Branch With Masked Squashing In A Superpipelined Prolog Processor

Ching-Long Su and Alvin M. Despain

CENG Technical Report 93-46

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-9143

October 1993

Branch With Masked Squashing In A Superpipelined Prolog Processor

Ching-Long Su and Alvin M. Despain

Advanced Computer Architecture Laboratory
University of Southern California
Los Angeles, CA 90089-2562, USA
Phone: (213) 740-9143
E-mail: csu@usc.edu

October 15, 1993

Abstract

The performance of a superpipeline processor heavily relies on its branch performance. Traditional branch strategies used in pipeline processors are delayed branches and branch with squashing. Delayed branches use safe instructions to fill delay slots. However, for a deeply pipelined processor, a compiler may not be able to find sufficient safe instructions to fill the branch delay slots. Branch with squashing takes advantage of using instructions in target basic blocks to fill the branch delay slots. However, the penalty of branch misprediction is large in superpipelined processors.

In this paper, we proposed a novel branch scheme, named branch with masked squashing, which take advantage of both delay branch and branch with squashing. The basic idea is to fill delay slots with safe instructions which may come from above or after the branch. For the remaining unfilled delay slots, we fill with instructions from the predicted target path. In the case of misprediction, only unsafe instructions are annulled. The safe instructions in branch delay slots are always executed. Simulation results show that this branch strategy performs better than traditional delayed branch and branch with squashing.

key words: micro-processor, branch scheme, and superpipeline.

1. Introduction

The speed and density of VLSI circuits has been significantly increased due to recently improved technology. Research in computer architecture is motivated to utilize large numbers of logic elements to improve performance. One method is to replicate multiple functional units in a single chip as in VLIW and superscalar architecture. We will explore in this paper, another possibility, superpipelined architecture [Jouppi 89].

The performance of a superpipelined processor depends critically on its branch performance. A deep pipeline design can potentially create extremely high throughput since the each pipeline stage is short. However, during program execution, several factors may break the pipeline flow. This results in less throughput than we expected. We focus here on pipeline breaks, which result from branch operations. In a pipeline processor, a likely branch target is usually predicted in order to keep fetching instructions into pipeline stages. Since the likely target instructions are fetched before the branch conditions are evaluated, these instructions may be squashed later when this branch goes another direction. The squashed instructions constitute a branch penalty. The branch penalty of a superpipelined processor is obviously larger than a design with a shorter pipeline.

In this paper, we mainly focus on the design issues of a branch on a superpipelined Prolog processor, named the Southern California Abstract Machine (SLAM). The goal of the SLAM project was to achieve the highest possible performance in a given technology, a 0.8u, 3-level metal CMOS process, with a deep pipeline design, to build a single processor with a peak rate of 33 million logical inferences (or 200 MIPS). This superpipelined processor not only supports generic RISC-like instructions, but also a powerful instruction set for effectively performing logic programming as in Prolog. During the design process of the SLAM chip, we considered two design goals simultaneously: high performance and code compaction.

In this paper, we investigate several branch strategies implemented in a superpipelined Prolog processor.These branch strategies include traditional delayed branch and branch with squashing [Smith 81], [Lee 84], [McFarling 86], [Dubey 91]. We also proposed a novel branch strategy, named branch with mask squashing, which allow the compiler to fill the delay slots with useful instructions as much as possible and fill predicted target instructions in the remaining delay slots. This scheme shows better performance than the traditional delayed branch and branch with squashing. To further study the impact of these branch strategies on the performance and static code size of various superpipeline processors, we also conducted experiments with variations of these branch strategies on SLAM with branch delay slots ranging from one to five.

The rest of the paper is organized as follows: Section 2 presents the high-level SLAM architecture. Section 3 briefly shows the methodologies used in this paper to evaluate branch strategies implemented in SLAM. The software environment (compiler, profiler, and simulator) are also introduced in this section. The detail of each branch strategy and the comparison among branch strategies are presented in Section 4. In Section 5, we provide analytical models for evaluating

performance and static code size of branch strategies on different number of branch delay slots. The results from analytic models are matched with the actual simulation results with the number of branch delay slots ranging from one to five. Finally, conclusion and final remarks are offered in Section 6.

2. SLAM Architecture

In this section, we briefly discuss the experimental machine used to evaluate various branch strategies. The SLAM architecture contains a CPU core, instruction cache, data cache, segment table, and fast tag logic. The CPU core contains a data path and a control path. Figure 1 shows a block diagram of SLAM. The control path has PC-chain, clock circuitry, and pipeline control circuits. The data path consists of a 5-port register file which contains 32 general registers, a fast carry-look-ahead adder, a barrel shifter, a logic unit, and circuits for data forwarding. The instruction cache is a 4KB, virtual-addressed, direct-mapped cache with 16-byte block. The data cache is a 4KB, virtual-addressed, write-back, direct-mapped cache with 16-byte block. The segment table has 64 entries.

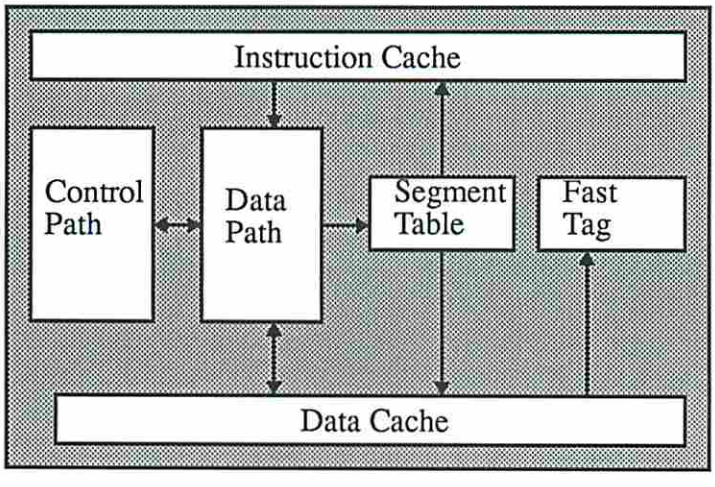


Figure 1 Block Diagram of the SLAM architecture

SLAM pipeline stages can be categorized into five phases based on their functionality. These five pipeline phases are Instruction Fetch (IF), Instruction Decode (ID), ALU EXecution (EX), Memory access (M), and Write Back (WB). Each pipeline phase contains two pipeline stages. Figure 2 shows the pipeline structure of SLAM. The SLAM processor is a 8-stage pipeline design in which pipeline stages are laid out as <IF1,IF2,ID1,ID2,EX1/M1,EX2/M2,WB1,WB2>. M1 and M2 stages are parallel with EX1 and EX2 stages. In other words, this design will not be able to support instructions which activate both ALU stages and memory stages. Pipeline delay slots in the SLAM pipeline structure are also presented in Figure 2. There is one cycle delay for load instructions and any following instructions which use the loaded data. There is a one cycle delay of store instruction and any following instructions which use the data address bus (e.g. load and store instructions), since the store instructions use data address bus for two consecutive cycles

(one for tag read and one for write operations). There is a delay cycle for compare instructions and following branch instructions, since the branch direction can be decided in the pipeline stage ID2, but the compare result can not be ready until the pipeline stage EX2. Finally, the branch delay slot in the SLAM is three cycles. There are three pipeline stages between the pipeline stage IF1 and ID2 where the branch decision is determined.

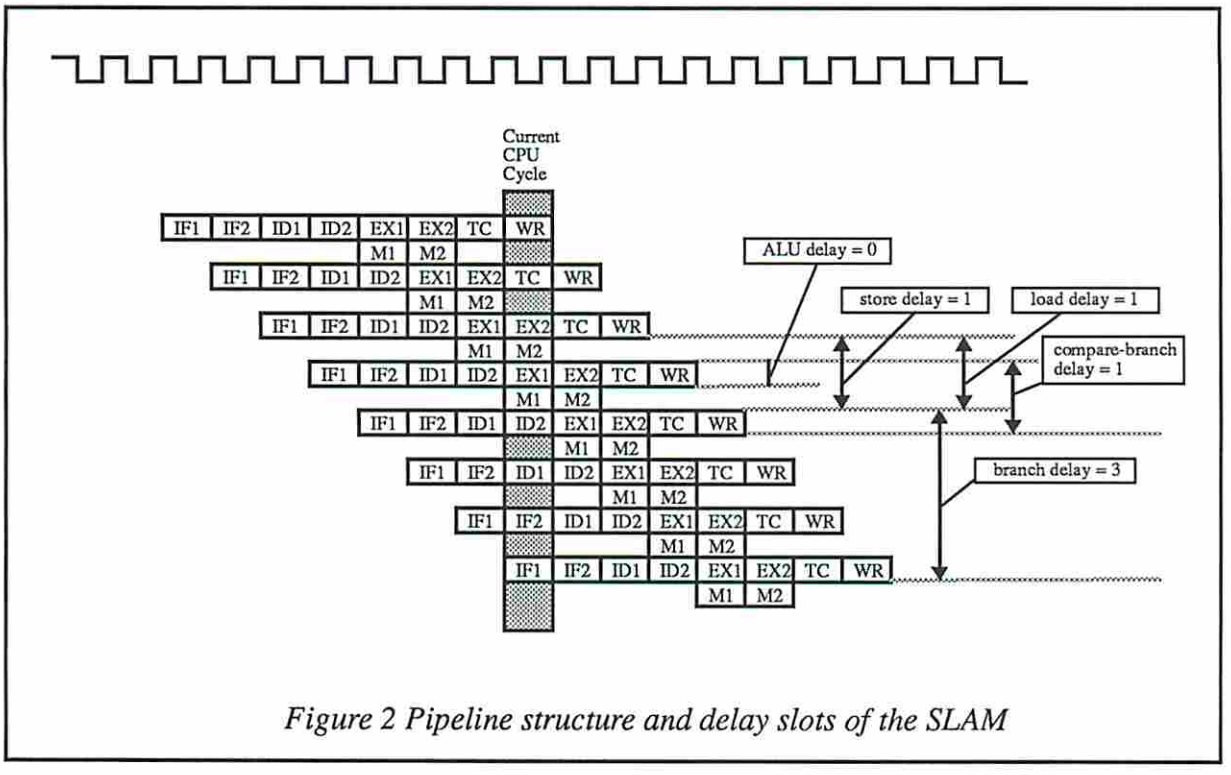


Figure 2 Pipeline structure and delay slots of the SLAM

3. Methodology

We consider experiments using a computer architecture evaluation system for SLAM which consists of a compiler front-end, an optimizing compiler backend, a parameterized instruction simulator, and a trace profiler.

The benchmark programs are compiled through the Aquarius Prolog compiler front-end into an intermediate representation (BAM code) [Van Roy 92]. The instruction set of the BAM code is designed in a way such that a Prolog program can be optimally compiled by the Aquarius Prolog compiler front-end, and a new platform for a target machine can easily be generated. The Aquarius optimizing backend [Su 92] performs further code translation, register allocation, instruction scheduling, peephole optimization, and assembly to a benchmark program in the BAM code. The output of the Aquarius optimizing backend is an object code of the target machine. In this paper, we will use the Aquarius optimizing backend for generating the SLAM object code.

An parameterized retargetable instruction-level simulator further takes the object code as an input and generates an execution trace of a given benchmark program in object code. This trace

will be read by a trace profiler which will generate run-time profiling information. This run-time profiling information can be read by users in order to monitor the quality of designs or read by a compiler backend in order to generate code based on this profiling information. Profiling information will be very useful when similar execution patterns can be found among program executions with different data sets.

3.1 Benchmarks

The benchmark programs used in this study are described in Table 1. These benchmark programs are selected from the Aquarius benchmark suite [Haygood 89]. Execution cycles for these benchmark programs ranges from thousands to millions of cycles. Applications of these benchmark programs includes list manipulation, data base queries, theorem provers, and computer language parsers.

Benchmark	Line	Instructions	Description
nreverse	145	1,138,655	Naive reverse of a 30-element list
qsort	225	4,560	Quicksort of a 50-element list
tak	397	1,064,197	A parser for language translation
circuit	1315	4,504,940	VLSI module generator
query	4395	4,487,201	Query a data base
zebra	985	350,761	A data base query
boyer	9918	27,494,723	An extract from a Boyer-Moore theorem prover
chat	114312	18,883,712	A small subset of English for database querying

Table 1 Benchmark programs

Figure 3 shows the dynamic instruction mix among these benchmark programs. The dynamic instruction mix is defined as the instruction mix from the execution trace. On average, branch instructions contributes 12.12% of the total execution trace.The call, return, and jump instructions all together make up 8.30% of the execution trace; the load and store instructions make up another 26.55% of the execution trace. The remaining 53.03% of the execution trace is arithmetic, logical, and move instructions. For a pipeline processor with three-cycle branch delay, the branch performance could cost as much as a 36% performance degradation.

Figure 4 shows static instruction mix among these benchmark programs. The static instruction mix is defined as the instruction mix of the static code. on average, branch instructions contributes 6.95% of the static instruction mix. The percentage of call, return, and jump instructions in the static instruction mix is 9.06%; the percentage of load and store instructions in the static instruction mix is 38.22%. The remaining 45.77% of the static instruction mix are arithmetic, logical, and move instructions. For a pipeline processor with three-cycle branch delay, a

bad branch strategy may cost as much as 27% in static code size.

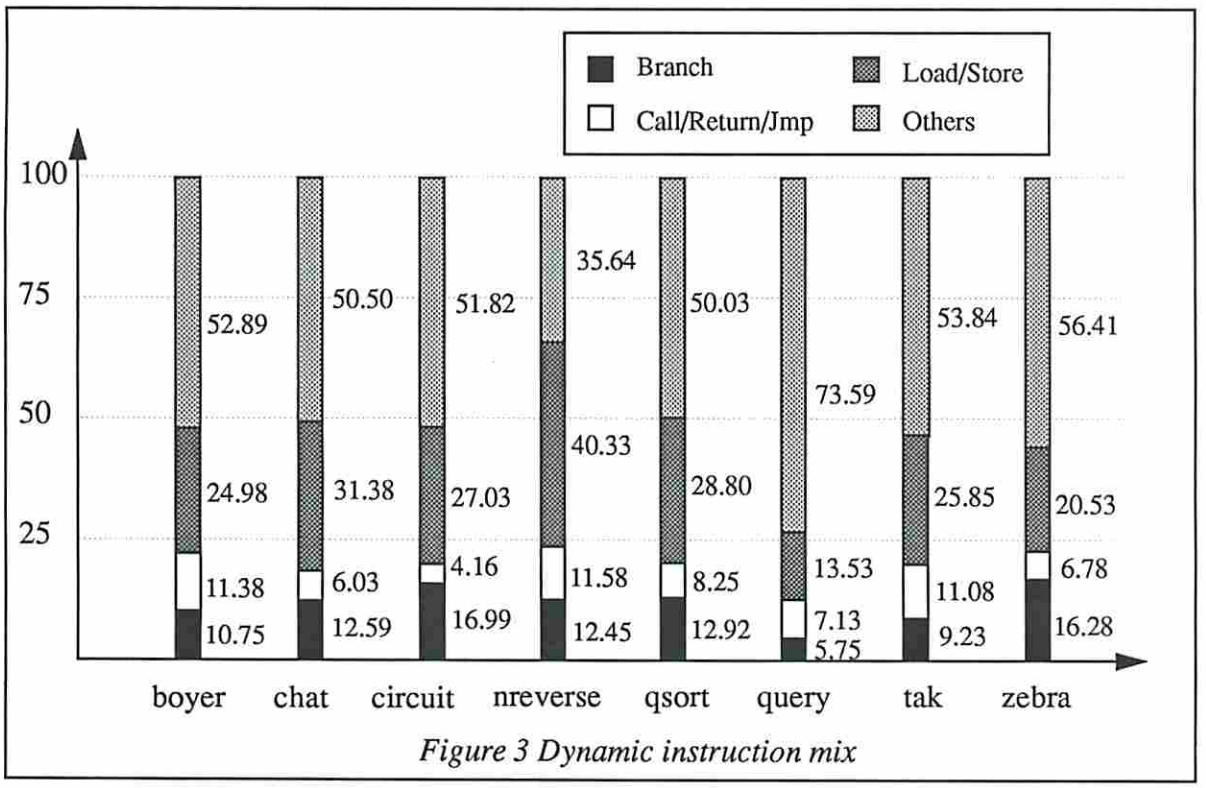


Figure 3 Dynamic instruction mix

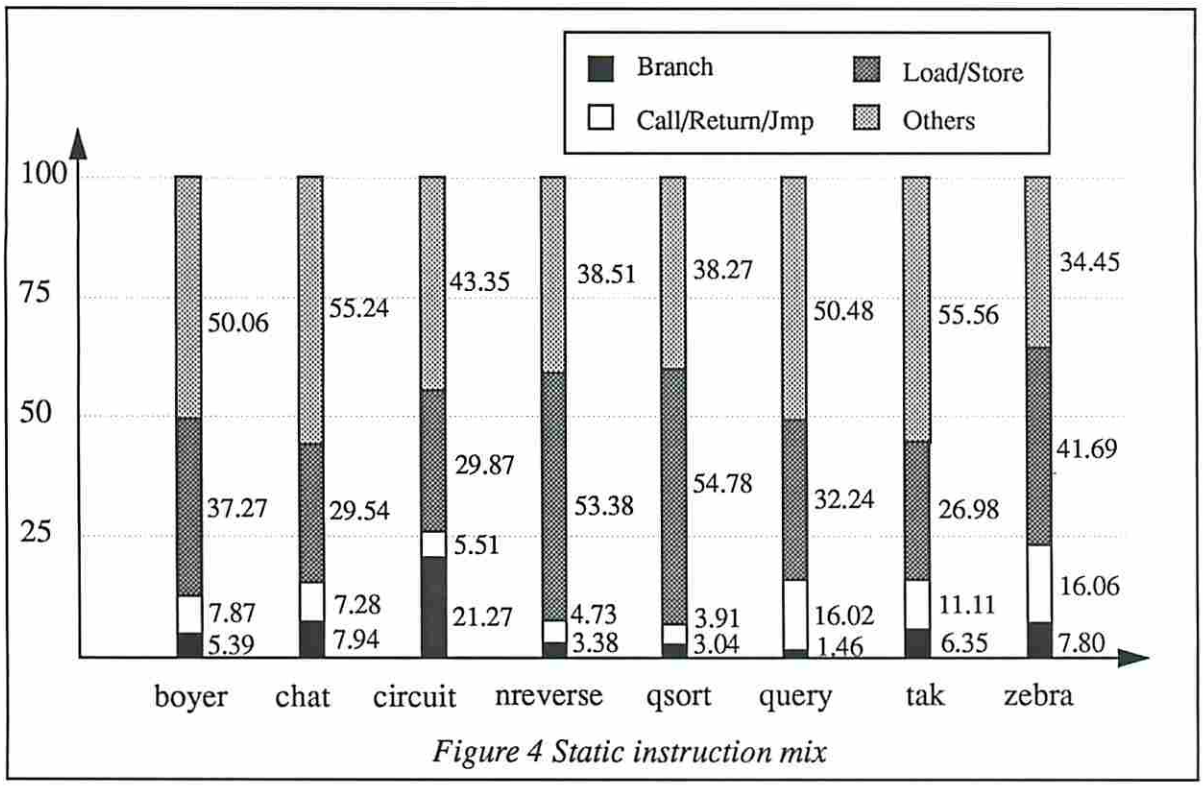


Figure 4 Static instruction mix

3.2 Retargetable Instruction scheduler

The retargetable instruction scheduler in SLAM, named SUPER-REORDERER, is used to fill branch delay slots. In SUPER-REORDERER, we apply a list scheduling algorithm. Given the DAG (Direct Acyclic Graph) of a basic block, the code generator of SUPER-REORDERER keeps a list of instructions that are ready to be executed without causing delays. In each iteration, the code generator selects an instruction from this list and also updates the list. Various heuristics are used to help the code generator make a better code schedule.

A post-branch instruction scheduling process is applied at the end of the code generation phase. Based on the branch instructions described in the target machine specification, the post-branch instruction scheduler finds suitable instructions, such as those instructions prior to that branch instruction, fall-through instructions, or those instructions in the target basic block, to feed into the branch delay slots. If no suitable instruction can be found, the code generator simply puts "NOPs" into the branch delay slots. For filling delay slots with instructions from target blocks, we extended a more advanced target inlining algorithm, called restricted target inlining. The original target inlining algorithm is proposed by [Hwu 92]. Hwu's algorithm assumes every instruction (including a branch instruction) in target blocks can be used to fill delay slots. However, in a practical machine, we may not be able to move local branch instructions very far away from their original basic block due to the restriction of their instruction fields representing target addresses. Therefore, instructions in target basic blocks may not be moved around freely. Because of this property, Hwu's algorithm needs to be modified in order to maintain correctness. More details of

the restricted target inlining used in SLAM are shown in [Su 93].

3.3 Parameterized Instruction-level Simulator

In order to correctly evaluate the performance of various target architectures, we wrote an instruction-level simulator. This simulator assumes memory accesses of all target machines are miss-free. This simplifies the machine specification and also allows efficient simulation. The output of the instruction-level simulator includes total execution cycles, total pipeline stall cycles, total annulled cycles due to branch misprediction, execution cycles for each instruction, and various execution traces for the trace profiler and cache simulator.

3.4 Profiling System

We wrote the profiling system used in this study. Previously existing profiling systems (e.g. *pixie*, *prof*/*gprof*, and *globin* rely on a technique best described as invasive profiling of the compiled code. This type of profiling inserts additional instructions into the object code of the program being profiled. These instructions, called instrumentation code, provide the mechanism whereby data about the execution of the program can be collected. During some intervals of program execution, the instrumentation code causes an interrupt and a jump to a special routine. The routines that are invoked by instrumentation code to collect and analyze the data are referred to as the profiling code. All these profiling systems need the compiler to insert instrumentation code, which may add a tremendous amount of complexity to the compiler. In addition, adding instrumentation code to the benchmark programs may make the optimal instruction scheduling very hard to achieve because instrumentation code skews data.

The profiling system of SLAM is a trace-driven profiling system. Instead of adding instrumentation code into the benchmark programs and providing profiling code, we directly add instrumentation code into the simulator. The simulator than produces the profiling trace which the trace-profiler takes as an input and gathers statistics. The advantages of this trace-driven profiling system are (1) The compiler does not need to be changed in order to insert instrumentation code into benchmark programs. (2) An optimizing instruction scheduler can reorder instructions without interference from instrumentation code. (3) Compiled benchmark programs do not need to be recompiled when we want to collect different profiling data. Only the simulator and trace-profiler need to be modified for collecting new profiling data. (4) Finer grain profiling data can be retrieved since the trace-driven profiling system directly monitors each individual instruction.

3.5 Measurement

We use an experimental model based on the performance and static code size of each branch instruction to obtain measurements. The performance is measured by CPB (Clock Per Branch). The CPB is defined as the number of cycles for a branch which includes the cycle for the branch instruction itself and the cycles for instructions which are not useful. The useful instructions are defined as normal instructions which belong to the program when the branch delay slot is zero. We do not consider instructions executed due to branch misprediction or NOPs as useful

instructions. Let the number of cycles for a program running on a machine with no branch delays be C_{zero}; let the number of cycles for a program running on the same machine with N-cycle branch delays be C_n. We assume the number of branch instructions in this program execution is N_b. CPB can be calculated by the following equation.

$$CPB = [(C_n - C_{zero}) / N_b] + 1$$

The static code size of a branch is measured by IPB (static Instructions Per Branch). The IPB is defined as the number of static instructions, which includes the branch instruction itself and any additional instructions inserted in the branch delay slots, associated with a branch instruction. Let the number of static instructions of a program compiled for a machine with no branch delay slots be S_{zero}; let the number of static instructions of a program compiled for the same machine machine with N-cycle branch delay slots be S_n.We also assume the number of branch instructions in the static code of this program is S_b. IPB can be calculated by the following equation.

$$IPB = [(S_n - S_{zero}) / S_b] + 1$$

For example, using a branch strategy on a machine with 3-cycle branch delays, a program has the following statistics,:

C_3 = 1,327,986 cycles
C_{zero} = 1,090,642 cycles
C_b = 178,489

S_3 = 7,895 instructions
S_{zero} = 7,048 instructions
S_b= 1,324

The performance of this branch strategy is 2.32 CPB. The static code size of a branch becomes 1.64 IPB.

4. Reduce Branch Penalties

To reduce the cost of a branch, we must minimize the cost of branch delays. We first briefly discuss traditional branch strategies for reducing the cost of branch delays used in contemporary pipeline microprocessors. These branch strategies include delayed branch, branch with squash, prophetic branch [Srivastava 93]. We then introduce a new branch strategy, called branch with masked squash, which is designed especially for achieving our design goals. Implementation of this new branch is also discussed.

4.1 Base Models

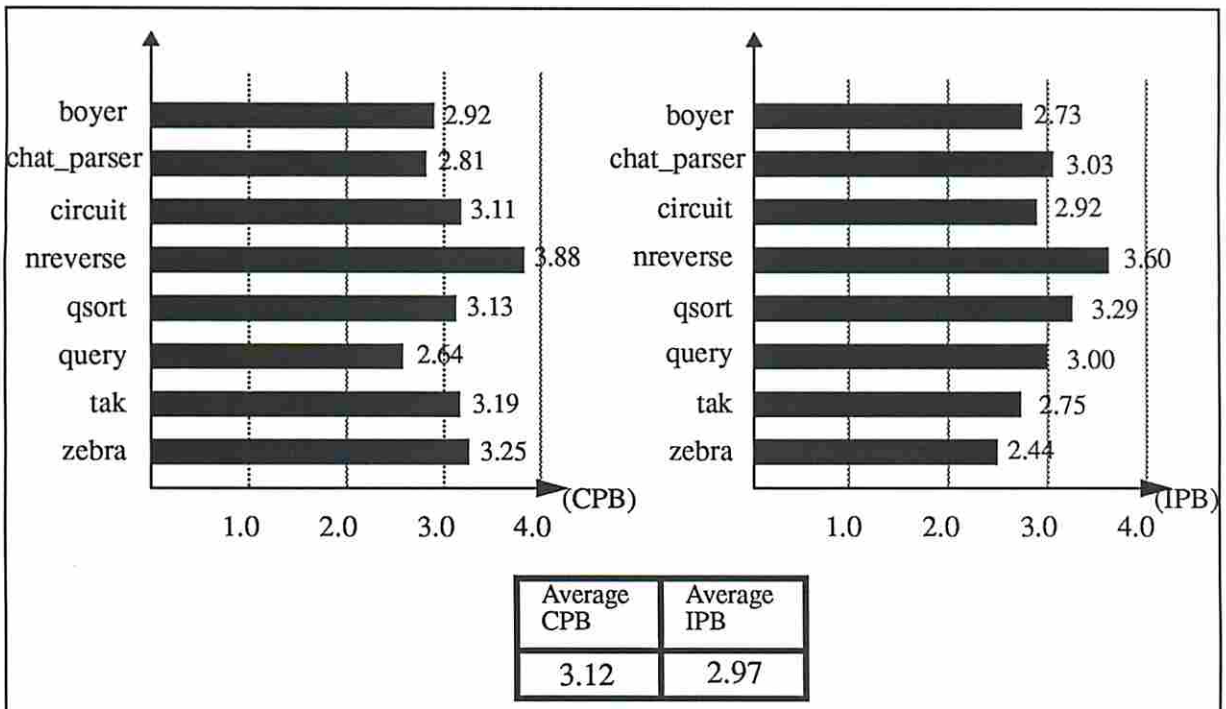
A simple branch strategy, name BASE, simply adds three NOP (No OPerations) instruc-

tions into the delay slots of any branch instruction. Another simple branch strategy, named STALL, does not add any instructions into delay slots. The machine simply stalls until the branch decision has been made. Both BASE and STALL are used as the base models for comparison to other branch strategies in terms of the performance and static code size. BASE has the worst branch performance (4.0 CPB) and static code size (4.0 IPB). STALL has also the worst performance (4.0 CPB). However, it has the best static code size (1.00 IPB) since code is not duplicated in STALL.

4.2 Delayed Branch

Instructions in the delay slots of a delayed branch are always executed. In order to reduce the cost of branch, a compiler must find sufficient safe instructions to fill these branch delay slots. Instructions in these branch delay slots can come from either above the branch or target basic blocks which do not affect program semantics regardless of which direction the branch goes. If there are not sufficient instructions that can be found by compiler to fill branch delay slots, NOPs are inserted into these unfilled branch delay slots. Delayed branches has been used successfully on several RISC machines. These machines, including the IBM-801, MIPS [MIPS 86], and VLSI-BAM [Holmer 92], have a one-cycle branch delay. The cost of delayed branch ranges from 1.3 to 1.5 cycles.

Figure 5 shows the branch performance and static code size of delayed branch with three-cycle branch delays. We assume the compare and branch operations are separate instructions. The average CPI of delayed branch is 3.12. Equivalently, only 0.88 safe instructions, on average, can be found by the compiler to fill the branch delay slots. The same phenomenon applies to the static code size. There are on average of 1.97 NOPs placed in the branch delay slots due to not enough save instructions being found.



Average CPB	Average IPB
3.12	2.97

Figure 5 Performance and static code size of delayed branch

4.3 Branch With Squashing

Branch with squashing takes advantage of filling delay slots with safe instructions and instructions from target basic blocks. In order to effectively use branch with squashing, the compiler must predict branch outcomes during compile-time. During run-time, if the prediction is correct, then instructions in delay slots are executed in the correct order. In other words, there will not be any branch penalty in this case. In the case of branch misprediction, instructions in the delay slots belong to the wrong target basic block. These instructions should not be executed. These instruction should be squashed by hardware.

The performance of branch with squashing depends on branch prediction at compile-time. A simple static branch prediction method is to predict branch taken if the branch direction is backward and not taken if the branch direction is forward. The reason of using this prediction strategy is that backward branches are likely to be branches for a loop and forward branches are likely to be for a fork operation. This branch prediction strategy has been successfully used in RISC processors. More advanced static branch prediction relies on run-time profiling. Profile information from previous runs can be very helpful for the next run, although the data sets used in the runs are different [Fisher 93].

We experiment with the branch with squashing strategy on our superpipelined processor.

For a correct branch prediction, the cost of a branch is only one cycle. For a misprediction, the cost of the branch becomes four cycles, in which one cycle is for the branch instruction itself and three cycles are for squashing unsafe instructions. To investigate the use of both delayed branch, branch with squashing, and the impact of profiling techniques, we experimented four different models as described below.

(I) Branch with squashing only (BWS)

For a backward branch, we copy instructions from branch target basic blocks into the delay slots. For a forward branch, we do not copy instructions. The delay slots are filled from instructions below this branch.

(II) Delayed branch and branch with squashing (BWS_DB)

Similarly to the previous model, we predict branch taken if it is a backward branch and not taken if it is a forward branch. We combine the previous model with a delayed branch scheme. For the case of a branch where the compiler can find sufficient instructions to fill all the delay slots, we use delayed branch. Otherwise, we use branch with squashing.

(III) Profile branch with squashing (P_BWS)

The model mainly uses profile information to reduce the overhead of branch with squashing. The compiler predicts a branch direction based on this profile information of previous runs.

(IV) Profile delayed branch and branch with squashing (P_BWS_DB)

Similarly to P_BWS, profile guided branch with squashing, we use profile information to reduce the overhead of branch with squashing. In addition, the compiler will use delayed branch if sufficient safe instructions can be found to fill the delay slots. Otherwise, the compiler will use branch with squashing based on the proifile information.

In Figure 6, we show the simulation results of the four varieties of branch with squashing. The performance and static code size of BWS is 2.27 CPB and 1.16 respectively. Compared to DB, which has 3.12 CPB and 2.97 IPB, the improvement of performance and static code size by using BWS are 37.44% and 60.94%. This improvement is not surprised because on average only 1.03 safe instructions can be found by the compiler to fill in the branch delay slots. In other words, there are on average of 1.97 NOPs in the delay slots for DB. For BWS, since most of branches in our benchmark programs are forward branches which do not duplicate the code, the static code size should be a lot less than DB. The performance of BWS with 50% misprediction is already 2.5 CPB. For BWS with lower misprediction rate, the performance should be significantly better than DB.

The BWS_DB strategy has 2.16 CPB and 1.08 IPB. Adding a delayed branch scheme in BWS slightly improves performance and static code size. Compared to BWS, BWS_DB has 5.10% improvement in performance and 6.89% improvement in static code size. This improvement comes from the case when all three branch delay slots filled by safe instructions.

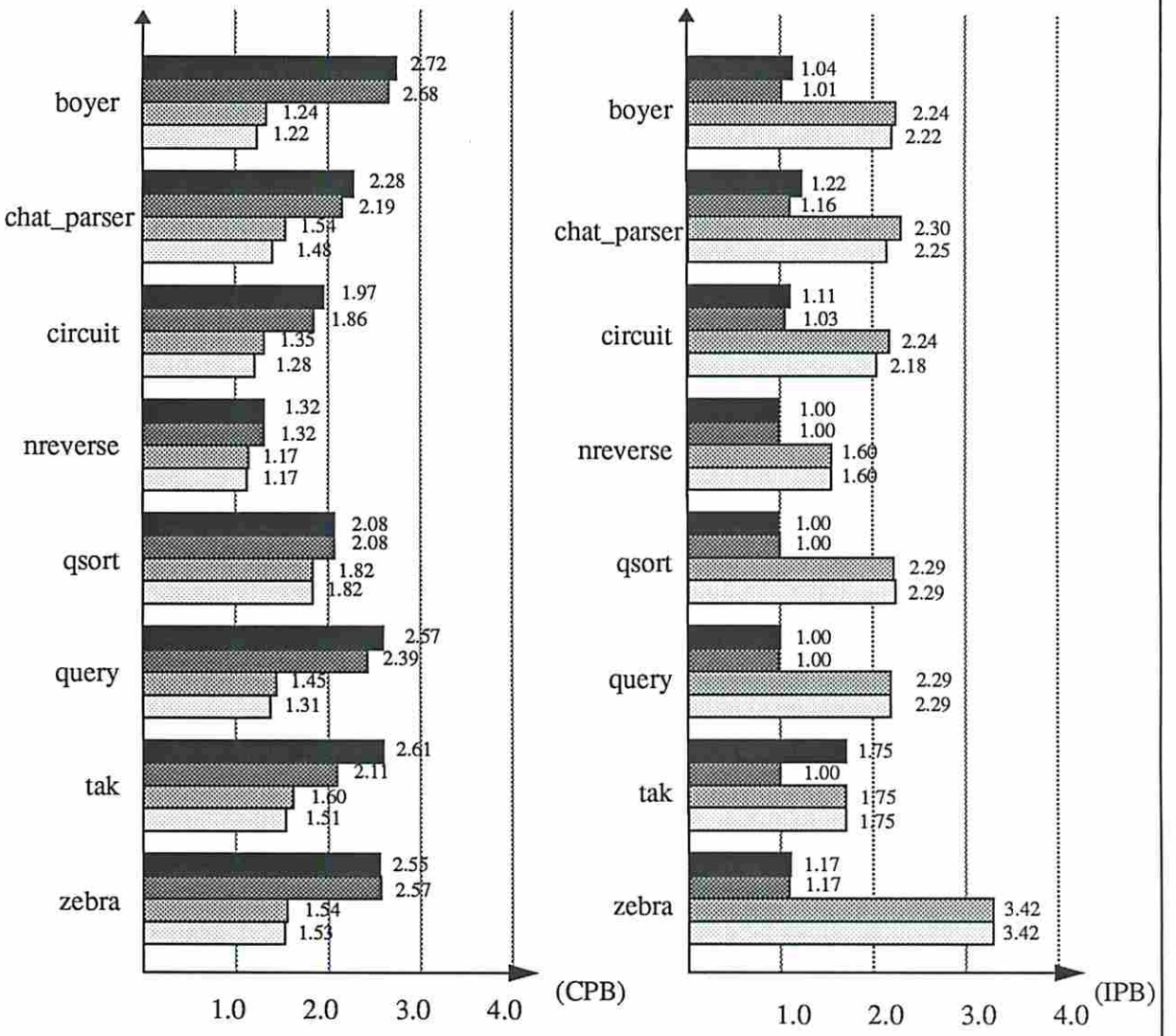
The P_BWS strategy has 1.46 CPB and 2.27 IPB. Adding profile information can significantly improve performance for BWS. Compared to BWS, the P_BWS improves performance 55.48%. However, the static code size is increased in P_BWS by 95.69%. This is because the profile information guides the branch directions which force some forward branches to be predicted as taken. This forward branch with the predicting branch taken will use instructions from predicted target basic blocks to fill delay slots, which causes the increased static code size for the branch.

The P_BWS_DB strategy has the best performance among these four branch strategies discussed above, although the static code size is as bad as P_BWS. The P_BWS_DB strategy has 1.42 CPI, which is a 2.80%, 52.11%, and 59.86% performance improvement over P_BWS, BWS_DB, and BWS. Similarly to the P_BWS strategy, the large static code size in the P_BWS_DB strategy, which is 2.23 IPB, comes from the code duplication of target instructions to fill delay slots for some forward branches guided by profile information.

4.4 Prophetic Branch

Prophetic branch is a new method for implementing branches by predicting branch taken without inlining target instructions. Branch delay slots in prophetic branch are always filled by instructions above the branch, including instructions which evaluate a branch condition (e.g. compare instructions). The basic idea is to specify the likely conditional branch before its condition evaluation. When the branch instruction is decoded and the likely target address is calculated, target instructions start to be fetched. These instructions from target blocks are continuously fetched until the condition is evaluated, which can be several cycles after the branch instruction. Since the compare instruction which evaluates the branch condition can be moved several cycles after a branch instruction, delay slots can be easily filled by safe instructions above this branch instruction. In addition, static code size of a program is not increased since no instructions from target basic block are duplicated.

Figure 7 shows an example. A sample instruction sequence is presented in Figure 7(a). Since btan is a predicted likely taken branch instruction, we need to implement target inlining in order to avoid delay cycles due to branches which is shown in Figure 7(b). Three target instructions are copied into delay slots of instruction btan. This results in increasing code size by three instructions. Figure 7(c) shows a version of this sample instruction sequence using prophetic branches. Branch delay slots are filled by instructions above the branches, including instruction *cmp(ltu,e,b)* which evaluates branch condition.



	Model	Average CPB	Average IPB
■	BWS	2.27	1.16
▒	BWS_DB	2.16	1.08
▒	P_BWS	1.46	2.27
□	P_BWS_DB	1.42	2.23

Figure 6. Performance and Static code size of branch with squashing

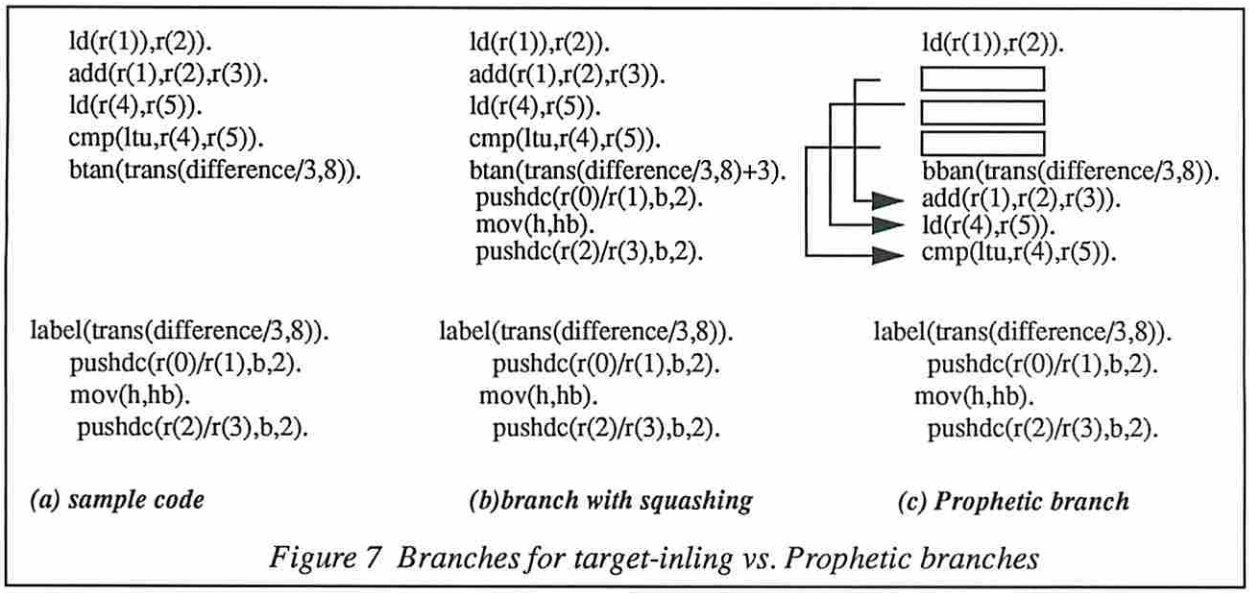


Figure 7 Branches for target-inling vs. Prophetic branches

Prophetic branches can help to avoid code duplication when a predicted branch is taken. However, when a prophetic branch is mispredicted, the penalty of misprediction is larger than traditional branches with squashing. For example, the penalty of misprediction in the branch with squashing is three cycles, which is shown in Figure 8. A prophetic branch will squash as many as five instructions due to branch misprediction in the same pipeline structure. This is because the evaluation of the branch condition is in pipeline stage EX2. In the case of misprediction, all instructions in the pipeline stages EX1, ID2, ID1, IF2, IF1 are from the wrong target basic blocks. These five instructions in the delay slots must be squashed in order to preserve the program semantics.

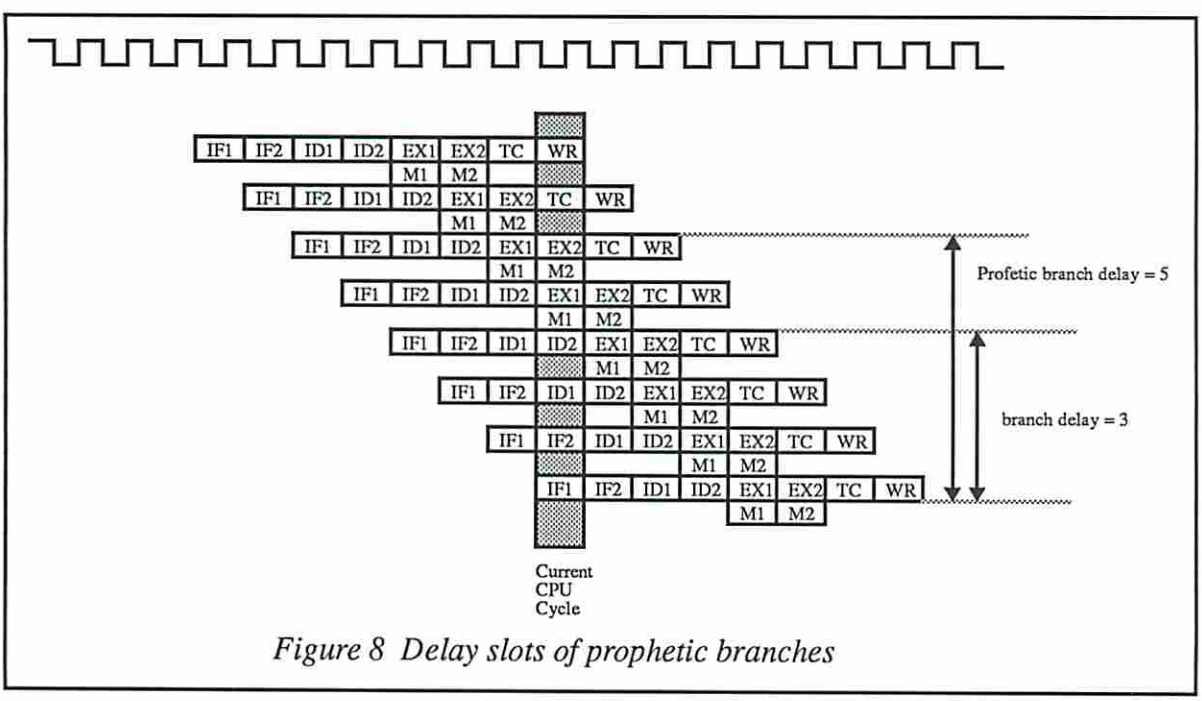


Figure 8 Delay slots of prophetic branches

To further understand how prophetic branches affect the performance, we select four different model for investigation. These four models, which are described below. The first model uses only prophetic branch. The rest of the models are combined prophetic branches alone with some other strategy.

(I) Prophetic branch only (PB)

For a backward branch, we use prophetic branch in which delay slots are filled from instructions above the prophetic branch instruction. For a forward branch, we simply use branch with squashing which predicts the branch is not taken. The delay slots are filled from instructions below this branch.

(II) Prophetic branch with delayed branch (PB_DB)

Similarly to the previous model, we use prophetic branch if it is a backward branch and branch with squashing if it is a forward branch. For the case of a branch with sufficient instructions to fill all the delay slots, we use delayed branch. Otherwise, we use prophetic branch.

(III) Profiled prophetic branch (P_PB)

This model is basically the PB strategy with the help of profile information to reduce the overhead of prophetic branch. The compiler then predicts branch directions based on this profile information of previous run.

(IV) Profiled prophetic branch with delayed branch (P_PB_DB)

Similarly to the model P_PB, we use profile information to reduce overhead of prophetic branches. In addition, the compiler will use delayed branch if there are sufficient safe instructions can be found to fill the delay slots. Otherwise, the compiler will use branch with squashing or prophetic branch based on the profile information.

Figure 9 shows the simulation results of these four varieties of prophetic branch. Since no code needed to be duplicated in these four branch strategies, the static code sizes of them are all 1.0 IPB.

The performance of PB is 3.13 CPI, which is about the same to DB. Unlike DB, which suffers performance due to having 1.88 NOPs on average for each branch, PB suffers low performance due to extra penalty of branch misprediction. Compared to the performance of the best branch strategies of branch with squash P_BWS_DB (which is 1.42 CPI), PB has only 45.37% of the performance.

The performance of PB_DB is 2.94 CPI, which is slightly better than PB. The gain, which is about 6.47%, in performance is mainly from replacing some branches in PB with delayed branches if the compiler can find three instructions to completely fill the delay slots.

The profile information significantly improves performance of the prophetic branch. The performance improvement is 76.83% on PB (P_PB has 1.77CPI) and 66.10% on PB_DB (P_PB_DB has 1.71 CPI). Compared to the performance of the best branch strategies of branch with squash P_BWS_DB, the best performance of the best branch strategies of prophetic branch P_PB_DB has only 83.04% of the performance. However, the static code size of a branch in P_PB_DB (which is 1.0 IPB) has only half of that in P_BWS_DB (which is 2.0 IPB).

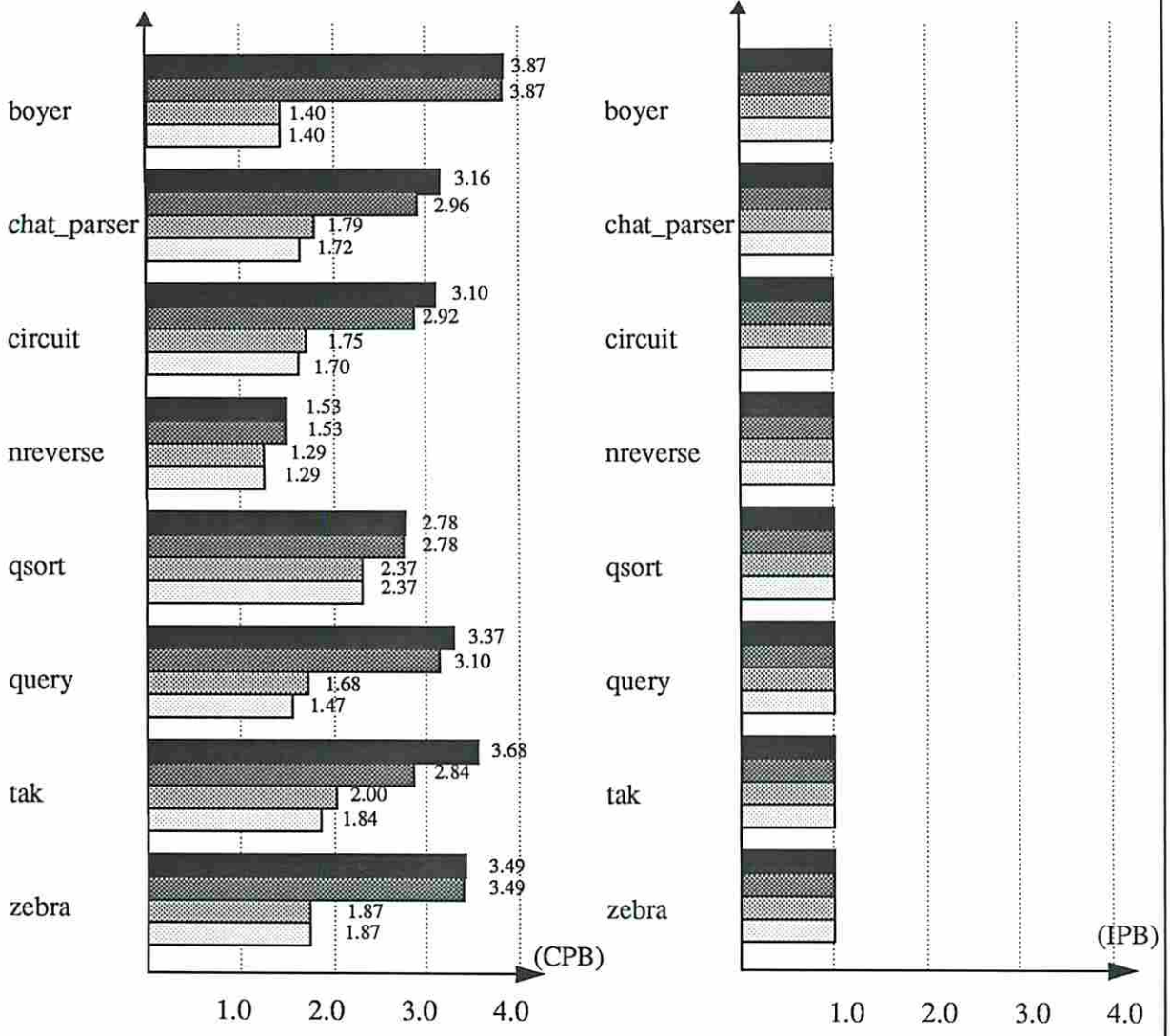
4.5 Branch With Masked Squash

In previous sections, we discussed three branch strategies, delayed branch, branch with squashing, and prophetic branch. Delayed branch takes advantages of using safe instructions above the branch to fill delay slots. In addition, hardware support for delayed branch is simple. However, there may not be a sufficient safe instructions found to fill the branch delay slots. Branch with squashing takes the advantage of using unsafe instructions from target basic blocks to fill the branch delay slots. However, the performance of branch with squashing heavily relies on the correctness of branch prediction. The penalty of misprediction may be very large since instructions in branch delay slots must be annulled. In addition, code duplication for target inlining can be significant. Prophetic branch can avoid code duplication while the branch direction is taken.

Figure 10 shows an example of branch with masked squashing. This example uses the same instruction sequence shown in Figure 10. In the case of a delayed branch, two instructions from above the branch, *ld(r(1)),r(2))* and *add(r(1),r(2),r(3))*, are used to fill delay slots. Since there are three delay slots that need to be filled and only two safe instruction can be found to fill in these delay slots, the third delay slot is filled by a NOP. In the case of branch with masked squashing, both two safe instructions are used to fill the delay slots. In addition, an unsafe instruction from the target basic block, *pushdc(r(0)/r(1),b,2)*, is used to fill the third delay slot. Moreover, since there is a load delay between the instructions *ld(r(1)),r(2))* and *add(r(1),r(2),r(3))*, we can also use this unsafe instructions to fill this load delay slot, which is shown in Figure 10 (c).

In order to use the branch with squashing scheme, we must define a scheme to specify what instructions are safe or unsafe. In general, there are two ways to specify unsafe instructions in the branch delay slots, which are described below.

The first scheme is that each instruction has a safe/unsafe bit associated with it. This safe/unsafe bit can be hard-wired in the instruction bits. When the compiler uses an unsafe instructions to fill a branch delay slot, the safe/unsafe bit associated in these instruction are set (meaning unsafe). If the branch is correctly predicted, all unsafe instructions act as normal safe instructions. If the branch is mispredicted, all instructions in the pipeline stages (before pipeline stage ID2) with safe/unsafe bit on are squashed.



	Model	Average CPB	Average IPB
■	PB	3.13	1.00
▩	PB_DB	2.94	1.00
▨	P_PB	1.77	1.00
□	P_PB_DB	1.71	1.00

Figure 9. Performance and staic code size for perphetic branch

In the second scheme, each branch instruction has mask bits to specify unsafe instructions in the branch delay slots. A simple implementation of this scheme is to assign one bit for each branch delay slot. If a delay slot is filled unsafe instruction, the corresponding mask bit in the branch instruction is set. The mask bits are passed through the control pipeline. If the branch is correctly predicted, these mask bits are inactivated. On the other hand, if the branch is mispre-

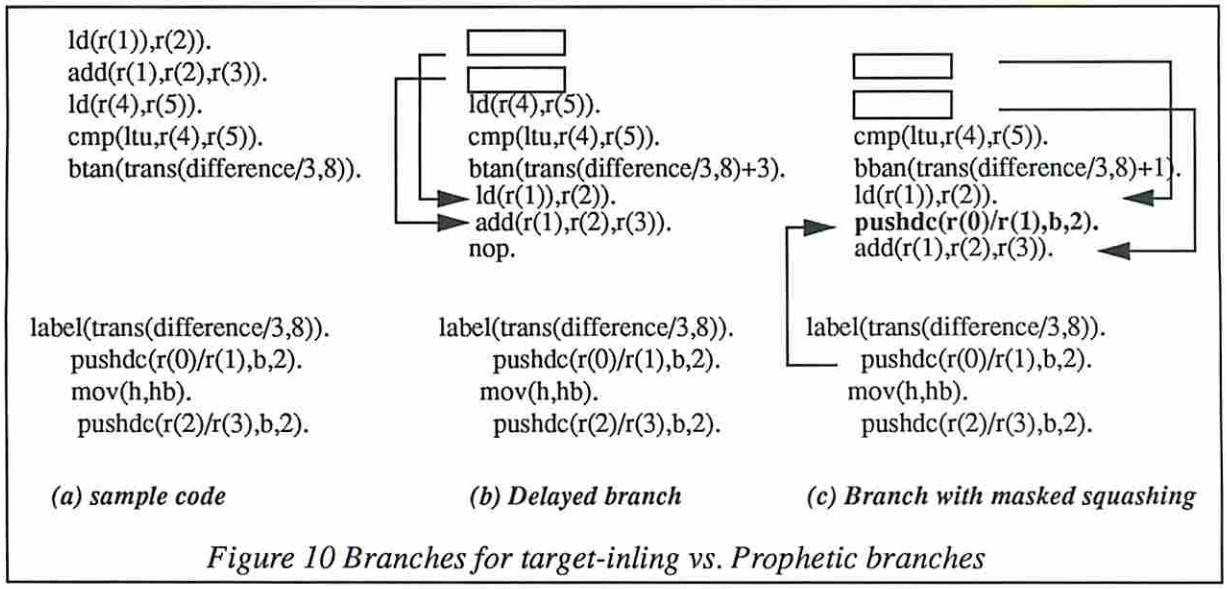


Figure 10 Branches for target-inling vs. Prophetic branches

dicted, these mask bits are activated. The first significant bit of the mask bits is used to specify whether the next instruction is safe or unsafe. If this bit is set, then the next instruction is squashed in the pipeline stage ID2. If this bit is not set, then the next instruction is executed. At the end of a cycle, the mask bits are shifted left one bit. This process will continue until all instructions in delay slots are handled.

To understand how branch with masked squashing affects branch performance, we select two different models for investigation, which are described below.

(I) Branch with masked squashing (BWMS)

For each branch, the compiler tries to find as many safe instructions above the branch as possible to fill the delay slots. If there are delay slots remaining unfilled, instructions from the predicted target basic block are used to fill these delay slots. For a forward branch, we simply predict branch is not taken. For a backward branch, we predict the branch is taken.

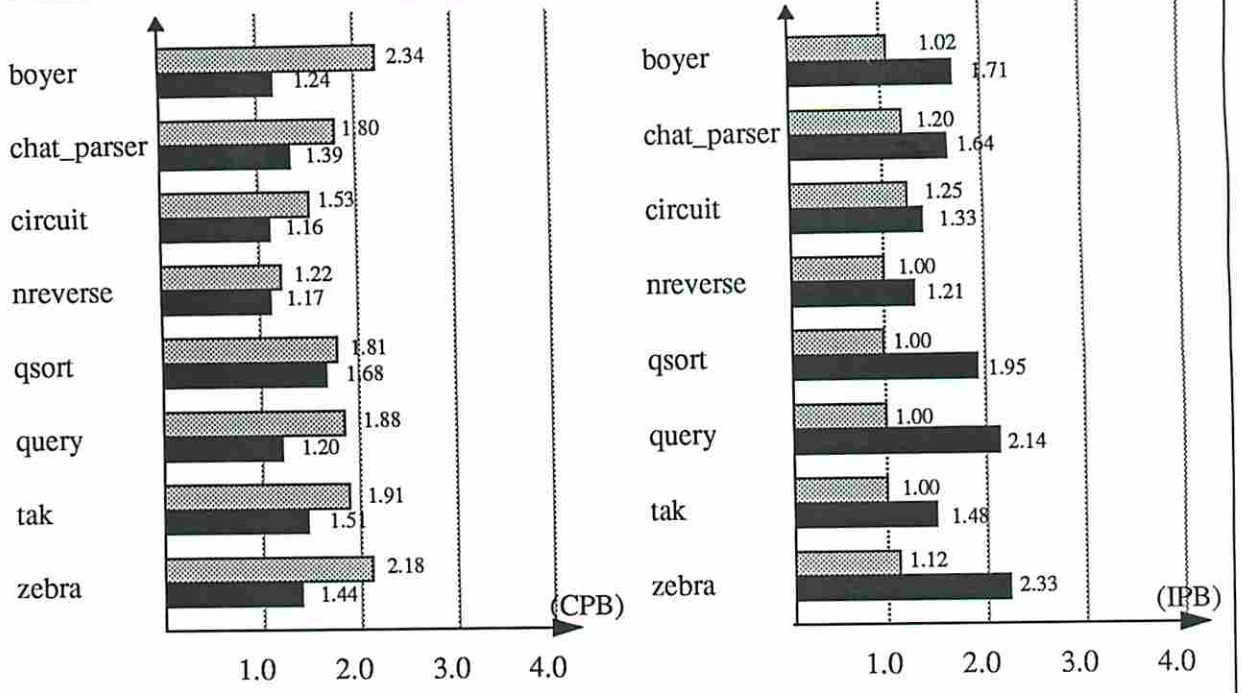
(II) Profile branch with masked squashing (P_BWMS)

The model is mainly using profile information to reduce overhead of misprediction for BWMS. The compiler then predicts branches based on this profile information.

Figure 11 shows simulation results of branch with mask squashing. BMWS has 1.87 CPB and 1.07 IPB. Without consider the help from profile information, BMWS has the best performance among the branch strategies discussed so far. The gain of performance mainly comes from the use of safe and predict target instructions in the branch delay slots.

The P_BWMS strategy has 1.36 CPB and 1.57 IPB. Profile information helps BWMS to improve performance 37.50%. However, the static code size of a branch is increased 53.64%. The

reason the increased static code size of a branch is increased is that profile information guides the branch direction which force some forward branch to be predicted as a taken branch. Code is duplicated when a forward branch predicts the branch is taken.



	Model	Average CPB	Average IPB
■	BWMS	1.87	1.07
▨	P_BWMS	1.36	1.57

Figure 11 Performance and staic code size for branch with masked squashing

4.6 Discussion

Figure 12 summarize the performance and static code size for the branch strategies discussed in this paper. The horizontal axis represents performance in terms of CPB. The vertical axis represents static code size of a branch in term of IPB. We located each branch strategy based on its (CPB, IPB). DB (3.12, 2.97) does not perform well on performance and static code size when there are three branch delay slots. The PB (3.13, 1.00) and PB_DB (2.94, 1.00) have the same static code size and similar performance. The BWS (2.27, 1.16) and BWS_DB (2.16,1.08) have significantly better performance than DB, PB, and PB_DB. In addition, the static code size is slightly worse than prophetic branch (about 10%), but significantly better than DB. Finally, BWMS, at (1.87, 1.07), has the best performance among all branch strategies discussed above. Compared to these branch strategies, BWMS has 15.51%, 21.39%, 57.22%, 67.38%, 66.84% better the performances than BWS_DB, BWS, PB_DB, DB, and PB. The static code size of a branch in BWMS is 7% larger than the branch strategies of prophetic branches and 1% less than

BWS_DB.

Adding profile information to these branch strategies can significantly improve the performance. These profile guided branch strategies are plotted as black nodes in Figure 12. Two prophetic branch models which use profile information are P_PB (1.77, 1.00) and P_PB_DB (1.71, 1.00). P_PB and P_PB_DB has 76.83% and 71.93% over PB and PB_DB respectively. Two profile guided models of branch with squashing are P_BWS (1.46,2.27) and P_BWS_DB (1.42, 2.23). P_BWS has 55.48% increased performance over BWS; P_BWS_DB has 52.11% increased performance over BWS_DB. Profile information also is a great help to BMWS. P_BMWS (1.36, 1.57) has 37.50% increased performance over BMWS. Among the branch strategies with profile information helps, P_BMWS has the best performance.

Adding profile information to the branch strategies other than prophetic branch can also increase the static code size. P_BWS, P_BWS_DB and P_BWMS increase by 48.90%, 51.57%, and 31.85% the static code size of a branch compared to BWS, BWS_DB, and BWMS, respectively. P_BMWS not only has better performance and but also has less static code size than P_BWS, P_BWS_DB, and DB.

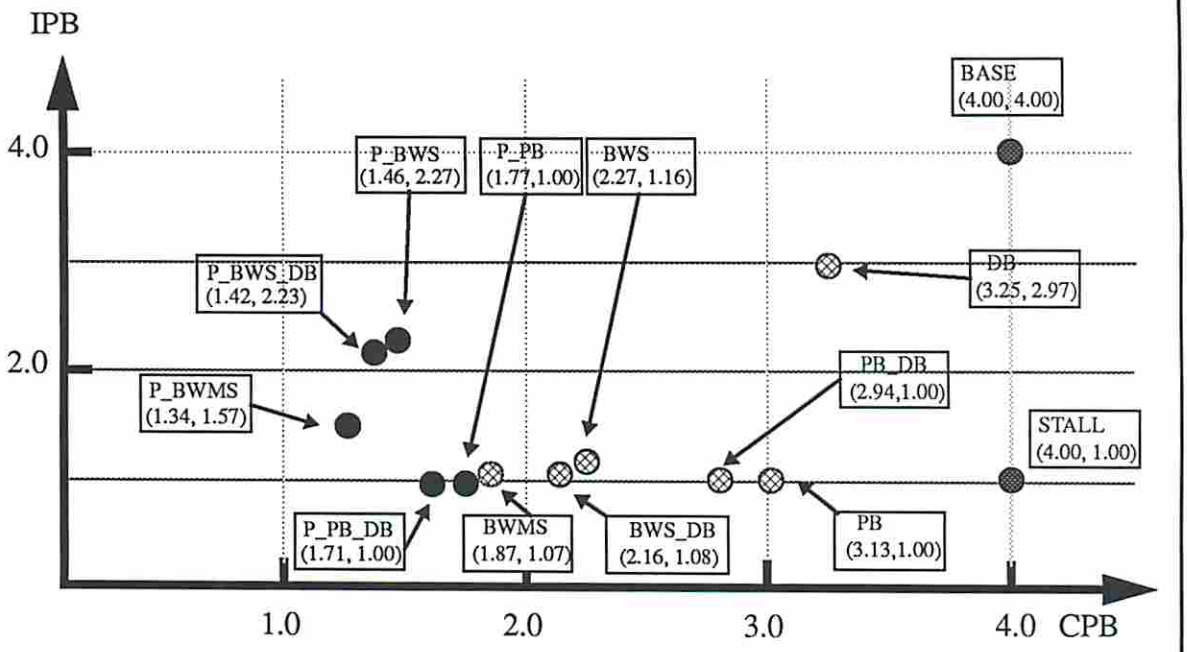
5 Scalability

In last section, we experimented with several branch strategies in SLAM. The result suggests that BMWS is a good choice in a superpipelined processor. Does this result hold when the number of branch delay slots is changed? In order to answer this question, we first provide analytic models for performance and static code size of the branch strategies. These analytic models can be easily used to analyze some relationship among these branch strategies. We also can use these analytic models to estimate the performance and static code size on any number of delay slots. We compare the results from actual simulation and from analytic models.

5.1 Analytic Models

For a better understanding of each branch strategy, we also provide an analytic model for the estimation of performance and static code size. Let P_d be the average number of safe instructions found by the compiler for each branch instruction in the execution trace. Let P_s be the average number of safe instructions found by the compiler for each branch instruction in a static program. Let D be the number of pipeline stages. We assume P_{miss} is the percentage of branch misprediction in the execution trace. We also assume P_{back} is the percentage of a backward branch in a static program.

Based on the above assumptions, we derive an analytic model of performance and static code size in terms of D, P_d, P_s, P_{miss}, and P_{back}. The performance of BASE is (1 + D) since there are D NOPs in the delay slots in addition to the branch instruction itself. Similar to BASE, the performance of STALL is also (1+D). The static code size of BASE is (1 + D) since there are D NOPs in the delay slots in addition to the branch instruction itself. For STALL, the static code size



Models	Discription
BASE	Fill three NOPs into the branch delay slots
DB	Delayed Branch
BWS	Branch with squashing
BWS_DB	Branch with squashing and delayed branch
BWMS	Branch with masked squashing
PB	Prophetic branch
PB_DB	Prophetic branch and delayed branch
P_BWS	Profile guided Branch with squashing
P_BWS_DB	Profile guided Branch with squashing and delayed branch
P_BWMS	Profile guided Branch with masked squashing
P_PB	Profile guided prophetic branch
P_PB_DB	Profile guided prophetic branch and delayed branch

Figure 12 Static Code Size vs. performance

is 1 since no code has been duplicated in this model. For DB, the analytic model is slightly more complicated. The percentage of a safe instruction in a delay slot is (P_d/D) in the execution trace. The percentage of a NOP in a delay slot is (1 - P_d/D). Since the number of delay slots is D, the average number of NOPs in the delay slots is D*(1 - P_d/D). The performance of DB is the average number of NOPs in the delay slots of a branch in the execution trace plus one, which is (1+ (D*(1 - P_d/D))). The analytic model for the static code size of DB is similar to the model for performance, except that P_d in performance model is replaced by P_s, yielding (1+ (D*(1 - P_s/D))).

The analytic model of the performance of BWS is (1 + D * Pmiss), since the misprediction of a branch cost D cycles and the branch instruction itself costs one cycle. For BWS_DB, all delay slots need to be filled by instructions from the target basic block unless there are at least D safe instructions found by the compiler. Since the percentage of delay slots filled by safe instructions is (P_d/D) in the execution trace, the percentage of all branch delay slots filled by safe instructions is $(p_d/D)^D$. In other words, the percentage of branches with squashing among all branches in the

execution trace is $(1 - (p_d/D)^D)$. These branches with squashing can be applied to BWS model, which is $P_{miss} * D$. Therefore, the analytic model of BWS_DB is $(1+ D*Pmiss*(1 - (p_d/D)^D))$. The analytic models for static code size of BWS and BWS_DB are similar to the performance model excepting P_d and P_{miss} are replaced with P_s and P_{back}.

The analytic models of performance of PB and PB_DB are similar to BWS and BWS_DB except that D is replaced by (D+2). This is because the misprediction penalty in prophetic branch is 2 cycles more than the misprediction penalty in normal branches. The static code size of PB and PB_DB are 1 since no code is duplicated.

The analytic model of performance of BWMS is $1 + D * P_{miss} * (1- P_d/D)$. It includes one cycle for the branch instruction itself and the misprediction penalty for squashing those unsafe instructions in the delay slots. These unsafe instructions are used to replace NOPs in DB. Since the percentage of delay slots filled by NOPs is $(1- P_d/D)$ and there are D delay slots for each branch instruction, the average misprediction penalty for squashing unsafe instructions is $D * P_{miss} * (1- P_d/D)$. The analytic model of static code size of BWMS is similar to the performance model excepting P_d and P_{miss} are replaced by P_s and P_{back}.

The analytic models of profile guided branch strategies are similar to the branch strategies without the help of profile information. The performance gains of profile-guided branch strategies is mainly from reducing value of P_{miss}. Profile information can increase or decrease the number of backward branches in a static program. Therefore P_{back} could be increased or decreased depending on the benchmark program.

Table 2 is a summary of the analytic models of all branch strategies. From this table, we can easily derived some conclusions.

Conclusion 1:

The performance of DB is always better than the performance of BASE and STALL.

Reason:

The performance of BASE and STALL is (1+D); the performance of DB is $(1 + D * (1 - P_d/D))$. Since P_d is always less than one and D is at least one, the value of $(1 - P_d/D)$ is always less than one. Therefore, the value of $(1 + D * (1 - P_d/D))$ is always less than $(1 + D)$. Therefore, The performance of DB is always better than the performance of BASE and STALL.

Conclusion 2:

The performance of DB is better than BWS only when $(1 - P_d/D)$ is smaller than P_{miss}. The static code size of DB is better than BWS only when $(1 - P_s/D)$ is smaller than P_{back}.

Reason:

	Average CPB	Average IPB
BASE	$1 + D$	$1 + D$
STALL	$1 + D$	1
DB	$1 + D * (1 - P_d/D)$	$1 + D * (1 - P_s/D)$
BWS	$1 + D * P_{miss}$	$1 + D * P_{back}$
BWS_DB	$1 + D * P_{miss} * (1-(p_d/D)^D)$	$1 + D * P_{back} * (1 - P_s/D)$
PB	$1 + (D+2) * P_{miss}$	1
PB_DB	$1 + (D+2) * P_{miss} * (1-(p_d/D)^D)$	1
BMWS	$1 + D * P_{miss} * (1- P_d/D)$	$1 + D * P_{back} * (1- P_s/D) * (1-(p_s/D)^D)$
P_BWS	$1 + D * P_{miss}$	$1 + D * P_{back}$
P_BWS_DB	$1 + D * P_{miss} * ((1-(p_d/D)^D)$	$1 + D * P_{back} * ((1-(p_s/D)^D)$
P_PB	$1 + (D+2) * P_{miss}$	1
P_PB_DB	$1 + (D+2) * P_{miss} * ((1-(p_d/D)^D$	1
P_PWMS	$1 + D * P_{miss} * (1- P_d/D) * (1-(p_d/D)^D)$	$1 + D * P_{back} * (1- P_s/D) * (1-(p_s/D)^D)$

D: The number of delay slots.
P_d: The average number of save instructions been filled into delay slots for a branch in the trace.
P_{miss}: The percentage of miss prediction of a branch in the trace.
P_s: The average number of save instruction been filled into delay slots for a branch in a static program
P_{back}: Thepercentage of backword branch of a branch in a statoc program

Table 2 Analytical Models

The performance of BWS is $(1 + D * P_{miss})$; the performance of DB is $(1 + D * (1 - P_d/D))$. The performance of DB is better than the performance of BWS only when P_{miss} is larger than $(1 - P_d/D)$. A similar reason for the static code sizes obtained by replacing P_{miss} and P_d to P_{back} and P_s.

Conclusion 3:

The performance of prophetic branch is always equal to or worst than branch with squashing. The static code size of prophetic branch is always smaller than branch with squashing.

Reason:

The performance of BWS is $(1 + D * P_{miss})$; The performance of PB is $(1 + (D+2) * P_{miss})$. Since P_{miss} is larger than zero, the performance of BWS is always better than the performance of PB. The same prove applies to BWS_DB and PB_DB. The static code size of PB and PB_DB is 1.00 IPB. The static code size of branch with squashing is larger than the value of $(1 + D * P_{back} * (1 - P_s/D))$. The static code size of branch with squashing is equal to 1.00 IPB only when P_{back} is zero or P_s is equal to D. Otherwise, the static code size of branch with squashing is always larger than prophetic branch.

Conclusion 4:

The performance of BMWS is always equal to or better than BWS, BWS_DB, PB, PB_DB and DB. The static code size of BWS is always smaller than BWS, BWS_DB, and DB.

Reason:

The performance of BWMS is $(1 + D * P_{miss} * (1- P_d/D))$. Since the value of P_{miss} is equal to or less than 1, the value of $(1 + D * P_{miss} * (1- P_d/D))$ is always equal to or larger than $(1 + D * (1- P_d/D))$, which is the performance of DB. Therefore, the performance of BWMS is better than the performance of DB. Since the value of $(1- P_d/D)$ is always equal to or less than 1, the value of $(1 + D * P_{miss} * (1- P_d/D))$ is always equal to or larger than $(1 + D * P_{miss})$, which is the performance of BWS. Therefore, the performance of BWMS is better than the performance of BWS. The performance of BWS_DB is $(1 + D * P_{miss} * (1-(p_d/D)^D))$. Since the value of $(1-(p_d/D)^D)$ is always equal to or less than the value of $(1- P_d/D)$, the performance of BWMS is always equal to or better than BWS_DB. We have already shown the performance of branch with squashing is always equal to or better than prophetic branches from Conclusion 3. We can conclude that the performance of BMWS is always equal to or better than BWS, BWS_DB, PB, PB_DB and DB. A similar result for can be applied to conclude that the static code size of BWS is always equal to or smaller than BWS, BWS_DB, and DB.

5.2 Experimental Results

In the previous section, we provided an analytic model of performance and static code size for all the branch strategies. In this section, we would like to experiment with the branch strategies on a superpipelined Prolog processor with various numbers of branch delay slots. The simulation results should match the results from analytic models.

Figure 13 shows the performance of branch strategies without help from the profile information with the number of branch delay slots ranging from one to five. The performance of all branch strategies decreases when the number of branch delay slots is increased. This agrees with the increasing D in the analytic models. In general, BASE has the worst performance among all branch strategies, except PB performs worse than BASE when the number of branch delay slots is only one. This irregular behavior of PB is mainly due to a significantly large misprediction penalty compared to the number of branch delay slot. In this case, the number of branch delay slots is one; the misprediction penalty is three. From the analytic models of PB and BASE, the performance of BASE is better than PB only when the value of $(D+2) * P_{miss}$ is larger than the value of D. In this special case, when D is equal to one, the value of P_{miss} must larger than 1/3.

The performance of DB is better than all prophetic branches when the number of delay slots is less than three. But a deeply pipelined machine, the prophetic branch is expected to be better than DB. However, when the number of delay slots is small, DB can perform much better than prophetic branches. From the analytic models of DB and prophetic branches, the performance of DB is better than all prophetic branches only when the value of $D * (1 - P_d/D)$ is smaller than the value of $(D+2) * P_{miss} * (1-(p_d/D)^D)$.

The performance of both BWS and BWS_DB is significantly better than DB, PB, and PB_DB. Especially when the number of delay slots is increased, the performance gains of BWS and BWS_DB over DB, PB, and PB_DB is also increased. From Conclusion 3, we have shown that the performance of BWS and BWS_DB are always better than the performance of PB, and PB_DB. Conclusion 2 suggests that the performance of BWS and BWS_DB is better than DB only when $(1 - P_d/D)$ is larger than P_{miss}. In the SLAM case, P_{miss} is less than 1/3. Therefore, the value of $(1 - P_d/D)$ is always larger than P_{miss}. BMWS shows the best performance on the number of delay slots ranged from one to five. This is also suggested by Conclusion 4.

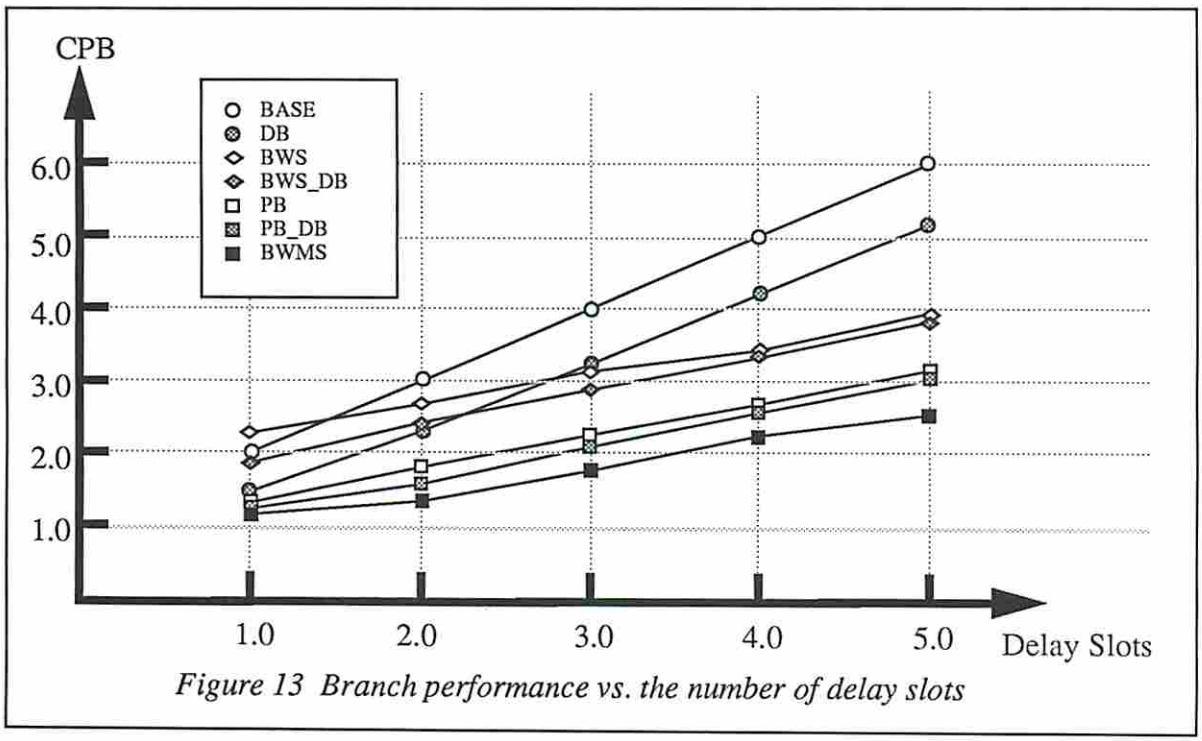


Figure 13 Branch performance vs. the number of delay slots

Figure 14 shows the performance of branches with the help of profile information on a machine with different numbers of delay slots. In general, profile information helps all branch strategies discussed in this paper, excepting BASE and DB which do not use profile information. The performance gain due to profile information is increased when the number of delay slots is increased. P_BWMS has the best performance among all branch strategies on machines with one to five delay slots. In the analytic models, since profile information only reduces the value of P_{miss}, the performance ratio among all branch strategies still holds except for BASE, STALL, and DB.

Figure 15 shows the static code size of branch strategies when the number of delay slots varies. BASE has the worst static code size among all branch strategies. DB also has worse static code size compared to most of the branch strategies. The remaining branch strategies (i.e. BWS, BWS_DB, PB, PB_DB, and BWMS) have very close static code sizes. This is because the value of P_{back} is very small in the SLAM benchmark suite. The impact of other factors (e.g. D and P_s)

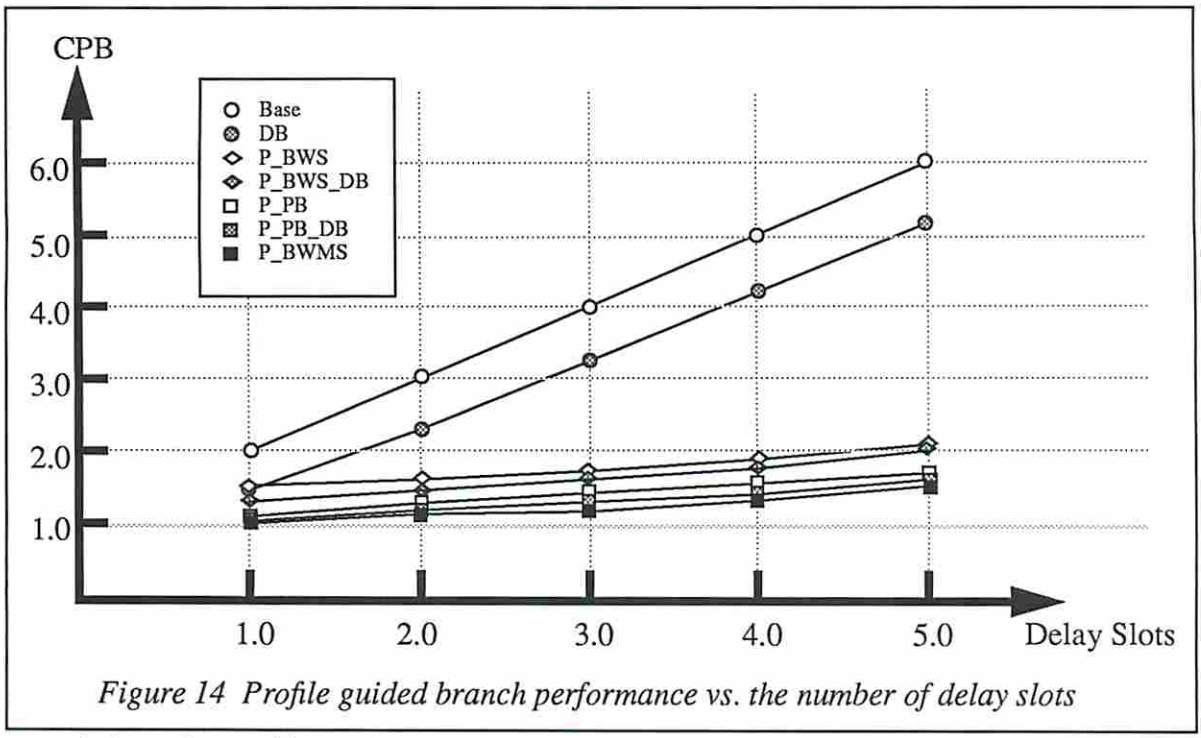


Figure 14 Profile guided branch performance vs. the number of delay slots

are not obvious due to this small P_{back}. The prophetic branch has the best static code size, which is 1.00 IPB, but only 5~10% better than BWS, BWS_DB, and BWMS. Comparing BWMS and prophetic branch (PB and PB_DB), BWMS has significantly better performance than prophetic branches. However, BWMS has worse static code size than prophetic branches. Compared to BWMS and branch with squashing (BWS and BWS_DB), both performance and static code size of BWMS are better.

Figure 16 shows the static code sizes of profile guided branches when the number of delay slots vary. Profile information can significantly increase the performance, though the static code size is also increased. The static code size of all profile guided prophetic branches is still 1.00 IPB. Compared to P_BWS and P_BWS_DB, P_BWMS has better performance and static code size.

6. Conclusion

In this paper, we have investigated several branch strategies used in modern pipelined processors. These branch strategies (i.e delay branch, branch with squashing, and prophetic branch) are useful only for a small number of branch delay slots. For a superpipelined processor, we need more aggressive branch strategies in order to avoid the large branch overhead.

We have proposed a novel branch scheme, called branch with masked squashing, which takes advantage of both delayed branch and branch with squashing. This scheme benefits from both compiler techniques and static branch prediction to optimally reduce penalty of misprediction. Since most modern microprocessors implement both delayed branch and branch with squashing, the only additional cost to implement branch with masked squashing is to specify

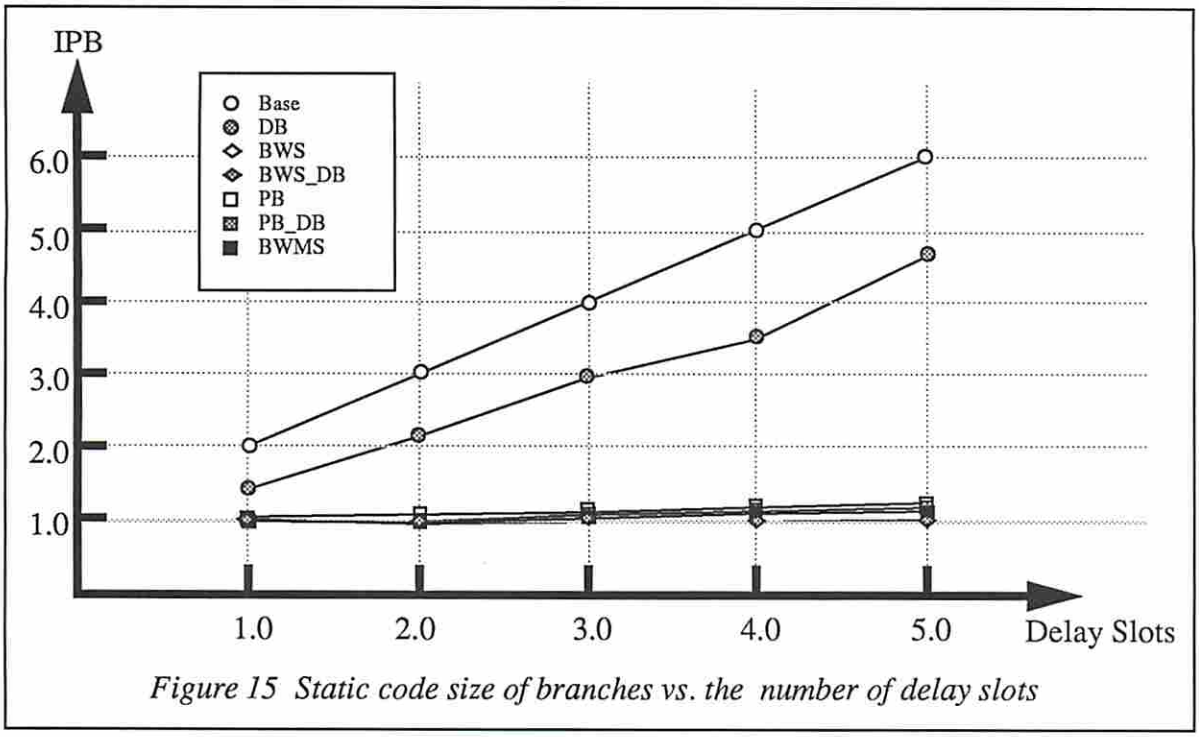


Figure 15 Static code size of branches vs. the number of delay slots

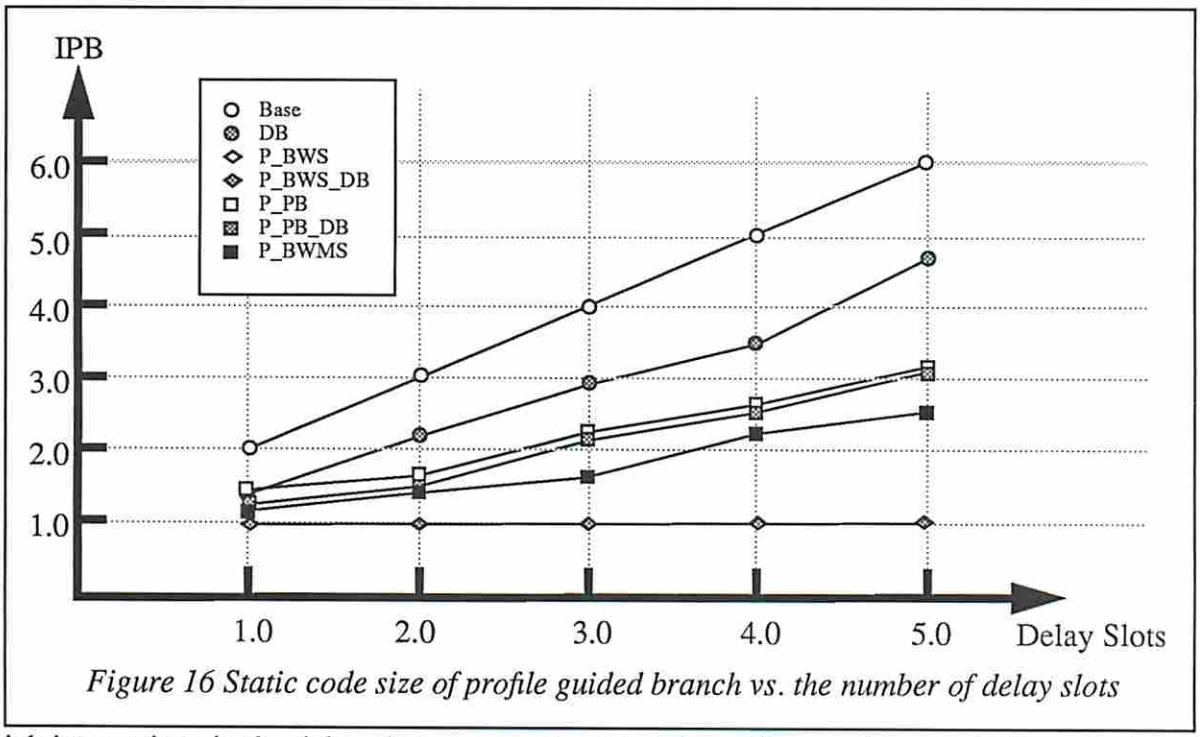


Figure 16 Static code size of profile guided branch vs. the number of delay slots

which instructions in the delay slots are unsafe. This information can be encoded into the branch instruction opcode or one bit per instruction can specify unsafe instructions.

We evaluate these branch strategies with the help of profile information. Profile information does improve performance of all branch strategies discussed in this paper. However, these

profile-guided branch strategies also increase static code size. Profile-guided branch with masked squashing has the best performance of all profile-guided branches.

Finally, we show experimental results for the branch strategies on machines with different numbers of branch delay slots. We provide analytic models to estimate the performance and static code size of a branch among the branch strategies. We also evaluate the actual performance and static code size by compilation and simulation. Both results from analytic models and simulation are perfectly matched. It suggests that the branch with squashing outperforms than other branch schemes no matter how many branch delay slots are required. It is not surprising since branch with masked squashing is built by taking advantages of delayed branch and branch with squashing. This study suggests that the branch with masked squashing is a very useful branch scheme for deeply pipelined processors.

Acknowledgment

The authors would to thank Steve Crago and Kevin Obenland for their comments and suggestions. The work was supported by ARPA under grant No. J-FBI-91-194.

References

[Dubey 91] P.K. Dubey and M.J. Flynn, "Branch Strategies: Modeling and Organization," IEEE Transactions on Computers, Vol. 40, No. 10, 1991.

[Fisher 81] J.A. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, Vol. 30, No. 7, 1981.

[Haygood 89] R. Haygood, "A Prolog Benchmark Suite for Aquarius," Technical Report, Computer Science Department, University of California, UCB/CSD 89/509, 1989.

[Holmer 90] B. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, W. Bush, and A. Despain. "Fast Prolog with an Extended General Purpose Architecture," The 17th Annual International Symposium on Computer Architecture, May 1990.

[Hwu 92] W.W. Hwu and P.P. Chang, "Efficient Instruction Sequencing with Inline Target Insertion," IEEE Transactions on Computers, Vol. 41, No.12, Dec. 1992.

[Jouppi 89] N.P. Jouppi, and D.W. Wall. "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," The 3rd International Conference on Architectural Support for Programming Languages and Operating System, 1991.

[Lee 84] J.K.F. Lee and A.J.Smith, "Branch Prediction Strategies and Branch Target Buffer Design," Computer, Jan. 1984.

[McFarling 86] S. Mcfarling, J. Hennssy, "Reducing the Cost of Branches," The 13th Annual International Symposium of Computer Architecture, 1986.

[MIPS 86] "MIPS language programmer's guide," MIPS Computer Systems, Inc., 1986

[Smith 81] J.E. Smith, "A Study of Branch Prediction Strategies," The 8th Annul Interna-

tional Symposium on Computer Architecture, May, 1981.

[**Srivastava 93**] A. Srivastava and A.M. Depain, "Prophetic Branches: A Branch Architecture for Code Compaction and Efficient Execution", MICRO-26, 1993.

[**Su 92**] C-L Su, "An instruction Scheduler and Register Allocator for Prolog Parallel Microprocessors," International Computer Symposium, 1992

[**Su 93**] Ching-Long Su, "An Modified Target Inlining Algorithm for SLAM", ACAL technical report, 1993

[**Van Roy 92**] P. Van Roy and A. M. Despain, "High-Performance Logic Programming with the Aquarius Prolog Compiler," January 1992.