Spotlight - A Low Complexity Highly Accurate Profile-Based Branch Predictor

Santhosh Verma, Benjamin Maderazo and David M. Koppelman
Department of Electrical and Computer Engineering
Louisiana State University
{sverma3, bmader1}@lsu.edu, koppel@ece.lsu.edu

Abstract

In an effort to achieve the high prediction accuracy needed to attain high instruction throughputs, branch predictors proposed in the literature and used in real systems have become increasingly more complex and larger over time. This is not consistent with the anticipated trend of simpler and more numerous cores in future multi-core processors. We introduce the Spotlight Branch predictor, a novel profile-based predictor which is able to achieve high prediction accuracy despite its simple design. Spotlight achieves high accuracy because complex decisions in the prediction process are made during an OS managed, one-time profile run instead of using complex hardware. We show that Spotlight achieves higher accuracy than Gshare as well as highly accurate and implementable predictors such as YAGS and the Hybrid Bimodal-Gshare predictor. It achieves an average reduction in misprediction rate of 20% over Gshare, 11% over Elastic History Buffer, 14% over Yags and 10% over Hybrid for a hardware budget of 8 kB. Spotlight is also compared to two difficult to implement neural predictors, the Path-based Neural and the Hashed Perceptron. It outperforms the Path-based Neural predictor at all sizes and the Hashed Perceptron at smaller hardware budgets. These results demonstrate that a simple profile-based predictor can achieve many of the benefits of more complex predictors. We also show that a single cycle latency implementation of Spotlight can be achieved without sacrificing accuracy by using an upstream replacement scheme.

1. Introduction

Modern microprocessors require highly accurate branch prediction to achieve high performance. In order to deliver this higher accuracy, the branch predictors proposed in literature and those used in real systems have become larger and more complex over time. A new class of predictors that have been proposed in recent years provide very high accuracy but require complex hardware and are unable to deliver single cycle predictions without using techniques which may require even further complexity. For example, neural based predictors such as the Hashed Perceptron [26] and the Path-based Neural [10] require complex computations and many table lookups that result in a latency of several cycles per prediction if clock frequencies are to be preserved. These approaches requiring complex hardware are counter to the widely anticipated trend in future multi-cores of simpler and more numerous cores and the need for low-power hardware solutions. Further, a single cycle latency is difficult to achieve without using either a faster first-level primary predictor whose prediction may be reversed later by the main branch predictor or other complicated schemes like ahead pipelining. The performance of these predictors might suggest that complex computations are difficult to avoid if higher prediction accuracy is to be achieved. While these computations may not be avoidable, they may not have to be done at prediction time or even at run time.

Branch predictors such as the Gshare [15] predictor use a fixed length history of the most recent branch outcomes for every prediction that they make. However, a predictable branch may only need to use some segment (s) of the global branch history most correlated to that branch . The neural predictors can accomplish this by assigning weights to segments of the global history and using these weights to select the most appropriate segments. Another approach that has not been widely explored is to select global history segments based on profiling data.

In this paper, we propose a simple profiling based branch predictor which is highly accurate over a range of predictor sizes and demonstrate a version of the predictor that is capable of delivering single cycle predictions. Our predictor, the Spotlight Branch Predictor, is able to achieve very high prediction accuracy despite its simple design because much of the decision making involved in making a prediction is based on profiled information. The basic structure of our predictor is similar to the Gshare predictor as it uses global branch history to access a pattern history table of two-bit saturating counters. However, instead of using a fixed section and fixed length of global history, the predictor uses profiling information to spotlight a specific section of the global history register. The predictor can achieve low latency because it involves one or two table lookups combined with simple combinational logic. This is in contrast to the neural class of predictors which involve computing the sum of several non-positive numbers and multiple table lookups. We show that Spotlight achieves higher accuracy than Gshare, highly accurate implementation practical predictors such as the Gshare-Bimodal Hybrid and YAGS [4] as well as the Elastic History Buffer (EHB) [27], a previously proposed profile-based predictor. Despite its far simpler design, it is also able to outperform the Path-based Neural predictor at all simulated sizes and the Hashed Perceptron at smaller hardware budgets. These results demonstrate that a well designed profile-based predictor with simple hardware can achieve (or even surpass) much of the performance benefits of more complex hardware schemes.

2. Prior Work

Tarlescu et al. propose the Elastic History Buffer (EHB) [27] predictor where the length of history used to predict a branch can vary and is determined using profiled data. Unlike our work, they ideally assume that profiled information can be obtained directly from the branch instruction in time for a prediction. Juan et. al [7] propose Dynamic History Length fitting, where the length of the GHR used is varied dynamically until one that works best for the code in execution is found. The major difference between these two predictors and ours is that we can select any segment of the global history starting at any point, not just the most recent bits. Further, our predictor uses the Agree scheme [24] to minimize collisions in the PHT. Stark et al. [25] use profiling to select one of N hash functions for indexing a prediction table, where a hash function k (k = 1,2N) uses the target address of the k most recent branches to produce the hash. Like our scheme, they also propose storing profiled information in branch instructions, caching this data at runtime in a table and organizing this table so that entries store data for downstream branches (see section 3.4). However, they do not simulate such a configuration to analyze its impact on prediction accuracy but instead only simulate a basic version of their predictor. Our approach differs in our use of branch history instead of path history and our consideration of arbitrary GHR sections starting at any point in the global history. Another important difference is that computing several hash functions and selecting amongst them is more complex than our indexing scheme. Each of these hash functions requires several rotations and XOR's to be performed on different target addresses in each cycle.

Profiling has been used by others to aid in branch prediction. Chang et al. [2] use profiling to classify branches based on their dynamic taken rates. Profiling data is used to predict highly biased branches statically. This allows them to optimize the hardware predictor by reducing PHT conflicts between highly biased and mixed direction branches. Sprangle et. al [24] propose the Agree predictor, and investigate a version of the predictor which uses profiling information to set a bias bit in the branch instruction. PHT conflicts which are harmful are reduced by updating PHT entries based on whether a branch agrees or disagrees with its hint bit, instead of taken and not taken rates. These two techniques improve performance by profiling the taken rates of individual branches. Several approaches to branch prediction adapt history length, globally or per-branch. The global branch predictor uses a global history register (GHR) of the most recent branch outcomes to index a pattern history table (PHT) of two-bit saturating counters. The Gshare [15] predictor is similar to global but it XORs the GHR with the PC of the branch. Chang et al. [2] observe that highly biased branches can be more accurately predicted with short histories. In one of their implementations, they use a short history length for highly biased branches and a long history length for mixed direction branches. Ever et. al [3] explore global histories up to a size of 32, were up to three of the most important branches over this history length are selected using an oracle mechanism for use in a hypothetical predictor. Their results indicate that considering longer global histories up to 20 bits can improve performance. Thomas et al.[28] use run-time data flow information to identify correlated branches over a long global history. An Affector Register file (ARF) contains dataflow information for every architectural register. Each entry in the ARF is a bit vector, where a bit represents whether a branch in the global history affected the architectural register. For each branch, an Affector Branch Bitmap is created by combining the affector registers of the branchs source operands and this bitmap is combined with the GHR to form the predictor index. Their technique uses 64 bits of GHR which are hashed down to the required number of index bits by using a fold and XOR hash. Our predictor is capable of combining the benefits of many of these approaches, as it can use not only GHR segments of any length, but can also explore longer global histories and use older segments of the GHR.

Many predictors have been proposed in recent years which can provide very high accuracy but are both implementation complex and are unable to deliver single cycle predictions without using complex hardware. Seznec [18, 20] analyses the O-GEHL predictor. This predictor can explore very long history lengths up to 128 bits but needs to compute the sum of M items to make prediction. Recent work using perceptrons improve predictor accuracy by using longer global histories without exponentially increasing hardware size. The idea of neural branch prediction was originally introduced by Vinton and Iridon [29]. Jiménez et al. [9] propose a dynamic Perceptron predictor which exploits long branch histories by increasing hardware table size linearly with the history length. A hash function of the branch address is used to select a single perceptron from a table and an output is computed by using the dot product of the perceptron and global branch history. However, the global perceptron has to determine the hash of the branch address and the dot product of the perceptron and global branch history, a process that can take up to 4 cycles even in an efficient circuit level implementation of the predictor [13]. To counter this problem, Jiménez [10] proposes the Path-based Neural Predictor which computes a running sum along the path leading up to each branch. This makes it possible for the latency to be reduced from four to two cycles. However, the Path-based neural predictor is extremely complex in its use of anywhere between 14 and 34 weight tables to make a prediction. Tarjan et al. [26] improve on the linear scaling of previous perceptron predictors by assigning multiple branches to a single weight in their Hashed Perceptron scheme. This predictor is able to outperform the path based neural predictor, hence demonstrating that the one-to-one mapping between weights and branch history bits that is used in other Perceptron predictors is not necessary. The neural approaches to branch prediction obtain high accuracy but unlike Spotlight, they are both very complex and difficult to implement with a single cycle latency because of the need for relatively complex calculations before arriving at a prediction. The Alpha EV6 [19] solves the latency issue by using a simple quick predictor which is backed up by a larger and more accurate predictor. Another approach to solving the latency problem is to use ahead pipelining [11, 17, 21, 26], where the computation required to make a prediction in a particular cycle is started several cycles earlier using the information available in the earlier cycle. However, these schemes do not mitigate the complexity of the hardware itself but instead are themselves very complex. There can also be a significant drop in accuracy when the "ideal" one-cycle latency versions of these predictors are compared to the pipelined versions. Furthermore, Tarjan et. al [26] point out that pipelined versions of branch predictors will need to checkpoint intermediate predictor data to account for the recovery process after a branch misprediction. The amount of data that needs to be checkpointed is dependent both on the complexity of the predictor and on the number of pipelined stages that the predictor uses to make a prediction. In [26], Tarjan et. al note that Perceptron-based predictors need to checkpoint a significant amount of data. While our predictor does use pipelining in its single-cycle version, the misprediction overhead will be much lower due to the lower complexity and the use of just two pipelined stages. Since Spotlight's pipeline is two stages checkpointing is not needed for uninterrupted predictions after a recovery, all it needs is the address of the mispredicted branch one cycle before prediction is to resume.

3. The Spotlight Branch Predictor

In this section, we provide an overview of the basic idea behind the Spotlight predictor. We then explain how profiling is used to determine the GHR bits used to make a prediction. We continue with a discussion of how this profiling information is stored so that it can be accessible to the predictor. Finally, we demonstrate a version of this predictor which is capable of delivering predictions with a single cycle latency.

3.1 Overview of Spotlight

The motivation for Spotlight is that the segment of the GHR which best correlates with the predicted branch can range widely both in terms of its distance from the predicted branch (its starting point) and the length of the segment from this starting point. Figure 1 shows the basic structure of the Spotlight predictor along with an example prediction. The predictor uses an n-bit global history register

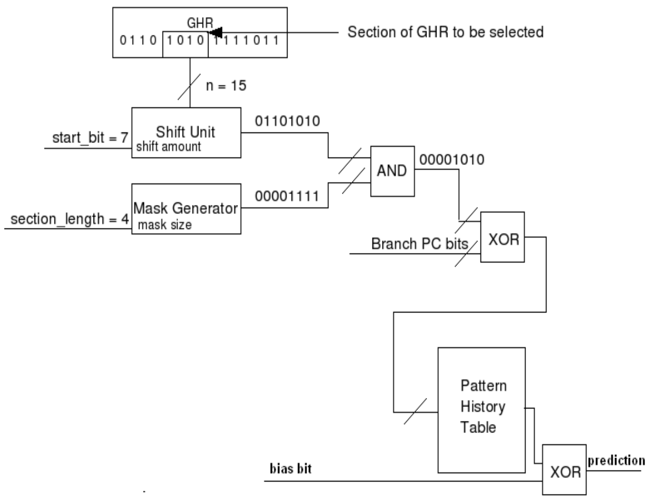


Fig. 1. Spotlight Direct

as its first level of history. For each prediction that needs to be made, a start_bit and a section_length is provided to the predictor. The start_bit represents the starting point of the GHR section that will be used for a prediction, and the section_length represents the length of this section. The start_bit and section_length are used to perform a combination of shift and mask logical operations on the global history so that a particular section of the GHR is selected (spotlighted). The example in the figure shows how a GHR section starting at the 7th bit with a length of 4 bits is selected. The selected segment of the GHR is XOR'd with the branch PC and this result is used to index a pattern history table (PHT) of two-bit saturating counters. Hence, the Spotlight predictor can use GHR bits starting at any point between 0 and n-1 with any length between 0 and the log_2 size of the PHT. Instead of using the PHT counters in the traditional Gshare approach, we use the Agree scheme[24]. A bias is stored with each branch and the PHT counter for that branch indicates whether its outcome is expected to agree or disagree with its bias. The bias represents whether a branch is majority taken or not-taken and is determined using profiling. This scheme reduces harmful PHT collisions, especially at smaller PHT sizes. Since Spotlight already uses profiled information to make predictions, the cost of including the Agree scheme is minimal while providing some performance benefit to the predictor.

3.2 The Profiling System

In the previous section, we see that the predictor requires a start bit and a section length to make a prediction. This information is determined using profiling. Since the predictor can select any part and any length of the GHR and the GHR can be larger than the PHT address, our profiling algorithm has to explore the entire subset of possible GHR segments. The value of starting_bit ranges from 0 to n-1, where n is the maximum number of GHR bits explored. The value of section_length ranges from 0 to size_lg, where size_lg is the log_2 of the PHT size of the simulated predictor. All contiguous GHR segments between these two ranges are considered. While the reader may be concerned about the exhaustive nature of this method, profiling is done in functional simulation mode and is hence very fast. Heuristic methods may be possible, such as testing coarser regions in an initial phase and finer regions in a second phase. We did not see the need for such methods because the speed of profiling was not an issue, the longest profiling simulation for the training inputs took about 10 hours with most benchmarks finishing much faster.

During profiling, a single PHT is instantiated for each combination of start_bit and section_length. The log size of an individual PHT is determined by the function min[min_pc_bits, section_length]. The PHT's are indexed using the XOR of a particular GHR segment and the PC of the branch being predicted. In this way, there is one PHT for each GHR segment and this PHT tracks the performance of that segment throughout the profiling benchmark. A minimum number of PC bits (min_pc_bits) is used for the PHTs to reduce conflicts between branches when the GHR segment for that PHT is very short. When a branch instruction is executed during the profile run, every PHT provides a prediction for that branch. The prediction of each PHT is compared to the actual outcome of the branch and prediction accuracy data is collected for the branch over all PHT's (and hence over all possible GHR segments).

At the end of the profiling run, we have the prediction accuracy over all PHT's for each static branch that was executed during profiling. Since each PHT represents a specific combination of start_bit and section_length, we can select the start_bit and section_length for a branch based on the most accurate PHT for that branch during the profile step.

In addition to looking at different combinations of start_bit and section_length, the profiler also considers the overall bias of a branch. If a branch is more than 50% taken, it is biased taken while a branch that is more than 50% not-taken is biased not-taken. If this bias percentage is greater than the accuracy of the most accurate PHT, we assume that this branch is best predicted using a bimodal type of predictor. While our proposed predictor does not have a separate bimodal table, using a start_bit of 0 and a section_length of 0 is a good approximation. This addition to the predictor provides minimal performance benefit but is included since there is no additional hardware cost.

3.3 Storing Profiled Information

The Spotlight profiling and annotation system is responsible for performing the profile analysis and the annotation of binaries for delivery at run-time to the Spotlight predictor. A goal is that it be transparent to both users and program developers and that it not interfere with code portability. This will be achieved using OS support and a technique we will call opcode borrowing as described below. Profile analysis would be performed by code written by the CPU manufacturer, or others familiar with the predictor; if necessary obfuscated, as with GPU drivers. A fully automatic profiling system would be fed by traces containing branch outcomes sampled from normally running binaries. By limiting trace size and relying on hardware support this sampling can be made unobtrusive. The OS would invoke a profile analysis program when a sufficient number of traces were collected from a large enough number of runs of the binary. The OS might profile a binary separately for each user for both privacy and performance reasons. By running at a low priority, profile analysis would be unobtrusive.

Spotlight needs to annotate a program binary with profile-collected branch information for delivery to the Spotlight predictor. Spotlight's approach is to place this information in special versions of existing instructions using a technique called opcode borrowing. The idea is to borrow unimplemented or rarely used opcodes to create special versions of common instructions that have space for Spotlight data. For example, the SPARC V9 [31] branch instruction has a 22-bit displacement field, the format encodes five types of branches, a non-branch, and two unused instructions. Those unused instructions can be borrowed for branch instructions that have, say, a 12-bit displacement and 10 bits of spotlight storage. It is not necessary that a branch alone carry information (or information for a successor), it can be spread out over any subset of instructions near the branch. With many add-by-1 and zero-offset loads, there is lots of unused instruction space. There still will be cases where there is not enough space, then the affected branch will use the default predictor. Branch prediction hints are available in many instruction sets, including SPARC V9, but these do not provide enough storage for Spotlight. Stark, et al [25] suggest augmenting an ISA for their variable-path length predictor. Such an augmentation would have to trade off hint storage space with preservation of unused opcode space. Since rarely used opcodes could be borrowed, opcode borrowing provides greater flexibility than ordinary implementation-dependent instructions.

OS support is needed for managing the annotated binaries in a way transparent to users. For portability reasons the OS would not replace the original binary with the annotated binary. Instead it would place the annotated binary in a shadow area and use it whenever the original binary was to be executed. A change to the original binary would invalidate the annotated one.

3.4 Single Cycle Predictions with Spotlight

In the previous section, we suggest that profiled information can be stored with branch instructions using either implementation independent or implementation specific extensions to the ISA. However, it is not possible to decode instructions in the same cycle as they are fetched in high clock frequency modern processors. Since the processor must predict a branch in the same clock cycle that it is fetched, a single cycle prediction cannot be made if we rely on obtaining information from the decoded branch instruction. If we are to obtain profiled information for a branch in time, we will have to access the data from a hardware structure instead of the decoded branch. We consider this limitation and modify Spotlight so that it can make single cycle predictions. A version of the modified predictor, which we call Spotlight BIT-Upstream (SingleTag), is shown in Figure 2. In this scheme, profiled

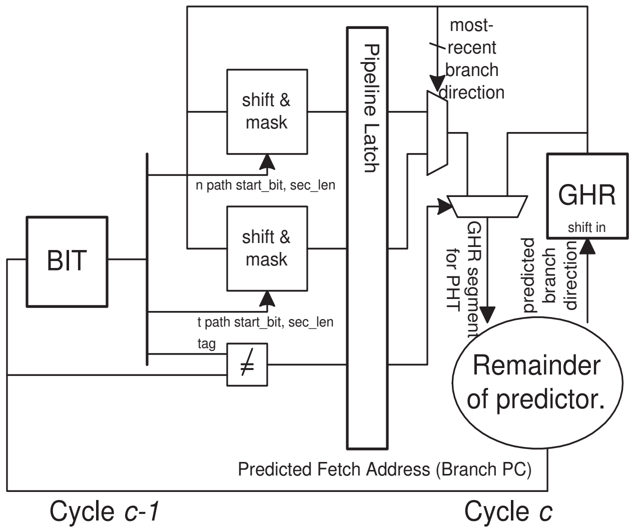

BIT
shift & mask
shift & mask
n path start_bit, sec_len
t path start_bit, sec_len
tag
≠
Pipeline Latch
most-recent branch direction
GHR segment for PHT
predicted branch direction
GHR
shift in
Remainder of predictor.
Predicted Fetch Address (Branch PC)
Cycle c-1
Cycle c

Fig. 2. Spotlight With Single Cycle Latency

information is first obtained from a decoded instruction and then stored in the Branch Information Table (BIT). A branch in the BIT will not store profiled information for its own use, it will instead store information about the next branch in the control flow of the program. A branch prediction is derived as follows. A branch that needs to be predicted in cycle *c* will perform a lookup in the BIT in cycle *c-1* using the lower bits of the PC of the branch being predicted in this cycle. Each BIT location can store information for two branches. If the entry is occupied by a conditional branch, one of these branches is on the taken path of the BIT entry occupant and the other is on the not-taken path. We access information for both of these branches and apply the shift and mask operations described in section 3.1(with minor modifications) to two separate copies of the GHR. As can be seen in Figure 2, one GHR represents the taken path following the branch in cycle *c-1* and the other represents the not-taken path. In order to reduce aliasing, each BIT entry is associated with a tag. If there is a tag miss, the default GHR is selected instead of the custom GHR for both paths. The default GHR has a starting_bit value of 0 and a section_length that is log_2 of the PHT size. The prediction for cycle *c-1* is available at the beginning of cycle *c* and can hence be used to select between the two copies of the GHR available in this cycle. One of these copies is selected and used to index the PHT. If PHT access in cycle *c* is on the critical path, the PHT can be indexed earlier using both copies of the GHR and one of the two predictions obtained can be selected after the PHT access.

This scheme is described as "BIT-Upstream (SingleTag)" because only one tag, which is based on the PC of the branch in cycle *c-1*, is stored in the BIT. An alternative would be to perform tag checking in cycle c using the PC of the branch in that cycle. This is likely to increase prediction accuracy but also increases the workload in cycle *c*, hence making it more difficult to achieve single cycle access. We call this second scheme Spotlight BIT-Upstream (DoubleTag) since the BIT would need to store two tags per entry, one for the taken path and one for the not-taken path from that entry.

The BIT is updated with the profile information for a branch after the branch is decoded. We may therefore expect a miss in the BIT the first time that a static branch is encountered, but subsequent accesses are expected to produce hits.

4. Experimental Results

4.1 Experimental Framework

We use the LSU RSIM simulator for the experiments done in this paper. This is an extensively modified version of RSIM [16] which does detailed simulation of a dynamically scheduled superscalar processor and memory system. The simulator implements a subset of the SPARC V9 ISA [31]. The results reported in this paper are obtained using 10 benchmarks from the SPEC 2000 integer benchmark suite. The benchmarks eon and twolf are not simulated. We use the full training inputs provided by SPEC to perform profiling. Full training inputs can be used because profiling is done in functional simulation mode and is hence very fast. The data collection runs are done using SPEC reference inputs (for most benchmarks) in the detailed timing simulation mode of RSIM. We use the SPEC 2000 Reduced input sets for Parser and Crafty since their reference inputs are very large. The configuration we used allowed a maximum of six instructions and one branch to be fetched per-cycle. We use a tool based on the Simpoint multiple simulation point methodology [23] to generate a set of weighted samples that are selected over the entire duration of each benchmark. Each sample has 100 million instructions and a 10 million instruction warm up period is used prior to each sample.

4.2 Experimental Evaluation

We evaluate the accuracy of Spotlight-Direct, which can access profiled data directly from the branch instruction and the modified versions discussed in section 3.4 which use the BIT to store profiled data. We compare their accuracy to Gshare, the profile-based EHB, as well as two highly accurate predictors with simple design, the YAGS predictor and a Gshare-Bimodal Hybrid predictor. We also evaluate the Hashed Perceptron and the Path-based Neural, two very high accuracy but implementation complex predictors which computes the sum of several non-positive numbers that are looked up from several tables. The Hashed Perceptron does 5 table lookups and computes their sum to make a prediction, while the Path-based neural can do between 14 and 34 table lookups per prediction.

The experiments are done over a wide range of total predictor storage sizes. For the Spotlight-BIT configurations, the size of the BIT is included in the total size of the predictor. Each entry stores a 4 or 6 bit tag as well as the start_bit and section_length information for two branches. We use a GHR history length of 32 during the profiling step for the Spotlight predictor over all sizes, i.e., Spotlight can select subsets of a 32 bit GHR. For the relevant competing predictors, we explore their design space to find the best configuration at a given size. The main configuration settings used for the simulated predictors is shown in Table 1.

4.3 Accuracy of Various Spotlight Configurations

In section 3.4, we demonstrated a version of Spotlight that can provide a prediction in the same cycle that a branch is fetched. In section 2, we discussed prior work that involves predictors which are not able to provide predictions in the same cycle as fetch unless complicated techniques such as ahead pipelining or secondary overriding predictors are used. In this section, we look at how Spotlight performs when the pipelined single-cycle latency schemes (BIT-Upstream) described in section 3.4 are compared to a version which assumes that profiled information is available directly from the branch (Direct) and another version which uses a BIT but assumes that all the tasks performed by the predictor can be performed in one cycle (BITDirect). Specifically, the BIT in BITDirect stores profiled information for the branch in the current cycle instead of the possible branches in the next cycle. Since Spotlight uses the Agree scheme for its PHT update, we also evaluate a version of Spotlight called Spotlight-Direct (No Agree) to evaluate the contribution of the Agree scheme to Spotlight's performance benefits. The PHT in this predictor is updated using the traditional Gshare method. From Figure 3, we can see that the two Spotlight BIT-Upstream configurations are close in accuracy to Spotlight-BITBasic starting at medium sizes (this corresponds to a PHT size of 4096 entries). The performance drop compared to Spotlight-Direct is larger but becomes less significant starting at a size of 5 kB. At the highest size, the average misprediction rate of the upstream configurations are almost identical to Spotlight-BITBasic. Therefore, our scheme is able to achieve single cycle latency without sacrificing much accuracy at medium and larger sizes. We also see that the performance difference between the SingleTag and DoubleTag schemes is minor. As one might expect, the gap between Spotlight-Direct and Spotlight-Direct (No Agree) declines as the PHT size increases. The difference is more significant at smaller sizes but becomes minor starting at a size of 1 kB. These results indicate that while Spotlight benefits due to Agree, the benefit becomes smaller with increasing PHT size and is not responsible for most of Spotlight's performance benefits.

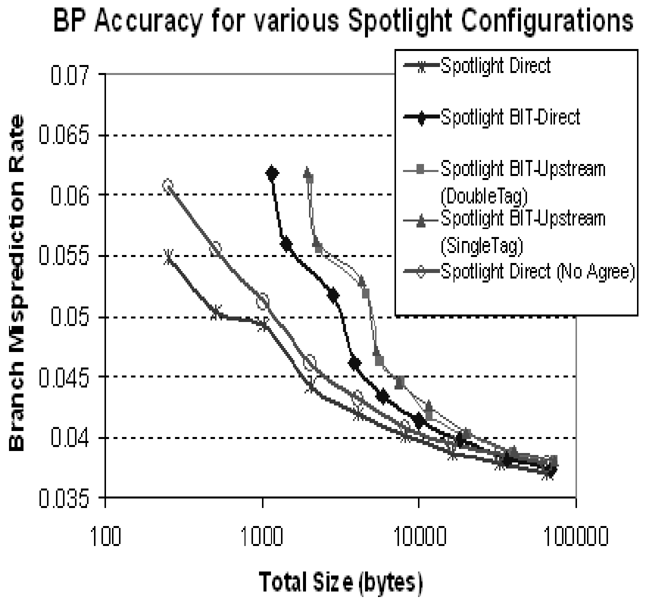


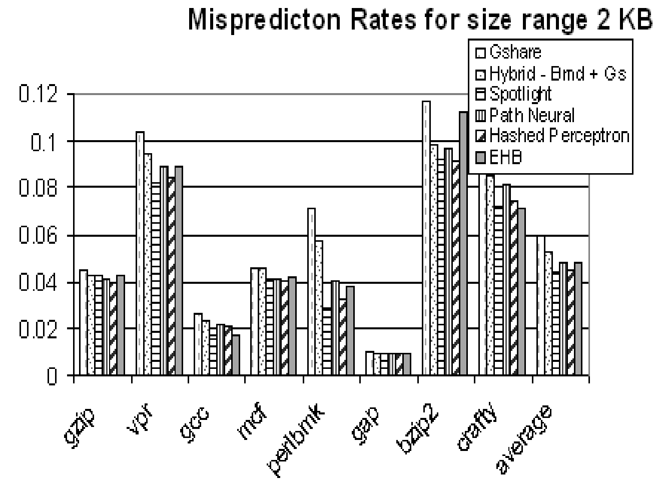
Fig. 3. Various Spotlight Configurations

4.4 Misprediction Rates

Figure 4 shows the average misprediction rates for the various predictors over the 10 SPEC integer benchmarks. It can be seen that Spotlight-Direct outperforms Gshare, EHB and Hybrid for all sizes simulated. This predictor achieves an average reduction in misprediction rate of about 20% over Gshare, 14% over Yags (Yags is not shown on the chart as it slightly under performs Hybrid at most sizes), 11% over EHB and 10% over Hybrid at a size of 8 kB. The Spotlight BIT-Upstream predictor is able to outperform Gshare as well as the higher accuracy Yags, EHB and Gshare-Bimodal Hybrid branch predictors at medium and higher size levels. Spotlight-Direct outperforms the Path-based Neural predictor, which is remarkable considering the far greater complexity of that predictor (it uses between 14 and 34 tables). It does not outperform the Hashed Perceptron predictor at medium and larger sizes but like the Path-based Neural, this predictor is more complex in its design and is also one of the most accurate predictors that have been proposed in literature. The fact that Spotlight-Direct is very competitive with the Hashed Perceptron at all sizes suggests that it could be used in a system with an overriding prediction mechanism similar to the Alpha EV6 [19]. In such a system, a first-level predictor would need to make a primary prediction which may be overridden by a Spotlight predictor which waits for profiled information from the decoded branch instruction. We must note that the version of the Hashed Perceptron and Path-based Neural that we simulated are ideal versions

TABLE I
PREDICTOR CONFIGURATIONS

Approx. Sz	Spotlight BIT		Yags			Hybrid		Hashed Perceptron			Path Neural	
	BIT Lg	PHT Lg	Tag Ln	PHT Lg	Bimdl Lg	PHT Lg	Bimdl Lg	Wt. Tbl Lg	Num. Tbls	Hist. Ln	Wt. Tbl Lg	Hist. Ln
2 kB	10	12	6	9	12	12	11	10	5	28	7	20
6kB	10	14	6	11	12	14	12	11	5	44	8	26
10kB	10	15	6	13	12	15	12	12	5	48	9	28
40kB	11	17	6	14	14	17	14	14	5	56	10	34



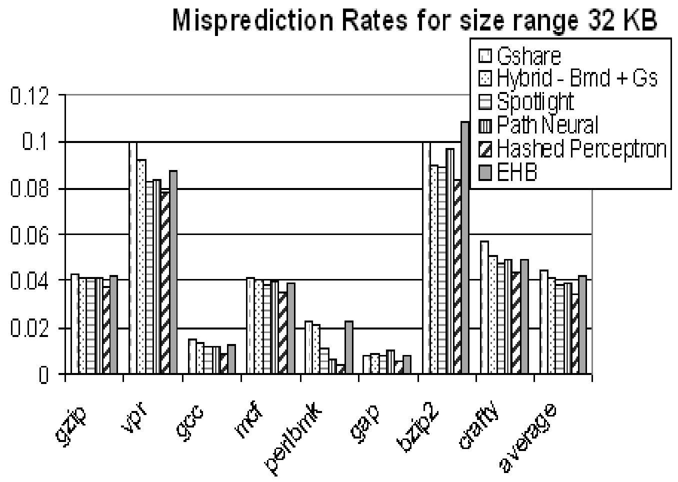


Fig. 5. Misprediction Rates for Various Branch Predictors

which assume that all computations can be done in the same cycle (we assume that all table lookups and the sum of non-positive numbers are done in that cycle). The version of EHB is similar to Spotlight-Direct as it assumes that data can be obtained directly from the branch instruction in time. Per benchmark data is shown in Figure 5 for most of these predictors at approximate sizes of 2 kB and 32 kB. We can see that the performance benefit provided by Spotlight occurs over a broad range of benchmarks.

4.5 Impact of Training Input Data on Spotlight's Accuracy

As mentioned above, we see in Figure 5 that Spotlight does well over a broad range of benchmarks. This result may seem surprising at first given that Spotlight relies on profiled information and either an inaccurate poorly designed training input or varying input data could impact its accuracy. It is beyond the scope of this paper to show that profiling is a robust and accurate technique. However, we note that profiling is used successfully by nearly all general-purpose compilers, such as gcc, and the evaluation and generation of accurate profiles has been widely studied in literature. For example, Wall [30] notes in early work that real profiles generated from different runs are very effective and are often nearly as good as perfect profiles. Hsu et. al [14] claim that profile data from training inputs can be used to reliably predict branch directions. We expect Spotlight to be even less sensitive to training data since it relies on using a set of correlated branches, and does not depend on the actual branch outcomes or input data to make predictions.

To evaluate how sensitive Spotlight is to training data, we simulate Spotlight using three different sets of training inputs for a few benchmarks. Each benchmark is evaluated using a perfect input, the training input and an arbitrary reference input. For the arbitrary reference input, an arbitrarily selected input is used for training and a different input is used for the reference run (for example, we train with gcc-ref-integrate and evaluate with gcc-ref-expr). For the perfect input case, the same input that is used for training is also used for the reference run. The results are shown in figure 6 for three different predictor sizes. We would expect the perfect input to be most accurate, followed by the training input and the arbitrary-ref input. The accuracy of Spotlight does not vary significantly for gcc and gzip across the three training inputs (the biggest difference in accuracy is about 0.15% for gcc). The results for bzip2 are interesting. The training input under performs even the arbitrary reference input while the arbitrary reference input is almost as accurate as the perfect input. This would seem to indicate that the training input for bzip2 is poorly designed. Fang et. al [5] observe that the SPEC 2000 bzip2 training input does not touch a lot of the code that reference inputs touch and this may be a factor. However, despite this limitation, Spotlight does very well for bzip2 compared to other branch predictors (see figure 5). This result suggests that Spotlight can be an effective branch predictor even for benchmarks which do not have

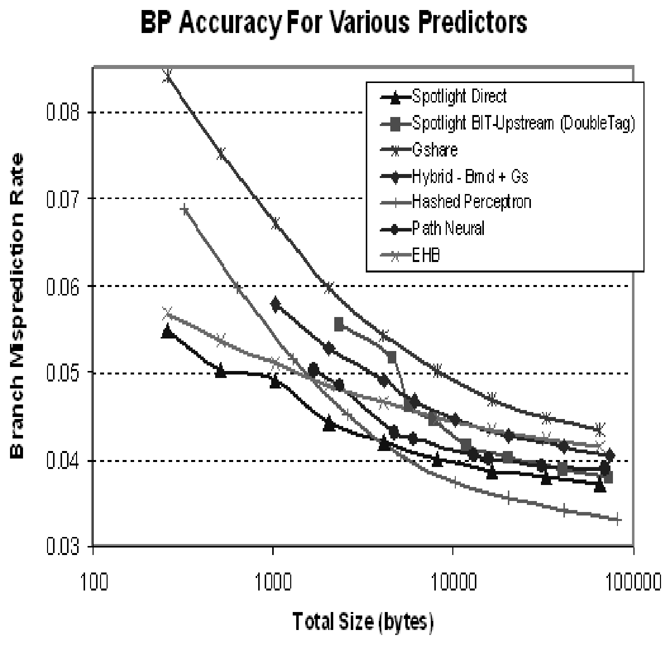


Fig. 4. Mispediction Rates for Various Branch Predictors

very accurate training data. We can gain an understanding

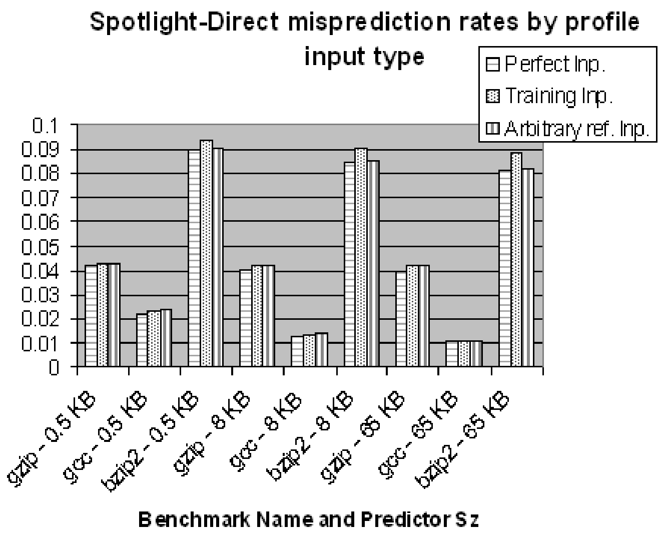


Fig. 6. Impact of training input data on Spotlight

of why Spotlight does well by looking at the chart in Figure 7 below. The data shows how frequently various sections of the GHR are used on average over the benchmarks. In the key listed on the right hand side, the first value describes the starting_bit of GHR segments and the second value describes the section_length. For example, the segments that comprise of the < 10 - > 7 bar are those GHR configurations which have starting_bit less than 10 and section_length greater than 7. The segment 0-0 is one that uses no GHR bits and is essentially a bimodal configuration (the difference between the two is PHT collisions in Spotlight). The segment 0-

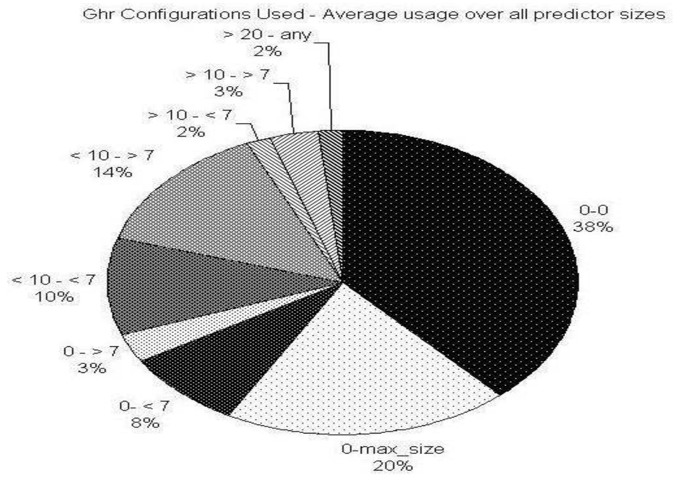


Fig. 7. Use of Various GHR configurations

max_size is the configuration that would be used for all predictions in a Gshare predictor and represents the default configuration mentioned in section 3.4.

The results demonstrate many of the reasons why Spotlight does well. The most common configuration used is the 0-0 bimodal configuration and this configuration along with the default GHR configuration of 0-max_size is used in approximately 58% of all dynamic branch predictions. These results suggest that Spotlight partly works like the Gshare-Bimodal Hybrid predictor even though unlike the actual Hybrid predictor, it does not have separate bimodal and chooser tables. On the other hand, one benefit of separate tables is that the bimodal component of the Hybrid predictor provides some accuracy while the Gshare component warms up. We saw in the previous section that Spotlight does better than the Hybrid predictor, and this along with the fact that other custom GHR configurations represent the remaining 42% of dynamic predictions suggest that Spotlight also benefits from these custom GHR configurations. We can also see that this remaining 42% comprises a very diverse set of GHR segments. They include significant amounts of short segments (section length < 7) starting at 0, long segments starting at 0, short segments starting at other values and so on.

5. Conclusions

In this paper, we proposed a profile-based branch predictor which delivers very high prediction accuracy despite its relatively simple hardware design. Spotlight achieves high accuracy because much of the decision making involved in making a prediction is done based on profiled data instead of using complex hardware. By using this approach, it is able to get performance comparable to some implementation complex schemes such as the Path-based Neural and the Hashed Perceptron. These predictors require several table lookups and compute the sum of several non-positive numbers before arriving at a prediction, an approach which is inconsistent with the anticipated trend of more numerous and simpler cores in future multi-core processors and the desire for low-power hardware solutions. The high accuracy of our predictor suggests that well designed profile-based predictors may have the potential to simplify branch prediction hardware by making reliable decisions based on profiled information. In future work, profiling may also be used successfully to augment current branch predictors as well as in other architecture areas.

There have been several profile-based predictors proposed in literature. However, many of these schemes assumed that profiled data can be obtained in time directly from the branch instruction. This is clearly an impractical assumption for a modern high clock frequency processor. While Spotlight can be used in an overriding prediction scheme like the one used in the Alpha EV6 [19], we show how a practical single-cycle implementation of a profile based predictor is possible by using a Branch Information Table to store profiled data instead of relying on obtaining it from the instruction. Our results in this paper demonstrate a BIT scheme can be used without sacrificing a significant amount of accuracy at medium and large predictor sizes. They further show that the performance loss is minimal even if it is necessary to decouple the BIT access from some of the other parts of the prediction process by storing information in a previous branch instead of the current branch.

REFERENCES

[1] David I. August, Daniel A. Connors, John C. Gyllenhaal, and Wen mei W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. *In Proceedings of the Third International Symposium on High-Performance Computer Architecture*, February 1997.

[2] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, Yale N. Patt. Branch Classification: a new mechanism for improving branch predictor performance. *In Proceedings of the 27th Annual ACM/IEEE Int'l Symposium on Microarchitecture*, pages 22-31, 1994.

[3] Marius Evers, Sanjay J. Patel, Robert S. Chappell, Yale N. Patt. An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work. *In Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52-61, June 1998.

[4] A.N. Eden and T.N. Mudge. The YAGS branch prediction scheme. *In Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, November 1998.

[5] Changpeng Fang, Steve Carr, Soner Onder, Zhenlin Wang. Feedback-directed memory disambiguation through store distance analysis. *In Proceedings of the 20th annual international conference on Supercomputing*, June 28-July 01, 2006.

[6] Intel Itanium Architecture Software Developers Manual. *Volume 3: Instruction Set Reference, revision 2.1*, October 2002.

[7] T. Juan, S. Sanjeevan, J. Navarro. Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch Prediction. *In Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[8] Daniel A. Jiménez, Calvin Lin. Branch Path Re-Aliaising. *In Proceedings of the 4th Workshop on Feedback Directed and Dynamic Optimization (FDDO-4)*, December, 2001 (co-located with MICRO 34).

[9] Daniel A. Jiménez, Calvin Lin. Dynamic Branch Prediction with Perceptrons. *In Proceedings of the 7th Int'l Symposium on High Performance Computer Architecture*, pages 197–206, January 2001.

[10] Daniel A. Jiménez. Improved Latency and Accuracy for Neural Branch Prediction. *In Proceedings of the ACM Transactions on Computer Systems*, pages 197-218, May 2005.

[11] Daniel Jiménez. Fast path-based neural branch prediction. *In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 243-252, December 2003.

[12] Daniel A. Jiménez, Heather L. Hanson, and Calvin Lin. Boolean formula-based branch prediction for future technologies. *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[13] Daniel A. Jiménez, Calvin Lin. Neural methods for Dynamic Branch Prediction. *ACM Transactions on Computer Systems*, 20(4):369-397, November 2002.

[14] Wei Chung Hsu , Howard Chen , Pen Chung Yew , Dong-Yuan Chen, On the Predictability of Program Behavior Using Different Input Data Sets. *In Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, p.45, February 03-03, 2002

[15] Scott McFarling. Combining branch predictors. Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

[16] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM Reference Manual Version 1.0. Rice University Dept. of Electrical and Computer Engineering, Technical Report 9705, August 1997.

[17] A. Seznec and A. Fraboulet. Effective ahead pipelining of the instruction address generator. *In Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 241-252, June 2003.

[18] Andrè Seznec. Analysis of the O-GEometric History Length Branch Predictor. *In Proceedings of the 32nd Annual International Symposium on Computer Architecture, pages 394-405*, June 2005.

[19] Andrè Seznec, Stephen Felix, Venkata Krishnan, Yiannakis Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. *In Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 296–306, Anchorage, AK, May 2002.

[20] A. Seznec. The O-gehl Branch Predictor. *In the first JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[21] A. Seznec. Revisiting the perceptron predictor. Technical Report PI-1620, IRISA Report, May 2004.

[22] R.L. Sites. Alpha Architecture Reference Manual. *Digital Press, Burlington, MA*, 1992.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[24] Eric Sprangle, Robert S. Chappell, Mitch Alsup, Yale N. Patt. The Agree Predictor: A mechanism for reducing branch history interference. *In Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 284-291, 1997.

[25] Jared Stark, Marius Evers, Yale N. Patt. Variable Length Path Branch Prediction. *In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[26] David Tarjan, Kevin Skadron. Merging Path and Gshare Indexing in Perceptron Branch Prediction. *In Proceedings of the ACM Transactions on Architecture and Code Optimization*, pages 280-300, September 2005.

[27] Maria-Dana Tarlescu, Kevin B. Theobald, Guang R. Gao. Elastic History Buffer: A Low-Cost Method to Improve Branch Prediction Accuracy. *In Proceedings of the 1997 International Conference on Computer Design (ICCD)*, October 1997.

[28] Renju Thomas, Manoj Franklin, Chris Wilkerson, Jared Stark. Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History. *In Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 314-323, June 2003.

[29] L. Vintan and M. Iridon. Towards a High Performance Neural Branch Predictor. *In Proceedings of the 9th International Joint Conference on Neural Networks*, pages 868873, July 1999.

[30] David W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. *WRL Technical Note TN-18*, December 1990.

[31] David L. Weaver and Tom Germond. The SPARC Architecture Manual, Version 9. *Sparc International and PTR Prentice Hall, Englewood Cliffs, NJ*, 1994.

[32] Joe Wetzel, Ed Silha, Cathy May and Brad Frey. PowerPC User Instruction Set Architecture. *Book 1, version 2.01*, September 2003.