A System Level Perspective on Branch Architecture Performance

Brad Calder Dirk Grunwald

Department of Computer Science
University of Colorado, Campus Box 430
Boulder, CO 80302-0430
calder,grunwald@cs.colorado.edu

Joel Emer

Digital Semiconductor
77 Reed Road (HLO2-3/J3)
Hudson, MA 01749
emer@vssad.hlo.dec.com

Abstract

Accurate instruction fetch and branch prediction is increasingly important on today's wide-issue architectures. Fetch prediction is the process of determining the next instruction to request from the memory subsystem. Branch prediction is the process of predicting the likely out-come of branch instructions. Many branch and fetch prediction architectures have been proposed, from simple static techniques to more sophisticated hardware designs. All these previous studies compare differing branch prediction architectures in terms of misprediction rates, branch penalties, or an idealized cycles per instruction.

This paper provides a system-level performance comparison of several branch architectures using a full pipeline-level architectural simulator. The performance of various branch architectures is reported using execution time and cycles-per-instruction. For the programs we measured, our simulations show that having no branch prediction increases the execution time by 27%. By comparison, a highly accurate 512 entry branch target buffer architecture has an increased execution time of 1.5% when compared to an architecture with perfect branch prediction. We also show that the most commonly used branch performance metrics, branch misprediction rates and the branch execution penalty, are highly correlated with program performance and are suitable metrics for architectural studies.

1 Introduction

When a branch is encountered in the instruction stream, it can cause a change in control flow. This change in control flow can cause branch penalties that can significantly degrade the performance of superscalar pipelined processors. To overcome these penalties, these processors typically provide some form of control flow prediction.

Instruction fetch prediction is used each cycle to predict which instructions to fetch from the instruction cache. In processors, such as the Alpha 21064, the next sequential instruction is fetched and speculatively executed from the instruction cache. If a branch instruction causes a change in control flow, the instruction fetched following the branch can not be used, and a new instruction must be fetched after the target address is calculated, introducing a pipeline bubble or unused pipeline step. This is called an instruction *misfetch penalty*, and is caused by waiting to identify the instruction as a branch and to calculate the target address.

Branch Prediction is the process of predicting the branching direction for conditional branches, whether they are taken or not, and predicting the target address for return instructions and indirect branches. The final destinations for conditional branches, indirect jumps and returns are typically not available until a later stage of the pipeline. To improve performance, the processor may elect to fetch and decode instructions on the assumption that the eventual branch target can be accurately predicted. If the processor mispredicts the branch destination or direction, instructions fetched from the incorrect instruction stream must be discarded, leading to several pipeline bubbles. This is called a branch *mispredict penalty*.

Instruction fetch prediction and branch prediction are both used to reduce pipeline penalties caused by changes in control flow. The Alpha 21064 processor uses these two prediction mechanisms in two steps; first performing instruction fetch prediction by predicting the next sequential instruction from the instruction cache, and second performing branch prediction in the decode stage. The destination for conditional branches are predicted and checked in the decode stage. If the fall-through is not predicted then the calculated target address must be fetched. If the decoded instruction is a return, the call prediction stack is used; if the instruction is an indirect jump, the "hint" field provided for indirect jumps is used. The call prediction stack is a four-entry circular stack used to mimic the call and return behavior of programs. The "hint" field is a 12-bit branch displacement used to indicate the likely destination of a given indirect branch. Some processors, such as the Intel Pentium and P6, combine both instruction fetch prediction and branch prediction into one hardware mechanism, allowing both the instruction fetch and branch prediction to be performed at the same time.

Usually, branch architectures are studied in isolation, ignoring other aspects of the system. In those studies, a variety of metrics are used to assess the performance of different branch architectures, but it is rare that the performance of an entire system (i.e., CPU and memory hierarchy) is reported. There are numerous reasons for this; the complexity of accurately simulating a complex architecture is daunting, the results are then specific to that architecture

Program	Measured Cycles	Zippy Cycles	% Diff	PAL Calls	%PAL Incr
doduc	3852.7	3705.0	-3.8	43	0.0
li	3840.2	3622.7	-5.6	32949	0.0
tomcatv	3428.5	3288.0	-4.1	128	0.0
espresso	798.9	822.1	2.9	86	0.2
gzip	559.2	621.7	11.1	67	0.7
od	545.3	589.1	8.0	193	2.9
porky	472.8	446.6	-5.5	241	0.7
gcc	406.9	403.3	-0.8	180	1.1
compress	305.0	318.8	4.5	208	3.4
bc	227.7	219.7	-3.5	535	0.6
indent	80.8	75.1	-6.9	133	3.1
cfront	68.7	65.8	-4.2	81	1.5
flex	34.2	30.7	-10.0	35	1.0
lic	17.9	15.4	-13.7	337	4.9

Table 1: Verification of Zippy architecture model, showing the number of actual executed cycles measured on an Alpha 3000-500 minus the PALcall cycles (column "Measured", in millions), the number of cycles estimated by Zippy while simulating the Alpha 3000-500 ("Zippy Cycles", in millions), the percent difference in the Zippy model and measured cycle counts (% Diff), the number of dynamic PAL calls executed in the measured programs (PAL Calls), and the percent increase in number of cycles executed if PAL calls were taken into account.

and it is difficult to translate the performance of that architecture to another, possibly different architecture.

However, there are advantages to system-level simulations. They provide the reader with a sense of the magnitude and importance of the problem being solved, and indicate where further effort should be placed. Furthermore, the performance metrics, such as execution time and cycles per instruction, are more understandable to the casual reader. Lastly, such studies provide a method to calibrate the performance metrics used in other studies with the impact on system level performance.

This paper provides a system level study of several branch prediction architectures, including recently proposed two-level correlated branch history architectures. We performed a system-level simulation of the Alpha AXP 21064, a dual-issue statically-scheduled processor. We first describe our experimental methodology in §2, and describe the architectures simulated in §3. The performance results from the system level study are discussed in §4. Section 5 measures the accuracy of common branch performance metrics. We then conclude in §6 and §7 with cautions to the reader to understand the limitations of this study.

2 Experimental Methodology

We will study several branch prediction architectures using information from direct-execution simulation. We used a pipeline-level simulator to measure program execution time and attribute portions of that time to different processor subsystems. We first describe the simulator used and compare measurements from that simulator to measurements from an actual system. We then describe the programs we measured and how we will compare branch architecture performance.

2.1 Simulation Methodology

We used Zippy, a cycle-level simulator for the DEC 21064 architecture that simulates all resources on the 21064, including the processor bus, first and second level caches, main memory, functional units and all hardware interlocks. Zippy is a direct-execution simulator; the program being measured is executed, and events generated by the program execution are fed to an event-driven simulator. The version of Zippy we used in this study used the ATOM instrumentation system [15] to instrument the applications. A considerable amount of analysis can be performed during the instrumentation phase. For example, interlocks and dependence between instructions can be computed statically. Some aspects of the simulation can be simplified as well; for example, we can statically determine if two instructions may issue concurrently, and perform a single instruction cache probe for the pair of instructions.

Zippy can be configured to use one of a number of processor and system configurations. The system we simulated is a DEC Alpha 3000-500 workstation, with a DECchip AXP 21064 processor clocked at 150Mhz. The system contains an 8KB on-chip direct-mapped instruction cache and an 8KB on-chip direct-mapped store-around data cache. The data cache is pipelined and can be accessed every cycle, although data is not available until 2 cycles after it has been accessed. There is a 512KByte second-level, unified direct-mapped cache with five cycle latency and access time. Main memory has a 24 cycle latency and 30 cycle access time. More information about the AXP 21064 can be found in the processor reference manual [5].

Table 1 compares the cycle count reported by Zippy to the actual cycle count taken for the same application, recorded by the processor cycle counter on an Alpha 3000-500 processor.[1] We ran each application 1000 times, and the measured values had a 95% confidence interval typically less than 0.06% of the mean.

The measurements in Table 1 show that the average difference between the cycle time reported by the Zippy model and the actual execution time is 6% for the programs we measured. There are a number of reasons why the estimated performance of Zippy would differ from the measured performance. First, a different virtual memory page layout algorithm was used in Zippy than the algorithm used by OSF on the Alpha 21064. For the estimated Zippy cycle times shown in Table 1, the layout of the pages in memory were random rather than the actual bin hopping algorithm used in OSF/1. The layout of pages in memory will cause cache conflicts in the physically indexed 2nd level cache and those penalties can be significant [1, 6]. For instance, the percent of cycles increased by 27% for `tomcatv` and decreased 10% for `gzip` when the virtual and physical mappings were identical.

Another difference is that Zippy simulates the processor executing a single program; it does not simulate the

[1] We could not accurately gather the cycle counts for `eqntott` and `wdiff` because these programs spawned threads, complicating the performance measurement. The Alpha 3000-500 we measured did not have the appropriate software to run `idl`.

Program	# Instructions in Millions	% Branches	Exec Time (s)	CPI	% of Cycles				
					Issues	Stalls	ICache	DCache	TLB
doduc	1149.9	9	24.249	3.195	25.4	47.7	9.7	17.0	0.2
tomcatv	899.7	3	27.654	4.657	16.3	12.6	0.1	70.9	0.2
compress	92.6	14	1.897	3.103	27.0	8.5	0.0	22.3	42.2
eqntott	1810.5	12	14.969	1.253	74.1	11.0	2.5	11.4	1.1
espresso	513.0	17	4.666	1.378	62.3	19.4	3.2	13.9	1.2
gcc	143.7	16	2.314	2.439	35.0	13.6	20.8	25.1	5.5
li	1355.1	18	20.718	2.317	36.7	17.4	15.5	30.3	0.0
bc	93.4	16	1.338	2.171	38.7	17.0	22.9	19.9	1.4
flex	15.5	15	0.185	1.811	47.9	19.1	11.9	20.6	0.4
gzip	309.5	13	3.363	1.646	54.5	19.2	1.6	24.6	0.0
indent	32.6	17	0.463	2.154	40.6	22.7	18.4	17.7	0.7
od	210.3	18	3.648	2.628	31.7	11.3	23.7	31.5	1.9
wdiff	76.2	17	0.984	1.956	43.5	41.5	0.1	14.9	0.0
cfront	16.5	13	0.392	3.594	23.7	10.3	23.2	29.5	13.3
idl	21.1	20	0.287	2.058	41.1	14.6	10.4	26.2	7.7
lic	5.9	17	0.093	2.361	36.2	19.6	20.0	19.9	4.3
porky	163.7	20	2.592	2.399	35.5	16.6	12.1	27.9	7.9
Avg	406.4	15	6.460	2.419	39.4	18.9	11.5	24.9	5.2

Table 2: Application performance from Zippy, assuming perfect branch prediction.

underlying operating system and other programs running on the system. Normally, when a process requests a system service, system code displaces parts of the instruction and data cache. These problems should be most visible in short-running programs or those that make a number of system calls. Table 1 confirms this; short-running programs such as `flex`, `indent` and `lic` have a high variance. On the Alpha, a "PALcall" is used to access system routines and for system-specific functions; the column showing the number of PALcalls shows that `li` executes many system calls, and also has a large variance. Other programs with a large difference in the execution time initiate I/O, which causes cache flushes and more system level disturbances. Even with these aspects of the system not being modeled by Zippy, the measurements in this experiment show that Zippy provides a reasonably accurate performance estimate for the 21064 architecture.

2.2 Programs Measured

We instrumented the programs from the SPECint92 benchmark suite and object-oriented programs written in C++. Other studies have noted that FORTRAN programs have very predictable branches. We simulated the SPECfp92 benchmarks and found that was true. Therefore we provide results for only two of the programs, `doduc` and `tomcatv`, and concentrate on C and C++ programs that are harder to predict. The programs were compiled on a DEC Alpha 3000-400 running OSF/1 V2.0, using either the DEC C, C++ or FORTRAN compiler. All programs were compiled with standard optimization (`-O`). For the SPECint92 programs, we used the largest input distributed with the SPECint92 suite. The first two columns in Table 2 show the number of instructions simulated and the percentage of those instructions that are branches for the programs we instrumented. The next two columns give the average execution time in seconds and the cycles-per-instruction (CPI) for each program, assuming perfect branch prediction. The remaining columns give the break down of cycles executed, and attributes those cycles to the instruction issues (Issues), stall cycles due to the inability to issue instructions (Stalls), instruction cache misses (ICache), data cache misses (DCache), and translation look-aside buffer misses (TLB).

The "Stalls" entry includes interlocks and data dependence constraints that can be computed statically. For example, the result of a load is not available until two cycles after it is issued. An instruction using the result of that load immediately following the load would have to stall at least that long. The cycles induced by dependence constraints are attributed to stalls, but any additional cycles induced by cache misses are attributed to that cache. We combined all cache misses from the first and second level cache into the ICache or DCache value.

2.3 Metrics

We used several metrics to compare branch architecture performance in this paper. The first two are the *increase in execution time* and *cycles per instruction*. We also compute a metric used in branch architecture papers, called the *branch execution penalty* (BEP). The BEP can be used to select one branch architecture over another.

Percent Increase in Execution Time (%IET) Throughout the paper, we will compare the execution time of each program to a system with an unobtainable "perfect" branch prediction architecture. Perfect branch prediction assumes that branches are never misfetched or mispredicted. Program execution using other branch architectures are specified as a percent increase in execution time, or slowdown, relative to the perfect branch execution architecture. The slowdown includes the affect of all system components.

Cycles Per Instruction (CPI) There is a considerable variation in the number of instructions issued by an individual program, as shown in Table 2. Cycles per instruction provides a more application-independent performance metric for a given architecture than the %IET. We also use the

CPI to determine if branch performance metrics predict program performance.

Branch Execution Penalty (BEP) Researchers have used the percent of mispredicted branches to compare the performance of different branch architectures. However, this metric is too simplistic for branch target buffer organizations because the miss rate does not take into account misfetched branches, and may exaggerate the performance differences in branch architectures. There are two forms of pipeline penalties we are concerned with: misfetching and misprediction. Each branch type can be misfetched, but only conditional branches, indirect jumps and returns can be mispredicted. The penalty for misfetching is less than the penalty for misprediction, and a processor design may tradeoff off one rate for another. We record the percentage of misfetched branches (%MfB) *and* the percentage of mispredicted branches (%MpB). It is often difficult to understand how these metrics interact and how they influence processor performance. Yeh & Patt [17] defined a formula to combine these branch penalties called the *branch execution penalty*:

$$\text{BEP} = \frac{\%\text{MfB} \times \text{misfetch penalty} + \%\text{MpB} \times \text{mispred penalty}}{100}$$

which reflects the average penalty suffered by a branch due to misfetch and misprediction. A BEP of 0.5 means that, on average, each branch takes an extra half cycle to execute; values close to zero are desirable. We have assumed a one cycle misfetch penalty and a five cycle mispredict penalty since these values are appropriate for the Alpha AXP 21064 architecture. Later we compare the performance predictability of the percent of mispredicted branches and the BEP by correlating these metrics to the programs CPI.

3 Branch Architecture Models

This section describes the branch architectures we simulated in this study. Table 3 gives a summary of these architectures. We simulated the DECchip 21064 implementation of the Alpha AXP architecture using perfect branch prediction (Perfect), no branch prediction (No-Pred), backwards taken/forwards not taken (BTFNT), the default 1-Bit branch prediction provided on the Alpha 21064 (21064) architecture, pattern history tables (PHT), branch target buffers (BTB), and two types of correlated conditional branch prediction architectures (PAs and GAg).

For Perfect branch prediction we assume that all branch targets are fetched correctly without any penalties, so there are no misfetch or mispredict penalties. This establishes an upper-bound on the influence that the branch architecture can have on the system architecture. For no branch prediction (No-Pred), we assume that the instruction fetch pipeline halts whenever a branch is encountered and the pipeline stalls until the correct branch destination address is known. This is the worst possible branch architecture, and establishes a lower bound on branch performance for the 21064 architecture. For No-Pred, all branches either cause a misfetch penalty or a mispredict penalty depending on the type of the branch. The only static branch prediction architecture we simulated was the simple backward-taken, forward not-taken (BTFNT) architecture. The BTFNT architecture predicts backwards branches are looping branches and will be taken, and forward branches are predicted as not taken.

Most processors now use some form of dynamic conditional branch prediction to improve the branch architecture performance. The Alpha 21064 used in the 3000-500 we simulated adds one bit to each instruction in the 8K direct mapped instruction cache to dynamically predict the direction of conditional branches. This bit is initialized with the sign-bit of the PC-relative displacement when the cache line is read in, initializing the dynamic 1-bit predictors to BTFNT prediction. Thus, the BTFNT rule is used the first time the branch is encountered, and a dynamic one-bit predictor is used thereafter.

Two-bit up-down saturating counters have been shown to effectively predict the direction of conditional branches and to outperform 1-bit branch predictors [10, 13]. The pattern history table (PHT) branch architecture is an example of an architecture using two-bit saturating up-down counters. It contains a table of two-bit counters used to predict the direction for conditional branches. In the direct mapped PHT architecture (PHT-Direct), the branch address is used to directly index into the pattern history table to find the two-bit counter to use when predicting the branch.

Pan *et al.* [11] and Yeh and Patt [16, 18] investigated *branch-correlation* or *two-level* branch prediction mechanisms. Although there are a number of variants, these mechanisms generally combine the history of several recent branches to predict the outcome of a branch. The *degenerate method* (GAg) of Pan *et al.* [11] uses a global history register to record the branch correlation. When using a 2^k entry table, the processor maintains a k-bit global history register that records the outcome of the previous k branches (e.g., a taken branch is encoded as a 1, and a not-taken branch as a 0). The register is used as an index into the pattern history table (PHT), much as the program counter is used for a direct-mapped PHT. This provides contextual information and correlation about particular patterns of branches. For the GAg method that we model, we use a variant described by McFarling [8]. His method used the exclusive-or of the global history register and the branch address as the index into the PHT (PHT-GAg).

Branch target buffers (BTB) have been used as a mechanism for branch and instruction fetch prediction, effectively predicting the behavior of a branch [7, 9, 12, 17]. The Intel Pentium is an example of a modern architecture using a BTB – it has a 256-entry BTB organized as a four-way associative cache. Only branches that are taken are entered into the BTB. If a branch address appears in the BTB and the branch is predicted as taken, the stored address is used to fetch future instructions, otherwise the fall-through address is used. The "BTB-2Bit" architecture models the architecture used in the Pentium, where each BTB entry contains a two-bit saturating counter to predict the direction of a conditional branch [7]. The Intel P6 architecture adds to this design by increasing the BTB to a 512 entry 4-way associative design and replaces the 2-bit counters with 4-bit history registers.

The "BTB-PAs" architecture models the architecture proposed by Yeh *et al* [17], where each BTB entry contains a 6-bit history register. This history register is used as the lower bits of an index into a PHT when predicting conditional branches. The upper bits of the index are the lower bits of the branch address. In both BTB-2Bit and BTB-

Perfect	All branches are correctly predicted, so there are no mispredict nor any misfetch penalties.
No-Pred	No Branch Prediction. All branches are either mispredicted or misfetched.
BTFNT	Backwards Taken, Forwards Not Taken. Static branch prediction.
21064	1-bit dynamic branch prediction for each instruction in the instruction cache.
PHT-Direct	Direct mapped Pattern History Table (PHT).
PHT-GAg	A single global history register is XORed with the program counter (PC) and used to index into the PHT.
BTB-2Bit	A BTB with each entry containing a 2-Bit counter for conditional branch prediction.
BTB-PAs	A BTB where each entry contains a 6-bit history register. This history register is used as the lower bits of an index into a PHT for predicting conditional branches. The upper bits of the index are the lower bits of the branch address.
BTB-GAg	A decoupled BTB that contains no conditional branch prediction information. Instead, conditional branches are predicted with the decoupled GAg, as in the PHT-GAg (XOR) architecture.
Infinite BTB's	An infinite BTB that only suffers from misprediction and cold-starts misses. We simulated this model for both the PAs and GAg (XOR) conditional branch prediction schemes.

Table 3: Architectures Simulated

PAs architectures, the branch prediction information (the two-bit counter and 6-bit history register), is associated or *coupled* with the BTB entry. Therefore, the dynamic conditional branch prediction information can only be used for branches in the BTB, and branches that miss in the BTB must use less accurate static prediction.

The "BTB-GAg" is an example of a *decoupled* design, where the branch prediction information is not associated with the BTB and is used for all conditional branches, including those not recorded in the BTB. This design is used in the PowerPC 604 [14]. The PowerPC 604 has a 64-entry fully associative BTB that holds the target address of the most recently taken branches, and uses a separate 512 entry PHT to predict the direction of conditional branches.

We simulated a number of configurations of these architectures. In particular, we varied the number of BTB entries, associativity of the BTB, and the size of the PHT. All of the BTB designs we simulated only update the BTB with taken branch addresses. This was shown to be more effective for both the coupled and decoupled architectures because the BTB is not filled with fall-through addresses that are easily calculated [2, 12]. Therefore, on a BTB miss for the coupled design, the branch is predicted as not taken. The other static prediction alternative is to predict the branch using the BTFNT scheme on a BTB miss. In our studies we have found that statically predicting the branches as not-taken performs better than BTFNT in this case, especially when compiler transformations are used to make the fall through path the most likely executed path [3].

4 Branch Architecture Performance

Throughout this section, the reader should be aware of the limitations of our study, and should not draw inferences beyond the systems we model. In particular, the Alpha AXP 21064 is a statically scheduled, dual-issue processor. Dynamically-scheduled processors may be more sensitive to branch architecture performance, because more instructions would be "in-flight," increasing the mispredict penalty. It is difficult to extrapolate how our results would differ with out-of-order execution, and this issue is the topic of a future study.

Table 4 shows the increase in execution time over an architecture using Perfect branch prediction for all the branch architectures we studied averaged over all programs. The different branch architectures are sorted by percent increase in execution time; architectures with poorer performance appear earlier in the table. Table 4 also shows the contribution of the instruction issues, stalls, caches, TLB, mispredict branch (MpB), and misfetched branch (MfB) penalties to the cycles needed to execute a program on average, as described earlier for Table 2. The last three columns give the actual percentage of mispredicted branches, misfetched branches, and the branch execution penalty, which combines the %MpB and %MfB into one metric. The %MpB contains all branches that are mispredicted, including indirect jumps, returns and conditional branches.

Table 4 shows that once a reasonable number of BTB entries have been added to a branch architecture, overall performance is best improved by adding resources to other architectural components, including TLB entries and the data cache. The results show that it is very important to have 2-bit conditional branch prediction, such as a PHT, although little performance improvement is seen as the size of the PHT is increased from a 512 entry PHT to a 4096 entry PHT. We found that increasing the PHT size for an infinite BTB beyond 32768 entries provided almost no improvement, because the remainder of the stalls arise from cold-start misses in the BTB and PHT.

Table 4 also shows the improved performance when a 64 entry BTB is added to the branch architecture. There is only a small improvement when using a 4-way associative BTB over a direct mapped BTB. Likewise, increasing the BTB from 64 to 512 entries results in only a very small performance improvement. The average increased percent of execution time for a 64 entry direct mapped BTB with a 512 entry PHT-GAg is 3.1%. In comparison a more costly 512 entry 4-way associative PAs with a 4096 entry PHT has an increased percent in execution time of 1.4%. Each decrease in execution time comes with a price, in both area and branch architecture complexity and potential increase in cycle time. Therefore, for a statically scheduled architecture like the Alpha 21064, the most aggressive branch prediction architecture that is justified by our experimentation is a direct mapped 64 entry BTB, with any 512 entry PHT configuration that we studied. For future architectures to be viable, they must achieve similar (or better) performance for cheaper hardware costs and make no sacrifice in cycle time [4].

Arch	BTB Size	A	PHT Size	%IET	CPI	% of Cycles							Branch Penalties		
						Iss	Stall	IC	DC	TB	MpB	MfB	%MpB	%MfB	BEP
No-Pred				27.16	2.981	30	15	8	18	4	23.04	0.67	86.85	13.15	4.47
BTFNT				11.10	2.647	35	17	10	22	5	9.47	2.32	30.95	39.00	1.94
21064				8.81	2.605	36	17	10	22	5	6.89	3.29	21.00	53.31	1.58
PHT-GAg			512	6.20	2.549	37	18	10	23	5	3.40	4.05	10.51	63.93	1.16
PHT-Direct			512	5.95	2.542	37	18	11	23	5	2.99	4.11	8.88	64.91	1.09
PHT-Direct			4096	5.77	2.538	37	18	11	23	5	2.73	4.14	8.01	65.39	1.05
PHT-GAg			4096	5.35	2.530	37	18	11	23	5	2.29	4.20	6.90	65.74	1.00
BTB-2Bit	64	1		4.27	2.512	38	18	10	23	5	4.92	0.50	14.72	8.03	0.82
BTB-PAs	64	1	512	3.86	2.504	38	18	10	23	5	4.50	0.51	13.51	8.03	0.76
BTB-PAs	64	1	4096	3.72	2.501	38	18	10	23	5	4.35	0.51	13.04	8.03	0.73
BTB-2Bit	64	4		3.65	2.500	38	18	10	23	5	4.27	0.43	12.92	6.91	0.72
BTB-PAs	64	4	512	3.22	2.492	38	18	11	24	5	3.82	0.43	11.69	6.91	0.65
BTB-GAg	64	1	512	3.13	2.489	38	18	11	24	5	3.12	1.02	9.62	15.51	0.64
BTB-PAs	64	4	4096	3.05	2.488	38	18	11	24	5	3.64	0.43	11.13	6.91	0.63
BTB-GAg	64	4	512	2.91	2.485	38	18	11	24	5	3.06	0.81	9.47	12.59	0.60
BTB-GAg	512	1	512	2.49	2.475	38	19	11	24	5	3.02	0.24	9.35	3.94	0.51
BTB-2Bit	512	1		2.40	2.471	38	19	11	24	5	2.94	0.13	8.73	2.20	0.46
BTB-GAg	512	4	512	2.37	2.472	38	19	11	24	5	3.02	0.06	9.32	1.22	0.48
BTB-GAg	Inf		512	2.33	2.471	39	19	11	24	5	3.01	0.00	9.29	0.04	0.46
BTB-GAg	64	1	4096	2.26	2.469	39	19	11	24	5	1.98	1.09	6.02	16.43	0.47
BTB-GAg	64	4	4096	2.04	2.465	39	19	11	24	5	1.92	0.87	5.87	13.44	0.43
BTB-2Bit	512	4		2.00	2.462	39	19	11	24	5	2.48	0.04	7.36	0.77	0.38
BTB-PAs	512	1	512	1.89	2.461	39	19	11	24	5	2.37	0.14	7.17	2.20	0.38
BTB-PAs	512	1	4096	1.74	2.458	39	19	11	24	5	2.17	0.14	6.58	2.20	0.35
BTB-GAg	512	1	4096	1.61	2.455	39	19	11	24	5	1.88	0.27	5.74	4.20	0.33
BTB-PAs	512	4	512	1.49	2.452	39	19	11	24	5	1.90	0.04	5.77	0.77	0.30
BTB-GAg	512	4	4096	1.49	2.452	39	19	11	24	5	1.87	0.07	5.71	1.35	0.30
BTB-GAg	Inf		4096	1.44	2.451	39	19	11	24	5	1.86	0.00	5.68	0.04	0.28
BTB-PAs	512	4	4096	1.35	2.448	39	19	11	24	5	1.72	0.04	5.22	0.77	0.27
BTB-PAs	Inf		512	1.35	2.448	39	19	11	24	5	1.71	0.00	5.09	0.03	0.25
BTB-PAs	Inf		4096	1.21	2.445	39	19	11	24	5	1.54	0.00	4.57	0.03	0.23
BTB-PAs	Inf		32768	1.17	2.444	39	19	11	24	5	1.48	0.00	4.40	0.03	0.22
BTB-GAg	Inf		32768	1.08	2.443	39	19	11	24	5	1.36	0.00	4.13	0.05	0.21
Perfect				0	2.419	39	19	12	25	5	0	0	0	0	0

Table 4: Average results for all the Branch Architectures sorted in terms of Percent Increase in Execution Time. The average cycles executed for each architecture is broken down as before.

Table 4 lets us directly compare the performance of decoupled GAg and coupled PAs architectures with equivalent BTB sizes. For the 64 entry BTB designs, the GAg scheme performs better than the PAs architecture because the coupled PAs architecture can only use the conditional branch prediction information on a BTB hit while the decoupled GAg uses the PHT predictors for all conditional branches. When increasing the size of the BTB to 512 entries, the PAs scheme performance usually surpasses that of the GAg, largely because of the increased BTB hit rate. However, this is not true in all cases; for example, a direct-mapped GAg with 512 BTB entries and 4096 PHT entries has (marginally) better performance than the corresponding PAs method. In general, the differences between these two architectures is too small to matter. We feel the GAg design should be used in an actual implementation, because the GAg design may be simpler to implement and update, and the PAs architecture requires extra storage, not needed in the GAg architecture, since it associates a 6-bit history register with each BTB entry.

5 The Accuracy of Branch Metrics

Branch architecture research, like all computer architecture research, is a computationally intensive undertaking. Normally, researchers try to identify a number of performance metrics that can be easily calculated, yet indicate the likely impact of a particular architectural feature. For example, the simulations used to accurately simulate the Alpha 21064 and calculate the CPI used in the previous sections were approximately 500-1000 times slower than a simulation that only calculated the branch execution penalty (BEP). Studies, including our own, have used the BEP for this very reason: it yields an intuitive performance metric, and it is significantly easier to calculate than the CPI. Other studies have used the percent of mispredicted branches (%MpB) for the same purpose.

Program	Correlation Coeff. (ρ^2) CPI *vs.* BEP	CPI *vs.* %MpB	BEP *vs.* %MpB
doduc	0.98	0.99	0.94
tomcatv	0.99	0.88	0.89
compress	1.00	0.92	0.92
eqntott	1.00	0.91	0.90
espresso	1.00	0.96	0.94
gcc	1.00	0.94	0.93
li	0.99	0.95	0.92
bc	0.99	0.96	0.93
flex	1.00	0.92	0.92
gzip	1.00	0.93	0.93
indent	1.00	0.95	0.95
od	0.99	0.97	0.94
wdiff	1.00	0.93	0.93
cfront	0.97	0.99	0.94
idl	1.00	0.95	0.95
lic	1.00	0.93	0.91
porky	0.99	0.94	0.90

Table 5: The square of the correlation-coefficient for each program across all branch architectures. A $\rho^2 = 1$ indicates that changes in the BEP due to a particular branch architecture are predictive of the CPI for that program using that architecture.

We were curious how accurately the BEP and %MpB predict the CPI for the architecture we examined. The standard statistical test for this problem is to compute the sample correlation coefficient, ρ. Given ρ for two series, ρ^2 represents the probability that the series are linearly related; $\rho^2 = 99\%$ indicates that 99% of the variation in one series is predicted by a change in the second series. The CPI and BEP for all branch architectures averaged over all programs resulted in $\rho^2 = 99.9\%$; i.e., over all the architectures studied, a change in the BEP is a very accurate predictor for the CPI. The corresponding measurement for the %MpB is $\rho^2 = 96.9\%$, indicating that the %MpB is a slightly less accurate predictor of the CPI than the BEP. This is only natural, since the BEP includes the misfetch penalty as well as the mispredict penalty.

The predictive accuracy of the BEP and %MfB is also consistent across a range of branch architectures. Table 5 lists ρ^2 for each program across all the branch architectures considered. Table 5 shows that CPI can be accurately predicted by the BEP and the %MpB, although the BEP is a more accurate metric. As expected, the BEP also predicts the %MpB with reasonable accuracy. It is essential to understand that we are asserting that the predictive quality of the BEP for the CPI only applies to the statically-scheduled architecture we model.

It is understandable that the BEP would be correlated with the CPI on the 21064 architecture. The nominal branch mispredict penalty is 5 cycles and the misfetch penalty is 1 cycle. In some circumstances, these penalties can vary. For example, if a load that precedes a conditional branch misses in the data cache, and the instruction that is the target of the branch needs the data, then the branch may be resolved before the miss finishes, masking the mispredict penalty that might have occured if the load had hit in the cache. Likewise, if only one instruction in an issue-pair can actually be issued, the processor has an additional cycle to determine the next fetch address, masking the misfetch penalty.

Note that the correlation coefficient does not indicate the degree of importance of the BEP. It simply indicates that the BEP and CPI are linearly correlated. The least-squares solution for all architectures over all programs indicates that CPI $= 2.35 + 0.073 \times$ BEP. Thus, although the BEP is important in the overall CPI, a difference of 1.0 in the BEP results in a 3.1% increase in the CPI on average. The branch architectures we examined had a BEP between 0.2 and 2.0.

6 Understanding the scope of this study

A valid question to ask is how do the results presented in this paper apply to future wide-issue architectures that have wider instruction issues, deeper pipelines, larger branch penalties or out-of-order execution? We feel it is important to understand the scope of our study, and the intent of evaluation we have conducted. All of the results we have described are relatively specific to the 21064 implementation of the Alpha architecture, although some general conclusions can be drawn from this study.

Within the limitations imposed by the particular architecture we studied, we feel that this study is important because it provides detailed system performance showing the actual gain in going from no branch prediction (27% increase in execution time) all the way down to Perfect prediction, using various branch prediction architectures. We are not aware of any previous study providing this detailed performance comparison for modern branch prediction architectures. We also feel that we have demonstrated that the BEP is an accurate predictor for the CPI for the class of architectures we considered. Therefore the BEP metric should be used when comparing different branch prediction architectures if the real CPI or execution time is not available; however we recommend that this be confirmed for architectures radically different than the 21064 architecture.

Our observations come with an important proviso – the importance of a branch architecture is highly dependent on the underlying architecture and the penalties introduced by other subsystems. For example, data cache overheads constitute $\approx$ 24% of the total execution time in the programs we measured. Larger first and second-level caches will reduce the cache overhead. Likewise, a dynamically-schedule architecture may be able to mask a larger fraction of the cache latency, depending on the processors ability to predict branches. Simulating the interaction between the operating system and an application or a mixture of applications may also indicate that other branch architectures are more desirable.

7 Conclusions

Microprocessor design is an art that balances many issues, including area and power budgets, design complexity, patent issues, robustness across a number of applications and a particular implementation technology. It is important to understand the contribution of individual architectural features in the context of the full cpu design. At

some point, architectural designs must address these issues, rather than solely address performance. This paper provides a system level study of several branch prediction architectures including recently proposed two-level correlated branch history schemes. We provide the performance comparison in terms of execution time using a full Alpha AXP 21064 architectural simulation model.

Our results show, for the processor we examined, that a significant performance gain is achieved by adding a small 512 entry PHT to the architecture, as was done in the Alpha 21064A. Little benefit is gained by increasing the size of the PHT. An additional reduction in execution time, half of that provided by adding the PHT, is seen by adding a small 64-entry direct mapped BTB, provided that this does not impact the cycle time. The results also show that increasing the BTB size and associativity provides minimal improvements. A processor using a 64 entry direct mapped BTB with a decoupled 512 entry PHT achieves an execution time within 3% of the execution time for prefect branch prediction.

Our results indicate that continuing to improve the branch behavior for processors and programs similar to the ones in this study is simply "polishing a round ball". This does not mean that reducing the BEP is an unimportant problem; rather, it means that for further reductions in the BEP to become meaningful, other pipeline stalls such as cache and TLB stalls first need to be greatly reduced on this type of architecture. Alternatively, researchers proposing new branch prediction architectures should demonstrate how they will work on a processor with dynamic out-of-order execution, deeper pipelines and wider issue widths. They should also emphasize designs that are faster, less complex, more testable or easier to implement. We feel a study similar to this one is needed to determine the importance of branch prediction on a dynamic execution architecture.

Acknowledgments

We could not have completed this study without significant equipment and software support from Digital Equipment Corporation, and the cooperation and assistance of the many people who have worked on ATOM and Zippy, particularly Mike McCallig, Alan Eustace, Amitabh Srivastava and Dave Webb. We would also like to thank the anonymous reviewers for their useful comments. Brad Calder was supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland. This work was funded in part by NSF grant No. ASC-9217394 and ARPA contract ARMY DABT63-94-C-0029.

References

[1] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 158–170, San Jose, California, 1994.

[2] Brad Calder and Dirk Grunwald. Fast & accurate instruction fetch and branch prediction. In *21st Annual International Symposium of Computer Architecture*, pages 2–11. ACM, April 1994.

[3] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251. ACM, 1994.

[4] Brad Calder and Dirk Grunwald. Next cache line and set prediction. In *22nd Annual International Symposium of Computer Architecture*, pages 287–296. ACM, June 1995.

[5] Digital Equipment Corperation, Maynard, Mass. *DECchip 21064 Microprocessor: Hardware Reference Manual*, October 1992.

[6] R. Kessler and M. Hill. Page placement algorithms for large direct-mapped real-index caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[7] Johnny K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 21(7):6–22, January 1984.

[8] Scott McFarling. Combining branch predictors. TN 36, DEC-WRL, June 1993.

[9] Scott McFarling and John Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. ACM, 1986.

[10] Ravi Nair. Optimal 2-bit branch predictors. *IEEE Transactions on Computers*, 44(5):698–702, May 1995.

[11] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Mass., October 1992. ACM.

[12] Chris Perleberg and Alan Jay Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, April 1993.

[13] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*, pages 135–148. ACM, 1981.

[14] S. Peter Song, Marvin Denman, and Joe Chang. The PowerPC 604 RISC microprocessor. *IEEE Micro*, 14(5):8–17, October 1994.

[15] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *1994 Programming Language Design and Implementation*, pages 196–205. ACM, June 1994.

[16] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch predictions. In *19th Annual International Symposium of Computer Architecture*, pages 124–134, Gold Coast, Australia, May 1992. ACM.

[17] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *25th International Symposium on Microarchitecture*, pages 129–139, Portland, Or, December 1992. ACM.

[18] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium of Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.