Next Cache Line and Set Prediction

Brad Calder and Dirk Grunwald

Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309–0430 USA
{calder,grunwald}@cs.colorado.edu

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. Several researchers have proposed very effective fetch and branch prediction mechanisms including branch target buffers (BTB) that store the target addresses of taken branches. An alternative approach fetches the instruction following a branch by using an index into the cache instead of a branch target address. We call such an index a *next cache line and set* (NLS) predictor. A NLS predictor is a pointer into the instruction cache, indicating the target instruction of a branch.

In this paper we examine the use of NLS predictors for efficient and accurate fetch and branch prediction. Previous studies associated each NLS predictor with a cache line and provided only one-bit conditional branch predictors. Our study examines the use of NLS predictors with highly accurate two-level correlated conditional branch architectures. We examine the performance of decoupling the NLS predictors from the cache line and storing them in a separate tag-less memory buffer. Our results show that the decoupled architecture performs better than associating the NLS predictors with the cache line, that the NLS architecture benefits from reduced cache miss rates, and it is particularly effective for programs containing many branches. We also provide an in-depth comparison between the NLS and BTB architectures, showing that the NLS architecture is a competitive alternative to the BTB design.

Keywords: Instruction fetch prediction, Branch prediction, Branch target buffers

1 Introduction

Modern superscalar processor designs are extremely sensitive to control flow changes. Changes in control flow, be they conditional or unconditional branches, direct or indirect function calls, or returns are not detected until those instructions are decoded. The target addresses for conditional, unconditional branches, and procedure calls are typically not calculated until the instruction is decoded. To keep the pipeline fully utilized, processors typically fetch the address following the most recent address. If the decoded instruction is a break in control flow, the previously fetched instruction 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.

The final destination for conditional branches, indirect function calls and returns are typically not available until a later stage of the pipeline. 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, instructions fetched from the incorrect instruction stream must be discarded, leading to several pipeline bubbles. This is called a branch *mispredict penalty*. In practice, pipeline stalls due to mispredicted breaks in control flow degrade a programs performance more than the misfetch penalty.

As processors issue more instructions concurrently, these penalties increase, and it is more likely that a branch will occur as more instructions are fetched per cycle, decreasing the likely-hood that the fall-through instruction will be executed. A branch target buffer (BTB) is one mechanism for efficiently predicting the next instruction fetch when a branch is encountered. In this paper we examine an alternative to the BTB called next cache line and set (NLS) prediction. A NLS predictor is a pointer into the instruction cache indicating the target instruction of a taken branch. Johnson [4] proposed a similar design using cache indices to predict the next instruction fetch. We propose an alternate organization that improves fetch prediction accuracy.

In this paper we examine two varieties of the NLS architecture. The NLS-cache is similar to the branch architecture described by Johnson, where each NLS predictor is associated with a cache line. The NLS-table uses NLS predictors stored in a separate direct mapped tag-less memory buffer. We also examine the effects of combining the NLS predictors with modern two-level correlated branch prediction architectures. Our results show that the NLS architecture's performance improves as the instruction cache miss rate is lowered, and that the NLS architecture is particularly effective for programs with many branches.

In §2, we describe prior branch prediction work. In §3 we describe an efficient BTB architecture and in §4 we describe the NLS architecture. We use trace-driven simulation to compare the performance of these two architectures. Section 5 describes the

ISCA '95, Santa Margherita Ligure Italy
© 1995 ACM 0-89791-698-0/95/0006...$3.50

programs we traced and how we analyzed them. In §6, we describe the NLS and BTB results and compare our NLS architecture to the cache index architecture proposed by Johnson. In §7, we provide a detailed performance comparison of the NLS and BTB architectures and we summarize our findings in §8.

2 Prior Branch and Fetch Prediction Research

This section briefly surveys prior work on branch prediction techniques used in this paper. Branch target buffers (BTB) have been used as a mechanism for branch and instruction fetch prediction, effectively predicting the behavior of a branch [1, 6, 9, 13, 15, 21]. The Intel Pentium is an example of a modern architecture using BTBs – 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. For each BTB entry, the Pentium uses a two-bit saturating counter to predict the direction of a conditional branch [6]. In this BTB architecture, the branch prediction information (the two-bit counter), is associated or *coupled* with the BTB entry. Thus, the dynamic prediction can only be used for branches in the BTB, and branches that miss in the BTB must use less accurate static prediction.

An alternative BTB architecture is the *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. In an earlier study [2], we found that decoupled designs performed better than coupled designs. This allows conditional branches that do not hit in the BTB to use dynamic prediction. The PowerPC 604 is an example of an architecture using a decoupled design [16]. 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 pattern history table (PHT) to predict the direction for conditional branches.

There are several different PHT variations. Pan *et al.* [12] and Yeh and Patt [20, 22] 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 simplest example is the *degenerate method* of Pan *et al.* [12]. When using a 2^k entry table, the processor maintains a k-bit shift register (the global history register) that records the outcome of previous branches (a taken branch is encoded as a 1, and a not-taken branch as a 0). The shift register is used as an index into the PHT, much as the program counter is used for a direct-mapped PHT. This provides contextual information and correlation about particular patterns of branches. Recently, McFarling [8] showed that combining branch history with the branch's address was more effective. His method used the exclusive-or of the global history register and the branch address as the index into the PHT.

The NLS and BTB architectures we study in this paper use a decoupled design with a separate PHT to predict the direction of conditional branches. For both of the architectures, we use McFarling's form of the two-level PHT [8]. In the next two sections we first describe the BTB architecture and then our alternative NLS architecture.

3 A BTB-based Instruction Fetch Architecture

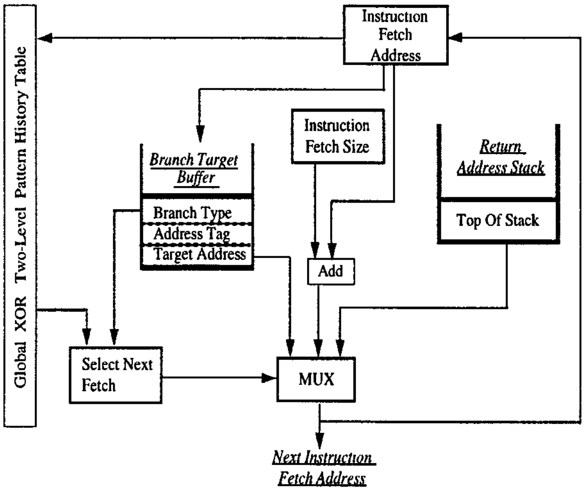

Figure 1: A schematic representation of a coupled BTB branch prediction architecture using two-level correlated branch prediction for conditional branches and a return stack for return instructions.

Figure 1 is a schematic representation of the decoupled BTB and PHT branch prediction and instruction fetch architecture we simulated. In Figure 1 the next instruction fetch address is concurrently offered to: the instruction cache, the BTB, and the PHT. The address is also used to compute the fall-through instruction's address. A 32-entry return address stack [5] predicts return instructions, and conditional branches are predicted using the pattern history table organization described by McFarling [8]. This is the degenerate scheme of Pan *et al* [12], where we XOR the global history register with the program counter and use this to index into a 4096 entry (1KByte) PHT. In this model, we store only taken branches in the BTB, since previous studies have shown this to be more effective [2, 13]. If a branch is not taken while it is in the BTB, we leave the branch (target address) in the BTB until it is removed due to the LRU replacement policy, since we might need the taken target address again in the near future. In this architecture, the BTB's main purpose is to provide the taken target address and the branch type.

4 Next Cache Line and Set Prediction Architecture

The NLS architecture is similar to the BTB architecture and is illustrated in Figure 2. The difference between these two architectures is the NLS architecture is a tagless table providing a pointer into the instruction cache to the next instruction to execute rather than the target address, as in the BTB. Like the BTB, the main purpose of the NLS architecture is to eliminate misfetch penalties by providing a pointer to the cache line and instruction that is the target of a branch. This allows the next instruction to be correctly fetched from the instruction cache while the branch instruction is decoded and the target address is calculated. The NLS predictor also predicts indirect jumps and provides the branch type.

As shown in Figure 2 there are three predicted addresses available for the next instruction fetch. These are the NLS predictor, the fall-through line (previous predicted line + fetch size), and the top of the return stack. Each NLS predictor contains the following fields:

Type Field: The following table shows the possible prediction sources represented by the NLS type field. The type field is

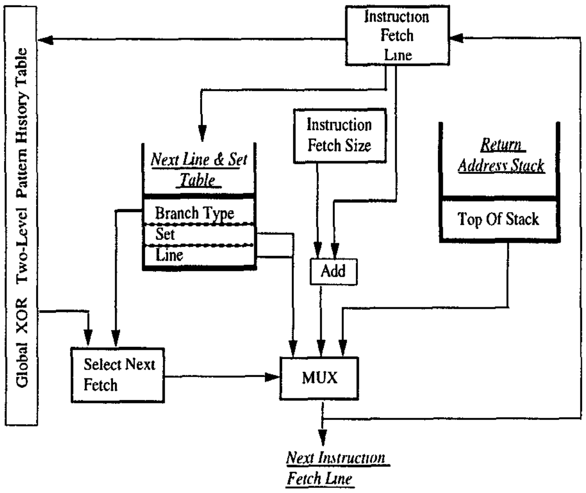

Figure 2: A schematic representation of the NLS-table architecture.

used to determine the proper prediction mechanism, shown in Figure 2, to use when fetching the next instruction. Unused NLS entries have "00" stored in the type field indicating the entry is invalid.[1]

		Branch Type	Prediction Source
0	0	Invalid Entry	
0	1	Return Instruction	Return Stack
1	0	Conditional Branch	NLS Entry, Conditional on PHT
1	1	Other Types of Branches	Always use NLS Entry

Line Field: This field contains the line number to be fetched from the instruction cache. The high-order bits indicate the line in the instruction cache and the low-order bits are used to indicate the actual instruction in that line.

Set Field: In a multi-associative instruction cache, the destination line may be in any set. The set field is used to indicate where the predicted line is located if a multi-associative cache is used. It is not needed for a direct mapped cache.

The NLS architecture assumes that during the instruction fetch stage of the pipeline, each instruction can easily be identified as a branch or non-branch instruction. The BTB does not have to make this assumption since an instruction is known to be a branch if it hits in the BTB. If the instruction set encoding does not contain such a distinguishing bit in the instruction, that information can be stored in the instruction cache or an instruction type prediction table, as described in [2]. Encoding this information in the instruction improves the fetch accuracy for the NLS architecture, since non-branch instructions fetch the fall-through address while branch instructions use NLS predictors.

If the instruction being fetched from the instruction cache indicates that it is a branch instruction, the NLS predictor is used and the type field is examined to choose among the possible next fetch addresses. Return instructions use the return stack. Unconditional branches and indirect branches use the cache line specified by the NLS entry. If the type field indicates a conditional branch, the architecture uses the prediction given by the PHT, as is done in the BTB architecture. If the branch is predicted as taken, the NLS line and set fields are used to fetch the appropriate cache line and instruction from the instruction cache. If the conditional branch is predicted as not-taken, the precomputed fall-through line address is used on the next instruction fetch.

The NLS entries are updated after instructions are decoded and the branch type and destinations are resolved. The instruction type determines the type field and the branch destinalion determines the set and line field. Only taken branches update the set and line field, but all branches update the type field. A conditional branch which executes the fall-through should not update the set and line field, since that would erase the pointer to the target instruction. For conditional branches, this allows the branch prediction hardware to use either the NLS predictor for taken conditional branches or to use the precomputed fall-through line, depending on the outcome of the PHT.

4.1 NLS-Table versus NLS-Cache

There are several possible variations on the basic NLS architecture design, and they share many common structures. Figure 2 shows one possible design. The intuition behind this architecture is that a branch target address is actually a pointer into the instruction cache. This pointer can be represented by an index pointing to the target instruction of a taken branch.

We considered two possible designs: "NLS-caches" and "NLS-tables". In the NLS-cache, we associate the NLS predictors with each cache line. Thus, the NLS entries share the instruction address tag with the cache line. There may be multiple NLS predictors per cache line and we studied various replacement policies and methods of associating the NLS predictors with specific instructions in a cache line. The second design, the NLS-table, is a simpler and more effective design that uses a tag-less direct-mapped table of NLS predictors. The table is indexed by the branch instruction's address. Both architectures use the NLS entries to predict the next line to fetch for a branch instruction, both architectures use the same conditional branch prediction and return-prediction mechanisms used in the BTB, and both designs replace the BTB with the NLS information.

The NLS-table has three advantages over the NLS-cache design and one disadvantage. These points arise because the NLS predictors are coupled with the cache lines in the NLS-cache design and they are decoupled from the cache in the NLS-table design. For the NLS-cache architecture, we found that associating two NLS predictors with an eight instruction cache line to be the most effective organization. This design restricts the use of the NLS predictors in the NLS-cache, since some cache lines may not have any branches while other cache lines may contain several branches. In contrast, the NLS-table uses the lower order bits of the branch instructions address to index into a tagless table. This allows a cache line to use as many NLS predictors as needed. The second advantage comes when an instruction cache line is replaced. The NLS-cache prediction information associated with a replaced cache line is discarded while the prediction information for the NLS-table is preserved across cache misses. The final advantage appears when examining different instruction cache sizes. As the instruction cache size doubles, the number of NLS-cache predictors must also double to achieve the same branch prediction performance. Therefore the NLS-cache size increases linearly with an increase in instruction cache size while the NLS-table size increases only logarithmically. This can greatly increase the cost of the NLS-cache design for large caches. There

[1] The type field is not needed for the NLS or BTB architectures if the type information can be easily extracted from the fetched instruction before the fetch cycle completes, or from the instruction cache if the information has been pre-decoded.

	# Insn's		Conditional Branches						Percentage of Breaks During Tracing				
Program	Traced	% Breaks	Q-50	Q-90	Q-99	Q-100	Static	%Taken	**%CBr**	**%IJ**	**%Br**	**%Call**	**%Ret**
doduc	1,149,864,756	8.53	3	175	296	1,447	7,073	48.68	81.31	0.01	4.97	6.86	6.86
espresso	513,008,174	17.12	44	163	470	1,737	4,568	61.90	93.25	0.20	1.88	2.29	2.39
gcc	143,737,915	15.97	245	1,612	3,742	7,640	16,294	59.42	78.85	2.86	5.75	6.04	6.49
li	1,355,059,387	17.67	16	52	127	556	2,428	47.30	63.94	2.24	7.74	12.92	13.16
cfront	16,529,540	13.66	69	833	2,894	5,644	17,565	53.18	73.45	2.17	6.40	8.72	9.26
groff	56,840,596	16.38	107	408	976	2,889	7,434	54.17	66.12	4.80	7.80	8.77	12.51

Table 1: Measured attributes of the traced programs.

is a disadvantage for the NLS-table in making it a tagless table, because prediction information from one branch may be erroneously used for another branch. Our results show that this effect is small for the NLS-table design when compared to the benefits of the three advantages mentioned above.

4.2 Using Next Line Addresses with the Instruction Cache

Unlike the BTB architecture, the NLS architecture does not have a full next target address to offer to the instruction cache. It only has the lower order bits of the full target address (the cache line index). This is not a problem for a direct mapped cache, since the tag check against the target address can be performed in the decode stage of the pipeline. When an associative cache is used, the cache needs to be slightly modified in order to properly use the next line address. The following two different approaches may be taken.

The traditional implementation of an associative cache selects the appropriate line from a set by performing a full tag comparison on the tags from the different sets. For all branch instructions, the set field in the NLS predictor is used to predict the the instruction cache set instead of performing the tag comparison. When the precomputed fall through line address is used, a full tag comparison is performed. The full fall-through address can be calculated by the time the cache needs to perform the tag comparison using the precomputed fall-through line address, the carry bit from the addition of the fall-through line address calculated in the previous cycle, and the previous instruction's tag.

The second approach to using next line addresses with an associative cache is more elegant and can lead to improved cache performance. In this approach we assume that each cache line has a *setfield* associated with it. This set field has the same use as the NLS set field, and it predicts the set where the fall-through line is located for each cache line. For each instruction cache lookup, either the NLS predictor's set field, for a branch instruction, or the previous cache line's set field, for a non-branch instruction, is used to predict the set for the current cache access. Since the set field is used on every cache access, only one cache set is driven at a time during the lookup and the tag comparison can be performed in the decode stage as if the cache where direct mapped. If the set prediction was incorrect and the tag does not match the destination address computed in the decode stage, the other sets in the cache need to be checked in order to find the correct entry or to find if there is a cache miss. This design is suitable for a two-way associative cache. If the first set prediction is incorrect, the remaining set is checked for the instruction. For higher degrees of associativity, other techniques may be applied when the set prediction is incorrect, but these are beyond the scope of this paper.

5 Experimental Methodology

We used trace driven simulation to quantify the performance for many BTB and NLS architecture configurations. We instrumented the programs from the SPEC92 benchmark suite and object-oriented programs written in C++. We simulated several programs but only show information for six programs because we felt this would be more useful than presenting less detailed results for more programs. We picked three of the programs (`gcc`, `cfront` and `groff`) because they have high instruction cache miss rates, execute a lot of branches, and the branches are hard to predict.

We used ATOM [17] to instrument the programs. Due to the structure of ATOM, we did not need to record traces and could trace very long-running programs. The programs were compiled on a DEC 3000-400 using either the DEC FORTRAN, C, or C++ compiler. All programs were compiled with standard optimization (`-O`). For the SPEC92 programs, we used the largest input distributed with the SPEC92 suite. The alternate programs include: `cfront`, version 3.0.1 of the AT&T C++ language preprocessor written in C++ and `groff` and a version of the `ditroff` text formatter written in C++. For these alternate programs, we used inputs we hoped would exercise a large part of the program.

Table 1 describes the branching activity of the programs we instrumented. The first columns list the number of instructions traced and the second column indicates the percentage of simulated instructions that could cause a break in control flow. The columns labeled 'Q-50','Q-90','Q-99' and 'Q-100' show the number of conditional branch instructions that contribute to 50, 90, 99 and 100% of all the executed conditional branches in the traced program. Thus, in doduc, three branch instructions constitute 50% of all executed conditional branches. The column labeled "static" represents the number of conditional branch sites in the program. The eighth column shows the percentage of executed conditional branches that are 'taken'. The last five columns describe the frequency of different type of branches encountered during tracing: conditional branches (**CBr**), indirect jumps (**IJ**), unconditional branches (**Br**), procedure calls (**Call**) and procedure returns (**Ret**).

5.1 Architectures Simulated

For each program, we simulated 8KB, 16KB, and 32KB instruction caches with 32 byte cache lines and 4 byte instructions. For each cache size, we simulated direct mapped, 2-way and 4-way associative LRU replacement caches. When simulating the NLS-cache architecture we used one to four NLS predictors per cache line with varying replacement policies. We found that two NLS predictors per cache line gave performance comparable to the NLS-table and BTB architectures. In this configuration, the first NLS predictor is associated with the first four instructions in the cache line and the

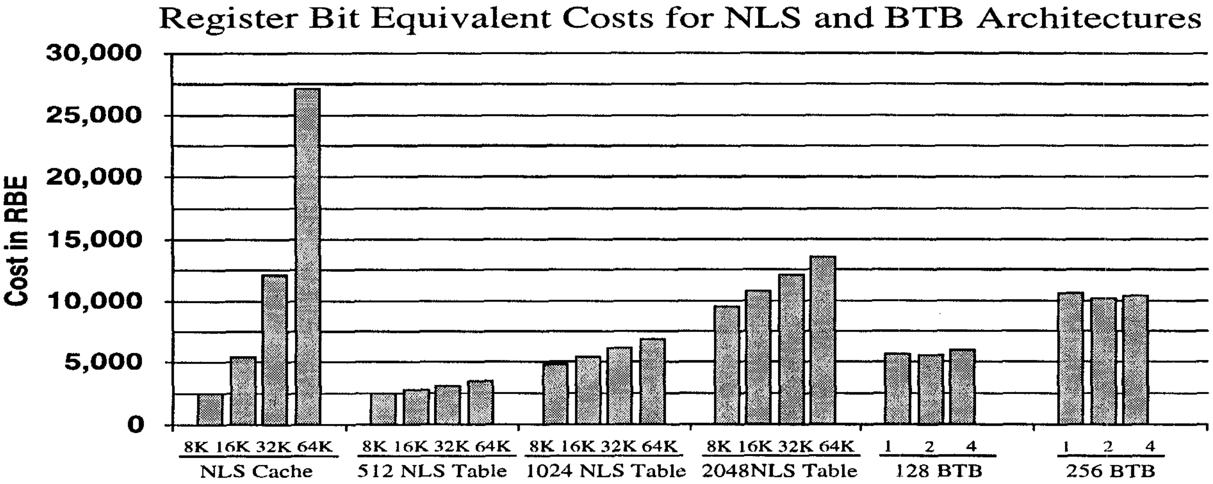

Figure 3: Register bit equivalent costs for the NLS-cache and a 512, 1024 and 2048-entry NLS-table for cache sizes of 8K, 16K, 32K and 64K, and for a 128-entry and 256-entry BTB with associativities of one, two and four.

second NLS predictor is associated with the last four instructions in the cache line. We only show results for this configuration of the NLS-cache architecture because of space limitations.

When simulating the NLS-table architecture, we simulated NLS-table sizes with 512, 1024 and 2048 NLS predictors. For the BTB architecture, we simulated 128-entry and 256-entry BTB organizations with direct mapped, 2-way and 4-way associativity with LRU replacement. Both the BTB and NLS architectures used a 32-entry return stack [5] to predict procedure returns and a two-level correlated 4096-entry pattern history table for conditional branches. The accuracy of the pattern history table is the same for both the BTB and NLS architectures. This allows us to isolate the fetch performance differences between the BTB and NLS architectures.

5.2 Performance Metrics

We compare the branch architectures using a number of performance metrics. We record the instruction cache miss rate since the NLS and BTB architectures may fetch different instructions,even for the same cache organization. We also record the percentage of misfetched branches (%MfB), and the percentage of mispredicted branches (%MpB). Note that a mispredicted branch is never counted as a misfetched branch and *visa versa*. It is often difficult to understand how each of these metrics influence processor performance. Yeh & Patt [21] defined the *branch execution penalty* to be:

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

The BEP reflects the average penalty suffered by a branch due to misfetch and mispredict penalties. With a BEP of 0.5, the average branch incurs a half-cycle execution penalty. The BEP provides a more intuitive understanding of how the two penalties interact, but requires us to use specific misfetch and mispredict penalties. For our results, we assume a one cycle misfetch penalty and a four cycle mispredict penalty, since these costs are reasonable for current superscalar architectures. In all of our BEP graphs we break the results into two parts. The top part shows the fraction of the BEP caused by the misfetch penalties and the lower part shows the fraction due to the mispredict penalties.

We also provide the *cycles per instruction* in order to illustrate overall program performance. We define CPI to be:

$$\text{CPI} = \frac{\begin{array}{c}(\#\ \text{Instructions executed} + \text{BEP} \times \#\ \text{branches} + \\ \#\ \text{I-cache misses} \times \text{miss penalty})\end{array}}{\#\ \text{Instructions executed}}.$$

We assumed a five cycle instruction cache miss penalty. With a CPI of 1.25, the average instruction takes 1.25 cycles to execute. Since we are simulating a single-issue architecture, the CPI can not be less than 1. The CPI does not include data cache misses or other resource conflicts.

6 Results

In order to compare the NLS architecture to the BTB we must first determine the resource costs for each architecture. In order to evaluate the area implementation costs of the NLS and BTB architectures, we used the register bit equivalent (RBE) model for on-chip memories proposed by Mulder *et al* [11], where one RBE equals the area of a bit storage cell.

Figure 3 shows the RBE costs for implementing the NLS and BTB architectures using Mulder *et al's* on-chip memory area model. The figure shows the RBE costs for the NLS-cache (with two NLS predictors per cache line) and a 512, 1024, and 2048 entry NLS-table for cache sizes of 8K, 16K, 32K and 64K. It also shows the RBE cost for a 128 entry and 256 entry BTB with associativities of one, two and four. The RBE cost of the NLS architecture depends on the size of the instruction cache. The NLS-table's RBE cost increases logarithmically as the instruction cache size increases, since the line field for each NLS predictor has to also increase. When the number of lines in the instruction cache are doubled, another bit must be added to each NLS predictor's line field. In the NLS-cache architecture, the number of NLS entries per cache line is constant, and as the cache size increases, the space devoted to NLS entries increases linearly. The RBE cost of the BTB architecture depends

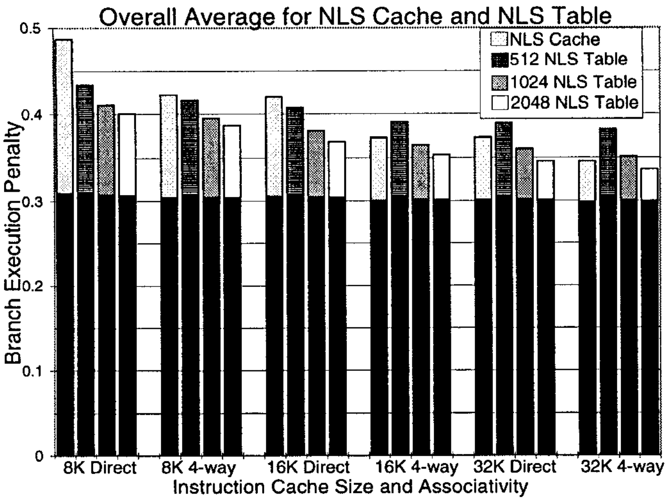

Figure 4: Branch execution penalty for the NLS-cache architecture and the 512, 1024 and 2048 entry NLS-table architectures for direct mapped and 4-way associative caches of size 8K, 16K and 32K. The BEP value is broken into two parts with the upper part representing the fraction of the BEP due to the misfetch penalty, and the lower part the mispredict penalty.

on the associativity of the BTB and the size of the address space not on the size of the instruction cache. In performing the BTB calculations, we assumed a 32-bit address space is used. If the address space is increased, the cost of the BTB would also increase.

6.1 Performance of the NLS Architecture

Figure 4 shows the branch execution penalty results averaged over the programs in Table 1. The figure shows results for the NLS-cache, and for 512, 1024 and 2048 entry NLS-tables for varying instruction cache sizes and associativities. Each branch execution penalty (BEP) value is broken into two parts with the upper part representing the fraction of the BEP due to the misfetch penalty, and the lower part the mispredict penalty.

Figure 4 shows that the NLS-table consistently outperforms the NLS-cache when architectures with equivalent costs are examined: the NLS-cache and the 512 NLS-table have equivalent costs when using an 8K instruction cache, the NLS-cache and the 1024 NLS-table have equivalent costs when using a 16K instruction cache, and the NLS-cache and the 2048 NLS-table have equivalent costs when using a 32K instruction cache. In terms of the RBE cost, the NLS-cache is practical for only small caches (8K and 16K), but even then, the NLS-table architecture has better performance when comparing architectures with equivalent costs. The difference in performance arises from the NLS-cache discarding useful prediction information for each instruction cache miss and because the predictors in the NLS-cache can only be used for the cache line they are associated with. In contrast, the NLS-table preserves prediction information across cache misses and a NLS-table predictor's use is not restricted to a given cache line. Figure 4 also shows that the difference in performance between the 512 and 1024 NLS-tables is small, and the difference between the 1024 and 2048 NLS-tables is smaller still. In the remainder of the paper we focus on the NLS-table design and only give results for the 1024 entry NLS-table.

6.2 Related Work

There are several branch prediction strategies related to the NLS design. Our NLS-table architecture was derived from the BTB: each uses a table holding pointers to branch destinations. The primary difference, besides eliminating the tag, is that the BTB encodes the full address, while the NLS encodes only the instruction cache line and set, allowing for larger NLS-tables.

Bray and Flynn [1] described a design similar to the NLS-cache that associated branch target addresses with each cache line. As in our study, they found approximately one entry per four instructions provided the most cost effective design.

Johnson [4], suggested the idea of using cache successor indices as in the NLS-cache architecture for instruction fetch and branch prediction. His architecture associated the cache indices with each cache line as the NLS-cache architecture does. The architecture he studied is slightly different than our NLS-cache design since he only considered using the index for one bit conditional branch prediction. With one bit prediction, the cache index stores either a pointer to the fall-through line or the target line for the next instruction fetch. In order to predict the fall-through line, the cache index is updated even when a non-taken branch is executed. By comparison, we only update the NLS predictor when taken branches are encountered to obtain improved branch prediction accuracy when using a decoupled PHT.

Variations on the NLS-cache design can be found in recent microprocessor architectures. The TFP microprocessor (MIPS R8000) [3] has a 1024 entry NLS-cache architecture similar to the design proposed by Johnson. It has one NLS predictor for every four instructions, and a one-bit branch predictor coupled with each NLS predictor. The UltraSPARC microprocessor also uses a similar 1024 entry NLS-cache design, associating an NLS predictor with every four instructions. Instead of using one-bit prediction as in the TFP, the UltraSPARC uses a 2-bit dynamic conditional branch predictor for every two instructions in the instruction cache.

The NLS-table design uses an independent table of next line and set predictors. This basic design was recently patented by Steely and Sager [18]. However, they have not published any performance comparisons, and the patented design only addresses direct mapped caches, while our design addresses both direct mapped and associative caches. Furthermore, the patented architecture uses a single "computed goto" register to store the destination of indirect jumps. By comparison, we use the NLS predictor to provide the predicted cache index for all branch destinations other than fall-through branches and return instructions. Although we developed our NLS architecture independently, there are several similarities as well as other differences; see [18] for more details.

6.3 Performance of the BTB Architecture

Figure 5 shows the average branch execution penalty (BEP) for the programs simulated in this paper. The 1024 entry NLS-table has better performance than the similar costing 128 entry BTB, even when the BTB has a high degree of associativity. The 1024 entry NLS-table and 256 entry BTB exhibit comparable performance even though the 1024 entry NLS-table has roughly half the RBE cost of the 256 entry BTB.

When comparing the NLS-table to the BTB, one must keep in mind that a direct mapped BTB has a shorter access time than an associative BTB. Figure 6 shows the estimated access time, in nanoseconds, for a 128 entry BTB and 256 entry BTB with direct mapped, two, and four way associativity. These estimates were derived using the CACTI timing model of Wilton and Jouppi [19].

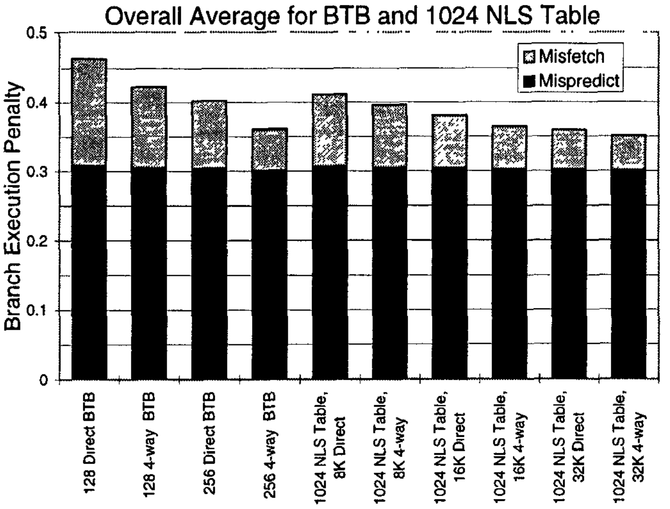

Figure 5: Branch execution penalty for the 1024 entry NLS-table architecture for direct mapped and 4-way associativity instruction caches of size 8K, 16K and 32K, and for a 128 entry and 256 entry BTB.

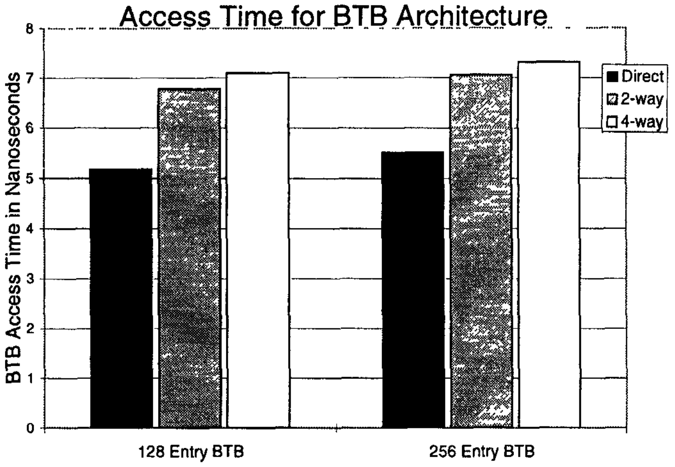

Figure 6: Access time for the BTB architecture with varying associativities. The relative values between the BTB access times are more important than the absolute values for a particular processor technology.

Their model derives access times for direct mapped and associative caches such as BTBs, but not for tag-less direct mapped memory buffers. Therefore we do not show the access times for the NLS-table, but we believe it would be similar to that of a direct mapped BTB. This figure shows the access time differences between direct mapped and associative cache structures. The differences arise from the extra time needed to perform the tag comparison. In a direct mapped cache, the tag comparison can be done in parallel while the data output is being driven to the next stage. The figure shows that the 4-way associative BTB access time is 30 to 40% longer than direct mapped BTBs of the same size. This should be considered when comparing the performance of the direct mapped BTB and NLS architecture to the associative BTB architectures since the cycle limitation of the instruction fetch may effect the entire machine. In [3], the designers of the TFP (MIPS R8000) microprocessor stated:

> *We evaluated several well-known branch prediction algorithms for layout size, speed, and prediction accuracy. The most critical factor affecting area was the infrastructure required to support a custom block: power ring and power straps to the ring, and global routing between the branch prediction cache and its control logic. Speed was a problem with tag comparisons for those schemes that are associative. Accordingly we chose a simple direct-mapped, one-bit prediction scheme which can be implemented entirely with a single-ported RAM.*

7 Comparison of NLS Table and BTB Architectures

Figure 7 compares the performance of the NLS and BTB architectures using the branch execution penalty (BEP) for the programs in Table 1. Each graph compares the direct mapped and 4-way set associative 128 and 256 entry BTBs to the 1024 entry NLS table. It was shown in §6 the 1024 entry NLS-table and the 128 entry BTB have similar implementation costs using the RBE model, that the 256 entry BTB implementation cost is twice that of the 1024 entry NLS-table, and that the access time of the associative BTB's is 30 to 40% longer than similar sized direct mapped structures.

The differences in the BEP between the BTB and NLS architectures is attributable to differences in the number of misfetched branches. Remember that the BTB and NLS architectures are not used to predict the direction for conditional branches. Conditional branch prediction information is stored in a separate pattern history table (PHT) and the conditional branches are predicted using the PHT. The NLS and BTB architectures are used to eliminate the misfetch penalty associated with the extra cycle taken to determine the branch type and to compute the target address for the next instruction fetch. Once the branch type and target line are predicted, the next fetch line can be chosen from the return stack, precomputed fall-through line, or the predicted target line. Both the NLS and BTB architectures are used to predict the destination for indirect jumps. Table 1 shows that indirect jumps constitute 0-5% of the breaks in the programs we instrumented. In Figure 7, any difference in the mispredict penalty for a given program is attributed to the variation in the mispredict penalty for indirect jumps across the different architectures. The figure shows that the difference in mispredict penalty across the different architectures is only noticeable for `groff`, and even then the difference is insignificant.

Figure 7 shows that the BEP for the NLS architecture decreases as the cache size increases or the cache associativity increases.

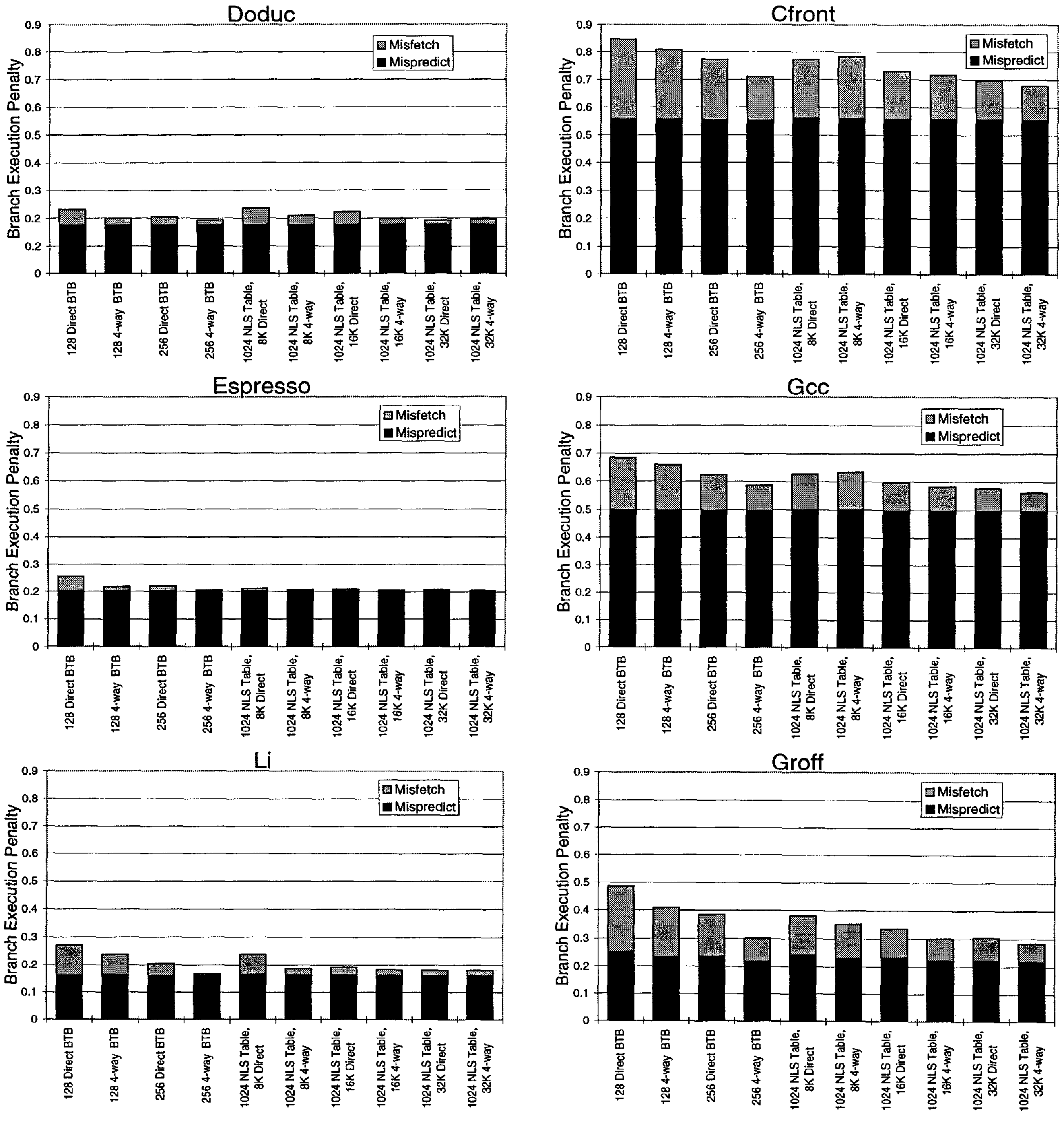

Figure 7: Performance comparison between the NLS and BTB architectures using branch execution penalty. Each value is broken into two parts. The top represents the fraction of the BEP due to the misfetch penalty and the lower part the fraction due to the mispredict penalty. The NLS results are given for an 8K, 16K and 32K instruction cache with direct mapped and four-way associativity. The BTB results are only shown once, since their results do not change for the different instruction cache configurations. The 1024 entry NLS table and the 128 entry BTB have equivalent implementation costs, and the cost for the 256 entry BTB is approximately twice that of the 1024 NLS-table.

Why? Recall that each NLS predictor indicates the cache line that should be fetched. The information associated with an NLS predictor is only useful if the actual destination of a branch is in the predicted location in the instruction cache. In smaller caches, the NLS predictors will often point to the proper cache line and set, but the desired instruction may not be present or may have been reloaded into a different set. With a NLS predictor, a branch destination that has been displaced from the instruction cache causes a misfetch penalty. When the current instruction is fully decoded, the misfetch is detected and the actual instruction is fetched. In this case, the misfetch penalty is associated with a cache miss. In contrast, the BTB always uses the full target address. This allows the BTB architecture to possibly locate the proper instruction in set associative caches, or to initiate an instruction cache miss a cycle earlier than the NLS architecture. If an associative cache is used, the NLS architecture would have to look in the other set on a misfetched branch, or do a full set lookup. When the cache miss rate is lowered, there is an increased probability that a cache line will still be resident when a NLS predictor is used. The BTB architecture will not benefit from the lower cache miss rate, and the there is no change in the BEP for varying cache configurations. Whole-program restructuring [7, 10, 14] is one technique that can be used to reduce the instruction cache miss rate at no additional architectural cost.

Why does the NLS architecture have significantly better BEP performance than the BTB for some programs, such as gcc, cfront and groff, but only slightly better or comparable performance for other programs, such as doduc and espresso? The program characteristics in Table 1 shows the programs that benefit most from the NLS architecture have more static branch sites then the programs that show little benefit. For example, in doduc, three individual branches constitute 50% of the branches encountered during program execution. One need only store those three branches to achieve 50% fetch accuracy. By comparison, gcc, cfront and groff have many more branches encountered during execution. The larger number of branches leads to capacity misses and conflicts in any prediction mechanism using a fixed-size resource. Because each NLS predictor in the NLS architecture is smaller than the comparable BTB entry, the NLS architecture has many more prediction entries using the same resources. Overall, the larger number of less-precise NLS predictors benefits program performance more than the fewer, more precise, BTB entries.

Figure 8 shows the average CPI for a single issue architecture with the different BTB and NLS configurations using 8KB, 16KB and 32KB direct-mapped and 4-way associative instruction caches. The Figure shows that the difference in performance is small among the configurations examined, with the 1024 entry NLS-table performing slightly better than the similar costing 128 entry BTBs. In examining the performance of individual programs, our results show that there is very little difference in performance between the NLS and BTB architecture for programs, such as espresso, that have a low cache miss rate. If the instruction cache miss rate is low, the probability of the instruction line indicated by the NLS predictor being in the instruction cache is very high. For programs that do not execute many branches, such as doduc (with 8.5% branches), there is also very little difference in the performance of the NLS and BTB architecture. If a program doesn't execute many branches, then neither the NLS nor the BTB architecture will suffer from capacity misses. For programs such as gcc, cfront and groff with many branches, the NLS architecture performs better than the BTB for most of the cache configurations examined, due to the large number of capacity misses in the BTB.

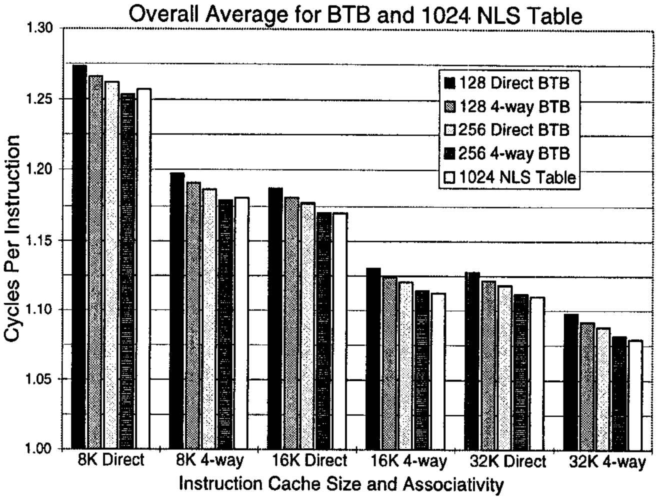

Figure 8: Performance comparison between the NLS and BTB architecture in cycles per instruction. The 1024 entry NLS-table and the 128 entry BTB have equivalent RBE costs, and the cost for the 256 entry BTB is approximately twice that of the 1024 NLS table.

A larger program address space poses several problems for BTB architectures, but is inconsequential for the NLS architecture. As the program address space increases, the tag size used to identify branches in the BTB must either increase, or the information stored in the BTB will become less precise. Likewise, the size of the destination address field must also increase, or the BTB architecture would store inaccurate target addresses. In our RBE calculations we assumed a 32-bit address space, so the target address stored in the BTB is 30 bits. If the address space was increased, the area needed by the BTB would also increase. By comparison, the NLS-table design does not use a tag nor does it store the full target address, so an increased address space has no effect on the size of the NLS-table. The size of an NLS predictor depends only on the number of lines in the cache and the number of instructions in the line. As the instruction cache size is increased the size of the NLS-table increases logarithmically. In contrast, an increase in cache size has no effect on the size of the BTB.

8 Conclusions

In this paper we have presented two alternative NLS architectures, the NLS-cache and NLS-table. Our results show that decoupling the NLS predictors from the instruction cache (NLS-table) performs better than Johnson's [4] approach of associating the NLS predictors with the cache line (NLS-cache). We found the NLS-cache is not a scalable design, because the number of NLS predictors increases linearly with the cache size. Our results also show that there is little benefit from increasing the NLS-table size from 1024 entries to 2048 entries.

The NLS-table is a tag-less, direct mapped buffer with better instruction fetch prediction than direct-mapped BTBs with similar costs. When comparing the performance of the NLS architecture to associative BTBs, one should keep in mind that the access time for an associative BTB is 30 to 40% longer than similar sized direct mapped structures. Our results show that the 1024 entry NLS-table performs better than the 128 entry BTB, with similar RBE costs. For a 256 entry BTB, the 1024 NLS-table had comparable performance for approximately half the RBE cost. The NLS-table can offer better performance than the BTB because the cost of an NLS entry is much less than a BTB entry, allowing the NLS-table to contain many more entries than BTB architectures with similar implementation costs. This allows the NLS-table to perform better than the BTB design especially for programs with many branches. For programs with fewer branches, the architectures have comparable performance.

The performance of the NLS architecture improves as the instruction cache miss rate is lowered, and its performance can be improved by using whole-program analysis, basic block reordering, and intelligent procedure layout. In contrast, improving the instruction cache miss rate has no effect on the branch performance of the BTB architecture. In this paper, we focused on the improvements offered by single-issue architectures and are currently investigating a number of design extensions for multi-issue architectures. Nothing in the design of the NLS architecture appears to be a problem for wide-issue architectures.

Acknowledgements

We'd like to thank Joel Emer, Alan Eustace, Keith Farkas, Dennis Lee and the anonymous reviewers for providing helpful comments. We'd also like to thank Amitabh Srivastava and Alan Eustace for developing ATOM, and Digital Equipment Corporation for an equipment grant. This work was partially supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies at University of Maryland, a DEC-WRL summer internship, in part by NSF grant No. ASC-9217394, and in part by ARPA contract ARMY DABT63-94-C-0029.

References

[1] Brian Bray and M.J. Flynn. Strategies for branch target buffers. In *24th Annual International Symposium and Workshop on Microprogramming*, pages 42–49. ACM, 1991.

[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] Peter Yan-Tek Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[4] Wen-mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th Annual International Symposium on Computer Architecture*, pages 242–251. ACM, 1989.

[5] Mike Johnson. *Superscalar Microprocessor Design*. Innovative Technology. Prentice-Hall. Inc., Englewood Cliffs, NJ, 1991.

[6] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *18th Annual International Symposium of Computer Architecture*, pages 34–42. ACM, May 1991.

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

[8] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1988.

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

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

[11] Johannes M. Mulder, Nhon T. Quach, and Michael J. Flynn. An area model for on-chip memories and its application. *IEEE Journal of Solid-State Circuits*, 26(2):98–105, February 1991.

[12] 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.

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

[14] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, June 1990.

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

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

[17] 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.

[18] Simon C. Steely and David J. Sager. Next line prediction apparatus for a pipelined computer system. US. Patent #5,283,873, Feb. 1994.

[19] Steven J. E. Wilton and Norman P. Jouppi. An enhanced access and cycle time model for on-chip caches. WRL Report 93/5, DEC Western Research Lab, 1993.

[20] 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.

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

[22] 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 on Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.