EXECUTION

The front-end stages of the pipeline prepare the instructions in either instruction window or reservation stations. The execution subsystem schedules and executes these instructions. All modern microprocessors use multiple execution pipes to increase parallelism. Performance gain is limited by the amount of parallelism found in the instruction window. The parallelism in today's system is limited by the data dependencies in the program.

Studies show that, in theory, high level of parallelism is achievable. In practice, however, this parallelism is not realized, even when enlarging the number of execution pipes. More parallelism requires higher fetch bandwidth, bigger instruction window,

and wider dependency tracker and scheduler. Enlarging such structures involves polynomial complexity for less than a linear performance gain (e.g., scheduling complexity is in the order of O^3 of the scheduling window). VLIW (Very Large Instruction Width) architectures such as IA64 EPIC and IBM avoid some of this complexity by using the compiler to schedule instructions.

Two hardware techniques have been very popular recently to solve the data dependencies in program: Value prediction [10, 21] and Instruction reuse [25]. These techniques are based on the fact that there is significant result redundancy in program [25-27], i.e., many instructions perform the same computation and, hence, produce the same result over and over again. Both techniques attempt to reduce the execution time of programs by alleviating the dataflow constraints. They use the redundancy in programs to determine, speculatively (Value Prediction) or non-speculatively (Instruction Reuse), the results of instructions without actually executing them. The advantage of doing so is that instructions do not have to wait for their source instructions to execute first; they can execute sooner using the results obtained by the above two techniques, thus, relaxing the dataflow constraint. The implementation of both techniques is to use a hardware table. Value prediction method is more efficient to catch redundancy in a program than Instruction reuse. However, since value prediction is a speculative technique, extra verification logic is needed to check the result. If a prediction is correct, the pipeline will continue without delay, and its dependent instructions can execute earlier that they would have otherwise. On the other hand, if a prediction is found to be wrong, all its dependent instructions need to re-execute with correct input value, and the pipeline is delayed by the latency of

verifying the prediction. Both technique collapses true dependencies by allowing dependent instructions, that would have executed sequentially, to execute in parallel.

3. BASELINE DESIGN

In this thesis, we try to speed up some critical logic in superscalar processor in order to increase the frequency without compromising IPC. The logic structures we have considered are adder, issue logic and register rename logic. Adder circuit delay is not related to issue width. However address calculation done by integer adders is the key operation for instruction fetch, branch prediction and data supply from memory [28]. Moreover, we are observing a trend in the growth of datapath width. Currently we are in transition from 32 bits to 64 bits. Designing very fast large adders has been a constant research topic [29, 30]. We believe adder may become a cycle limiter also in the future. The latter two are key structures used to exploit ILP in a wide-issue superscalar microprocessor and generally considered as single cycle function logic that are proved to be difficult to pipeline inside. We called these structures cycle limiter. In order to see the performance improvement of our work, a baseline microarchitecture is needed to compare with. There are different ways to implement an out-of-order issue microarchitecture. Our baseline superscalar uses a centralized issue window structure. It basically combines the reorder buffer and instruction window together, and can provide precise interrupt [2, 15 and 31]. We briefly describe the three structures used in our baseline machine in the following sections.

3.1 ADDER

Many instructions contain add. Load, store and branch use adder for address calculation. Arithmetic instructions use adder for add, subtract, multiply and divide.

Adder is one of the key performance structures used in function units. There are many different kinds of adders. Due to performance requirement, most of the current high performance processors employ one of the known parallel adders [32].

An n-bit adder is just a combination circuit. It can be written by a logic formula whose form is a sum of products and can be computed by a circuit with two levels of logic. The formula for the ith sum can be written as

$$s_i = a_i b_i \#c_i\# + a_i \#b_i c_i\# + a_i \#b_i \#c_i + a_i b_i c_i \quad (3.1)$$

where c_i is both the carry-in to the ith adder and the carry-out from the (i-1)-st adder. a_i and b_i are the two inputs at ith bit.

Since c_i is the only term that depends on previous inputs, we introduce the following formula to calculate c_i.

$$p_i = a_i + b_i \,,\; g_i = a_i b_i \,,\; c_i = g_{i-1} + p_{i-1} c_{i-1} \quad (3.2)$$

where p_i and g_i are called propagation and generation term for ith bit respectively. If g_{i-1} is true, then c_i is certainly true, so a carry is generated. Thus g is for generate. If p_{i-1} is true, then if c_{i-1} is true, it is propagated to c_i. If we re-write the above equation recursively, then

$$c_i = g_{i-1} + p_{i-1}\, g_{i-2} + p_{i-1} p_{i-2} g_{i-3} + \ldots + p_{i-1} p_{i-2} \ldots p_1 g_0 + p_{i-1} p_{i-2} \ldots p_{i1} p_0 c_0 \quad (3.3)$$

An adder that computes carries using equation (3.3) is called a carry-lookahead adder, or CLA. A CLA requires one logic level to form p and g, two levels to form the carries, and two for the sum, for a grand total of five logic levels. However, a carry-lookahead adder on n bits requires a fan-in of n+1 at the OR gate as well as at the rightmost AND gate. Also, the p_{n-1} signal must drive n AND gates. In addition, the

rather irregular structure and many long wires in above design makes it impractical to build a full carry-lookahead adder when n is large.

However, we can build up p's and g's in steps to reduce fan-in. By defining the group propagation and generation term P and G, for any j with $i < j$, $j+1 < k$, we have the recursive relations

$$G_{ik}=G_{j+1,k} + P_{j+1,k}G_{ij},\ P_{ik}=P_{ij}P_{j+1,k},\ c_{k+1}=G_{ik} + P_{ik}c_i,\ s_i = a_i \wedge b_i \wedge c_i \quad (3.4)$$

Equation (3.4) says that a carry is generated out of the block consisting of bits i through k inclusive if it is generated in the high-order part of the block (j+1, k) or if it is generated in the low-order part of the block (i,j) and then propagated through the high part. These equations will also hold for $i \leq j < k$ if we set $G_{ii} = g_i$ and $P_{ii} = p_i$.

A four bits CLA is shown in Figure 3.1. At the top of the diagram, input numbers $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$ are converted to p's ad g's using cells of type A. The various P's and G's are generated b combining cells of type B in a binary-tree structure. By feeding c0 in at the bottom of this tree, all the carry bits come out at the top. Each cell must know a pair of (P,G) values in order to do the conversion, and the value it needs is written inside the cells. There is a one-to-one correspondence between cells, and the value of (P,G) needed by the carry-generating cells is exactly the value known by the corresponding (P,G) generating cells.

There are different kinds of parallel adders besides Carry Look Ahead (CLA): Brent-Kung Adder (BKA), Kogge-Stone Adder (KSA) and Carry Select Adder (CSA), They all have comparable asymptotic performance when they are implemented in CMOS with either static or dynamic circuits [33]. That is, their critical path delay is asymptotically proportional to *log* (N), where N is the number of bits of the adder.

The cost complexity of parallel adders approaches N^2 when fan-in and fan-out of gates used are fixed.

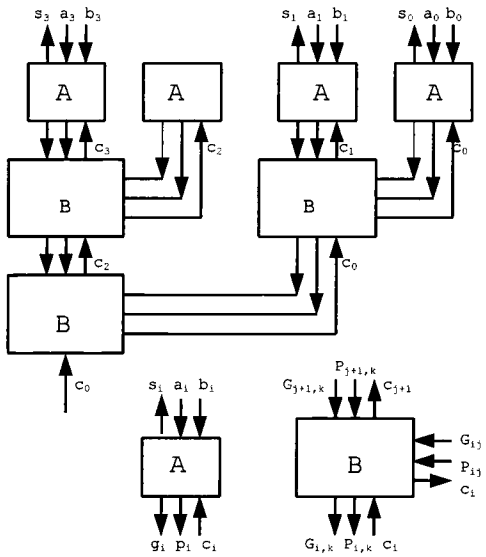


Figure 3.1 Four bits complete carry-lookahead tree adder

3.2 REGISTER RENAME LOGIC

Register renaming eliminates storage conflicts (anti- and output dependencies) for registers. When an instruction is decoded, its destination register is assigned to a physical register (renamed). Usually the number of physical registers is greater than the number of architectural or logical registers. When a later instruction refers to a previously renamed destination register (with its logical binding), it must be able to traverse the renaming and obtains the value stored inside the corresponding physical register or just the tag of the physical register if the value has not yet been produced. Thus, the register rename logic is used to translate logical register designators into physical register designators. Logically, this is accomplished by accessing a mapping table with the logical register designator as the index. From [15, 16], there are two different implementations: RAM and CAM. In the RAM scheme, the number of entries (i.e., rows) in the mapping table is equal to the number of logical registers and is independent of the number of physical registers. However the mapping table's entry length (i.e., columns) of the RAM scheme depends on the number of checkpoints needs to be stored. As we issue more instructions per cycle we need to predict over nested branches that will increase the width of the mapping table. The CAM scheme, on the other hand, has fixed table width but requires a larger number of entries. We use the CAM structure in our baseline machine. A block diagram of the renaming logic is shown in Figure 3.2 (in this figure the horizontal entries are rows, R is a logical register, and P is a physical register). It consists of a set of physical registers, a mapping table and a priority encoding logic block. The number of entries in the mapping table is equal to the number of physical registers. When a decoded

instruction enters into the rename logic, its destination register is assigned a new entry in the physical register and the corresponding physical register is stored with the logical register binding. The same decoded instruction's source registers binding will be used to lookup the mapping table associatively. Since it is possible that a logical

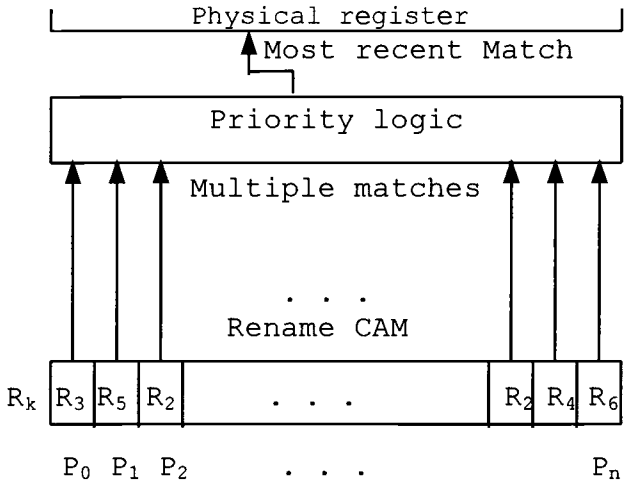


Figure 3.2 Rename CAM and priority logic

register can match multiple physical registers due to earlier instructions specify the same destination registers, the result from this associative lookup is channeled into the priority encoding logic. The priority encoder converts the multiple ones into a single active line to be used to access the physical register. The critical path of register rename using this scheme is the time for mapping table lookup and the priority encoding logic when multiple matches are found. The delay will be longer as we

increase the number of physical registers. In the worst case, when the matched entry is at the head of the mapping table, N-bit adder-like ripple structure will be formed through the entire priority encoder. Let m_i stand for the ith mapping table has a content that maps the upcoming logical register, and s_i means the mapping is selected as the latest. Also assume the upcoming logical register index is l, and the content of ith mapping table is l_i. So we get

$$s_i = s_{i-1}\# s_{i-2}\# \ldots s_1\# s_0\# m_i,\ m_i = l \text{ xnor } l_i \tag{3.5}$$

If we compare the terms in above equation to those of an adder, l and li correspond to the two adder inputs ai and bi, $s_{i-1}\# s_{i-2}\# \ldots s_1\# s_0\#$ corresponds to carry term ci.

A carry look ahead structure (parallel-prefix) can be used to make it associatively parallel, and the delay will be in the order of *log* (N), where N is number of physical registers. Similar to CLA we discussed before, we can generate the priority chain by steps. We can construct two types of block A and B. Block A has two inputs m_{in} and s_{in}, two outputs m_{out} and s_{out}. m_{in} means there is a local match and it sends out a request m_{out} to be considered as the latest match. s_{in} means the request has be granted from upper level of priority logic so the block sends out a grant signal s_{out} to select the local match as the latest match. The logic in block A are

$$s_{out} = m_{in}\, s_{in},\quad m_{out} = m_{in} \tag{3.6}$$

Block B has two input request m_{in0} and m_{in1} from adjacent two bits. The block sends out a request m_{out} to upper level. If the upper level grant the request, it sends in s_{in} and the block give grant as s_{out0} and s_{out1}based on the priority of the two input requests. If we assume bit 0 has higher priority, the logic in block B is

$$s_{out0} = m_{in0}\, s_{in},\ s_{out1} = m_{in1}\, s_{out0}\#\, s_{in},\quad m_{out} = m_{in0} + m_{in1} \tag{3.7}$$

A four bits parallel priority encoding logic is shown in Figure 3.3. Assume bit 0 has higher priority, s_{in} at the bottom block is hardwired to logic "1".

For a wide issue superscalar machine, generally multiple instruction will be renamed at the same time. Thus the comparing and priority logic will also include the earlier instructions in the current rename group.

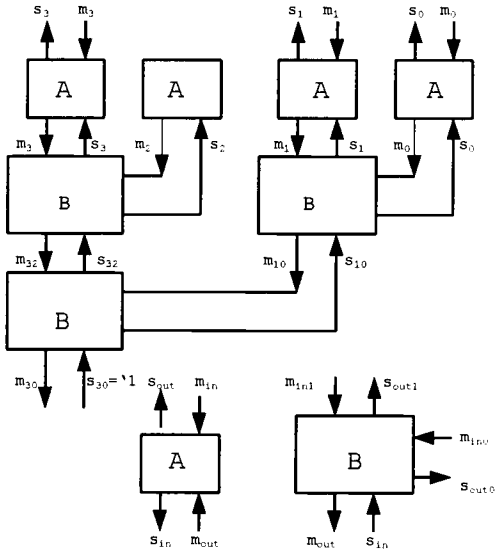


Figure 3.3 Four bits priority encoding logic

3.3 INSTRUCTION ISSUE LOGIC

The issue logic contains three different parts, and all of them are speed critical [14, 15, 16]. When an instruction is finished from the functional unit, it will write back data to its destination register. The status of its dependent instructions will be updated by the write back instruction. This is done by broadcasting the tag associated with the result register to all the instructions in the issue window. The broadcasting tag will compare the tag of each source operand of the instructions in the window. If there is a match, that particular operand is marked ready. If all of the operands are marked ready, the instruction is ready to issue. This hardware is usually referred to as the *wakeup logic*. If multiple dependent instructions are ready to issue, there may be contentions on issue bandwidth and functional unit. A *selection logic* is needed to arbitrate which ready instruction to be issued first. Every ready instruction raises a request signal to the selection logic, if the request is accepted, the ready instruction receives a grant signal, and it is issued to the functional unit. There are different kinds of selection policy, and oldest-first policy, which grant instruction occurs earliest in program order first, is one of most popular policies. In a superscalar machine, since out-of-order issued instructions usually retire in-order, this policy is very necessary because issuing earlier instruction first can resolve dependencies quicker and committing earlier instruction first can leave space in the instruction window for newly decoded instructions. The basic structure of the selection logic is shown in Figure 3.4. When a ready instruction is granted to issue, write back data of the instruction it depends on will be bypassed from output of the corresponding functional unit to the source register. The delay of wakeup-selection-bypass logic increases with

increasing issue window size. The selection logic will start to check the request of instructions from earliest to latest in program order, which is the order of RUU [34] from head to tail. In the worst case, when the only request is from tail of RUU, an adder like ripple carry will be formed through all entries of RUU. Let r_i stand for the request from a waken instruction resides in ith RUU location, and s_i means the request is the oldest. So we get

$$s_i = s_{i-1} \# s_{i-2} \# \ldots s_1 \# s_0 \# r_I \tag{3.8}$$

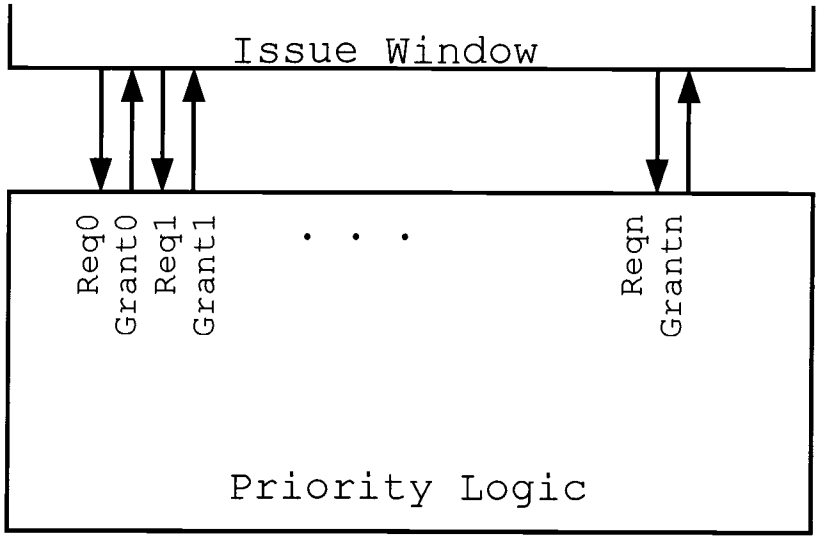


Figure 3.4 Issue selection logic

This is the similar structure as in rename logic. A carry look ahead structure can be used to make this process parallel and the delay is the order of *log (N)*, where N is the window size. For wakeup and bypass logic, the RC delay dominates the circuit speed. Circuit simulation shows that RC delay is more sensitive to window size than logic gate level [15]. For the multiple issue case, the delay analysis will be similar.

4. LOGIC LEVEL SPECULATION TO SPEEDUP CRITICAL LOGIC

Previous study [35] shows that for random input data, the average carry length of a CLA is only 1/3 of its maximum carry chain. Moreover, other works have shown that there is redundancy exits in programs [25-27], i.e., many instructions perform the same computation with the same or similar input data pattern repeatedly. This could be used for adder output speculation. For example, in address calculation, one of the input to the adder is static. Moreover the other operand is usually incrementing with a regular stride. Therefore the actual adder delay is much shorter than the worst case maximum delay. We use the *approximation* technique described in the introduction section by generating part of the whole carry chain. As for the register renaming logic, we believe that the renaming will mostly happen among instructions close to each other, so we employ the *approximation* method described previously and use a simpler priority encoding logic. For issue logic, we again use the *approximation* method. We only select among a small group of instructions close to the head of instruction queue to issue, because these instructions are relatively older ones and should be issued and retired earlier. This results in simpler and faster selection logic. Due to this selection strategy, the wakeup and bypass logic can be prioritized to work on the corresponding instructions closer to the head of instruction queue first, and work on rest of the instructions later. Because of the approximation techniques, the total pipestages of the machine are shorter, the dependency chain will be resolved faster, and results in higher IPC. As other prediction methods, logic level speculation is not 100% accurate. If the prediction is wrong, the false speculated instruction has to be re-issued and re-

executed. This will cause more resource contention, and the dependency chain will be resolved even slower than the baseline structure. If the prediction accuracy goes down to a certain point, the speculatively architecture will perform worse than the baseline architecture. So we can only work on the logic structure whose behavior is highly predictable. If the prediction accuracy is high enough to overcome the replay penalty of false speculation, a performance improvement is expected. Also the wrongly speculated instruction output will trigger its dependent instructions to start execution and produce more false results. These false results will trigger their own dependent instructions to execute, and cause a chain reaction resulting in large overhead and overall performance loss. Therefore it is important to stop the write-back of the speculative instructions as soon as the false prediction is detected. We describe the details of our design and analysis used in the following sections.

4.1 ADDER

The critical path of an adder is its full carry chain. Current microprocessors all use some type of parallel adders that generate all carries through a tree structure first and then consume the results of the chain to provide the sums. For an N-bit adder, we denote the individual bits of the two input operands as a_i, b_i and intermediate carries as c_i (i=0, 1, …, N-1). Each intermediate carry signal - c_i depends on all its previous input bits. i.e.,

$$c_i = f(a_{i-1}, b_{i-1}, a_{i-2}, b_{i-2}, \ldots, a_0, b_0) \quad (4.1)$$

Thus, in order to generate the correct final result, we must consider all input bits (look ahead all inputs) to obtain the final carry out. However in real programs, inputs

to the adder are not completely random and the effective carry chain is much shorter for most cases. That means we can build a faster adder with much short carry chain to approximate the result. We propose an approximated design which considers only the previous k inputs (lookahead k-bits) instead of all previous input bits for the current carry bit. i.e.,

$$c_i = f(a_{i-1}, b_{i-1}, a_{i-2}, b_{i-2}, \ldots, a_{i-k}, b_{i-k}) \text{ where } 0 < k < i+1 \text{ and } a_j, b_j = 0 \text{ if } j<0 \quad (4.2)$$

We have discussed previously that the delay cost of calculating the full carry chain length of N bits using a parallel adder structure is proportional to *log (N)*. Therefore, if we let $k = \sqrt{N}$, our new approximation adder only need half of the delay ($log\sqrt{N} = \frac{1}{2} log\ N$). We can also derive the probability of having a correct result with only k previous inputs considered assuming random inputs. We will go through the detail derivation in the Appendix. The prediction rate of an N-bit adder with k bits carry chain is:

$$P(N, k) = (1 - \frac{1}{2^{k+2}})^{N-k-1} \quad (4.3)$$

Figure 4.1 illustrates this relationship between prediction rate of add and the prediction carry chain length (look-ahead length - k) graphical. For example, a 64-bit approximation adder with 8-bit ($8 = \sqrt{64}$) look-ahead gives correct result 95% of the time assuming random input data. An example design with k = 4 is shown in Figure 4.2.

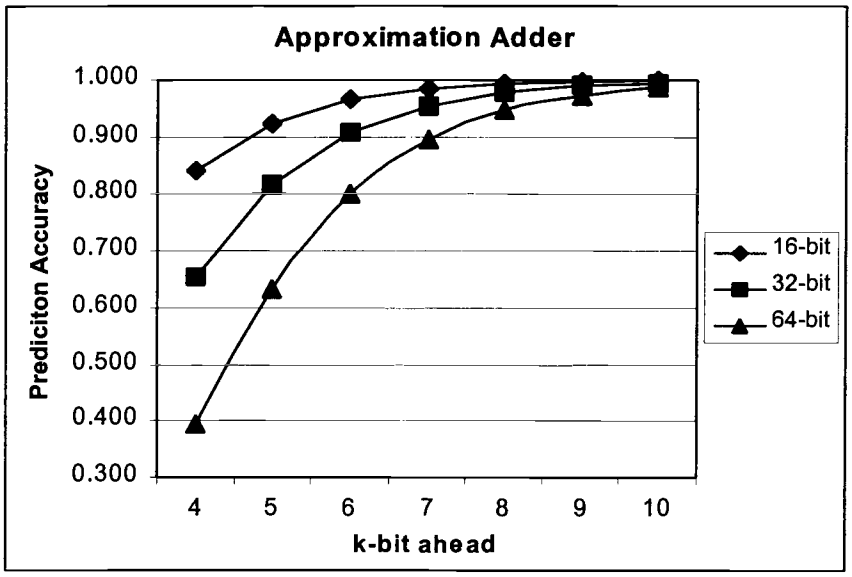


Figure 4.1 Prediction rate vs. # of bit look-ahead for 16, 32 and 64-bit adder

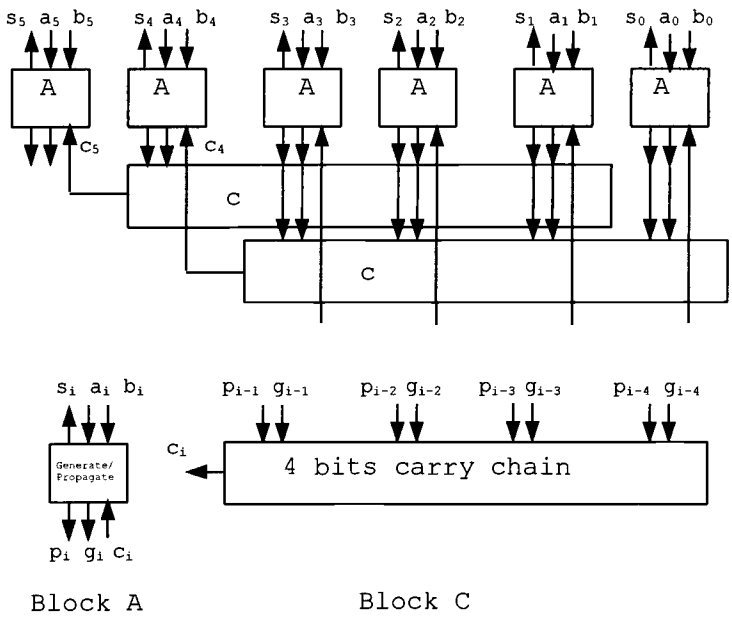


Figure 4.2. An example approximation adder design with k=4

4.2 RENAME LOGIC

As mentioned previously, the critical path of the register rename logic is the delay of the associative lookup and the priority logic when multiple matches are found. By experimenting with benchmarks, we found that dependent instructions may have spatial locality. In other words, they are most likely to be close to each other. Thus, we propose to use a smaller CAM to implement the mapping table. The CAM table basically contains a portion of the whole map. When a new instruction enters the rename logic, its destination binding is assigned a new physical binding. The mapping table is updated if the table is not full. Otherwise the oldest one is dropped to leave

room for the newly renamed destination binding. At the same time the source bindings are used to lookup the partial CAM. If there is no physical mapping found in the small CAM but the mapping does exist in the full CAM, A mis-speculation occurs. Since the number of inputs to the priority encoder is equal to the number of entries in the smaller CAM, the delay for the rename logic is also smaller. In order to double the speed, we propose to use a much smaller CAM table containing only the latest $\sqrt{N}$ number of instruction's register mapping table in it, where N is the window size. Because of the locality property of register dependency, we hope to get most of the reading operation from the rename logic correctly. Beside the faster (approximation) renaming logic, we still keep a regular full CAM and the associated full length priority encoder. It will be used to recover the mis-speculation and provide the correct renaming result in the next cycle.

4.3 ISSUE LOGIC

We use the same idea as rename logic by targeting the issue logic on the earliest $\sqrt{N}$ entries (N = window size), so that the issue logic only needs to consider waking up, selecting and bypassing data to instructions within $\sqrt{N}$ entries to the head of RUU. Since the wakeup and bypass delay are RC dominated, and RC delay is more sensitive to the window size, we will have more than twice speed up in these two logics. So the total speculative issue logic delay will be less than half of the issue logic in baseline microarchitecture if only $\sqrt{N}$ entries are considered. There is no replayed needed for the approximated issue logic since there is no false result generated.

However, some issue bandwidth or functional units may be wasted because there may not be enough ready instructions in $\sqrt{N}$ number of entries (N = window size).

4.4 HARDWARE IMPLEMENTATION AND RECOVERY

4.4.1 IMPLEMENTATION COST

Our new microarchitecture uses the speculative adder, rename and issue logic as described previously. A duplicated normal adder and rename logic is also included in the machine being sampled at a slower frequency. The size of the above mentioned logic-level speculation logic for rename and issue is smaller than the original logic used in the baseline machine, since the speculative window size is scaled down (in our case the size is the square root of the original size). For 64 bits priority encoding logic, the total cost of hardware is 64*A + 64*(1/2 + 1/4 + 1/8 + ... + 1/64)*B. For 8 bits speculative priority logic, the cost is 8*A + 8*(1/2 + 1/4 + 1/8)*B, only 1/8 of the normal hardware. For an N-bit adder with k-bit carry look-ahead, a total of N k-bit adders are needed. When k is large, the new design may have a significantly large area. Fortunately, from our benchmark experiment, 4 bits of carry look-ahead can achieve an average of 85% prediction rate for 64 bits adder (random inputs give only 40% accuracy), this is due to the redundancy in program data. The total cost of hardware for 64 bits adder is 64*A + 64*(1/2 + 1/4 + 1/8 + ... + 1/64)*B. For 64 bits approximation adder with 4 bits predictor, the total cost of hardware is 64*A + 64*(1/2 + 1/2)*B. Even though the two cases have the same total number of gates, the normal adder has long routings from LSB to MSB. On the other hand, for speculative adder, each piece of small carry chain only has local wire routings, so the device size can be very small and layout can be rather compact. In sub-micro technology, most function units are routing limited, the area

saving for speculative adder could be an order of magnitude. Thus, in general, our duplicated hardware used to speculate is smaller in size than the checking hardware. This is different from DIVA processor proposed by Austin [36], which requires an almost identical hardware as the checker. Both approaches speculate on circuit timing and both can avoid metastability. The other cost of hardware is that the verification adder and rename priority logic needs to be duplicated in order to match the slower verification frequency to faster execution frequency. So the overall extra cost for approximation adder and priority rename logic is 100%. Since in baseline microarchitecture, the adder and rename logic account for less than 1% of the total gates and area, the increase of area and power for approximation method is relatively small.

4.4.2 RECOVERY COST

After the verification logic finished, the result is compared with corresponding "speculative result". If they match, no other action is required. Otherwise instructions, which generate a false result, will be issued again and write back with the correct result from verification logic. We assume that it takes an extra cycle for the slow (original) logic to finish and verify the speculative result. Also, as soon as the false speculation is known, the write back of the speculative instruction is stopped so that it won't trigger the next dependent instructions. For issue speculation, there won't be any false result generated, so no replay is needed.

The issue mechanism in the superscalar microarchitecture is event triggered. This means an instruction will check the readiness of all of the source registers and decide to send a request to issue only when new data is written to any of the source registers. This can happen in two cases:

I. In rename stage, if all source register data are available, either in physical register it matched with, or direct from architectural register file, then the instruction is ready to issue immediately.

II. In write back stage, when an instruction finishes execute and write back data, its dependent instructions will be waked up, instructions with all source data available are ready to issue.

We now discuss the detail on how the newly proposed microarchitecture handles speculation and recovery. In our design, RUU has the same content as baseline microarchitecture except every entry has flags showing the bogus speculation, one per each source register. We call it value prediction flag (VPF). Initially all VPFs are reset. The VPF of a register will be set when the verification logic finds out that the speculation done on the corresponding instruction before is wrong, or that register is written back by an earlier instruction whose VPF has been set. The VPF will be cleared when the corresponding register is written back by an earlier instruction whose VPF is cleared. VPF will gate the write back of the instructions so that they won't contaminate its dependents. Because it takes one extra pipestage for the verification logic to figure out the result of the speculation, VPF will be updated one cycle later than the speculation stage. If an instruction's write back stage is immediately following it speculation stage, it will trigger its dependent instruction to issue because VPF hasn't been set yet. However, after the dependent instruction issues, its VPF will be assigned and its write back will be stopped if false speculation happens. Since updating VPF for the dependent instructions can be done in parallel with their executions, it won't degrade the performance. We didn't use speculative adder for

branch instruction. The reason is that branch will be resolved in the next cycle immediately after the adder calculates the address, and before VPF of the branch instruction is assigned. The false speculation of adder will cause spurious branch mis-predictions. In other words, a correctly predicted branch will be considered mis-predicted because the adder that is used to calculate target address and to verify the branch prediction is wrong. The penalty of recovering from spurious branch mis-predictions will be higher than the benefits we get from the speculation of add. For rename speculation, because it happens at the front end of the machine pipeline, the VPF of the false speculated instruction would be set before the branch is resolved. So no spurious branch miss-predictions will happen.

5. THEORETICAL PERFORMANCE STUDY

Research done by Emma and Davidson et. al. [1] shows that as the number of pipestage is increased, data dependencies and branches monotonically degrade the pipeline performance (in terms of clock cycles per instruction). The longer the pipeline is, the more cycles of penalty the data dependency and branch mis-prediction will cause. However, increasing the pipeline length will increase clock frequency monotonically. These two opposite factors will decide the optimal pipeline length based on specific technology.

In this thesis, we want to study the impact of logic-level data speculation on how it improves the performance by overcoming the effect of data dependencies on long pipelines. As we have presented in the previous chapter, the key idea of this method is to reduce the pipeline length by speculating the result of long delay functional structures. The baseline model and speculative model runs at the same frequency. In order to keep the same frequency, the execution time of a functional unit in the baseline machine must be broken up and requires 2 stages. However the logic speculation approach allows the same functional unit to run faster and uses only one stage or one cycle but with replay penalty. It is obvious that under same frequency, the model with shorter pipeline will suffer less from data dependency and branch mis-predictions. However, the wrongly speculated result will be replayed so that more functional unit write back bus bandwidth will be occupied and draw back the performance gain. Table 5.1 lists the symbol used in our performance comparison.

PR	Prediction rate of the speculative logic
DR	data dependency rate for the instructions, i.e. the probability that data dependency exists between 2 adjacent instructions
FR	functional unit write back bus occupancy rate
P_B	overall branch miss rate, i.e. the probability that an arbitrarily selected instruction is a branch and the branch prediction is wrong
C_{Dstall}	Stalled cycle corresponding to DR
C_{Bstall}	Stalled cycle corresponding to P_B

Table 5.1 Symbol used in performance study

Notice that the overall branch miss rate is the product of branch miss rate and branch frequency. Since our goal is to evaluate data dependency, the branch prediction factor can be simplified as one term. Also we assume only in-order issue and commit. The reason for in-order assumption is because out-of-order machine is fairly complicated structure for theoretical analysis. And we can see in later chapter that the theoretical result does match the simulation result performed on the out-of-order microarchitecture.

Since we assume the same frequency for both models, the performance depends mainly on cycle per instruction (CPI). The general formula for CPI with data dependency and branch penalty is

$$CPI = 1 + DR * C_{Dstall} + P_B * C_{Bstall} \quad (5.1)$$

Assume C_{Bstall} is 3 cycles in either model for simplicity, and C_{Dstall} is 1. For baseline pipestage structure, we get

$$CPI = 1 + DR + P_B * 3 \quad (5.2)$$

For pipeline structure with speculative functions, there are four extreme cases considering data dependency and prediction rate factors,

(a) all instructions are independent and the prediction rate is 100%

(b) all instructions are independent and the prediction rate is 0

(c) all instructions are dependent and the prediction rate is 100%

(d) all instructions are dependent and the prediction rate is 0

For case (a) and (c), when all predictions are correct, there is no data dependency penalty.

For case (b), the verification logic will re-issue the instruction in the next cycle. This means extra write back slot for the re-issued instruction. While the impact of extra write back slot on the performance is complicated, we can approximate the relationship by following method: If the functional unit write back bus bandwidth occupancy rate (FR) is 100% for the original instructions, the extra instruction will always stalled one cycle. If the occupancy rate is 50% for the original instruction, the extra instruction will not be stalled and CPI will be the same as that of fully pipelined machine. For linear approximation, a straight line between this two point will be the function of C_{Dstall} vs. FR, so we get

$$C_{Dstall}(FR) = 2 * FR - 1 \tag{5.3}$$

For case (d), the analysis is similar to case (b). The difference is that since all instructions are dependent, the pipeline will be stalled for one cycle even when there is no limitation on write back bandwidth. This means when FR is 50%, the pipeline will be one cycle, and when FR is 100%, the pipeline will be stalled two cycles. By applying linear approximation, we get,

$$C_{Dstall}(FR) = 2 * FR \quad (5.4)$$

Combine all the cases together with branch prediction term, we get

$$CPI = 1 + (2 * FR - 1) * (1 - DR) * (1 - PR) + 2 * FR * DR * (1 - PR) + P_B * 3 \quad (5.5)$$

Let the speedup be the ratio of baseline CPI and speculative CPI. Figure 5.1, 5.2, 5.3 show the speedup when FR is 0.5, 0.8 and 0.95. These figures also assume P_B = 0.05, which means small branch mis-prediction impact. From the diagram, we can see that speedup rate increases with dependency rate and prediction rate monotonically. When functional unit occupation rate is high, the speculative performance is more likely to be sacrificed since replay instructions cause more penalties in write back bandwidth. For the case where FR is more than 50% in figure 5.2 and 5.3, when prediction rate or dependency rate is low enough, the speculative microarchitecture performance is even lower than that of the baseline processor. In an extreme case, when dependency rate is 0, the speedup increases with prediction rate, but maximum speedup rate is 1, means there is no speedup even with perfect prediction. If the prediction is not perfect, then performance actually decreases. The result shows that the speculative method only benefits the performance where there is high instruction dependency rate. In another extreme case when prediction rate is 0, the speedup increases with dependency rate but always lower than 1. This means some minimum prediction rate is required for the performance improvement in speculation method.

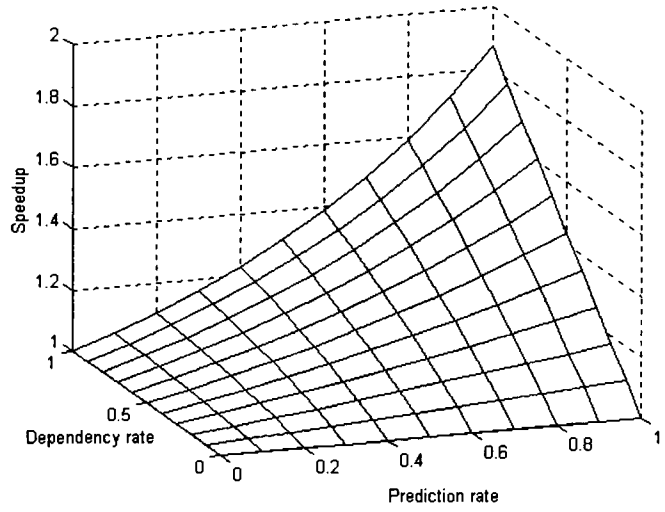


Figure 5.1 Speedup by speculative execution vs. PR and DR (FR=0.5)

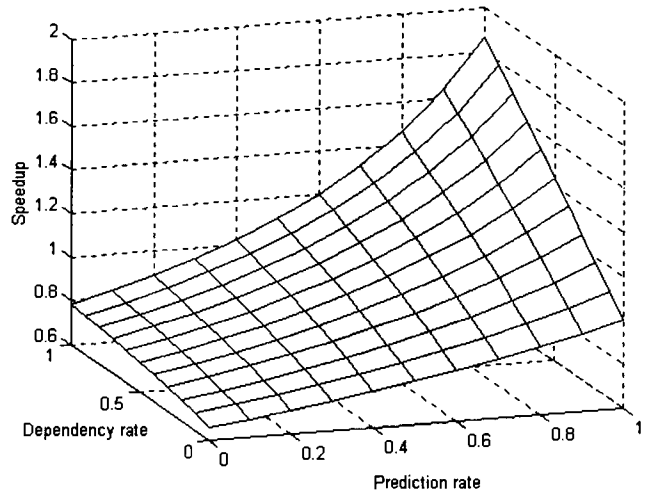


Figure 5.2 Speedup by speculative execution vs. PR and DR (FR=0.8)

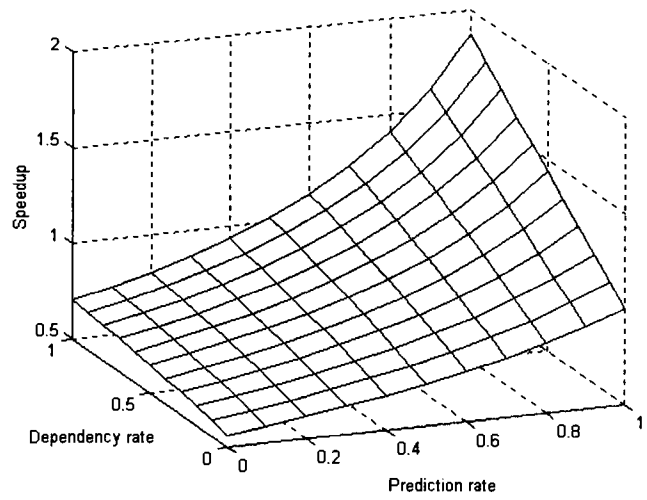


Figure 5.3 Speedup by speculative execution vs. PR and DR (FR=0.95)

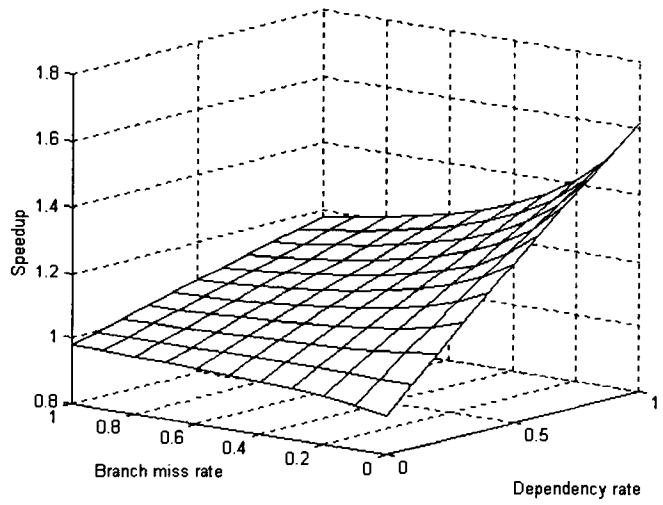


Figure 5.4 Speedup by speculative execution vs. P_B and DR (PR=0.85, FR=0.8)

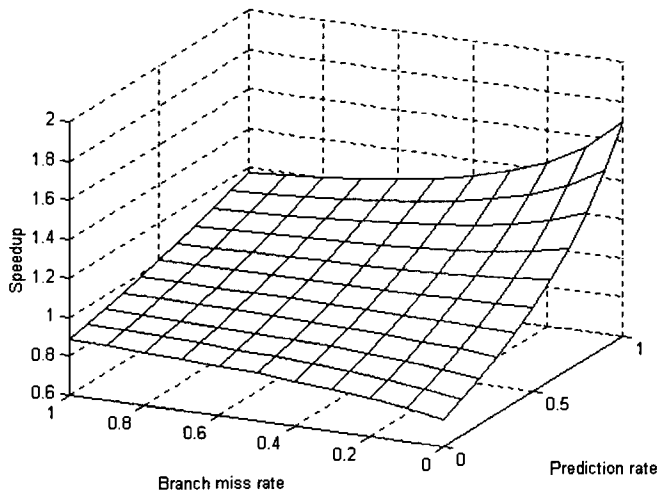


Figure 5.5 Speedup by speculative execution vs. P_B and PR (DR=0.85, FR=0.8)

Figure 5.4 shows the impact of overall branch mis-prediction rate and dependency rate to the performance when data prediction rate is high (0.85) and write back occupancy rate is medium (0.8). At the lower data dependency rate side when the speculative performance is low, the performance speedup increases when P_B increases. While at the higher data dependency rate side, when the speculative speedup is high, the speedup decreases when P_B increases. In the first case, data speculation suffers more on replay penalty than speculation performance gain due to lack of dependent instructions. In the second case, the baseline model suffers more on dependency stall penalty than performance gain from not replaying wrongly speculated instruction. Higher P_B causes more performance loss on both models at the same rate and thus neutralizes the performance loss by data dependency stall or data speculation penalty.

So the overall branch mis-prediction rate is in favor of the worse case model affected by dependency rate. Figure 5.5 shows the relationship of speedup in terms of P_B and PR. For the same reason, the overall branch mis-prediction rate will neutralize the impact by data prediction rate. This analysis means good branch prediction rate is important for data speculation speedup to take effect.

6. SIMULATION RESULT

We use SimpleScalar [22] tool set to compare the performance of our speculative microarchitecture with the baseline machine. Assume both models run with the same frequency. In the baseline machine, in order to keep up the frequency the cycle limiter logic blocks all take 2 cycles. While in the new speculative machine with approximation circuits, these same logic blocks take only 1 cycle. However the speculative machine will need to replay when the result is incorrectly generated and incur miss speculation (replay) penalty. Independent simulation experiment is performed for each of the above mentioned cycle limit logic - rename logic, issue logic and adder, with the assumption that only one of them is the main performance limiter.

6.1 SIMULATOR IMPLEMENTATION

Simplescalar tool set is a C platform that implements instruction trace simulation for superscalar in-order/out-of-order microarchitecture. It uses several sub-routings to simulate the basic stages of microprocessor: ruu_fetch() for instruction fetch (IF); ruu_dispatch() for instruction decode and register rename (ID); ruu_issue() for scheduling and execution (EX); lsq_refresh() for memory access (M); ruu_writeback() for instruction writeback (W); ruu_commit() for instruction commit. Since we want to simulate the effect of speculative rename logic, adder and issue logic, sub-routing ruu_dispatch(), ruu_issue() and ruu_writeback() need to be modified. In ruu_dispatch(), a smaller rename table is constructed and takes one cycle

to finish. A duplicated verification table is also build to validate the result in the next cycle. If the result is correct, the program goes on, otherwise, it asserts the VPF flag. In ruu_issue(), it checks that if an instruction has its entire source register data available and assigns a functional unit to it. If there are more ready to issue instruction that number of functional unit or writeback bandwidth, the priority logic will pick the oldest instruction to issue first. A smaller priority encoding logic is construct to look for instructions close to head side of RUU. We also set the speculative adder latency to be one cycle. In ruu_wirteback(), a written back instruction will search for its dependent instructions and wake the up. If the written back instruction has its VPF set, all its waken up instruction will set their VPF. If the former one proved to be wrong, it will be re-issued, so with all its dependent ones, and the re-issued written back instruction will clear its VPF since it is verified to be correct.

6.2 SIMULATION WITH VARIOUS MICROARCHITECTURE

We run eight integer benchmarks from the spec95 suite, using the reference input database. First, we set the RUU window size = 64, issue width = 4, integer adder number = 4, integer multiplier number = 1, and run 2 billion instructions for each benchmark. Then by shuffling the parameters: window size of 16, 32, issue width of 8, integer adder number of 8 and integer multiplier number of 2, we run each benchmark for 500 million instructions. These parameters are listed in Table 6.1 and 6.2. The speedup results and speculation accuracy are summarized in Figure 6.1-6.6. The speedup is basically the ratio of IPC with baseline machine normalized to one. Bars labeled HM in all figures are the harmonic mean over all the benchmarks simulated.

Instruction fetch	4 inst. per cycle
Instruction cache	16K byte, Direct mapped, 32 byte line, 6 cycle miss latency
Branch Predictor	Bimodel, 2048 BTB entries with 2 bit saturating counter
Issue mechanism	Out-of-order issue, commit at 4 operations per cycles, load may execute when all prior store addresses are known
Functional units	2 load/store, 4 fp adders, 1 fp MUL/DIV
FU latency (total/issue)	load/store 1/1, int ALU 1/1, int MUL 3/1, int DIV 29/19, fp adder 2/1, fp MUL 4/1, fp DIV 12/12, fp SQRT 24/24
Data cache	16K byte, 4 way set associate, 32 byte line, 6 cycle miss latency

Table 6.1 Common parameters of base simulator

	Issue width	RUU, LSQ size	Functional units (Integer)	Spec window	Spec carry chain	Inst. Count million
I4R64	4	64, 64	ALU=4,MUL=1	8	4	2000
I8R64	8	64, 64	ALU=8,MUL=2	8	4	500
I4R32	4	32, 32	ALU=4,MUL=1	4	4	500
I4R16	4	16, 16	ALU=4,MUL=1	4	4	500

Table 6.2 Parameters of four cases of base simulator

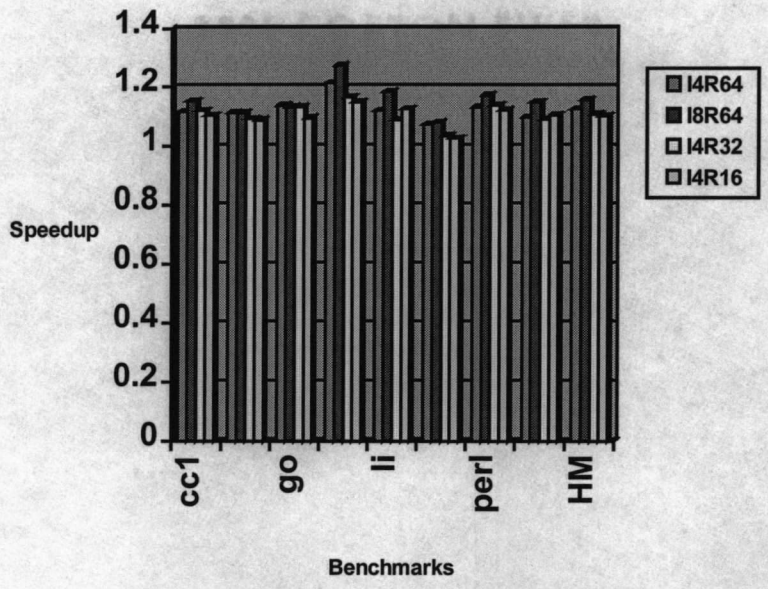


Figure. 6.1 Speedup by logic-level speculation of rename logic

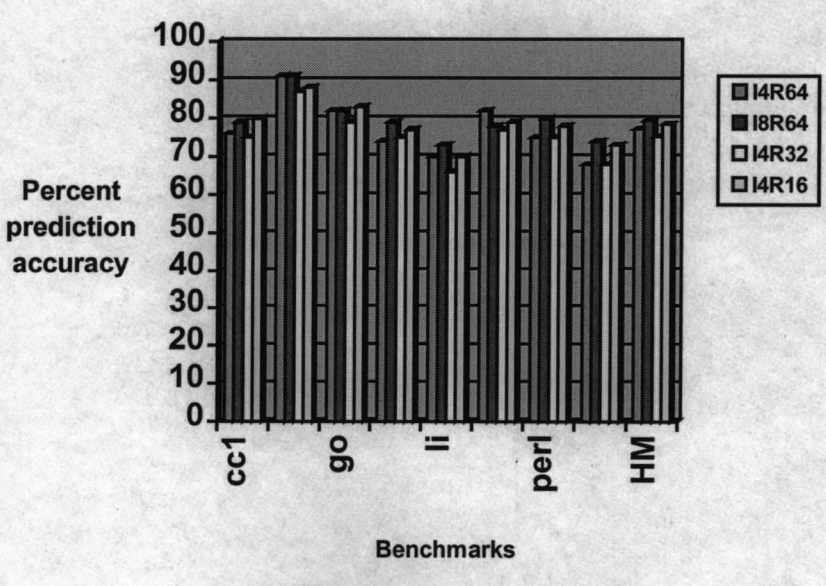


Figure 6.2 Percent of approximation accuracy for rename logic

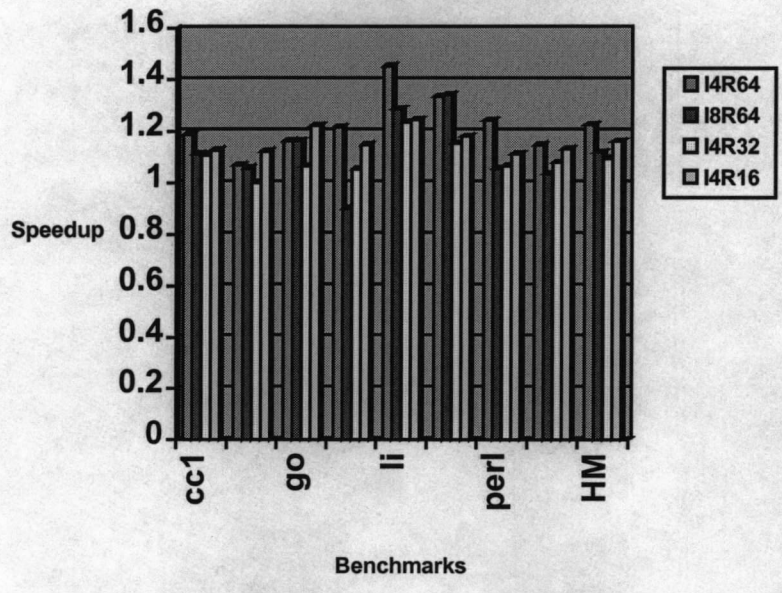


Figure 6.3 Speedup by logic level speculation of issue logic

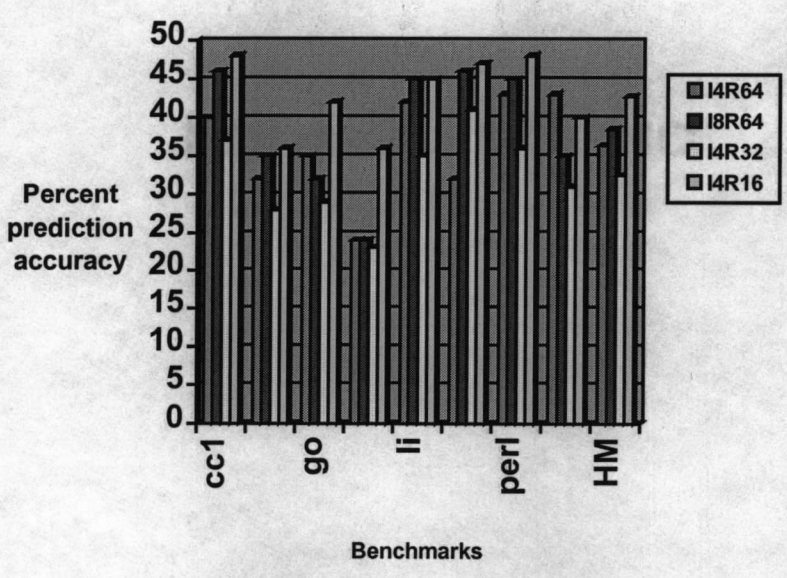


Figure 6.4 Percent of accuracy for the approximation issue logic

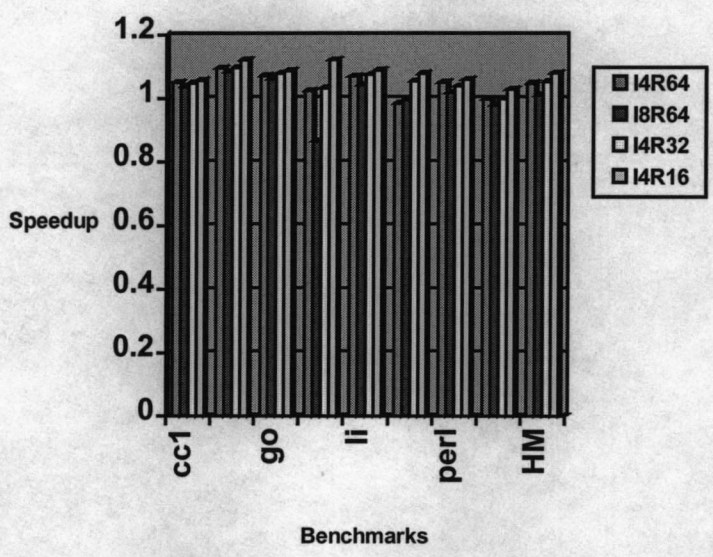


Figure 6.5 Speedup by logic level speculation with approximation adder

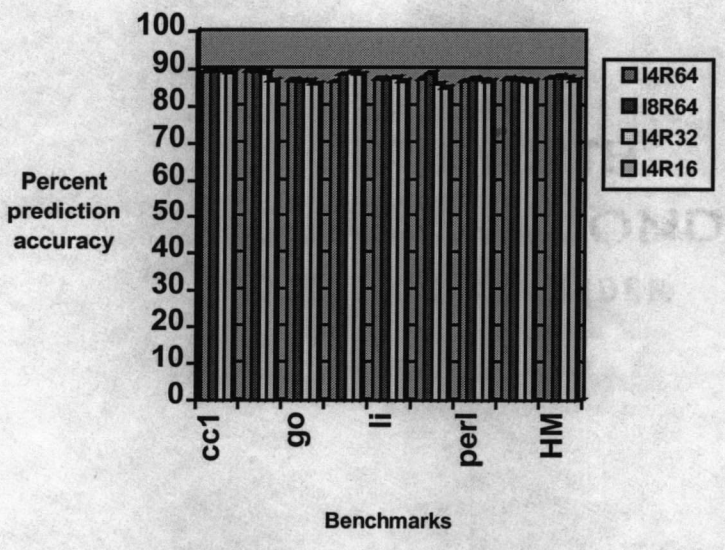


Figure 6.6 Percent of accuracy for approximation adder

From these diagrams, we can see that logic-level speculation method described does improve the overall performance of the new microarchitecture. For adder and rename logic, high prediction rate is also achieved. For adder speculation, the performance improvement is less than the other two speculations. This is because addition completes close to the back end of the machine, it is more likely to pollute the dependent instructions by false write back and cause more penalties. By reducing window size, the adder speculation performance relative to the baseline machine increased. This is because smaller number of independent instructions is available in a smaller issue window. So the speculation is more important and efficient to resolve dependencies. On the other hand, increasing issue width and number of function units degrades the relative performance, since wider issue width, larger window size and more functional units potentially cause larger instruction level parallelism, and the mis-speculation penalty will overcome the performance gain by resolving dependency chain. However, for rename and issue speculation, the speculative window size will change to match the baseline window size so that to achieve the circuit speedup of twice fast. This will compromise the relationship between relative performance and window size, issue width and functional unit. For case I8R64, which means wide issue, large window and more functional unit, the relative performance of ijpeg degrades a lot in issue and add speculation. The predication accuracy of issue speculation means the percentage of ready instructions in speculation window over the total ready instructions. It is as low as 24% for ijpeg, causing huge waste of execution bandwidth. Since ijpeg is a computational intensive program, it is full of independent data processing instructions, which means there are fewer dependencies than other

benchmarks. This explains the low performance gain with issue and adder logic-level speculation.

6.3 SIMULATION WITH TYPICAL MICROARCHITECTURE

We can also do the simulation by varying one parameter at a time in a typical wide issue microarchitecture. With the same common parameter in Table 6.1, and setting RUU window size = 64, issue width = 8. For the functional unit (FU) factor, we try to keep the same number of integer adder and make it unlimited but limit the writeback bandwidth. So the number of integer adder is set to 8. We consider three functional unit writeback bandwidth (FR) situations: (a) Extremely lack of FR -- 2; (b) moderately lack of FR -- 4; (c) Unlimited FR -- 8. Both in-order and out-of-order cases are simulated in order to correspond to our theoretical study in chapter 5. Three benchmarks (gcc, compress95 and ijpeg) are picked for the study. Instruction count is 50 million per each program. Table 6.3 and 6.4 show the simulation result.

Benchmark	Writeback bandwidth			Dependency rate	Prediction rate
	2	4	8		
	Performance speedup				
gcc	1.0	1.01	1.05	22%	88%
compress95	1.02	1.07	1.08	40%	90%
ijpeg	0.96	1.0	1.04	9%	96%

Table 6.3 Performance speedup vs. writeback width, dependency rate

Benchmark	Prediction rate	Performance speedup	Writeback width	Dependency rate
gcc	67%	0.985	4	22%
	79%	1.0		
	88%	1.018		
Compress95	64%	0.967	4	40%
	77%	1.0		
	89%	1.076		
ijpeg	83%	0.89	4	9%
	92%	0.98		
	97%	1.0		

Table 6.4 Performance speedup vs. prediction rate (FR=4)

In Table 6.3, the performance speedup of all three benchmarks increase with the increase of writeback bandwidth (FR). When FR=2, the bandwidth is very limited, we can see that speculation doesn't have much benefit. For ijpeg, the speculative performance is even worse than that of baseline machine. When FR=8, the bandwidth is high, so all three programs see 4% to 8% performance boost over baseline. The speculative performance is also related to data dependency rate. ijpeg has the lowest data dependent rate, so its relative performance speedup is less than the other two benchmarks, and can be worse than baseline with limited writeback width. Compress95 has the highest data dependent rate, so as the highest relative speedup.

In Table 6.4, we can see clearly that when prediction rate increase, the relative speedup also increase for all three benchmarks. When prediction rate is low enough, the speculative performance can even be worse than that of baseline machine.

From above analysis, we can see that the experimental result match what we have predicted in theoretical analysis in Chapter 5.

7. CONCLUSION

In this thesis, we first try to identify some possible cycle limiters in a superscalar microprocessor, namely adder, rename logic and issue logic and analyze their speed path. Then we propose a logic level speculation method – *approximation* to speedup these critical logic blocks. For adder, carry chain is generated by a subset of the input data. For rename and issue logic, we only target on a subset of instructions in the issue window. For adder and rename logic, the corresponding verification logic must be duplicated to detect the correctness of speculation. In case of false speculation, the instruction will be replayed. Our simulation of SPEC95 benchmarks with different window size, issue width and number of function units shows performance improvement for this newly proposed microarchitecture over the baseline machine. Our conclusion is that logic level speculation method is a potential way to speedup some cycle limiting logic structures and achieve better performance in wide issue superscalar microprocessor. Approximation method works better on programs with more dependencies than that with high ILP originally. The extra hardware cost both for duplicated logic blocks and verification logic is somewhat limited.

In the future, we can use approximation method on x86 instruction decoding, integer adder data bypassing or any potential circuit speed path in microprocessor design.

REFERENCES

[1] Philip G Emma and Edward S. Davidson, "Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance, " IEEE Transactions on Computers, VOL. C-36, NO. 7, July 1987

[2] James E. Smith, and Gurindar S. Sohi, "The Microarchitecture of Superscalar Processors," in Proceedings of the IEEE, Volume: 83 12, Dec. 1995, pp. 1609 –1624.

[3] P. Michaud, A. Seznec, and S. Jourdan, "Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors," in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 1999, pp. 2 –10.

[4] S. Dutta, and M. Franklin, "Control flow prediction schemes for wide-issue superscalar processors," IEEE Transactions on Parallel and Distributed Systems, Volume: 10 4, April 1999, pp. 346 –359.

[5] Sangyeun Cho; Pen-Chung Yew; Gyungho Lee, "Decoupling local variable accesses in a wide-issue superscalar processor," in Proceedings of the 26th International Symposium on Computer Architecture, 1999, pp. 100 –110.

[6] J. Farrell and T. C. Fischer, "Issue Logic for a 600-MHz Out-of-Order Execution Microprocessor," IEEE J. of Solid State Circuits, Vol. 33, No. 5, May 1998, pp. 707-712.

[7] S. J Patel, D. H. Friendly and Y. N. Patt, "Evaluation of design options for the trace cache fetch mechanism," IEEE Transactions on Computers, Volume: 48 2, Feb. 1999, pp.193 -204

[8] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in Proceedings of 22nd Annual International Symposium Computer Architecture, 1995, pp. 392 –403.

[9] C. B. Zilles, J. S. Emer and G. S. Sohi, "The use of multithreading for exception handling," in Proceedings of 32nd Annual International Symposium on Microarchitecture, 1999, pp. 219 –229.

[10] P. Marcuello, J. Tubella, and A. Gonzalez, "Value prediction for speculative multithreaded architectures," in Proceedings of 32nd Annual International Symposium on Microarchitecture, 1999, pp. 230 –236.

[11] S. Wallace, D. M. Tullsen and B. Calder, "Instruction recycling on a multiple-path processor," in Proceedings of Fifth International Symposium On High-Performance Computer Architecture, 1999, pp. 44 –53.

[12] J. -M. Parcerisa, and A. Gonzalez, "The synergy of multithreading and access/execute decoupling," in Proceedings of Fifth International Symposium On High-Performance Computer Architecture, 1999, pp. 59 –63.

[13] H. Akkary, and M. A. Driscoll, "A dynamic multithreading processor," in Proceedings of 31st Annual International Symposium on Microarchitecture, 1998, pp. 226 –236.

[14] S. Cotofana, and S. Vassiliadis, "On the Design Complexity of the Issue Logic of Superscalar Machines," in Proceedings of the 24th Euromicro Conference, 1998, pp. 277 –284.

[15] Subbarao Palacharla, Norman P. Jouppi, J. E. Smith, "Complexity-Effective Superscalar Processors," in Proceedings of the 24th Int. Symp. on Computer Architecture, June 1997.

[16] Subbarao Palacharla, Norman P. Jouppi, J. E. Smith, "Quantifying the Complexity of Superscalar Processors," Technical Report CS-TR-96-1328, University of Wisconsin-Madison, November 1996.

[17] R. Bechade et. al., "A 32b 66 MHz 1.8 W microprocessor," in Digest of Technical Papers of the 41st IEEE International Solid-State Circuits Conference, 1994, pp. 208 –209.

[18] D. Dobberpuhl et. al., "A 200 MHz 64 b dual-issue CMOS microprocessor," in Digest of Technical Papers of the 39th IEEE International Solid-State Circuits Conference, 1992, pp. 106 -107, 256.

[19] H. Sanchez et. al., "A 200 MHz 2.5 V 4 W superscalar RISC microprocessor," in Digest of Technical Papers of the 43rd IEEE International Solid-State Circuits Conference, 1996, pp. 218 -219, 448.

[20] T. Fischer, and D. Leibholz, "Design tradeoffs in stall-control circuits for 600 MHz instruction queues," ," in Digest of Technical Papers of the 45th IEEE International Solid-State Circuits Conference, 1998, pp. 232 -233, 442.

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

[22] D. C. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin Computer Science Technical Report #1342, June 1997.

[23] SPEC. "SPEC Benchmark Suite Release 1.0," Santa Clara, Calif., October 2, 1989.

[24] Geppert, L.; Perry, T.S. "Transmeta's magic show [microprocessor chips]" IEEE Spectrum, Volume

[25] Avinash Sodani and Gurindar S. Sohi, "Dynamic Instruction Reuse," Proceedings of the 24th International Symposium on Computer Architecture (ISCA), June, 1997.

[26] Avinash Sodani and Gurindar S. Sohi, "An Empirical Analysis of Instruction Repetition," in Proc. of 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), Oct 1998.

[27] Avinash Sodani and Gurindar S. Sohi, "Understanding the Differences between Value Prediction and Instruction Reuse," in Proceedings of 31st International Symposium on Microarchitecture (MICRO-31), Nov-Dec 1998.

[28] Y. Shintani et. al., "A Performance and Cost Analysis of Applying Superscalar method to Mainframe Computers," IEEE Trans. On Computers, Vol. 44, No. 7, July 1995, pp. 891-902

[29] Wei Hwang; Gristede, G.; Sanda, P.; Wang, S.Y.; Heidel, D.F, "Implementation of a Self-resetting CMOS 64-bit Parallel Adder with Enhanced Testability," IEEE Journal of Solid-State Circuits, Volume: 34 8, Aug. 1999, pp. 1108 –1117.

[30] L. A. Lev et. al., "A 64-b microprocessor with multimedia support ," IEEE Journal of Solid-State Circuits, Volume: 30 11 , Nov. 1995 , pp. 1227 -1238.

[31] Mike Johnson, *Superscalar Microprocessor Design*. Prentice Hall Series in Innovative Technology. 1991.

[32] C. Nagendra, M.J. Irwin, and R.M. Owens, "Area-time-power tradeoffs in parallel adders," Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on Volume: 43 10 , Oct. 1996 , pp. 689 –702.

[33] T. Lynch, and E. Swartzlander, "The redundant cell adder," in Proceedings. of the 10th IEEE Symposium on Computer Arithmetic, 1991, pp. 165 – 170.

[34] G. Sohi, "Instruction Issue Logic for High Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," IEEE T. on Computers, Vol. 39, No. 3, March 1990, pp.349-359.

[35] R. Ramachandran and S. L. Lu, "Carry Logic," Wiley Encyclopedia of Electrical and Electronics Engineering, Edited by John G. Webster, 1999.

[36] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in Proceedings of the 32nd Annual International Symposium on Microarchitecture, 1999, pp. 196 –207.

APPENDIX

PREDICTION RATE FOR APPROXIMATION ADDER

Theory: For N bits adder with k bits carry generator (predictor), the prediction rate is $(1 - 1/2^{k+2})^{N-k-1}$

For N bits adder with k bits carry generator (predictor), for each bit i, assume inputs are a_i, b_i and carry in c_i, (i = 0, 1, 2, …, N-1)

$s_i = a_i$ xor b_i xor c_i

Notation:
$P^k(c_i\ Y)$, the probability of carry c_i is correct with k bit predictor
$P^k(c_i\ N) = 1 - P^k(c_i\ Y)$, the probability of carry c_i is wrong with k bit predictor
$P(c_i=c)$, the probability of carry c_i equal c without prediction
$P^k(c_{i+1}\ Y|c_i\ Y)$, the probability of carry c_{i+1} is correct with k bit predictor when carry c_{i+1} is correct with its own k bit predictor

Since s_i is correct if and only if c_i is correct with k bit predictor

$P^k(s_i\ Y) = P^k(c_i\ Y)$

Their probabilities are equal.

Lemma 1. $P^k(c_i\ N) = (1/2^k) * P^k(c_{i-k}=1)$ (1)

Proof:

Let k=2,
$(c_3\ Y) \Rightarrow a_2b_2 = (00|01)$ OR
$(01|10)$ AND
$(a_1b_1 = (00|11)$ OR
$(01|10)$ AND $c_1=0)$

$(c_3\ N) \Rightarrow a_2b_2 = (01|10)$ AND
$(a_1b_1 = (01|10)$ AND $c_1=1)$

So $P^2(c_3\ N) = (1/2) * (1/2) * P(c_1=1)$

Lemma 2. $P(c_{i-k}=1) = (1/4) + (1/2) * P(c_{i-k-1}=1)$ (2)

Proof:

$(c_3=1) \Rightarrow a_2b_2 = (11)$ OR $(01|10)$ AND $c_2 =1$

So $P(c_3=1) = (1/4) + (1/2) * P(c_2=1)$

Lemma 3. $P^k(c_i\ Y) = 1 - 1/2^{k+1} + 1/2^{i+1}$ (3)

Proof:

From (2), and also $P(c_0=1)=0$, which means no carry at bit 0,
We can get
$P(c_{i-k}=1) = (1/4) * (1 + 1/2 + 1/4 + \ldots + 1/2^{i-k-1})$
$= (1/2) * (1 - 1/2^{i-k})$

insert into (1)

$P^k(c_i\ N) = (1/2^{k+1}) * (1 - 1/2^{i-k})$

$=> P^k(c_i\ Y) = 1 - P^k(c_i\ N)$
$= 1 - 1/2^{k+1} + 1/2^{i+1}$

Lemma 4. $P(c_{i+1}\ Y|c_i\ Y) = 1 - (1/2^k) * (1/4)$ (4)

Proof:

Let k=3,

$c_5 = f(a_4,b_4,a_3,b_3,a_2,b_2)$
$c_4 = f(a_3,b_3,a_2,b_2,a_1,b_1)$

$(c_5\ Y) => a_4b_4 = (00|01)$ OR
$(01|10)$ AND
$(a_3b_3 = (00|11)$ OR
$(01|10)$ AND
$(a_2b_2 = (00|11)$ OR
$(01|10)$ AND $c_2=0)$

$(c_4\ Y) => a_3b_3 = (00|01)$ OR
$(01|10)$ AND
$(a_2b_2 = (00|11)$ OR
$(01|10)$ AND
$(a_1b_1 = (00|11)$ OR
$(01|10)$ AND $c_1=0)$

$P^3(c_5\ N) = (1/2) * (1/2) * (1/2) * P(c_2=1)$
$P^3(c_5\ N|c_4\ Y) = (1/2) * (1/2) * (1/2) * P^3(c_2=1|c_4\ Y)$

The only condition that satisfy (c_2=1) AND (c_4 Y) is when (a_1b_1=11)
So $P^3(c_2=1|c_4\ Y) = P(a_1b_1=11) = 1/4$

So $P^3(c_5\ N|c_4\ Y) = (1/2^3) * (1/4)$
$\Rightarrow P(c_{i+1}\ Y|c_i\ Y) = 1 - P^3(c_5\ N|c_4\ Y)$
$= 1 - (1/2^k) * (1/4)$

Starting from any bit i, the probability of c_i, c_{i+1}, ... c_{N-1} of the N bits adder are all predicted correctly with k bits predictor is

$$P^k(N,i) = P^k(c_i\ Y) * P(c_{i+1}\ Y|c_i\ Y) * P(c_{i+2}\ Y|c_{i+1}\ Y) * \ldots * P(c_{N-1}\ Y|c_{N-2}\ Y)$$

Since it is always predicted correctly for bit starting from k and below, the total prediction probability is

$P^k(N) = P^k(N,k+1)$, insert (3) and (4)

$$P^k(N) = (1 - (1/2^k)*(1/4))^{N-k-2} * (1 - 1/2^{k+1} + 1/2^{k+2})$$
$$= (1 - 1/2^{k+2})^{N-k-1}$$

QED