# Branch Target Buffer Design and Optimization*

*Chris H. Perleberg*
*Alan Jay Smith*

Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

The Branch Target Buffer (BTB) can reduce the performance penalty of branches in pipelined processors by predicting the path of the branch and caching information used by the branch. This paper discusses two major issues in the design of BTBs with the theme of achieving maximum performance with a limited number of bits allocated to the BTB design. First is the issue of BTB management - when to enter and discard branches from the BTB. Higher performance can be obtained by entering branches into the BTB only when they experience a branch taken execution. A new method for discarding branches from the BTB is examined. This method discards the branch with the smallest expected value for improving performance, outperforming the LRU strategy by a small margin, at the cost of additional complexity.

The second major issue discussed is the question of what information to store in the BTB. A BTB entry can consist of one or more of the following: branch tag (i.e. the branch address), prediction information, the branch target address, and instructions at the branch target. A variety of BTB designs, with one or more of these fields, are evaluated and compared. This study is then extended to multilevel BTBs, in which different levels have different amounts of information per entry. For the specific implementation assumptions used, multilevel BTBs improved performance over single level BTBs only slightly, at the cost of additional complexity. Multi-level BTBs may provide significant performance improvements for other implementations, however.

Design target miss ratios for BTBs are also developed, so that the performance of BTBs for real workloads may be estimated.

December 23, 1989

* The material presented here is based on research supported in part by the National Science Foundation under grant MIP-8713274, by NASA under consortium agreement NCA2-128, by the State of California under the MICRO program, and by Philips Laboratories/Signetics, Digital Equipment Corporation, Apple Computer Corporation, and International Business Machines Corporation.

# Branch Target Buffer Design and Optimization*

*Chris H. Perleberg*
*Alan Jay Smith*

Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

## 1. Introduction

Most modern computers use pipelining to significantly increase performance. Pipelining divides the execution of each instruction into several pieces, normally called pipeline stages. A typical pipeline might have five pipeline stages, including instruction fetch, instruction decode, operand fetch, execution, and result writeback. It is important that the operation time of each pipeline stage is almost identical, since the rate at which instructions flow through the pipeline is limited by the slowest pipeline stage. In the optimal case, five instructions could be operated on simultaneously in this pipeline, each in a separate pipeline stage. This optimal case, though seldom achieved in practice, would produce a five fold performance gain over the same processor designed without a pipeline. More details about pipelining can be found in [Ramamoorthy76] and [Kogge81].

Branch instructions can reduce the performance of pipelines by interrupting the steady flow of instructions into the pipeline. To execute a branch, the processor must decide whether the branch is taken, calculate the branch's target address, and then, if the branch is taken, fetch instructions from the target address. Branches hurt performance because the pipelined processor needs to know the path of the branch (in order to fetch more instructions) before the path of the branch has been determined. Thus, when a branch occurs, the processor has two choices, either wait for the branch to finish executing before fetching more instructions, or continue fetching instructions, possibly from the wrong path. Either choice hurts performance, although the second choice hurts less, especially if the path of the branch is predicted correctly most of the time.

The Branch Target Buffer (BTB) can be used to reduce the performance penalty of branches by predicting the path of the branch and caching information about the branch [Lee84]. This paper discusses two major issues in the the design of BTBs, with the theme of achieving maximum performance with a limited number of bits allocated to the BTB design. First, the question of BTB management, when to enter branches into the BTB, and how to discard branches from the BTB. Second, the question of what information to keep in the BTB. Four types of information have been kept in BTBs, including branch tags, the branch target address, prediction information, and instruction bytes from the branch target address. A BTB contains a subset of these four types of information, and a multi-level BTB contains several distinct subsets, each in a separate level of the BTB. Higher performance BTBs generally contain a larger subset of the possible information types than lower performance BTBs. As a sidenote, a comparison of BTB and instruction cache miss rates is performed, with the purpose of showing that BTB miss rates can be derived from the

* The material presented here is based on research supported in part by the National Science Foundation under grant MIP-8713274, by NASA under consortium agreement NCA2-128, by the State of California under the MICRO program, and by Philips Laboratories/Signetics, Digital Equipment Corporation, Apple Computer Corporation, and International Business Machines Corporation.

instruction cache miss rates that exist in literature [SmithAJ87].

This paper is organized as follows. In the following section, a number of solutions to the branch problem are surveyed. The methodology used to generate the results in this paper (trace driven simulation) is then presented, followed by a discussion of optimizations in the design of BTBs. Branch prediction is covered, and the issue of BTB management. A discussion on miss rates of BTBs and instruction caches is followed by a consideration of the issue of what information to keep in a BTB. Conclusions are then provided.

## 2. Solutions to the Branch Problem

This survey of solutions to the branch problem is intended to give a brief explanation of each solution, and provide the names of a few machines that have used each solution. Further discussion is available in [Lee84], [Lilja88], and [McFarling86].

**Static Branch Prediction:** As discussed in the introduction, if the path of a branch can be correctly predicted, the branch penalty can be reduced. Predicting all branches not taken or predicting all branches taken (Lee and Smith found 68 percent of branches to be taken [Lee84]; we found 70 percent of branches to be taken) is the simplest type of static prediction. Predicting branches based on the direction of the branch (forward or backward) and/or the branch opcode is also possible [SmithJ81]. The National Semiconductor NS32532 uses the branch opcode and the branch direction to predict 71 percent of branches correctly [Maytal89]. A static prediction bit in each branch instruction, used to indicate which path the branch will probably take, is also effective. The bit can be set using heuristics and execution profiling [McFarling86]. The Ridge Computers Ridge-32 [Folger83], the Intel 80960 [Hinton89], and the AT&T CRISP [Ditzel87] all use a static prediction bit. An average prediction rate of 85 percent has been reported for the static prediction bit method [McFarling86].

**Dynamic Branch Prediction:** Dynamic branch prediction predicts branch paths based on the runtime execution history (the sequence of taken and not taken executions, represented with one bit per execution) of each branch. One proposed dynamic method uses a two bit counter state machine which takes the branch execution history as input and outputs the predicted branch path for the next branch execution [SmithJ81]. Another method examines the last $n$ executions (taken or not taken) of a branch, and using statistics gathered beforehand to calculate the chance of a taken branch, predicts the branch path [Lee84] [Widdoes77]. Widdoes proposes keeping bits with each instruction in the instruction cache for storing branch execution history [Widdoes77]. Machines that use dynamic branch prediction include the S-1 [Hailpern79], the MU5 [Rau77], several TRON microprocessors [Sakamura87], the Edgecore Edge 2000 (Motorola 680X0 compatible) [Manual87] [Walter89], and the NexGen processor (Intel 80386 compatible) [Stiles89a].

**Delayed Branch:** Executing $n$ instructions after a branch whether the branch is taken or not often allows useful instructions to be executed while the branch path is resolved and branch target address calculated. The $n$ instructions (usually $n$=1) are called delay instructions. For more flexibility, some machines allow the execution of the delay instructions to be skipped, depending on the value of a "squash" bit in the branch and the path of the branch, taken or not taken [McFarling86]. The delayed branch technique is used in most RISC processors, and requires intelligent compilers for optimal use [DeRosa87] [Gross82] [McFarling86] [Patterson81].

**Loop Buffers:** Loop buffers are small, high speed buffers used for instruction prefetching and fast execution of instruction loops. If a loop is completely contained within a loop buffer, the loop can be executed from the high speed buffer, instead of accessing slower main memory for the instructions. Multiple loop buffers allow multiple non-contiguous loops to execute from the buffers. Machines using loop buffers include the CDC-6600 with eight 60-bit words [Thornton64], the CDC-7600 with twelve 60-bit words [Elrod70], the IBM 360/91 and the IBM 360/195

with eight 64-bit words [Anderson67] [Murphey70], the Cray-1 with four 128-byte buffers [Russell78], and the Cray-XMP with four 256 byte buffers [Hwang84].

**Multiple Instruction Streams:** To avoid the penalty of a wrong branch prediction, both paths of a branch can be fetched and executed, and the results of the incorrect path discarded once the branch is resolved. This method requires duplication of hardware, early calculation of the branch target address, high memory and register file bandwidth, and a method for handling more than one branch in the pipeline. This technique has been used in mainframes, including the IBM 370/168 and the IBM 3033 [IBM73] [IBM78].

**Prefetch Branch Target:** Many computers predict all branches not taken and pay a penalty if the branch is taken. This penalty may be reduced if instructions from the branch target address are fetched while the branch (and the instructions in the not taken branch path) is executing. The IBM 360/91 prefetches two double-words (each 64 bits long) from the target address [Anderson67].

**Prepare to Branch:** The "Prepare-to-Branch" and "Load-Look-Ahead" instructions of the Texas Instruments ASC computer direct the machine to prefetch from the target address of an upcoming branch, rather than from the sequential addresses after the branch [Gaulding75]. The CHoPP supercomputer has a similar instruction [Kartashev90]. Use of these instructions with branches that are usually taken can increase performance.

**Shared Pipeline Multiprocessor:** An $n$ stage pipelined processor executing $n$ independent instruction streams simultaneously, each with a single instruction occupying one of the pipeline stages, eliminates the performance penalty of branches. Within an individual instruction stream, each instruction completes execution before the next instruction begins execution, so branch instructions do not have to wait during execution for prior instructions to finish executing. Each independent instruction stream receives a $1/n$ fraction of total processor power. The shared pipeline multiprocessor concept was implemented in the Denelcor HEP computer [Jordan83], and is examined in [Shar74].

**Branch Folding:** The AT&T CRISP processor uses a technique called branch folding to reduce branch execution time to zero [Ditzel87]. CRISP uses a decoded instruction cache; each non-branch instruction is transformed into a line of microcode in a cache of decoded instructions. Each entry includes a next address field pointing to the next line of microcode to be executed. Unconditional branches are simply "folded" into the next address field of the previous line of microcode. No branch execution time is necessary once the unconditional branch is in the decoded cache. Conditional branches make use of a second next address field in the microcode and static branch prediction. A bit in the branch instruction predicts whether the branch will be taken or not, and the predicted next address field is selected to find the next line of microcode to execute. If the predicted path is found to be wrong once the condition flag is resolved, the incorrect path results are discarded, and the correct path is executed. Like the unconditional branch, the conditional branch has been folded into the previous line of microcode, resulting in a zero cycle execution time for a correctly predicted branch in the decoded cache. Note that branch folding does not require a decoded instruction cache. Branch folding can be implemented with a branch target buffer that caches instructions from both the taken and not taken branch paths of each branch.

**Branch Target Buffer:** The Branch Target Buffer (BTB) can be used to reduce the performance penalty of branches by predicting the path of the branch and caching information about the branch [Lee84]. Up to four types of information can be cached in a BTB: a tag identifying the branch the information corresponds to (usually the branch address), prediction information for predicting the path of the branch, the branch target address, and instruction bytes from the branch target address. A BTB with all four types of information works as follows. As each instruction is fetched from memory, the instruction address is used to index into the BTB. If a valid BTB entry

is found for that address, the instruction is a branch. The branch path is predicted using the branch's prediction information. If the branch is predicted taken, the pipeline is supplied with the cached instructions in the BTB, and the processor begins fetching instructions from the target address, offset by the number of bytes of instructions cached in the BTB. If the branch is predicted not taken, the processor continues fetching sequentially after the branch. After the processor finishes executing the branch, it checks to see if the BTB correctly predicted the branch. If it has, all is well, and the processor can continue sequentially. If the branch was predicted incorrectly (or the branch was taken and the target address changed), the processor must flush the pipeline and begin fetching from the correct branch path. In either case, the branch prediction information and branch target address (if changed) must be updated after the branch. Machines that use a variation of the BTB include: the Advanced Micro Devices Am29000 [AMD88] and the General Electric RPM40 [Lewis88] with branch target caches (no prediction information, just a cache of target instructions); the Mitsubishi M32 with a BTB containing dynamic prediction information, branch tag, and instructions from the target address (no mention of target address being stored) [Yoshida87]; the Edgecore Edge 2000 with a direct map 1024 entry BTB containing one dynamic prediction bit, a branch tag, and the branch target address for each entry [Walter89]; and the NexGen processor with a fully associative LRU replacement BTB containing prediction information, branch tag, branch target address, and instructions from the target address [Stiles89a] [Stiles89b].

# 3. Methodology

## 3.1. Trace Driven Simulation

To date, there exists no generally accepted model of program behavior with respect to the performance of BTBs. As a result, we use trace driven simulation to generate the data presented. Trace driven simulation uses a recording (known as a trace) of the execution of a program as input to a software design simulator. Each trace contains a record of the data and instruction addresses issued by the program and the actual instruction bytes fetched by the program. Using a number of these traces as input to a software simulator can give accurate performance numbers, within the accuracy of the simulator itself, and allows a flexibility in varying design parameters that is very difficult to achieve in a hardware prototype. There are, however, reasons to be cautious when dealing with the results of trace driven simulation [SmithAJ85]. Traces tend to be short recordings (less than 1 CPU second) and may not be representative of the expected workload. They often do not include the effects of context switching, operating sytem code execution, and input/output. In addition, traces are often dependent on the implemention of the computer architecture the trace is recorded on (e.g. number of bytes fetched per instruction fetch), and may not match the assumptions of the simulated design. We have used a wide variety of traces of both small and large programs from a number of different machines and believe that our results are representative of what could be expected in practice.

## 3.2. Description of Traces

A total of 32 program traces are used in this paper. Four processors are represented in the traces, including the MIPS Co. R2000 (MIPS), the Sun Microsystems SPARC (SPARC), the DEC VAX 11/780 (VAX), and the Motorola 68010 (68k) [DEC77] [Fujitsu87] [Kane89] [Motorola89]. The traces are divided into six workloads, so the individual characteristics of the workloads can be observed in the results [Lee84]. The workloads include COMP (MIPS compiler related traces), FP (MIPS floating point traces), TEXT (MIPS text processing traces), SPARC (SPARC traces), 68k (68010 traces), and VAX (VAX traces). The large number of traces for the MIPS reflects the fast, flexible, and bug free tracer "pixie" that MIPS Co. provides for its computers. The small number of traces for the SPARC is due to difficulty with the

SPARC tracer. The 68k traces are described in [Grochowski86]. The VAX traces are described in [Henry83]. In general, the traces, in order of significance (and confidence in the results they generate), are the MIPS, the SPARC, the 68k, and the VAX.

In some of our discussions, we refer to "static" and "dynamic" instructions. The instructions in the execution trace are dynamic instructions. The instructions in the executed program that appear in the trace are static instructions. Thus, with a two instruction loop that is executed 10 times, there are two static instructions, and 20 dynamic instructions.

The BTB simulators used in this paper use a reduced trace as input, which is composed of all control transfer (branch) instructions (which average 19 percent of the "static" instructions), both conditional and unconditional [DEC77] [Fujitsu87] [Kane89] [Motorola89]. As a result, all of the data presented is for BTBs that handle both conditional and unconditional branches.

The individual workloads and traces are described below. For a summary of statistics on the traces and workloads, see Table #1. A complete table of data is presented in [Perleberg89].

(1) **COMP:** MIPS Compiler Traces. This workload is comprised of traces of MIPS compiler related programs. The traces are of significant real world programs, as the names, and numbers in Table #1 show. There are seven traces in this workload:

| | |
|---|---|
| asm | MIPS Co. assembler, as01.21 |
| ccom | MIPS Co. C Compiler front end, ccom1.21 |
| fcom | MIPS Co. F77 Fortran Compiler front end, fcom1.21 |
| ld | MIPS Co. link editor, ld1.21 |
| ugen | MIPS Co. microcode generator, ugen1.21 |
| uopt | MIPS Co. microcode optimizer, uopt1.21 |
| upas | MIPS Co. Pascal Compiler front end, upas1.21 |

(2) **TEXT:** MIPS Text Traces. This workload is comprised of traces of MIPS text processing programs. There are four traces in this workload:

| | |
|---|---|
| emacs | GNU Emacs compiled on the MIPS. |
| grep | GNU grep version 1.1 compiled on the MIPS. |
| nroff | MIPS Co. version of nroff. |
| vi | MIPS Co. version of vi. |

(3) **FP:** MIPS Floating Point Traces. This workload is comprised of floating point intensive programs compiled on the MIPS using the MIPS Co. fortran77 compiler. There are four traces in this workload:

| | |
|---|---|
| doduc | F77 high energy physics program (doduc.f) [Doduc89]. |
| integ | F77 integration program (integral.f) |
| mold | F77 molecular dynamics program (moldyn.f) |
| spice | F77 circuit simulator program (spice2g6) [UCB87]. |

(4) **SPARC:** SPARC Program Traces. This workload is comprised of three traces, not quite as significant as the MIPS traces.

| | |
|---|---|
| ccom2 | Old (unknown) version of Sun 4 C Compiler front end. |
| esp2 | Trace of espresso, logic minimization program. |
| gbis2 | Trace of GNU Bison, 'yacc' like program. |

(5) **VAX:** VAX Program Traces. This workload is comprised of seven traces [Henry83].

| Table #1: Average Workload and Trace Statistics | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | COMP | TEXT | FP | SPARC | VAX | 68k | Work Ave | Trace Ave |
| CPU | MIPS | MIPS | MIPS | SPARC | VAX | 68k | | |
| **Dynamic Instr** | 2161788 | 2428066 | 2439411 | 1976181 | 1045646 | 681475 | 1788761 | 1644401 |
| **Static Instr** | 15737 | 11696 | 14157 | 4860 | 2832 | 3978 | 8877 | 8619 |
| **Object Instr** | 54516 | 48287 | 60658 | 43235 | na | na | na | na |
| **Object Bytes** | 218064 | 193148 | 242632 | 172940 | na | na | na | na |
| **Stat Branch Instr** | 17.47 | 19.93 | 12.29 | 21.11 | 26.05 | 23.69 | 20.09 | 20.71 |
| **Stat Branch Bytes** | 17.47 | 19.93 | 12.29 | 21.11 | 13.06 | 20.76 | 17.44 | 17.23 |
| **Stat Cond Instr** | 9.74 | 12.97 | 8.03 | 11.77 | 16.04 | 11.60 | 11.69 | 11.90 |
| **Stat Uncond Instr** | 7.73 | 6.96 | 4.26 | 9.34 | 10.01 | 12.10 | 8.40 | 8.81 |
| **Branch Taken** | 72.84 | 67.80 | 69.91 | 59.29 | 70.79 | 69.50 | 68.36 | 69.39 |
| **Cond Taken** | 64.48 | 63.44 | 59.49 | 44.18 | 61.14 | 56.62 | 58.23 | 59.73 |
| **Dyn Blk Instr** | 5.65 | 5.00 | 10.38 | 4.60 | 3.38 | 4.27 | 5.55 | 5.26 |
| **Dyn Blk Bytes** | 22.61 | 19.98 | 41.53 | 19.72 | 14.26 | 11.97 | 21.68 | 20.22 |
| **Stat Blk Instr** | 5.86 | 5.03 | 8.92 | 4.75 | 3.96 | 4.25 | 5.46 | 5.27 |
| **Stat Blk Bytes** | 23.46 | 20.10 | 35.68 | 19.01 | 17.60 | 14.25 | 21.68 | 20.85 |
| **Obj Blk Instr** | 5.38 | 4.70 | 7.08 | 4.46 | na | na | na | na |
| **Obj Blk Bytes** | 21.52 | 18.79 | 26.56 | 17.85 | na | na | na | na |

| Definitions | |
|---|---|
| Dynamic Instr | Average number of dynamic instructions |
| Static Instr | Average number of static instructions |
| Object Instr | Average number of instructions in the entire program (object text segment) |
| Object Bytes | Average number of bytes in the entire program (object text segment) |
| Stat Branch Instr | Percentage static branch instructions of all static instructions |
| Stat Branch Bytes | Percentage bytes of static branch instructions of all static instruction bytes |
| Stat Cond Instr | Percentage static conditional branch instructions of all static instructions |
| Stat Uncond Instr | Percentage static unconditional branch instructions of all static instructions |
| Branch Taken | Percentage of branches that are taken |
| Cond Taken | Percentage of conditional branches that are taken |
| Dyn Blk Instr | Dynamic basic block size in instructions, including branch instruction |
| Dyn Blk Bytes | Dynamic basic block size in bytes, including branch instruction |
| Stat Blk Instr | Static basic block size in instructions, including branch instruction |
| Stat Blk Bytes | Static basic block size in bytes, including branch instruction |
| Obj Blk Instr | Object (text segment) basic block size in instructions, including branch instruction |
| Obj Blk Bytes | Object (text segment) basic block size in bytes, including branch instruction |
| Work Ave | Average of all the workload average values |
| Trace Ave | Average of all of the individual trace values |

**Table #1:** This table contains statistics on the 32 traces we used. The values are all averages, including the workload averages (COMP, TEXT, FP, SPARC, VAX, and 68k), the overall trace average for the 32 traces (Trace Ave), and the average of the workload averages (Workload Ave). More information can be found in [Perleberg89].

| | |
|---|---|
| awk | Unix awk |
| ls | Unix ls |
| otmdl | Parser/Constructor, written in Pascal, uses set operations |
| sedx | Unix stream editor sedx |
| spic | Spice circuit simulator |
| troff | Unix troff |
| ymerge | Parse table compacter, written in C |

(6) **68k:** 68010 Program Traces. This workload is comprised of seven traces [Grochowski86].

| | |
|---|---|
| as | Unix assembler as |
| egrep | Unix egrep utility |
| fort1 | F77 matrix manipulation program |
| fort1a | Trace of 'fort1' compiled with optimizer |
| ls | Unix ls program with '-R' flag set |
| stat | Trace analyzer 'stat' |
| stata | Trace of 'stat' compiled with optimizer |

## 4. BTB Design Optimization

The theme of this paper is to maximize BTB performance with a limited number of bits allocated to the BTB implementation. Describing this problem mathematically helps to clarify the issues. To begin, it is necessary to understand the performance impact of different amounts and types of information stored for each branch in the BTB.

Potential BTB performance increases as the information content of each entry of the BTB increases, and as the number of entries increases. Increased information content requires more bits of storage, as does an increased number of entries. The simplest BTB with the lowest level of performance contains only prediction information. Predicting the branch direction can reduce the branch penalty. A medium complexity BTB which gives equal or higher performance contains the branch tag, prediction information, and the branch target address. With the branch target address available before it is normally calculated, fetching can occur earlier at the branch target. A high complexity BTB with still higher performance contains the branch tag, prediction information, target address, and instruction bytes from the target address. This design can decrease the delay in fetching instructions from the target. Increasing the number of types of information (e.g. prediction information, target address, target instruction bytes) stored for each branch improves performance. In addition, increasing (per entry) the amount (bits of storage) of each type of information also improves potential performance.

An optimization problem exists if a maximum performance BTB is to be designed with a limited number of bits. As both greater numbers of entries and greater amounts of information per entry increase performance, the problem is selecting the number of entries and the amount of information per entry that will maximize performance. Which design provides better performance, a large number of entries with a small amount of information per entry, or a small number of entries with a large amount of information per entry?

The following simple mathematical BTB model is intended to help clarify the problem. This model gives insight into how to maximize the performance of a BTB design that has been added onto a processor that normally predicts all branches not taken. The equation presents the probable savings per branch of an optimal BTB designed with $N$ storage bits. Some

simplifications have been made in setting up this equation, as noted below.

$$Max\left\{\sum_i h(i)\left[t(i)ptt(i)V(i) - (1-t(i))ptnt(i)W(i)\right]\right\} \quad such\ that \quad \sum_i num\_bits(i) \le N$$

where:

$N$ = number of storage bits in the BTB

$h(i)$ = probability that branch $i$ is referenced (executed)

$t(i)$ = probability that branch $i$ will be taken

$ptt(i)$ = probability of predicting branch $i$ taken, given that branch $i$ is taken

$ptnt(i)$ = probability of predicting branch $i$ taken, given that branch $i$ is not taken

$V(i)$ = cycle savings for correct taken prediction of branch $i$

$W(i)$ = cycle cost for incorrect taken prediction of branch $i$

$num\_bits(i)$ = number of bits in BTB allocated to branch $i$

Note that $ptt(i)$ and $ptnt(i)$ are zero for branches not in the BTB, so the sum is only effected by branches in the BTB.

The first product in the equation, when summed up, is the expected cycle savings due to the BTB, in the case of a correct taken prediction. The second product in the equation, summed up, is the expected cycle cost of an incorrect taken prediction. The difference of these two sums is the net cycle savings per branch due to the BTB. The case of a correct not taken prediction has no performance penalty with or without the BTB (unless branch folding is used, in which case there may be a savings), so it does not appear in the equation. The case of incorrect not taken prediction is not included, since the cost is the same with or without a BTB (except when the BTB contains target instructions).

The problem of selecting the best BTB design is described by the equation above. For a fixed number of bits, increasing the performance of entries in the BTB is equivalent to increasing $V(i)$ and decreasing $W(i)$, and decreases the number of entries. Increasing the number of entries in the BTB adds more branches into the sum, and decreases the information per entry. Finding the best performance for a limited number of total bits in the BTB requires a careful balance between performance per entry and number of entries.

The simple BTB model supports three concepts used later in this paper.

(i) *If a branch does not have potential for improving performance, do not enter it into the BTB.* The BTB model indicates that the BTB should be filled with the branches that have the most potential for improving performance. Placing into the BTB branches with no performance value displaces branches that may have performance value.

(ii) *When it is necessary to discard a branch from the BTB, discard the branch with the least potential performance value.* The BTB should contain the branches with the most potential for improving performance. A replacement strategy based on this concept might discard the branch that is least likely to be referenced AND least likely to be taken. A not taken branch that is in the BTB has the same performance as one that is not in the BTB, so discarding a branch that is not likely to be taken has little penalty.

(iii) *A multi-level BTB, each level possibly containing different amounts/types of information per entry, may be able to maximize performance by achieving a better balance of number of entries and quantity of information per entry.* Note that $V(i)$ and $W(i)$ in the BTB model are functions of $i$. The BTB performance may be maximized if $V(i)$ and $W(i)$ are not

constant. If branches with high performance potential are allocated more bits per entry, and branches that have less performance potential are allocated fewer bits per entry (but more entries), higher overall performance may be achieved. The constraint of a limited number of storage bits for the BTB design exists, so not all branches can be given the large size (higher performance) BTB entries.

## 5. Branch Prediction

In this paper, dynamic branch prediction is used, in which predictions are based on probability of taken branch statistics (as a function of $n$ bits of branch execution history, known as prediction bits) compiled from all 32 traces. Since every branch has a startup period during which $n$ bits of execution history are not available (only 1 to $n$-1 bits are available in this period), statistics were also compiled for partial bit strings of execution history, from 1 to $n$-1. The statistics give the measured probability of a taken branch for the 32 traces as a function of $n$ bits of execution history. As a result, predictions based on these statistics are the best possible predictions (yielding the highest correct prediction rate) for the 32 traces of any fixed, (i.e. non-learning or adaptive) method that only uses $n$ bits of execution history as input. Probability of taken and probability of occurrence statistics for 6 branch history bits for each of the workloads and for the 32 trace average are given in [Perleberg89]. These statistics are appropriate for BTBs that enter branches on the first execution. Similiar statistics for BTBs that enter branches on taken branch executions only can also be found in [Perleberg89].

Achieving the highest prediction rate is not the same as achieving the highest performance. Four cases are possible when a prediction is made: predict not taken, branch not taken (zero cycle cost); predict not taken, branch taken (cycle cost $b$); predict taken, branch not taken (cycle cost $c$); and predict taken, branch taken (cycle cost $d$). Maximizing the prediction rate is accomplished by predicting branch taken whenever the probability $p$ of a taken branch is greater than 50 percent. Maximizing performance (minimizing cycle cost) is accomplished by predicting branch taken when the cost of predicting taken is less than the cost of predicting not taken (i.e. when $(1-p)*c + p*d < p*b$). As this equation requires implementation dependent numbers ($b$,$c$,$d$), all prediction rates presented in this paper are based on predicting taken if the probability of taken is greater than 50 percent. Later, in our discussion of multi-level BTBs, for which a single prediction rate is not a useful measure, performance is maximized using a set of cycle costs for the four cases described above.

### 5.1. Best Case Prediction Rates

As a reference point, it is useful to know the realistic upper limit for prediction rates. To determine this upper limit, two infinite size BTB simulators were designed (BTB#1 and BTB#2). Design BTB#1 takes in branches on their first execution and keeps them in the BTB until the end of the trace. Design BTB#2 takes in branches on their first taken execution and keeps them in the BTB until the end of the trace. Both BTBs predict all instructions that are not in the BTB as not taken. Note that the BTB prediction rate is independent of the amount of information per entry in the BTB, although the BTB must contain prediction bits and a branch tag (i.e. this is not a hash table BTB).

BTB#1 and BTB#2 prediction rates are presented in Table #2 (the full set of results is presented in [Perleberg89]). The prediction rates in each column under a workload name are simple workload averages, and the **Average** column is the average of the workload averages. Table #2 includes the effect of target address changes in the prediction rate by considering a target change occurring during a taken branch to be a misprediction. Note that target changes do not affect a BTB that contains only prediction bits (no target address or instructions are stored). It happens that virtually all target changes occur for branches that are always taken (e.g. subroutine

| Table #2: Branch Prediction Rates (including target change effects) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Prediction Bits** | **BTB Design** | **COMP** | **TEXT** | **FP** | **SPARC** | **VAX** | **68k** | **Average** |
| 0 | BTB#1 | 67.21 | 66.13 | 61.78 | 56.80 | 68.85 | 64.96 | 64.29 |
| 0 | BTB#2 | 81.27 | 80.86 | 81.04 | 73.04 | 81.54 | 76.68 | 79.07 |
| 1 | BTB#1 | 86.30 | 94.83 | 85.03 | 87.51 | 89.71 | 85.55 | 88.15 |
| 1 | BTB#2 | 86.30 | 94.83 | 85.03 | 87.51 | 89.71 | 85.55 | 88.15 |
| 2 | BTB#1 | 87.81 | 95.45 | 86.72 | 88.69 | 91.56 | 86.27 | 89.42 |
| 2 | BTB#2 | 87.81 | 95.45 | 86.72 | 88.69 | 91.56 | 86.27 | 89.42 |
| 4 | BTB#1 | 88.66 | 96.02 | 87.85 | 90.61 | 93.36 | 88.21 | 90.79 |
| 4 | BTB#2 | 88.66 | 96.02 | 87.84 | 90.61 | 93.35 | 88.20 | 90.78 |
| 8 | BTB#1 | 89.19 | 96.18 | 89.28 | 91.78 | 93.70 | 89.00 | 91.52 |
| 8 | BTB#2 | 89.19 | 96.18 | 89.27 | 91.77 | 93.67 | 88.97 | 91.51 |
| 16 | BTB#1 | 90.14 | 96.40 | 89.90 | 92.93 | 95.25 | 91.48 | 92.68 |
| 16 | BTB#2 | 90.13 | 96.39 | 89.89 | 92.91 | 95.22 | 91.45 | 92.67 |

**Table #2:** Prediction Rates for BTB#1 and BTB#2, both of infinite size. BTB#1 enters all branches as they are executed. BTB#2 enters branches only as they are taken. These prediction rates are upper limits on the prediction rates possible from finite BTBs.

| Table #3: Prediction Rate Reduction due to Target Change | | | | | | |
|---|---|---|---|---|---|---|
| **COMP** | **TEXT** | **FP** | **SPARC** | **VAX** | **68k** | **Average** |
| 5.20 | 1.41 | 7.78 | 2.35 | 1.61 | 4.18 | 3.75 |

**Table #3:** This table indicates the constant reduction of the prediction rate due to the effect of target address changes.

| Table #4: Lee and Smith's Percentage Target Changes [Lee84] | | | | | | |
|---|---|---|---|---|---|---|
| **IBM/CPL** | **IBM/BUS** | **IBM/SCI** | **IBM/SUP** | **PDP11** | **CDC6400** | **Average** |
| 4.2 | 2.1 | 4.4 | 1.4 | 12.0 | 2.9 | 4.5 |

**Table #4:** Lee and Smith's percentage probability of target change for their workloads [Lee84].

returns) and thus the effect of target changes is to decrease prediction accuracy by a constant amount. Those constants can be found in Table #3, and are identical for BTB#1 and BTB#2. Equivalent constants for the six workloads of Lee and Smith are shown in Table #4 [Lee84]. Figure #1 shows the average prediction rate for BTB#2 (including and not including the effect of target changes) as a function of the number of prediction bits.

The BTB#1 and BTB#2 prediction rates are very similiar, except for the case of zero prediction bits. For zero prediction bits, the BTB#2 prediction rate is higher than BTB#1's prediction rate. This is due to the fact that for zero prediction bits, BTB#2 only predicts as taken those branches that are in the BTB. Since BTB#2 only enters branches into the BTB on a taken branch

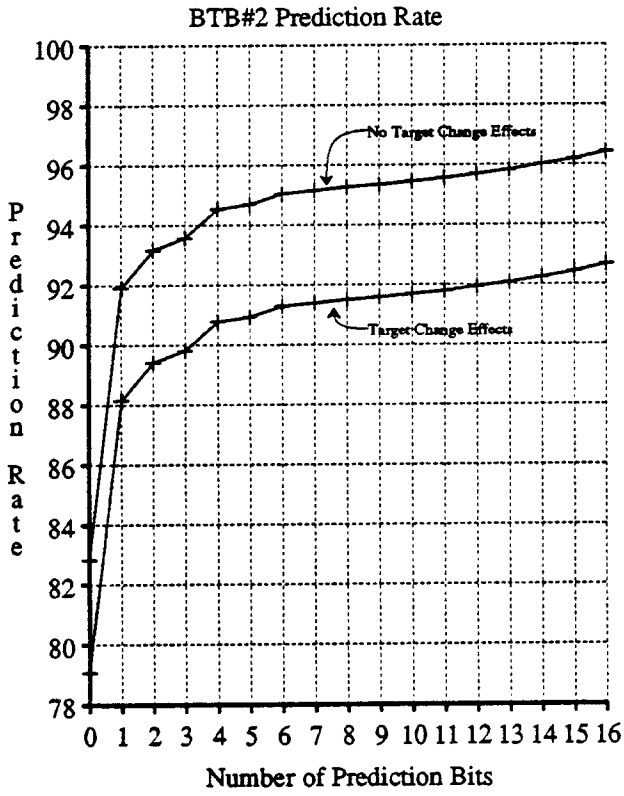


**Figure #1:** BTB#2 (best case) prediction rates as a function of number of prediction bits.

execution, branches that are never taken or are initially not taken are correctly predicted, thus boosting the prediction rate. BTB#1 enters branches into the BTB on the first execution, predicting the branches as taken thereafter, for the zero prediction bit case. Thus, the prediction rate of BTB#1 is approximately the average percentage of taken branches. Note that even with zero prediction bits, a BTB can be useful by caching the branch target and instructions at the branch target; in such a design, all branches in the BTB are predicted as taken.

| Table #5: Prediction Rate Comparison | | | |
|---|---|---|---|
| **Prediction Bits** | **Lee & Smith** | **BTB#1** | **BTB#2** |
| 0 | 67.4 | 68.0 | 82.8 |
| 1 | 88.7 | 91.9 | 91.9 |
| 2 | 92.0 | 93.2 | 93.2 |
| 3 | 92.7 | 93.6 | 93.6 |
| 4 | 93.3 | 94.5 | 94.5 |
| 5 | 93.5 | 94.7 | 94.7 |

**Table #5:** Comparison of BTB#1 and BTB#2 prediction rates to those of Lee and Smith [Lee84].

The prediction rates are very close to those of Lee and Smith [Lee84]. Table #5 contains three columns of prediction rates, one for the average of prediction rates from Lee and Smith, one for the BTB#1 prediction rate (calculated in a similiar fashion to Lee and Smith), and one for the BTB#2 prediction rate. These prediction rates do not include the effect of target address changes, so our values are from Table #2 with the **Average** constant from Table #3 added in. Lee and Smith use the percentage of taken branches as the prediction rate for zero prediction bits, basically the same as BTB#1, which is not comparable to the BTB#2 zero prediction bit case for reasons explained above. Lee and Smith generated prediction rates for up to 5 prediction bits. For up to four prediction bits, the prediction rate increases sharply. For more than 4 prediction bits, Figure #1 indicates that the prediction rate grows slowly.

## 5.2. Effect of Context Switching on Prediction Rates

Multitasking computer systems frequently experience context switches. During a context switch, the processor stops executing one program and begins executing another. The BTB normally contains invalid data after a context switch, and is flushed. As a result, context switching has a detrimental effect on the prediction rate of the BTB. In order to observe this effect, the infinite size BTB#2 simulator was altered to flush the BTB after every $N$ instructions. Prediction rates were generated for $N$ equal to 1000, 3160, 10000, 31600, 100000, 316000, and 1000000. The results are shown in Figure #2, with curves for 0, 1, 2, 4, 8, and 16 prediction bits (the full set of results are presented in [Perleberg89]). Each curve represents the average prediction rate of the six workload averages. The prediction rates include the effect of branch target changes (i.e. target change on a taken branch is considered a misprediction).

Examining Figure #2, it is apparent that context switching significantly reduces the prediction rate, and correspondingly, the performance benefit of a BTB. With $N$=10000, two prediction bits attain a 84 percent prediction rate while with no context switching they attain a 89 percent prediction rate. For context switches that occur less than every 10000 instructions, the effectiveness of adding additional prediction bits to increase the prediction rate is limited. For $N$ above 10000 instructions, adding prediction bits has a stronger effect on the prediction rate. Some of the traces contain fewer instructions than $N$ (for some $N$), so that for $N$ greater than the number of instructions in the trace, the context switching does not effect the prediction rate for the trace.

The curve for zero prediction bits in Figure #2 is interesting. Apparently, with zero prediction bits, context switching can increase the prediction rate. This anomally is due to the characteristics of the zero prediction bit BTB, discussed in the previous section. The zero prediction bit BTB correctly predicts taken branches that are in the BTB, and not taken branches that are not in the BTB. Context switches clear the BTB of branches that are experiencing not taken executions, allowing them to be predicted correctly. These correct predictions counter the mispredictions it takes to get taken branches back into the BTB, producing a maximum prediction rate for a finite context switch rate.

# 6. BTB Management

Management of BTBs is concerned with the issue of entering and discarding branches from the BTB. Good management schemes significantly improve the performance of a BTB. In this section, the first two optimization concepts presented in the **BTB Design Optimization** section are used to present alternative BTB management schemes.

## 6.1. Entering Branches into the BTB

The first concept presented in the **BTB Design Optimization** section is: *If a branch does not have the potential for improving performance, do not enter it into the BTB*. Two methods for entering branches into the BTB are evaluated here. The first method is to enter branches into the

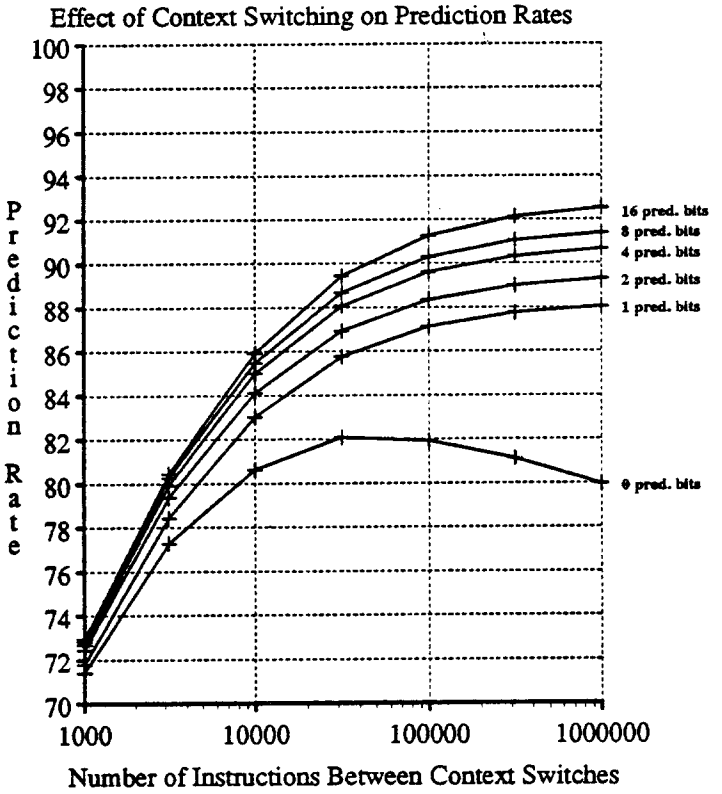


**Figure #2:** Effect of context switching on the prediction rate of BTB#2. Each of the curves is for BTB#2 with a constant number of prediction bits.

BTB when branches are first executed (design BTB#3). This obvious method has an apparent drawback. If a branch is entered into the BTB on a not taken execution, the BTB will not improve performance while the branch continues not taken executions, and will mispredict the first taken execution (with no performance improvement) that the branch experiences. In addition, if the BTB contains target instruction bytes, entering the branch on a not taken execution requires extra memory bandwidth to fetch the target instruction bytes. The second method for entering branches into the BTB is to enter branches on their first taken execution (design BTB#4). This method avoids entering into the BTB entries of limited usefulness, which may themselves displace more useful entries. Thus, the BTB does not have to deal with branches that are never taken and which have no performance penalty that the BTB can reduce. If the BTB contains target instructions, entering branches on taken execution allows the target instructions to be cached in the BTB at no cost, since they are normally fetched on a taken branch execution.

The two methods of entering branches into the BTB were simulated. Design BTB#3 enters branches into the BTB on the first branch execution, is set associative, and uses LRU (least recently used) replacement. Design BTB#4 enters branches into the BTB on the first branch taken execution, is set associative, and uses LRU replacement. Both BTBs use branch tags to uniquely identify the branches. The BTBs were simulated for 0 to 16 prediction bits, 16 to 2048 entries, and set sizes of 1 to 16. The prediction rates presented are averages of the workload averages, and include the effect of target address changes (i.e. a target change on a taken branch execution is considered an incorrect prediction).

| Table #6: Comparison of Methods of Entering Branches into a BTB | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Number of BTB Entries | | | | | | | |
| Prediction Bits | BTB Design | Set Size | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 |
| 0 | BTB#3 | 4 | 47.60 | 50.63 | 55.58 | 58.97 | 61.46 | 63.18 | 63.92 | 64.18 |
| 0 | BTB#4 | 4 | 62.72 | 69.47 | 74.68 | 77.10 | 78.32 | 78.96 | 79.00 | 79.05 |
| 1 | BTB#3 | 4 | 59.39 | 67.68 | 75.31 | 80.67 | 84.20 | 86.58 | 87.63 | 88.01 |
| 1 | BTB#4 | 4 | 63.61 | 71.27 | 77.92 | 82.11 | 85.28 | 87.10 | 87.83 | 88.08 |
| 2 | BTB#3 | 4 | 59.74 | 68.20 | 76.22 | 81.72 | 85.36 | 87.81 | 88.88 | 89.27 |
| 2 | BTB#4 | 4 | 64.01 | 72.03 | 78.88 | 83.21 | 86.47 | 88.33 | 89.09 | 89.33 |
| 4 | BTB#3 | 4 | 59.97 | 68.59 | 76.89 | 82.65 | 86.51 | 89.11 | 90.23 | 90.64 |
| 4 | BTB#4 | 4 | 64.25 | 72.50 | 79.57 | 84.19 | 87.65 | 89.64 | 90.43 | 90.70 |
| 8 | BTB#3 | 4 | 60.13 | 68.81 | 77.31 | 83.15 | 87.13 | 89.79 | 90.95 | 91.37 |
| 8 | BTB#4 | 4 | 64.43 | 72.85 | 80.01 | 84.70 | 88.30 | 90.34 | 91.16 | 91.42 |
| 16 | BTB#3 | 4 | 60.29 | 69.02 | 77.61 | 83.71 | 87.94 | 90.82 | 92.07 | 92.52 |
| 16 | BTB#4 | 4 | 64.60 | 73.09 | 80.41 | 85.35 | 89.19 | 91.40 | 92.29 | 92.58 |

**Table #6:** Comparison of two methods of entering branches into a BTB. Higher prediction rate indicates higher performance. BTB#3 enters branches into the BTB on the first branch execution, while BTB#4 enters branches on the first branch taken execution.

Table #6 presents the prediction rates for the two BTB designs, for set size of 4, and number of prediction bits 0, 1, 2, 4, 8, and 16 (the full set of results is presented in [Perleberg89]). As expected, BTB#4 significantly outperforms BTB#3 in every case, but especially for low numbers of entries in the BTB. Entering branches into the BTB on the first taken execution avoids placing useless information in the BTB that displaces useful information. For the 32 traces, 17 percent of all static branches are never taken, and these never taken branches constitute 12 percent of all dynamic branch executions. These branches never enter BTB#4, but they enter BTB#3, hurting performance. Not only does BTB#4 have higher performance, it also has a simplier design. The BTB is notified to enter a branch whenever a branch is taken that does not exist in the BTB. If the BTB contains target instructions, they can be stored into the BTB as the instruction fetch stage of the processor fetches from the target address of the taken branch.

Table #6 indicates that the number of entries in a BTB has a significant impact on the prediction rate of the BTB. With two prediction bits, prediction rates range from 64 percent with 16 entries to 89 percent with 2048 entries. The growth in prediction rates (as a function of BTB size) levels off after 512 entries, so for the traces used, a 512 entry BTB would have about the same performance as a 2048 entry BTB, at a much lower cost.

## 6.2. Discarding Branches from the BTB

The second concept presented in the **BTB Design Optimization** section is: *When it is necessary to discard a branch from the BTB, discard the branch with the least performance value.* It is not possible to determine the branch with the least performance value in a real system, but it is possible to find a branch that probably has the least performance value. The Least Recently Used (LRU) algorithm is known by experience to work well for replacing (discarding) entries in caches, although the random algorithm works better with small caches, avoiding the worst case behavior of the LRU algorithm for loops that are larger than the small cache

[SmithJ83]. The LRU algorithm replaces the entry that has been accessed least recently. Design BTB#4, described above, uses LRU replacement.

A BTB has an added level of complexity over a cache, since each entry can contain prediction bits. The prediction bits provide more information that can be used in finding the branch with the least performance value. If a branch's prediction bits strongly predict not taken, it may be reasonable to discard the branch, as the BTB does not contribute to the performance of a not taken branch. But if this branch has a high probability of being referenced, it will probably be executed soon, and may predict taken in the future, and thus probably has some performance value. What is needed is a algorithm that replaces the branch that is least likely to be used and least likely to be taken. The Minimum product of Probability of reference and Probability of taken (MPP) algorithm is a good approximation. This algorithm replaces the branch with the minimum product of probability of reference and probability of branch taken. The probability of reference can be obtained empirically as a function of LRU bits that the algorithm maintains. The LRU bits indicate the order in which the branches have been referenced. The probability of branch taken is obtained from the branch prediction statistics. Table #7 gives the probability of reference values for three BTB sizes and a set size of four (LRU bits=0 is the most recently referenced entry in the set, LRU bits=3 is the least recently referenced entry in the set). Table #8 gives the probability of a taken branch as a function of three prediction bits of branch history. Table #9 presents an ordered list of the possible MPP products with a 128 entry 4 way set associative three prediction bit BTB (combination of Tables #7 and #8, disregards partial history strings). Probability of taken and probability of occurrence statistics for 6 branch history bits for each of the workloads and for the 32 trace average can be found in [Perleberg89].

| Table #7: Probability of Reference (Set Size of 4) | | | | |
|---|---|---|---|---|
| | LRU bits: Order of Reference | | | |
| BTB Entries | 0 | 1 | 2 | 3 |
| 16 | .5111 | .2199 | .1464 | .1225 |
| 32 | .5504 | .2123 | .1405 | .0969 |
| 64 | .6147 | .2030 | .1073 | .0750 |
| 128 | .7016 | .1729 | .0812 | .0443 |
| 256 | .7771 | .1401 | .0550 | .0277 |
| 512 | .8450 | .1060 | .0337 | .0153 |
| 1024 | .8967 | .0781 | .0188 | .0064 |
| 2048 | .9417 | .0474 | .0084 | .0025 |

**Table #7:** For several sizes of a 4 way set associative BTB, this table gives the probability of reference of the entries in each set as a function of the LRU bits. The most recently accessed entry of the set has LRU bits=0, while the least recently accessed entry of the set has LRU bits=3.

The MPP algorithm can be implemented relatively cheaply. A set size of two or four entries is common in caches, so only one or two LRU bits are required. Context switching has a severe impact on the performance of large numbers of prediction bits so one to five prediction bits is reasonable. Probability of reference is a function of the LRU bits, and probability of taken is a function of the prediction bits. Therefore, at most seven bits of input are required to calculate the product for MPP for each entry. A total of four input bits will probably be typical. Four input bits indicate 16 possible values. What is important is not the actual value of each product of

| Table #8: Probability of Taken Branch | | |
|---|---|---|
| **Branch History N = Not_taken T = Taken** | **Probability of Occurrence** | **Probability of Taken Branch** |
| NNN | .1305 | .0780 |
| NNT | .0151 | .3405 |
| NTN | .0207 | .5186 |
| NTT | .0249 | .6794 |
| TNN | .0154 | .3258 |
| TNT | .0307 | .6469 |
| TTN | .0250 | .7914 |
| TTT | .7377 | .9766 |

**Table #8:** As a function of a branch's history, this table gives the probability that the next execution of the branch will be taken, and the probability with which this branch history will occur.

probability of reference and probability of taken but the relative ordering of the products. Each branch's product can therefore be encoded into four bits, and can be generated by a simple four bit table with four bits of input. The four bit results of the branches in a set can be quickly compared to find the branch with the minimum product.

To compare the LRU and MPP algorithms, two BTB simulators were constructed. Design BTB#4 is set associative, uses LRU replacement, and enters branches on taken executions. Design BTB#5 is set associative, uses MPP replacement, and enters branches on taken executions. Both BTBs contain branch tags and prediction bits. The BTBs were simulated for 0 to 16 prediction bits, 16 to 2048 entries, and set sizes of 1 to 16. The prediction rates presented are averages of workload averages, and the effect of target changes is included in the prediction rate (i.e. a target change on a taken execution is a misprediction). The full table of prediction rate data is in [Perleberg89]. Table #10 contains prediction rate data for BTB#4 and BTB#5 with set sizes of four and number of predictions bits 0, 1, 2, 4, 8, and 16. Examining Table #10, the MPP replacement algorithm (BTB #5) outperforms the LRU replacement algorithm (BTB #4) except for a few BTB sizes with 16 prediction bits. However, the prediction rate difference between the two designs is trivial, and probably does not justify the extra complexity of the MPP design.

## 7. Relationship between BTB and Instruction Cache Miss Rates

We hypothesized that a BTB and a instruction cache should have similiar miss rates if the following conditions are met:

(i) *The cache line is one instruction in length.*

(ii) *The cache size is the BTB size (number of entries) multiplied by the average basic block size.*

Our reasoning is that a BTB is simply an instruction cache that only holds branches. Assume the basic block size is $n$ instructions (lines). One branch exists in every basic block (i.e. on average, one of every $n$ instructions is a branch). Therefore, the instruction cache has $n$ misses for every miss in the BTB and $n$ hits for every hit in the BTB. If the two conditions above are met, the miss ratio should be the same.

| Table #9: Order of Possible MPP Products for 128 Entry 4 Way Set Associative Three Prediction Bit BTB | | |
|---|---|---|
| **LRU Bits** | **Prediction Bits** | **MPP Products** |
| 0 | TTT | .6852 |
| 0 | TTN | .5552 |
| 0 | NTT | .4767 |
| 0 | TNT | .4539 |
| 0 | NTN | .3638 |
| 0 | NNT | .2389 |
| 0 | TNN | .2286 |
| 1 | TTT | .1689 |
| 1 | TTN | .1368 |
| 1 | NTT | .1175 |
| 1 | TNT | .1118 |
| 1 | NTN | .0897 |
| 2 | TTT | .0793 |
| 2 | TTN | .0643 |
| 1 | NNT | .0589 |
| 1 | TNN | .0563 |
| 2 | NTT | .0552 |
| 0 | NNN | .0547 |
| 2 | TNT | .0547 |
| 3 | TTT | .0433 |
| 2 | NTN | .0421 |
| 3 | TTN | .0351 |
| 3 | NTT | .0301 |
| 3 | TNT | .0287 |
| 2 | NNT | .0276 |
| 2 | TNN | .0230 |
| 3 | NTN | .0230 |
| 3 | NNT | .0151 |
| 3 | TNN | .0144 |
| 1 | NNN | .0135 |
| 2 | NNN | .0063 |
| 3 | NNN | .0035 |

**Table #9:** Table lists in order of MPP product value the 32 possible products for a 128 entry 4 way set associative BTB with three prediction bits. The products from branch history strings of less than three are not presented. Note that the MPP algorithm would replace the entry (one of four) that has the minimum product value.

| Table #10: Comparison of BTB Replacement Strategies | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Number of BTB Entries | | | | | | | |
| Prediction Bits | BTB Design | Set Size | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 |
| 0 | BTB#4 | 4 | 62.72 | 69.47 | 74.68 | 77.10 | 78.32 | 78.96 | 79.00 | 79.05 |
| 0 | BTB#5 | 4 | 62.72 | 69.47 | 74.68 | 77.10 | 78.32 | 78.96 | 79.00 | 79.05 |
| 1 | BTB#4 | 4 | 63.61 | 71.27 | 77.92 | 82.11 | 85.28 | 87.10 | 87.83 | 88.08 |
| 1 | BTB#5 | 4 | 63.92 | 71.50 | 78.14 | 82.28 | 85.35 | 87.13 | 87.84 | 88.08 |
| 2 | BTB#4 | 4 | 64.01 | 72.03 | 78.88 | 83.21 | 86.47 | 88.33 | 89.09 | 89.33 |
| 2 | BTB#5 | 4 | 64.17 | 72.11 | 79.02 | 83.26 | 86.52 | 88.35 | 89.09 | 89.34 |
| 4 | BTB#4 | 4 | 64.25 | 72.50 | 79.57 | 84.19 | 87.65 | 89.64 | 90.43 | 90.70 |
| 4 | BTB#5 | 4 | 64.50 | 72.56 | 79.68 | 84.25 | 87.71 | 89.65 | 90.44 | 90.70 |
| 8 | BTB#4 | 4 | 64.43 | 72.85 | 80.01 | 84.70 | 88.30 | 90.34 | 91.16 | 91.42 |
| 8 | BTB#5 | 4 | 64.69 | 72.91 | 80.11 | 84.77 | 88.34 | 90.35 | 91.16 | 91.43 |
| 16 | BTB#4 | 4 | 64.60 | 73.09 | 80.41 | 85.35 | 89.19 | 91.40 | 92.29 | 92.58 |
| 16 | BTB#5 | 4 | 64.85 | 73.14 | 80.46 | 85.33 | 89.19 | 91.39 | 92.29 | 92.57 |

**Table #10:** This table contains a comparison of the MPP and LRU replacement strategies. BTB#4 uses LRU replacement and BTB#5 uses MPP replacement. The MPP strategy has a slightly higher prediction rate in most cases.

The hypothesis was tested with a BTB and a cache simulator. The BTB simulator (design BTB#3, the same design as was used previously) is set associative with LRU replacement and enters branches on their first execution (both taken and not taken). The BTB sizes are powers of two. The cache simulator is set associative with LRU replacement and a four byte line size. Cache sizes are not powers of two since the cache size is equal to the BTB size multiplied by the basic block size. The six workloads were fed into the simulators, each workload using its own average basic block size to size the cache. The BTB and cache miss rates can thus be compared for equality within each workload. A tabulation of the resulting miss rates can be found in [Perleberg89].

Figure #3 displays the cache and BTB#3 miss rates for the COMP, TEXT, and FP workloads with set size of four. Figure #4 displays the cache and BTB#3 miss rates for the SPARC, VAX, and 68k workloads with set size of four. It is immediately apparent that the cache and BTB miss rates are nearly equivalent. The four byte line size worked very well for the MIPS (COMP, TEXT, FP) and SPARC (SPARC) traces, since MIPS and SPARC only have four byte instructions. In Table #1, by dividing **Dyn Blk Bytes** with **Dyn Blk Instr**, we can calculate that the VAX traces have an average instruction length of 4.2 bytes and the 68k traces have an average instruction length of 2.8 bytes. The curves in Figure #4 for the VAX are not compensated for this slightly larger instruction length. The 68k curves in Figure #4, however, have been compensated by multiplying the cache miss rate by a factor of 4.0/2.8. The compensation is necessary since a single miss in the cache reads in 4.0/2.8 instructions instead of the assumed 1 instruction, thus lowering the cache miss rate by a factor of 2.8/4.0. Once compensated, the BTB and cache miss rates for the 68k are virtually equivalent.

With set sizes of 1 and 2, the cache and BTB miss rates differ more than for set sizes of 4 or 8. Generally, with set size of one or two, the BTB miss rate is higher than the cache miss rate. The reason for this is simple. Programs tend to have code that is slightly ordered with respect to

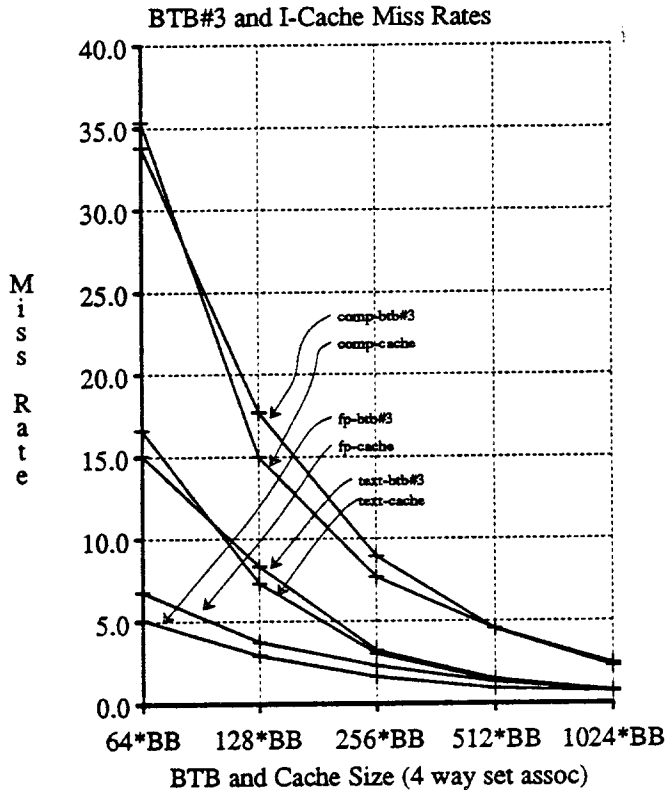


**Figure #3:** BTB#3 and corresponding instruction cache miss rates for the workloads COMP, TEXT, and FP. This figure shows that the BTB and cache miss rates can be computed from one another.

powers of two, and a cache with a power of two number of sets will have more collisions than one that has an non-power of two number of sets. The effect of collisions is greatest when the set size is small [Hill89].

The BTB#3 design (branches entered on first execution) has been shown to not perform as well as the BTB#4 design (branches entered on first taken execution). It is interesting, therefore, to compare their miss rates. The complete miss rate data for BTB#3 and BTB#4 can be found in [Perleberg89].

Figure #5 contains the BTB#3 and BTB#4 miss rates for the COMP, TEXT, and FP workloads (set size of four), and Figure #6 contains the BTB#3 and BTB#4 miss rates for the SPARC, VAX, and 68k workloads (set size of four). The miss rate curves indicate that BTB#4 has significantly lower miss rates than BTB#3.

The relationship between the BTB#3 and BTB#4 miss rates depends on the percentage of taken branches and the BTB size. BTB#3 enters branches into the BTB on the first execution of the branches, and BTB#4 enters branches on the first taken execution of the branches. Therefore, as the percentage of taken branches increases, the BTB#4 miss rate approaches the BTB#3 miss rate. The effect of BTB size on the relationship between BTB#3 and BTB#4 miss rates can be observed by examining the two cases, infinite BTB size and small BTB size. In an infinite size BTB, BTB#4 will not miss on those branches that are never taken (17 percent of static branch instructions in our traces), so the number of misses for BTB#4 is the number of misses for BTB#3 (number of static branches) minus the number of static branches that are never taken. For

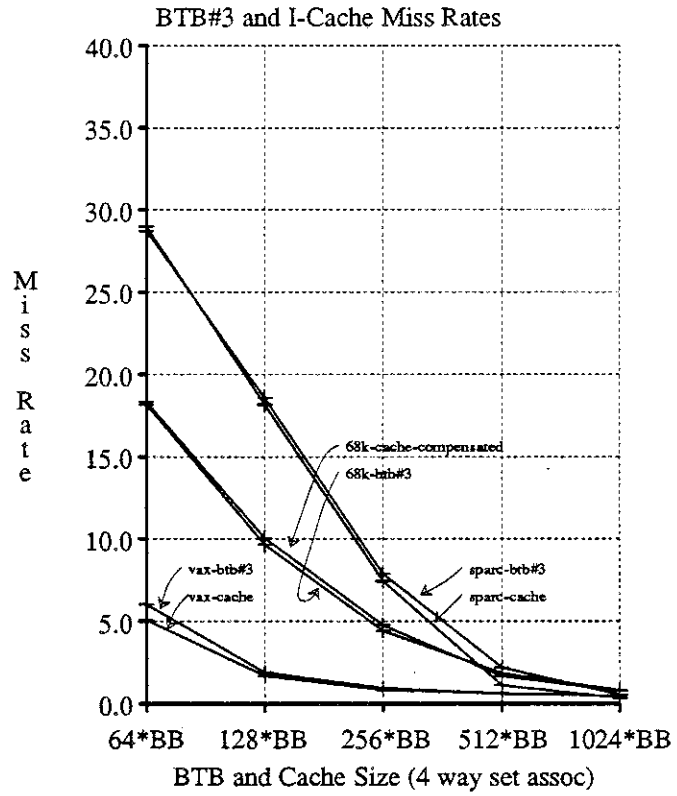


**Figure #4:** BTB#3 and corresponding instruction cache miss rates for the workloads SPARC, VAX and 68k. This figure shows that the BTB and cache miss rates can be computed from one another.

our traces, the resulting miss rate for BTB#4 is 83 percent of the BTB#3 miss rate. As BTB size decreases, entering not taken branches generates additional misses by displacing useful branches. The result is that BTB#3's miss rate grows relative to BTB#4's miss rate, as BTB size decreases.

| Table #11: Miss Rates for BTB#3 and BTB#4 and Ratio of Miss Rates BTB#4/BTB#3 | | | | |
|---|---|---|---|---|
| **Entries** | **Set Size** | **BTB#3 Miss Rate** | **BTB#4 Miss Rate** | **BTB#4/BTB#3** |
| 16 | 4 | 46.82 | 25.90 | .553 |
| 32 | 4 | 29.06 | 15.74 | .542 |
| 64 | 4 | 14.87 | 8.34 | .561 |
| 128 | 4 | 7.55 | 4.42 | .585 |
| 256 | 4 | 3.58 | 2.14 | .598 |
| 512 | 4 | 1.65 | 0.97 | .588 |
| 1024 | 4 | 0.81 | 0.52 | .642 |
| 2048 | 4 | 0.53 | 0.41 | .774 |

**Table #11:** Table gives the miss rates of BTB#3 and BTB#4 and the ratio of their miss rates as a function of BTB size.

Table #11 contains the BTB#3 and BTB#4 miss rates and their ratio as a function of BTB size for a 4 way set associative BTB. Notice that for BTB#3 miss rates that are greater than 1.0%, the BTB#4/BTB#3 ratio ranges between .542 and .598 (a relatively small range) with an average value of .57. With less than a 1.0% miss rate for BTB#3, the ratio rises sharply. A simple model of the BTB#4/BTB#3 miss rates ratio is therefore .57 for BTB#3 miss rates greater than 1.0%, and undefined for BTB#3 miss rates less than 1.0% (alternatively, this region could be assigned a worst case miss rate ratio of 1.0). This model works reasonably well for set sizes other than four as well. This model does not take into account the effect of the taken branch percentage, so workloads with taken branch percentages other than 70 percent may need a different model.

## 7.1. BTB Design Target Miss Ratios

Given the relationship that we have obtained between BTB and instruction cache miss ratios, we can derive *design target miss ratios* for BTBs. Design target miss ratios are those to be expected for an "average" workload on a real machine in normal operation. They were defined and created in [SmithAJ85] and [SmithAJ87]; the methodology is described in those papers. To convert instruction cache design target miss ratios to design target miss ratios for BTBs, the average basic block size and the average instruction size in the workloads used are needed. We have combined our data with that from some other published sources. Peuto and Shustek extracted extensive information from seven IBM 370 traces that indicate average basic block size of 24.22 bytes and an average instruction size of 3.68 bytes [Peuto77]. Lee and Smith used four IBM 370 workloads with a total of 15 traces that have an average basic block size of 18.98 bytes, assuming an average instruction length of 3.68 bytes [Lee84]. Our 7 VAX traces indicate an average basic block size of 4.2 bytes and basic block size of 14.26 bytes. The average instructions lengths bracket 4 bytes per instruction, so for simplicity, 4 bytes per instruction is assumed in the conversion calculations. The basic block sizes bracket our 32 trace average of 20.22 bytes so this basic block size is assumed.

Using data from [SmithAJ87] for a 4-way set associative instruction cache with 4-byte line size, LRU replacement, and cache sizes from 32 bytes to 32768 bytes, BTB miss ratio data was generated. Note that miss ratios for caches with line sizes greater than 4 bytes can be converted to an equivalent 4-byte line size miss ratio using the ratios of miss ratios data in [SmithAJ87]. Figure #7 contains the resulting miss ratios for BTB designs BTB#3 and BTB#4 (four way set associative, LRU replacement). Table #12 contains BTB design target miss ratio data for 2 to 1024 BTB entries.

# 8. Information Stored in the BTB

The information stored in a BTB strongly influences the performance of the design. Four types of information have been stored in BTBs: branch tags (uniquely identifying the branch), prediction information, branch target address, and target instruction bytes. In addition, in a BTB design described below, the instruction bytes immediately following the branch instruction are stored in the BTB, allowing branch folding to occur (eliminating branch execution time). In order to get an idea of the performance potential of the above types of information, we will consider their effect using the five stage pipeline mentioned in the introduction. Seven possible single-level BTB designs will be discussed using this example pipeline. The five pipeline stages are: instruction fetch, instruction decode, operand fetch, execution, and result writeback. All the pipeline stages normally take one cycle to execute.

For a processor with branch instructions that have a simple uniform target address encoding, the target address can be available after the instruction decode stage, if all instructions (branch and non-branch, since the instruction type is not determined until after decode) undergo

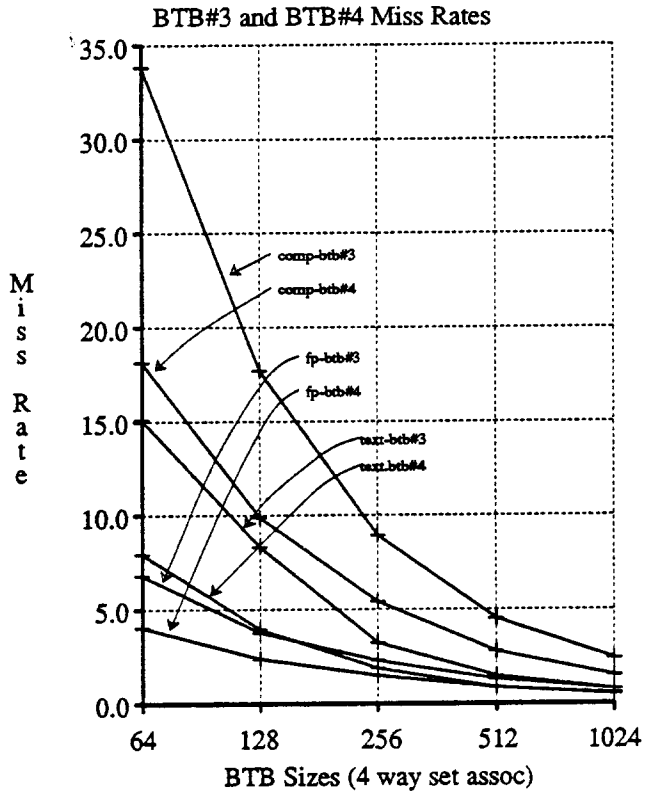


**Figure #5:** BTB#3 and BTB#4 miss rates for the workloads COMP, TEXT, and FP as a function of the number of BTB entries. BTB#4 has significantly lower miss rates than BTB#3.

target address calculations during the instruction decode stage. In the following designs, it is assumed that the target address is available after the decode stage. The conditions for a conditional branch are stable at the end of the operand fetch stage (i.e. at the end of the execution stage of the previous instruction), at which time instructions can be fetched from the target address if the branch is taken. With one cycle random access memory, this pipeline's branch penalty for a taken branch is two cycles.

With the pipeline and assumptions given above, seven BTB designs are discussed below.

i) **Hash Table of Prediction Information:** This simple BTB is accessed with the branch instruction address (modulo the table size) during the instruction decode (or fetch) stage to obtain prediction information. If the decoded instruction is a branch, the predicted branch path is fetched. In the example pipeline, this design saves a cycle for a correctly predicted taken branch (removes the delay between target address available and conditions stable). The prediction information must be updated after every branch execution. Note that the hash table (in which no branch tag exists) allows collisions to occur (multiple branches accessing the same entry) which reduces the prediction rate. The Mitsubishi GMICRO/100 uses a one prediction bit 256 entry BTB of this type [Sakamura87].

ii) **Branch Tag and Prediction Information:** A branch tag identifies a branch instruction with an entry in the BTB before the decoding stage of the processor determines the instruction is a branch. Using a branch tag can remove a cycle of delay (the decode stage time) in the example pipeline if early branch identification is useful. However, in our pipeline, since the target address is not available until after the instruction decode, this BTB design will not reduce the branch

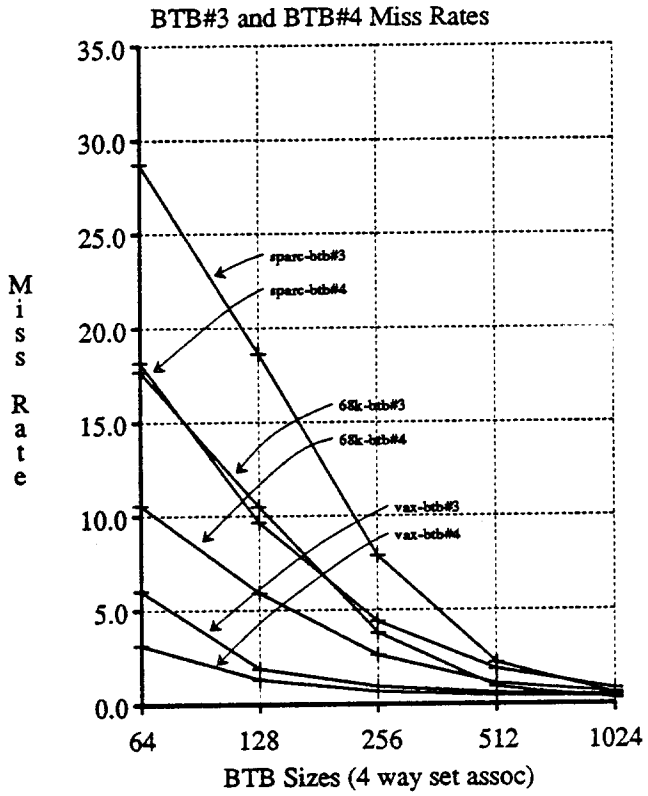


**Figure #6:** BTB#3 and BTB#4 miss rates for the workloads SPARC, VAX, and 68k. BTB#4 has significantly lower miss rates than BTB#3.

penalty more than BTB (i), but it will eliminate collisions (multiple branches accessing the same entry in the hash table) at the cost of supporting the branch tag.

**iii) Hash Table of Prediction Information and Target Address:** This BTB can remove the delay from the time of identifying the branch instruction to the time of conditions stable for a correctly predicted taken branch. In our pipeline, the branch is identified after the decoding stage when the target address is available, so this BTB has no performance benefit over BTB (i). However, if a delay exists between the time the instruction is identified as a branch and the time of target address available, this BTB will remove that additional delay, unlike BTB (i). For example, in our pipeline, if branches have a very simple encoding, they may be identified before going through the decoding stage, saving an additional cycle over BTB (i) for a correctly predicted taken branch. As with BTB (i), the hash table allows collisions to occur.

**iv) Branch Tag, Prediction Information, and Target Address:** In this BTB, the branch tag improves performance for our pipeline, unlike BTB (ii). This BTB removes the delay from the time of branch identification to the time of conditions stable. Since the branch tag identifies the branch one cycle earlier than the decoding stage, this BTB saves two cycles from the taken branch delay. The Edgecore E2000 uses a 1024 entry direct map BTB of this type, with one prediction bit, branch tag, and target address [Walter89].

v) **Target Instructions:** This BTB is often called a branch target cache. It is basically an instruction cache that caches instructions from the target addresses of branches. It has been used in the Am29000 and the GE RPM40 [AMD88] [Lewis88]. Hill compares it with instruction caches and instruction buffers [Hill87].

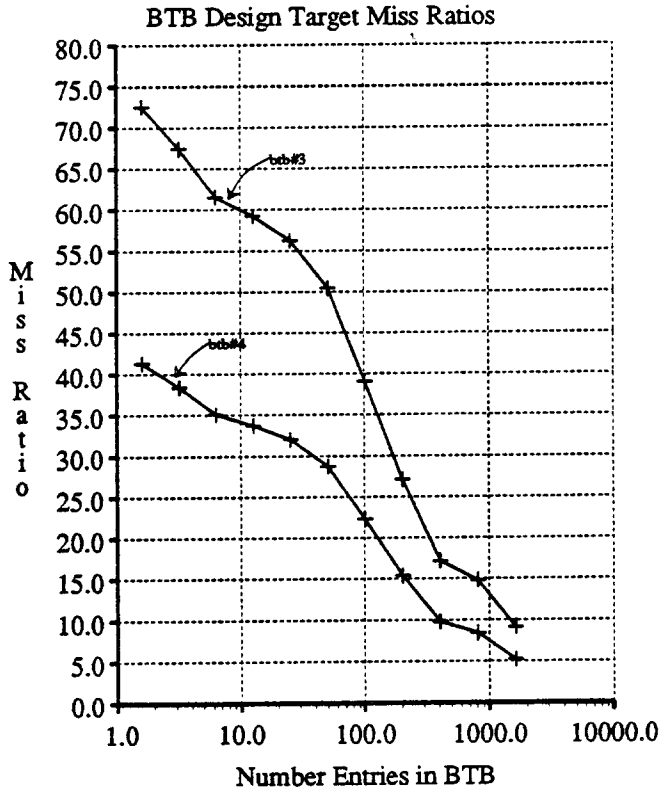


**Figure #7:** BTB design target miss ratios, converted from cache design target miss ratios in [SmithAJ87].

**vi) Branch Tag, Prediction Information, Target Address, and Target Instructions:** This BTB removes the delay from identifying a branch to receiving target instructions for a taken branch. In our pipeline, with one cycle random access memory, this BTB does not improve performance over BTB (iv). However, if the memory is static column memory, with a 2 cycle initial column access, and one cycle for same column accesses thereafter, the availability of the target instructions will reduce the taken branch delay by one additional cycle (total of three cycles saved). The target instructions can also reduce the delay in deep pipelines. This type of BTB is used in the NexGen processor [Stiles89a].

**vii) Branch Tag, Prediction Information, Target Address, Target Instructions, and Instructions after the Branch:** This BTB can implement branch folding, eliminating the execution time of branches [Ditzel87]. This BTB identifies a branch during instruction fetch, and instead of decoding the branch, decodes an instruction (taken from the BTB) from the predicted branch path. Extra hardware is required to decode and execute the branch in parallel, to verify the predicted branch path and target address. Simple branch encodings can reduce the decoding hardware required. This BTB allows branches to execute in zero time.

## 8.1. Tradeoffs Between Size and Number of Prediction Bits

Not only does the type of information contained in the BTB have an impact on cost and performance, the quantity of each type of information has an impact as well. In particular, Table #6 shows that significant tradeoffs can be made between size of the BTB (BTB#4) and number of prediction bits. For example, if a BTB with a prediction rate of 88 percent is required, both a one

| Table #12: Design Target Miss Ratios | | |
|---|---|---|
| Number BTB Entries | Miss Ratio BTB#3 | Miss Ratio BTB#4 |
| 2 | 70.8 | 40.4 |
| 4 | 65.4 | 37.3 |
| 8 | 60.7 | 34.6 |
| 16 | 58.2 | 33.2 |
| 32 | 54.2 | 30.9 |
| 64 | 46.6 | 26.6 |
| 128 | 35.0 | 20.0 |
| 256 | 23.8 | 13.6 |
| 512 | 16.4 | 9.3 |
| 1024 | 12.9 | 7.4 |

**Table #12:** BTB design target miss ratios, converted from cache design target miss ratios in [SmithAJ87].

prediction bit 2048 entry BTB and a two prediction bit 512 entry BTB will suffice. If each BTB entry has enough storage bits such that the number of prediction bits is insignificant, then the two prediction bit BTB has the same performance as a one prediction bit BTB which uses four times as many storage bits. If a BTB with a prediction rate of 89 percent is needed, a two prediction bit 1024 entry BTB can be used, or, for about half of the storage bits (assuming the number of prediction bits is small compared to the total number of bits per entry), a four prediction bit BTB with 512 entries will do. The potential cost savings encourage the designer to be careful in selecting a BTB configuration.

## 8.2. Multi-Level BTBs

The third concept presented in the **BTB Design Optimization** section is: *A multi-level BTB, each level possibly containing different amounts/types of information per entry, may be able to maximize performance by achieving a better balance of number of entries and quantity of information per entry.* Within the constraint of a finite number of storage bits allocated to a BTB design, a multi-level BTB may maximize performance.

To test this concept of multi-level BTBs, four BTB designs were simulated. The multi-level BTBs have up to three levels. The first level, the highest performance level, contains a branch tag, prediction bits, target address, and target instruction bytes for each entry. The second level, the medium performance level, contains a branch tag (one design eliminates this), prediction bits, and a target address for each entry. The third level, the low performance level, is a hash table of prediction bits. Higher performance levels require more storage bits than the lower performance levels.

Note that a simple prediction rate does not adequately represent the performance of a multi-level BTB. To obtain a measure of performance, implementation dependent cycle costs are required. Each of the three levels has different cycle costs for the four possible cases when a branch executes: predict not taken, branch not taken; predict not taken, branch taken; predict taken, branch not taken; predict taken, branch taken. Two sets of cycle costs, **costx** and **costy**, are used, and appear in Table #13. The predict not taken, branch not taken case has a zero cycle cost

| Table #13: Cycle Cost Sets Costx and Costy | | | | | | |
|---|---|---|---|---|---|---|
| | costx | | | costy | | |
| Four Branch Cases | level1 | level2 | level3 | level1 | level2 | level3 |
| Pred. not taken, branch not taken | 0 | 0 | 0 | 0 | 0 | 0 |
| Pred. not taken, branch taken | 4 | 4 | 4 | 4 | 4 | 4 |
| Pred. taken, branch not taken | 4 | 4 | 4 | 4 | 4 | 4 |
| Pred. taken, branch taken | 0 | 2 | 3 | 0 | 1 | 2 |

**Table #13:** The cycle costs for the two cost sets, **costx** and **costy**, as a function of BTB level and the four possible branch cases.

for all three levels and both cost sets. For simplicity, the two incorrect prediction cases (predict not taken, branch taken; predict taken, branch not taken) have a 4 cycle cost for both cost sets. The Amdahl 470 V/6 has a 4 cycle penalty when a branch is taken [Amdahl76], and the CLIPPER has a 3 to 5 cycle penalty [Hollingsworth89]. Note that in general, the two incorrect prediction cases need not have the same costs. Normally the sequential instruction stream is available, so an incorrect taken prediction costs less than a incorrect not taken prediction, in the second and third levels.

For the predict taken, branch taken case, the cycle costs are more interesting. In **costx**, a static column memory is assumed, with a three cycle initial column access, and one cycle access in that column thereafter. The target address is available one cycle after fetching the branch, and the conditions (for conditional branches) are stable one cycle after that. Level one entries have a zero cycle penalty by caching 8 bytes of instructions to make up for the extra two cycle delay in the initial three cycle target fetch. This assumes one instruction (4 bytes) executed every cycle. Level two entries have a two cycle penalty, due to the extra two cycle delay in the initial three cycle target fetch. Level three entries have a three cycle penalty, due to the one cycle to calculate the target address, and the two cycle delay in the initial target fetch.

For **costy**, the predict taken, branch taken case is similiar. A static column memory is again assumed, with a two cycle initial column access, and one cycle per access in that column thereafter. The target address is available one cycle after branch instruction fetch, and the conditions are stable two cycles after that. The level one entries have a zero cycle penalty by caching 4 bytes of target instructions to eliminate the extra cycle of the initial two cycle target access. The level two entries have a one cycle penalty, the extra cycle on the initial target access. The level three entries have a two cycle penalty, the cycle to calculate the target address plus the extra cycle of the initial target access.

Using the cost sets **costx** and **costy**, four multi-level BTB designs were simulated. All the designs use global LRU replacement where needed so small regular size changes can be made in the various levels. (In a real implementation, levels one and two would be set associative). Branches in lower levels are promoted to the highest level on a branch taken execution so that the target instructions can be fetched without increasing memory demands. Entries at higher levels are demoted one level at a time as they are replaced at the higher levels. The number of prediction bits used in all levels is fixed at two. Table #14 contains the approximate number of bits required by each entry of the levels in the designs. Justification for the bit allocations is provided in the following paragraphs.

BTB#6 contains level one and level two. For **costx**, level one requires approximately 32 bits of branch tag, prediction bits, 32 bits of target address, and 8 bytes of target instructions, for a

| Table #14: Bits Per Entry As Function of BTB Design, Cost Set, and Level | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | BTB#6 | | BTB#7 | | BTB#8 | | BTB#9 | |
| | costx | costy | costx | costy | costx | costy | costx | costy |
| level1 | 128 | 96 | 128 | 96 | 128 | 96 | 128 | 96 |
| level2 | 32 | 32 | 64 | 64 | 64 | 64 | | |
| level3 | | | | | 2 | 2 | 2 | 2 |

**Table #14:** Number of storage bits required per entry of the four multi-level BTBs, as a function of cost set and BTB level.

rough total of 128 bits per entry. For **costy**, level one requires branch tag, prediction bits, target address, and only 4 bytes of target instructions, for a rough total of 96 bits per entry. In BTB#6, level two is implemented as a hash table (no branch tag) containing prediction bits and target address for an approximate total of 32 bits for both **costx** and **costy**. This level two depends on a simple branch instruction encoding that allows the branch to be identified at the end of instruction fetch. Note that this hash table will experience collisions (multiple branches accessing the same entry) that will reduce performance.

BTB#7 is identical to BTB#6 except that level two contains a branch tag that increases the number of bits per entry to roughly 64.

BTB#8 is a full three level BTB. Levels one and two are identical to BTB#7. Level three is a 1024 entry hash table of two prediction bit entries. Note that collisions do occur in level three.

BTB#9 contains level one (same as BTB#6) and level three, each entry of which requires two bits.

### 8.2.1. Results of Simulating Multi-Level BTBs

Each of the four multi-level BTB designs was simulated with two cost sets (**costx** and **costy**), three values for total number of bits in the BTB (4096, 8192, 16384), and nine different bit allocations between the levels of the BTBs for each of the six possible cost sets and total number of bits combinations. The resulting cycle savings per branch data is given in full in six tables in [Perleberg89]. Six reduced tables presenting the average cycle savings per branch are presented here (Table #15 to Table #20). Tables #15 and #16 are for 4096 total bits and cost sets **costx** and **costy** respectively. Tables #17 and #18 are for 8192 total bits and cost sets **costx** and **costy** respectively. Tables #19 and #20 are for 16384 total bits and cost sets **costx** and **costy** respectively.

It is important to understand that the results presented are dependent on the cost sets, the bit allocations per entry, and the BTB designs. A multi-level BTB may or may not be practical, strongly depending on the specific implementation.

The cycle savings per branch results for BTB#6 and BTB#7 indicate that it is worth allocating all available bits to the high performance level one, since any other bit allocation between levels one and two increases the cycle savings per branch by at most 1.5 percent, and in most cases makes performance worse. This is true for all three total bit sizes, and over both cost sets, **costx** and **costy**.

BTB#8, the full three level BTB, achieves optimum results with a 1024 entry level three and all the remaining bits allocated to level one. Bits allocated to level two did not improve

| Table# 15: Average Cycle Savings Per Branch, 4096 bit BTB, costx | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BTB#6 Entries | | | BTB#7 Entries | | | BTB#8 Entries | | | BTB#9 Entries | | | Cycle Savings | | | |
| lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | BTB#6 | BTB#7 | BTB#8 | BTB# 9 |
| 0 | 128 | 0 | 0 | 64 | 0 | 0 | 32 | 1024 | 0 | 0 | 2048 | 0.903 | 0.902 | 0.903 | 0.522 |
| 4 | 112 | 0 | 4 | 56 | 0 | 2 | 28 | 1024 | 4 | 0 | 1792 | 1.298 | 1.309 | 1.195 | 1.115 |
| 8 | 96 | 0 | 8 | 48 | 0 | 4 | 24 | 1024 | 8 | 0 | 1536 | 1.411 | 1.424 | 1.300 | 1.294 |
| 12 | 80 | 0 | 12 | 40 | 0 | 6 | 20 | 1024 | 12 | 0 | 1280 | 1.515 | 1.523 | 1.369 | 1.435 |
| 16 | 64 | 0 | 16 | 32 | 0 | 8 | 16 | 1024 | 16 | 0 | 1024 | 1.561 | 1.574 | 1.412 | 1.556 |
| 20 | 48 | 0 | 20 | 24 | 0 | 10 | 12 | 1024 | 20 | 0 | 768 | 1.589 | 1.605 | 1.484 | 1.610 |
| 24 | 32 | 0 | 24 | 16 | 0 | 12 | 8 | 1024 | 24 | 0 | 512 | 1.601 | 1.628 | 1.509 | 1.668 |
| 28 | 16 | 0 | 28 | 8 | 0 | 14 | 4 | 1024 | 28 | 0 | 256 | 1.599 | 1.638 | 1.530 | 1.679 |
| 32 | 0 | 0 | 32 | 0 | 0 | 16 | 0 | 1024 | 32 | 0 | 0 | 1.665 | 1.665 | 1.556 | 1.665 |

| Table# 16: Average Cycle Savings Per Branch, 4096 bit BTB, costy | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BTB#6 Entries | | | BTB#7 Entries | | | BTB#8 Entries | | | BTB#9 Entries | | | Cycle Savings | | | |
| lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | BTB#6 | BTB#7 | BTB#8 | BTB#9 |
| 0 | 128 | 0 | 0 | 64 | 0 | 0 | 32 | 1024 | 0 | 0 | 2048 | 1.465 | 1.420 | 1.506 | 1.136 |
| 5 | 112 | 0 | 5 | 56 | 0 | 2 | 28 | 1024 | 5 | 0 | 1792 | 1.674 | 1.645 | 1.660 | 1.560 |
| 10 | 96 | 0 | 10 | 48 | 0 | 5 | 24 | 1024 | 10 | 0 | 1536 | 1.747 | 1.719 | 1.733 | 1.711 |
| 16 | 80 | 0 | 16 | 40 | 0 | 8 | 20 | 1024 | 16 | 0 | 1280 | 1.785 | 1.759 | 1.771 | 1.777 |
| 21 | 64 | 0 | 21 | 32 | 0 | 10 | 16 | 1024 | 21 | 0 | 1024 | 1.793 | 1.774 | 1.805 | 1.867 |
| 26 | 48 | 0 | 26 | 24 | 0 | 13 | 12 | 1024 | 26 | 0 | 768 | 1.794 | 1.779 | 1.829 | 1.884 |
| 32 | 32 | 0 | 32 | 16 | 0 | 16 | 8 | 1024 | 32 | 0 | 512 | 1.802 | 1.792 | 1.847 | 1.945 |
| 37 | 16 | 0 | 37 | 8 | 0 | 18 | 4 | 1024 | 37 | 0 | 256 | 1.796 | 1.781 | 1.855 | 1.937 |
| 42 | 0 | 0 | 42 | 0 | 0 | 21 | 0 | 1024 | 42 | 0 | 0 | 1.777 | 1.777 | 1.867 | 1.777 |

**Table #15:** Average cycle savings per branch for the four BTB designs (BTB#6-BTB#9) with a total of 4096 storage bits and cost set **costx**.

**Table #16:** Average cycle savings per branch for the four BTB designs (BTB#6-BTB#9) with a total of 4096 storage bits and cost set **costy**.

performance. BTB#8 outperforms BTB#6 and BTB#7 in four of the six cost set/number of bits combinations. In terms of cycle savings per branch, BTB#8 outperforms BTB#6 and BTB#7 by a maximum of 5 percent.

BTB#9 performs the best of all, in one configuration (**costy**, 4096 bits, 32 level one entries, 512 level three entries) outperforming designs with all bits allocated to level one by about 10 percent. For this best case, if we assume one cycle per non-branch instruction, basic block size of 5.26 instructions, and 70 percent branches taken, BTB#9 speeds up processor performance by 32 percent, and BTB#6 or BTB#7, with all bits allocated to level one (42 level one entries), speed up processor performance by 28 percent. So in this case, the multi-level BTB produces a net processor speedup of 4 percent over a single level BTB. In practice, the net speedup will be less than 4 percent since the average non-branch instruction will not execute in one cycle.

| Table# 17: Average Cycle Savings Per Branch, 8192 bit BTB, costx | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BTB#4 Entries | | | BTB#5 Entries | | | BTB#6 Entries | | | BTB#7 Entries | | | Cycle Savings | | | |
| lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | BTB#4 | BTB#5 | BTB#6 | BTB#7 |
| 0 | 256 | 0 | 0 | 128 | 0 | 0 | 96 | 1024 | 0 | 0 | 4096 | 0.969 | 0.968 | 0.992 | 0.529 |
| 8 | 224 | 0 | 8 | 112 | 0 | 6 | 84 | 1024 | 8 | 0 | 3584 | 1.498 | 1.504 | 1.483 | 1.304 |
| 16 | 192 | 0 | 16 | 96 | 0 | 12 | 72 | 1024 | 16 | 0 | 3072 | 1.680 | 1.680 | 1.643 | 1.567 |
| 24 | 160 | 0 | 24 | 80 | 0 | 18 | 60 | 1024 | 24 | 0 | 2560 | 1.758 | 1.765 | 1.737 | 1.685 |
| 32 | 128 | 0 | 32 | 64 | 0 | 24 | 48 | 1024 | 32 | 0 | 2048 | 1.826 | 1.837 | 1.797 | 1.811 |
| 40 | 96 | 0 | 40 | 48 | 0 | 30 | 36 | 1024 | 40 | 0 | 1536 | 1.859 | 1.878 | 1.845 | 1.869 |
| 48 | 64 | 0 | 48 | 32 | 0 | 36 | 24 | 1024 | 48 | 0 | 1024 | 1.891 | 1.911 | 1.890 | 1.932 |
| 56 | 32 | 0 | 56 | 16 | 0 | 42 | 12 | 1024 | 56 | 0 | 512 | 1.910 | 1.940 | 1.913 | 1.971 |
| 64 | 0 | 0 | 64 | 0 | 0 | 48 | 0 | 1024 | 64 | 0 | 0 | 1.950 | 1.950 | 1.932 | 1.950 |

| Table# 18: Average Cycle Savings Per Branch, 8192 bit BTB, costy | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BTB#4 Entries | | | BTB#5 Entries | | | BTB#6 Entries | | | BTB#7 Entries | | | Cycle Savings | | | |
| lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | BTB#4 | BTB#5 | BTB#6 | BTB#7 |
| 0 | 256 | 0 | 0 | 128 | 0 | 0 | 96 | 1024 | 0 | 0 | 4096 | 1.545 | 1.520 | 1.595 | 1.145 |
| 10 | 224 | 0 | 10 | 112 | 0 | 8 | 84 | 1024 | 10 | 0 | 3584 | 1.858 | 1.835 | 1.881 | 1.734 |
| 21 | 192 | 0 | 21 | 96 | 0 | 16 | 72 | 1024 | 21 | 0 | 3072 | 1.932 | 1.909 | 1.967 | 1.887 |
| 32 | 160 | 0 | 32 | 80 | 0 | 24 | 60 | 1024 | 32 | 0 | 2560 | 1.969 | 1.956 | 2.011 | 1.984 |
| 42 | 128 | 0 | 42 | 64 | 0 | 32 | 48 | 1024 | 42 | 0 | 2048 | 1.982 | 1.978 | 2.047 | 2.035 |
| 53 | 96 | 0 | 53 | 48 | 0 | 40 | 36 | 1024 | 53 | 0 | 1536 | 2.001 | 1.998 | 2.068 | 2.086 |
| 64 | 64 | 0 | 64 | 32 | 0 | 48 | 24 | 1024 | 64 | 0 | 1024 | 2.007 | 2.009 | 2.087 | 2.111 |
| 74 | 32 | 0 | 74 | 16 | 0 | 56 | 12 | 1024 | 74 | 0 | 512 | 1.995 | 2.009 | 2.103 | 2.109 |
| 85 | 0 | 0 | 85 | 0 | 0 | 64 | 0 | 1024 | 85 | 0 | 0 | 2.008 | 2.008 | 2.111 | 2.008 |

**Table #17:** Average cycle savings per branch for the four BTB designs (BTB#6-BTB#9) with a total of 8192 storage bits and cost set **costx**.

**Table #18:** Average cycle savings per branch for the four BTB designs (BTB#6-BTB#9) with a total of 8192 storage bits and cost set **costy**.

The limited speedup of a multi-level BTB compared to a single level BTB suggests that the additional complexity of the multi-level BTB is not cost effective. If the complexity of the multi-level BTB requires significant chip area, converting this area to storage bits for a simple single level BTB may produce equal or better performance. But multi-level BTBs are strongly dependent on the implementation. BTB#9 produced optimal performance for **costy** which gives lower BTB levels more performance value per entry than **costx**. RISC processors, with delayed branches, and very small (usually one cycle) branch penalties, will probably not find that multi-level BTBs, or any BTB for that matter, significantly improve performance. CISC processors, however, with deep pipelines and larger branch penalties, may find multi-level BTBs useful, and almost certainly will find that a single level BTB significantly improves performance.

| Table# 19: Average Cycle Savings Per Branch, 16384 bit BTB, costx | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BTB#6 Entries | | | BTB#7 Entries | | | BTB#8 Entries | | | BTB#9 Entries | | | Cycle Savings | | | |
| lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | BTB#6 | BTB#7 | BTB#8 | BTB#9 |
| 0 | 512 | 0 | 0 | 256 | 0 | 0 | 224 | 1024 | 0 | 0 | 8192 | 1.017 | 1.044 | 1.040 | 0.531 |
| 16 | 448 | 0 | 16 | 224 | 0 | 14 | 196 | 1024 | 16 | 0 | 7168 | 1.736 | 1.761 | 1.723 | 1.576 |
| 32 | 384 | 0 | 32 | 192 | 0 | 28 | 168 | 1024 | 32 | 0 | 6144 | 1.907 | 1.922 | 1.879 | 1.819 |
| 48 | 320 | 0 | 48 | 160 | 0 | 42 | 140 | 1024 | 48 | 0 | 5120 | 1.993 | 2.004 | 1.981 | 1.949 |
| 64 | 256 | 0 | 64 | 128 | 0 | 56 | 112 | 1024 | 64 | 0 | 4096 | 2.048 | 2.055 | 2.045 | 2.034 |
| 80 | 192 | 0 | 80 | 96 | 0 | 70 | 84 | 1024 | 80 | 0 | 3072 | 2.064 | 2.066 | 2.073 | 2.065 |
| 96 | 128 | 0 | 96 | 64 | 0 | 84 | 56 | 1024 | 96 | 0 | 2048 | 2.067 | 2.076 | 2.087 | 2.090 |
| 112 | 64 | 0 | 112 | 32 | 0 | 98 | 28 | 1024 | 112 | 0 | 1024 | 2.065 | 2.083 | 2.096 | 2.104 |
| 128 | 0 | 0 | 128 | 0 | 0 | 112 | 0 | 1024 | 128 | 0 | 0 | 2.085 | 2.085 | 2.104 | 2.085 |

| Table# 20: Average Cycle Savings Per Branch, 16384 bit BTB, costy | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BTB#6 Entries | | | BTB#7 Entries | | | BTB#8 Entries | | | BTB#9 Entries | | | Cycle Savings | | | |
| lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | lev1 | lev2 | lev3 | BTB#6 | BTB#7 | BTB#8 | BTB#9 |
| 0 | 512 | 0 | 0 | 256 | 0 | 0 | 224 | 1024 | 0 | 0 | 8192 | 1.604 | 1.633 | 1.644 | 1.148 |
| 21 | 448 | 0 | 21 | 224 | 0 | 18 | 196 | 1024 | 21 | 0 | 7168 | 2.002 | 2.033 | 2.031 | 1.892 |
| 42 | 384 | 0 | 42 | 192 | 0 | 37 | 168 | 1024 | 42 | 0 | 6144 | 2.084 | 2.102 | 2.113 | 2.045 |
| 64 | 320 | 0 | 64 | 160 | 0 | 56 | 140 | 1024 | 64 | 0 | 5120 | 2.124 | 2.138 | 2.159 | 2.130 |
| 85 | 256 | 0 | 85 | 128 | 0 | 74 | 112 | 1024 | 85 | 0 | 4096 | 2.131 | 2.145 | 2.175 | 2.157 |
| 106 | 192 | 0 | 106 | 96 | 0 | 93 | 84 | 1024 | 106 | 0 | 3072 | 2.133 | 2.147 | 2.182 | 2.175 |
| 128 | 128 | 0 | 128 | 64 | 0 | 112 | 56 | 1024 | 128 | 0 | 2048 | 2.127 | 2.147 | 2.188 | 2.190 |
| 149 | 64 | 0 | 149 | 32 | 0 | 130 | 28 | 1024 | 149 | 0 | 1024 | 2.122 | 2.141 | 2.192 | 2.197 |
| 170 | 0 | 0 | 170 | 0 | 0 | 149 | 0 | 1024 | 170 | 0 | 0 | 2.138 | 2.138 | 2.197 | 2.138 |

**Table #19:** Average cycle savings per branch for the four BTB designs (BTB#6-BTB#9) with a total of 16384 storage bits and cost set **costx**.

**Table #20:** Average cycle savings per branch for the four BTB designs (BTB#6-BTB#9) with a total of 16384 storage bits and cost set **costy**.

## 9. Conclusions

A Branch Target Buffer can reduce the performance penalty of branches in pipelined processors by predicting the path of the branch, and caching information that the branch needs to execute quickly. Using a set of program traces and software BTB simulators, BTB performance data was generated. Prediction rates for infinite size BTBs, using a dynamic prediction method, increased significantly up to four prediction bits, after which the prediction rate increased slowly up to our maximum measured size of 16 prediction bits. Context switching was found to have a significant negative effect on the prediction rates. With a context switch occurring every 10000 instructions, two prediction bits have a prediction rate of 84 percent, as compared to 89 percent without context switching.

BTB management, when to enter and discard branches from the BTB, was explored. We found that entering branches into the BTB only on a branch taken execution yields better results than entering branches on both taken and not taken executions. Entering branches on taken

executions avoids entering into the BTB branches that can't improve performance at the expense of discarding branches that can improve performance. A new method of discarding BTB entries was discussed and compared to the simple LRU replacement strategy. This new method discards branch information that is both not likely to be referenced (i.e. low in the LRU stack), and not likely to be useful (i.e. predicts not taken). This method performed slightly better than simple LRU.

We found a relationship between miss ratios for BTBs and instruction caches. This allowed us to convert miss rates for caches to BTB miss rates and vice versa. Using this capability, design target miss ratios for instruction caches were taken from the literature and converted to design target miss ratios for BTBs.

Finally, the nature of BTB entries was discussed. The type and amount of information stored in a BTB has a significant impact on the performance of the BTB. Several BTB designs were presented and evaluated, including entries with one or more of the following information types: branch tag, prediction bits, branch target address, and branch target instructions. Varying the amount of individual types of information has a significant impact, as shown by examining the tradeoffs possible between number of prediction bits and number of entries in the BTB. Multi-level BTBs were considered, i.e. BTBs that vary the information content in each level in the BTB. The multi-level BTBs simulated, with specific cycle cost and implementation assumptions, did outperform single level BTBs, but by a small margin. However, depending on the implementation, multi-level BTBs may significantly improve performance.

## 10. Bibliography


[AMD88] Advanced Micro Devices, "Am29000 Streamlined Instruction Processor User Manual," Advanced Micro Devices, 1988

[Amdahl76] Amdahl, "Amdahl 470 V/6 Machine Reference Manual," Amdahl, Sunnyvale, Calif., 1976

[Anderson67] D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo, "The IBM System/360 Model 91: Machine Philosphy and Instruction-Handling," IBM Journal of Research and Development, Jan. 1967, pp. 8-24

[DeRosa87] J.A. DeRosa and H.M. Levy, "An Evaluation of Branch Architectures," Proc. 14th Ann. Symp. Computer Architecture, 1987, pp. 10-16

[DEC77] Digital Equipment Corporation, "VAX 11/780 Architecture Handbook," Digital Equipment Corportion, 1977

[Ditzel87] D.R. Ditzel and H.R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," Proc. 14th Ann. Symp. Computer Architecture, 1987, pp. 2-9.

[Doduc89] N. Doduc, "Fortran Execution Time Benchmark," unpublished report, V.20, March 1989

[Elrod70] T.H. Elrod, "The CDC 7600 and SCOPE 76," Datamation, Apr. 1970, pp. 80-85

[Folger83] D. Folger and E. Basart, "Computer Architecture - Designing for Speed," Spring COMPCON 1983, pp. 25-31

[Fujitsu87] Fujitsu Microelectronics, Inc., "MB86900 RISC Processor Architecture Manual," Fujitsu Microelectronics, Inc., 1987

[Gaulding75] S.N. Gaulding and D.P. Madison, Jr., "Optimization of Scalar Instructions for the Advanced Scientific Computer," Spring COMPCON 1975, pp. 189-193

[Grochowski86] E.T. Grochowski, "An Instruction Tracer for the Motorola 68010," UC Berkeley Masters Report, 1986

[Gross82] T.R. Gross and J.L. Hennessy, "Optimizing Delayed Branches," Proc. 15th Ann. Workshop on Microprogramming, Oct. 1982, pp. 114-120

[Henry83] R.R. Henry, "VAX Address and Instruction Traces," unpublished report, 1983

[Hill87] M.D. Hill, "Aspects of Cache Memory and Instruction Buffer Performance," UC Berkeley Technical Report UCB/CSD 87/381, November 1987

[Hill89] M. Hill and A.J. Smith, "Evaluating Associativity in CPU Caches," Univ. of Wisconsin Computer Sciences Technical Report #823, Feb. 1989, to appear in IEEE TOC, Dec. 1989

[Hinton89] G. Hinton, "80960 - Next Generation," Spring COMPCON 1989, pp. 13-17

[Hollingsworth89] W. Hollingsworth, H. Sachs, and A.J. Smith, "The CLIPPER Processor: Instruction Set Architecture and Implementation," Communications of the ACM, Feb. 1989, pp. 200-219

[Hwang84] K. Hwang and F.A. Briggs, "Computer Architecture and Parallel Processing," McGraw-Hill, 1984, pp. 714-729

[IBM73] IBM, "IBM Maintenance Library System/370 Model 168 Theory of Operation/Diagrams Manual," Vol. 2, 1973, IBM, Poughkeepsie, N.Y.

[IBM78] IBM, "IBM Maintenance Library 3033 Processor Complex Theory of Operation/Diagrams Manual," Vols. 1-3, Jan. 1978, IBM, Poughkeepsie, N.Y.

[Jordan83] H.F. Jordan, "Performance Measurements on HEP - A Pipelined MIMD Computer," Proc. 10th Ann. Symp. Computer Architecture, 1983, pp. 207-212

[Kane89] G. Kane, "MIPS RISC Architecture," Prentice Hall, 1989

[Kartashev90] S.P. Kartashev and S.I Kartashev, "Supercomputing Systems," Van Nostrand Reinhold, 1990, pp. 106-153

[Kogge81] P.M. Kogge, "The Architecture of Pipelined Computers," McGraw-Hill, 1981

[Lee84] J.K.F. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," Computer, Jan. 1984, pp. 6-22.

[Lewis88] D.K. Lewis, J.P. Costello, and D.M. O'Connor, "Design Tradeoffs for a 40 MIPS (Peak) CMOS 32-bit Microprocessor," Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers & Processors, Oct. 1988, pp. 110-113

[Lilja88] D.J. Lilja, "Reducing the Branch Penalty in Pipelined Processors," Computer, July 1988, pp. 47-55.

[Manuel87] T. Manuel, "Getting Mainframe Power out of a CISC Supermicro," Electronics, Sept. 3, 1987, pp. 66-69

[McFarling86] S. McFarling and J. Hennessy, "Reducing the Cost of Branches," Proc. 13th Ann. Symp. Computer Architecture, 1986, pp. 396-403.

[Motorola89] Motorola, "M68000 8-/16-/32-Bit Microprocessors User's Manual," Prentice Hall, 1989

[Murphey70] J.O. Murphey and R.M. Wade, "The IBM 360/195," Datamation, Apr. 1970, pp. 72-79

[Patterson81] D.A. Patterson and H. Sequin, "RISC-I: A Reduced Instruction Set VLSI Computer," Proc. Eighth Symp. Computer Architecture, May 1981, pp. 443-458

[Perleberg89] C.H. Perleberg, "Branch Target Buffer Design," UC Berkeley Computer Science Division Technical Report No. UCB/CSD 89/553, December 1989

[Peuto77] B.L. Peuto and L.J. Shustek, "An Instruction Timing Model of CPU Performance," Proc. 4th Ann. Symp. Computer Architecture, Mar. 1977, pp.165-178

[Ramamoorthy77] C.V. Ramamoorthy and H.F. Li, "Pipeline Architectures," Computing Surveys, March 1977, pp. 61-102.

[Rau77] B.R. Rau and G.E. Rossman, "The Effect of Instruction Fetch Strategies upon the Performance of Pipelined Instruction Units," 4th Ann. Symp. Computer Architecture, 1977, pp. 80-87

[Russell78] R.M. Russell, "The CRAY-1 Computer System," Comm. of the ACM, Jan. 1978, pp. 63-72

[Sakamura87] K. Sakamura, "TRON Project 1987," Springer-Verlag, 1987

[Shar74] L.E. Shar and E.S. Davidson, "A Multiminiprocessor System Implemented Through Pipelining," IEEE Computer, Feb. 1974, pp. 42-51

[SmithJ81] J.E. Smith, "A Study of Branch Prediction Strategies," Proc. Eighth Symp. Computer Architecture, May 1981, pp. 135-148

[SmithJ83] J.E. Smith and J.R. Goodman, "A Study of Instruction Cache Organizations and Replacement Policies," Proc. 10th Symp. Computer Architecture, June 1983, pp. 132-137

[SmithAJ85] A.J. Smith, "Cache Evaluation and the Impact of Workload Choice," Proc. 12th Symp. Computer Architecture, June 1985, pp. 64-74.

[SmithAJ87] A.J. Smith, "Line (Block) Size Choice for CPU Cache Memories," IEEE Transactions on Computing, Sept. 1987, pp. 1063-1075.

[Stiles89a] D.R. Stiles and H.L. McFarland, "Pipeline Control for a Single Cycle VLSI Implementation of a Complex Instruction Set Computer," Spring COMPCON 1989, pp. 504-508

[Stiles89b] D.R. Stiles (of NexGen Microsystems), personal interview concerning branch prediction cache of NexGen processor, Sept. 25, 1989

[Thornton64] J.E. Thornton, "Parallel Operation in the Control Data 6600," AFIPS Fall Joint Comp. Conf., 1964, pp. 33-40

[UCB87] U.C. Berkeley CAD/IC Group, "SPICE2G.6," March 1987

[Walter89] S. Walter (of Edgecore), personal interview concerning branch cache in Edge 2000, Sept 20, 1989

[Widdoes77] L.C. Widdoes, Jr., "Jump Prediction," Stanford University (unpublished draft), February 1977

[Yoshida87] T. Yoshida and T. Enomoto, "The Mitsubishi VLSI CPU in the TRON Project," IEEE Micro, Apr. 1987, p. 24