# Toward Virtualizing Branch Direction Prediction

Maryam Sadooghi-Alvandi, Kaveh Aasaraai, Andreas Moshovos
Department of Electrical and Computer Engineering, University of Toronto
{alvandim, aasaraai, moshovos}@eecg.utoronto.ca

***Abstract*—This work introduces a new branch predictor design that increases the perceived predictor capacity without increasing its delay by using a large virtual second-level table allocated in the second-level caches. Virtualization is applied to a state-of-the-art multi-table branch predictor. We evaluate the design using instruction count as proxy for timing on a set of commercial workloads. For a predictor whose size is determined by access delay constraints, accuracy can be improved by 8.7%. Alternatively, the design can be used to achieve the same accuracy as a non-virtualized design while using 25% less dedicated storage.**

## I. Introduction

Modern processors use branch prediction to predict branch outcomes, allowing them to fetch ahead in the instruction stream, increasing concurrency and performance. Predicting a branch outcome generally consists of using the branch address and some additional information to look up predictions in one or more tables of counters. This method achieves good accuracy, but since the tables have a limited number of entries, some branches are inadvertently forced to use the same entries. This results in a phenomenon known as *aliasing*. Increasing the branch predictor table size(s) is a simple way of reducing aliasing. However, larger tables come at the cost of a larger area budget and a longer access latency. It has been shown that increasing delay to improve prediction accuracy is almost never a good tradeoff [1].

*Predictor Virtualization* (PV) [2] proposes a methodology for increasing the effective capacity of predictors without necessarily increasing their latency. PV uses the memory hierarchy to transparently store predictor metadata in a *virtual predictor table*, while using the on-chip resources to record active predictor entries. Whenever the dedicated storage is not enough, data is spilled to and fetched from the virtual table.

Applying PV to branch outcome predictors could be used to provide the best balance between the predictor's accuracy, area overhead, and latency. Augmenting the dedicated branch predictor tables with a large virtual second-level table is beneficial, not only because the second-level table does not come at the cost of larger dedicated storage, but also because, if orchestrated properly, the delay of the predictor can remain constant at the latency of accessing the smaller dedicated tables.

The key challenge in virtualizing a predictor is tolerating the access latency of the virtual table. Some predictors naturally benefit from virtualization as they can inherently tolerate higher latencies [2]. However, a straightforward application of PV techniques is not feasible for branch predictors, since branch predictors are delay sensitive. Other delay-sensitive predictors, such as Branch Target Buffers (BTBs), have been successfully virtualized by exploiting the inherent locality in their access streams in order to *prefetch* metadata [3]. BTBs use branch addresses for their accesses, which inherit the spatial and temporal locality present in program instructions. Branch predictor access streams do not exhibit such locality, since they use a hash of the branch address and other data (such as branch history) to index their tables. To take advantage of PV, the design of branch predictors has to be revisited with virtualization in mind.

This paper explores virtualization techniques as applied to branch direction predictors. As most branch predictors were designed with area constraints in mind, it is useful to rethink branch predictor design in the absence of area constraints and in the context of virtualization; Virtualization makes area constraints a secondary factor by providing the predictor with a large virtual space. Virtualization is applied to a state-of-the-art multi-table branch predictor, the TAgged GEometric History Length predictor (TAGE) [4]. Our design augments a single table in the predictor with a large virtual second-level table.

We propose a paging scheme for the design of such a heirarchical predictor, in which the virtual table is broken into a collection of sub-tables, each dedicated to a set of instruction memory blocks. This allows us to exploit some of the locality that is natural in the instruction fetch stream. However, contention is increased as branches are now limited to using a single sub-table based on where they appear in the instruction memory space. Experimental results show that the paging scheme can improve accuracy by 8.7%, for a predictor whose size is determined by access delay constraints. Alternatively, the design can be used to achieve the same accuracy as a non-virtualized design while using 25% less dedicated storage. Paging introduces a small overhead of just 704 bits of extra dedicated storage (off the critical path).

The rest of the paper is organized as follows. Section II presents a background on the L-TAGE branch predictor. In Section III, we introduce the paging scheme and describe its architecture. In Section IV, we describe the methodology used for our tests, presenting the results of our experiments in Section V. Finally, we review related work in Section VI, and conclude in Section VII.

## II. Background: The L-TAGE Predictor

In this work we virtualize the TAGE predictor, the main component of the L-TAGE, a state-of-the-art branch direction predictor [4]. The L-TAGE predictor is composed of a TAGE predictor and a loop predictor (which tries to identify regular loops with constant iteration counts). Figure 1 shows the

**978-3-9810801-8-6/DATE12/ © 2012 EDAA**

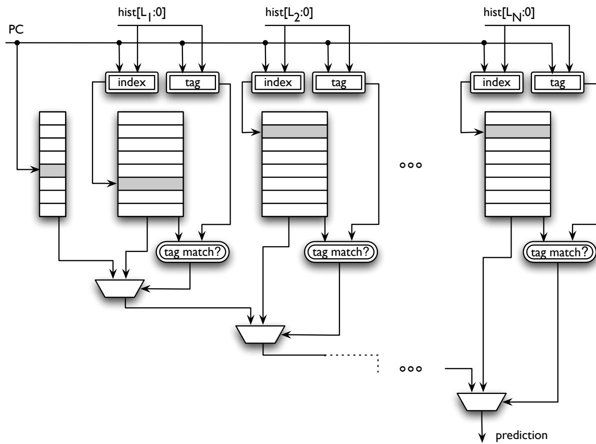


Fig. 1: The TAGE Predictor with N Tagged Tables

organization of the TAGE predictor. It consists of a base bimodal predictor and a set of partially tagged predictor tables. Each tagged table is indexed through an independent function of global and path histories and the branch PC. The set of global history lengths used forms a geometric series. The base predictor is in charge of providing a default prediction and is implemented as a simple PC-indexed 2-bit counter bimodal table. An entry in the tagged table consists of a 3-bit prediction, an unsigned 2-bit useful counter, and a tag.

At prediction time, all tables are accessed simultaneously. The base predictor provides a default prediction, while the tagged tables provide a prediction only on a tag match. If there are no tag matches, the default prediction is used. Otherwise, two predictions from the two longest history length tables with a tag match (or bimodal, if there is only one tag match) are considered. In general, the longest history prediction is used unless the entry is deemed as newly allocated, in which case the other prediction is used. The interested reader is referred to the original L-TAGE description for additional detail.

## III. PAGING SCHEME

In this scheme, one of the tagged predictor tables is augmented with a large second-level table. While the first-level table is accessible within a single cycle, the second-level table (placed in the L2 cache) has a much longer latency. Due to the delay-sensitive nature of branch prediction, the longer latency of the second-level needs to be hidden by fetching several correlated entries on each miss, thus amortizing the cost of the fetch. In general, branch predictor hash functions are designed to randomize the predictor access stream in order to reduce aliasing. The increased *perceived* capacity of the table helps to allievate aliasing, allowing us to introduce some locality in the access stream. Specifically, we limit the randomness to only the lower portion of the index bits, while using the branch PC bits directly as the upper portion of the index. In effect, the predictor table is divided into several smaller sub-tables, known as *pages* in our terminology, where each branch uses the entries within a specific page. This introduces coarse grain locality in the branch predictor access stream; The page sequence inherits the spatial and temporal locality present in program instructions. While the sequence of accesses within a page may not exhibit locality, the stream of pages accessed do.

Only the two levels of this predictor component are divided into pages, while the other components remain unchanged. The entry organization in both levels remains the same as in the original predictor. The first level table serves as a cache for the most recently accessed pages of the second level. Branches are assigned specific pages on the table based on their PC. The instruction address space is conceptually divided into blocks, and branches in one *instruction block* use the corresponding page for making prediction and allocating new entries.

The prediction mechanism is unaware of the existence of pages and is unaffected by the introduction of the second-level table. The only change to the prediction mechanism is the change in the index hash function for this component, so that each branch, based on its PC, accesses entries within a specific page. To make a prediction, only the first-level table is accessed directly. Updates are also made only on the first-level table. The second-level table is not accessed directly by the prediction mechanism. Instead, a separate engine swaps pages between the two levels as necessary.

### A. Architecture

Figure 2 shows the architecture of the paged design. We refer to the two levels of the virtualized predictor component as the L1 and L2 tables. We also refer to this component as a whole as long table, or *ltable* for short, since it appears virtually longer than the other predictor tables.

*1) Virtualization Engine:* The Virtualization Engine is responsible for maintaining pages in the two levels. The branch address is used to determine the page number, and the corresponding page is requested from the L2 table in case of a miss. At the same time, the residing page, if dirty, is written back to the L2. On a page miss, prediction is not stalled. Instead, the prediction mechanism uses the entries currently residing in the L1. In these cases, a branch may inadvertently use data in a page different than its own while waiting for the correct page to be fetched. The entry's tag provides a second layer of filtering so that the effect of aliasing is insignificant in such cases. Similarly, a branch may allocate entries in a page different than its own. Conceptually, this allows repeating patterns to take advantage of entries in other pages to make a prediction in the meantime. In addition, pages are swapped to keep the L1 in sync with the current instruction block, regardless of whether ltable provides the final prediction or not. Consequently, the sequence of pages swapped is a characteristic of the application alone, and is independent of the predictor configuration.

*2) Index Hash Function:* The index hash function is modified so that each branch allocates and uses entries in a specific page. Figure 2 shows how the bits of the branch PC are used in the index hash function. In the figure, $p$ is equal to $log_2$ of the page size, $b$ is equal to $log_2$ of the instruction block size, $a$ is $log_2$ of the number of pages that fit in the L1, and $n$ is $log_2$ of the number of pages that fit in the L2. The parameter $X$ is used to vary the instruction block size as a function of the page size, such that $b = p + X$. For each branch, bits $[b + n - 1 : b]$

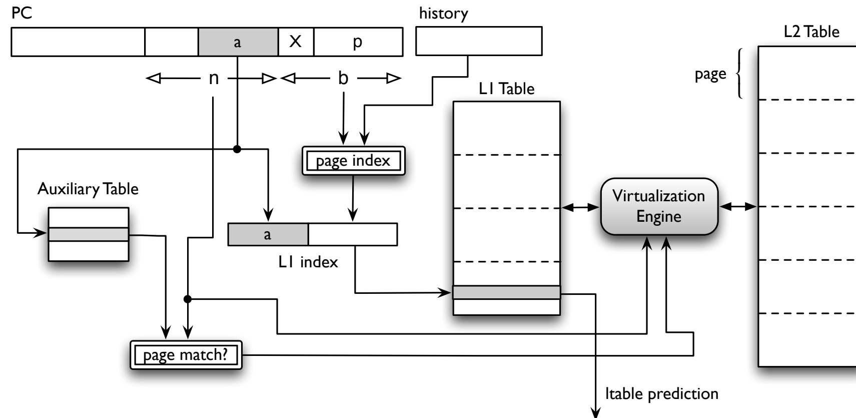


Fig. 2: Paging Scheme Architecture

determine the corresponding page number. The page number is a function of the branch address only and preserves the locality of the branch stream at a coarser grain. The position of a page in the L1 table is determined by the lower $a$ bits of the page number, bits $[b+a-1:b]$ of the branch PC.

As shown in Figure 2, the index to the L1 table is the $a$ bits of the branch PC that determine the location of the corresponding page in the L1, concatenated with the page index (index within the page). The page index hash function parallels that of the original predictor, with minor modifications. The original hash function compresses the information bits (PC, global and path histories) to $t$ bits, where $t$ is the $log_2$ of the number of table entries. Instead in the new hash function, the same information bits are compressed to $p$ bits, the $log_2$ of the page size.

*3) Auxiliary Table:* Tags for pages currently residing in the L1 are kept in the Auxiliary Table. The table has $2^a$ entries, each entry consisting of $n$ bits for the page number, as well as a single dirty bit, which indicates whether the page has been modified since it was fetched. Only dirty pages are written back to the L2. The auxiliary table represents an additional dedicated storage of $2^a \times (n+1)$ bits. Since the Auxiliary Table is only used by the Virtualization Engine, it is not on the critical path of making a prediction.

## IV. Methodology

### A. Simulation Infrastructure

To evaluate the designs explored in this work, we use the functional simulation infrastructure provided for the second Championship Branch Prediction Competition. Results are measured using functional simulation of two billion instructions of a set of three commercial workloads. For each workload, a trace of the branch addresses, corresponding outcomes and intervening instruction counts is collected on Flexus, a full-system simulator based on Simics, configured as a functional in-order uniprocessor running the UltraSPARC ISA.

To emulate the L2 table's delay, we employ a request queue model. The L2 table is modeled with separate read and write ports, and a request queue is managed for each of these ports. At most one read and one write request is serviced per cycle. We assume that this table is pinned in the L2 cache. Each request contains the number of the requested cache line, in addition to being stamped with the cycle number in which the request is made. For write requests, a snapshot of the data at the time the request is made is also passed along with the request. To simulate a latency of $N$ cycles, a request made at time t is serviced at the beginning of cycle $t+N$. The simulator fetches four instructions per cycle (fetch width of four).

We use the following benchmarks: the TPC-C v.3.0 online transaction processing (OLTP) workload running on IBM DB2 v8 ESE, the TPC-H workload running on IBM DB2 v8 ESE representing a decision support system (DSS), and finally the SPECweb99 benchmark running over Apache HTTP Server v2.0. The web server is driven with separate simulated client systems, the results present the server activity.

The only processor parameters outside the branch prediction unit considered in this work are the L2 cache line size and latency, which we assume to be 64 bytes and 20 cycles respectively. All virtualized designs described use a 64 KB second-level table, equivalent to 1 K L2 cache lines.

The primary metric used in this work to measure prediction accuracy is *mispredictions per kilo instructions* (MPKI). This metric is a better measure of prediction accuracy than misprediction rate, as it includes information about both the branch frequency, as well as the predictor's ability to correctly predict these branches. MPKI is also more representative of the performance impact of the predictor's accuracy. For example, a poor misprediction rate in a program with few branches may not have a significant impact on performance.

### B. TAGE Parameters

The original TAGE contains 12 tagged tables. We vary the number of tagged tables, $N$, from 1 to 12 since one of our objectives is to use virtualization to reduce the predictor's dedicated budget and complexity. In this way, we can determine if a non-virtualized design can be replaced with a virtualized design with fewer tables but equivalent accuracy. We use the notation $TAGE_{N,M}$ to refer to a predictor with $N$ tagged tables each with $M$ entries. The entry sizes remain unchanged from the original predictor. The prediction and useful counters are 3 and 2 bits each, and the tag widths vary for each table.

| Number of Tagged Tables (N) | Number of Entries |
|---|---|
| 1 | 16K |
| 2 | 8K |
| 3-4 | 4K |
| 5-8 | 2K |
| 9-12 | 1K |
| **Bimodal Table** | 64K |

TABLE I: Delay-Constrained Table Size Configurations

The original TAGE geometric function used for calculating history lengths takes a minimum and maximum history length, and computes a history length for each table based on the number of predictor tables. The minimum and maximum history lengths used in the original TAGE are 4 and 640 bits respectively. This allows a wide range of history lengths for a predictor with a reasonable number of tables. For a predictor with fewer tables, however, this results in very short and very long history lengths with few moderate values in between. To allow a better range of history lengths for smaller $N$, we tested the predictor with a simple geometric function: $L_i = L_1 \times 2^{i-1}$. This resulted in better accuracy for predictors of up to five components. Therefore, we use two sets of geometric functions: the new function for $N <= 5$ and the original for $N > 5$.

For our baseline predictor, we allow the predictor capacity to be set by constraints on the predictor access time rather than its area. The predictor is allowed as much capacity as it can access to make a prediction in a single cycle. We use the CACTI cache modeling tool [5] to estimate the latency of the predictor structures and choose the largest configuration that can provide a prediction in one cycle. The tagged tables are modeled as caches, and the bimodal table as an SRAM array. We optimistically assume a 4 Ghz processor. To mitigate the effects of discrepancies introduced by our simplified model for the predictor access time, we allow delays within a 20% margin of the 250 ps cycle time, up to 300 ps. Table I summarizes the results of this analysis. We refer to these baseline configurations as delay-constrained or delay-limited, since the predictor size is determined not by area constraints, but by the delay to access its tables.

## V. Tests and Results

In this section, we first describe the experiments used to explore the design space of the paged TAGE predictor. In order to allow these effects to be shown more clearly, we use a fixed size for all of the predictor tables, as opposed to using the delay-constrained configurations established using CACTI (Table I). For the final results (Section V-B), we apply the parameters determined in the design exploration phase to the delay-constrained configurations and analyze the effects.

In the design space exploration phase, the tagged table are fixed at 2K entries each ($TAGE_{N,2K}$). This is the size determined by CACTI for the range of $5 <= N <= 8$. We use this size since the baseline $TAGE_{8,2K}$ predictor achieves the best MPKI of all the delay-constrained configurations, as will be shown in Section V-B. The size of the bimodal table is 16 K entries, unchanged from the original predictor.

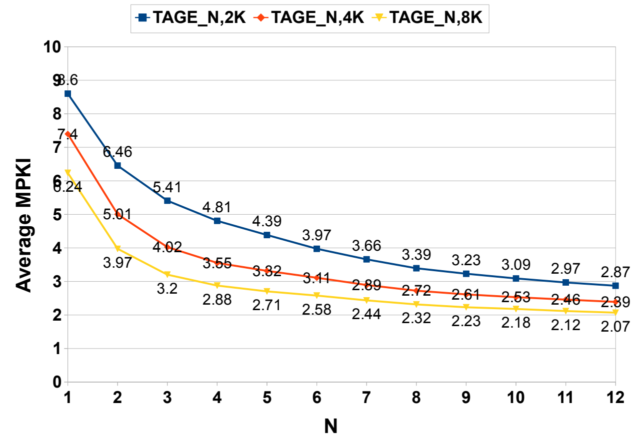


Fig. 3: TAGE MPKI as a function of the number of tables (x-axis) and the capacity of each table (curves)

### A. Design Space Exploration

*1) TAGE Baseline and Potential for Virtualization:* Figure 3 shows the average MPKI of the original TAGE predictor with three different capacities: $TAGE_{N,2K}$, $TAGE_{N,4K}$, and $TAGE_{N,8K}$ as a function of $N$. The figure shows that the predictor's accuracy can be improved both by increasing the table count and the table sizes. This observation suggests that increasing the capacity of TAGE can improve accuracy and motivates our attempt to virtualize TAGE.

*2) Choosing Which Table to Virtualize:* Since the paging mechanism is applied to a single predictor table, we ensure that increasing the perceived capacity of one table provides enough improvement to justify a paged design. We also determine the table that utilizes the increased capacity most efficiently to improve accuracy. To do this, we model an idealized version of the predictor by replacing one of the predictor tables with a large table equal in size to the L2 table, representing the virtualized component (ltable). The table is not broken into pages and can be thought of as a single large page. The accuracy of this predictor represents an upper bound on the improvement that can be achieved through paging. For all of our experiments, we use virtual table with 32 K entries (approximately 64 KB).

Figure 4 shows the average MPKI of the idealized predictor for different values of $N$. The x-axis shows which table was replaced with a large table, while each line represents a different $N$. The average MPKI of the baseline predictor (with no ltable) is also shown at $N = 0$ for comparison. In all cases, the first few tables benefit the most from an increased capacity. Intuitively, we expect the first few tables which use shorter history lengths to gain the most benefit, since the behaviour of most branches depends on the outcome of a few recent branches, while a smaller portion of branches depend on long history lengths. The best MPKI is roughly achieved with $ltable = 2$ for $2 <= N <= 6$, and $ltable = 3$ for $N > 6$. We use this ltable configuration in the rest of this study.

*3) Introducing Pages:* To simplify the design of the virtualized predictor, we choose the page size that fits in a single 64 B L2 cache line. Since each entry is about 2 B, this corresponds to a 32-entry page size. Figure 5 shows the average MPKI of the predictor with 32-entry pages. Also shown for comparison are

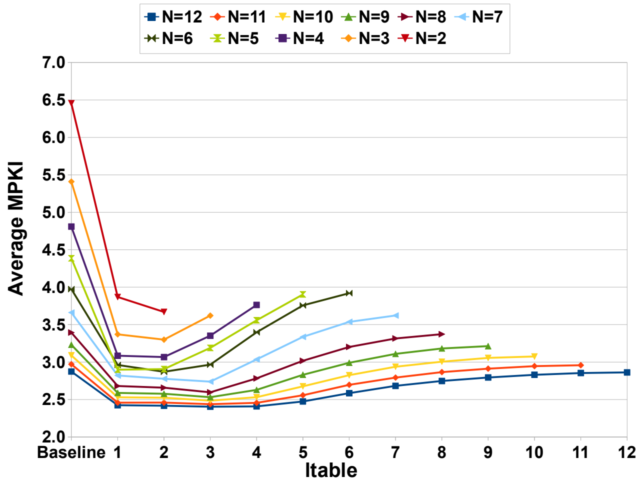


Fig. 4: Design space sweep for choosing which table to virtualize

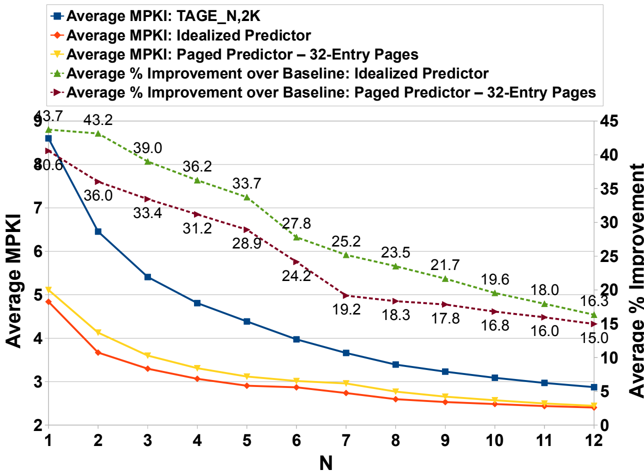


Fig. 5: Maximum Potential Improvement with a Page Size of 32 Entries

the average MPKI of the baseline predictor $TAGE_{N,2K}$ and the idealized predictor from the previous section. The two dashed plots show the percentage improvement in the accuracy of the idealized predictor and the paged predictor over the baseline. The difference between the two lines represents improvement loss due to page size constraints. It can be seen that even with paging, significant room for improvement persists.

*4) Instruction Block Size:* In the previous section, we assumed that the instruction block size (IBS), the size of a block in the instruction address space whose branches share a predictor table page, and the page size are equal. In our experiments, we found that for this IBS, a page would have to be swapped into the L1 table once every three conditional branches. We omit the detailed results due to space limitations and summarize the key finding. Specifically, we found that using an IBS eight times larger than the predictor page results in the best tradeoff between page swap frequency and improvement in MPKI. Specifically, this cuts the page swap frequency down to a third, from 33% down to 11%.

Figure 6 shows the average MPKI of this predictor for different values of $N$. Also shown for comparison are the average MPKI of $TAGE_{N,2K}$ and the predictor with equal

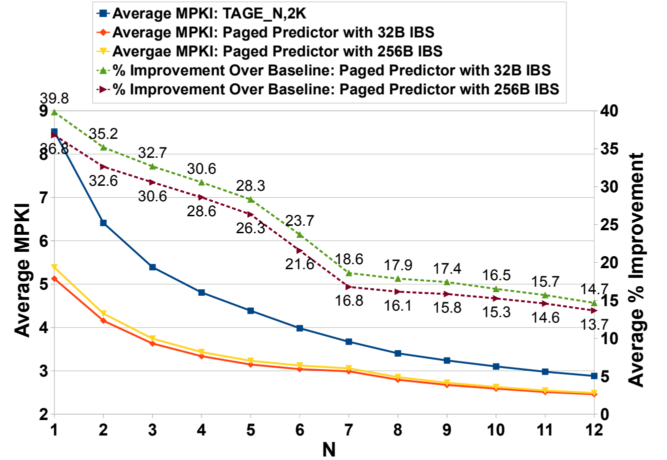


Fig. 6: Maximum Potential Improvement with an IBS of 256 B

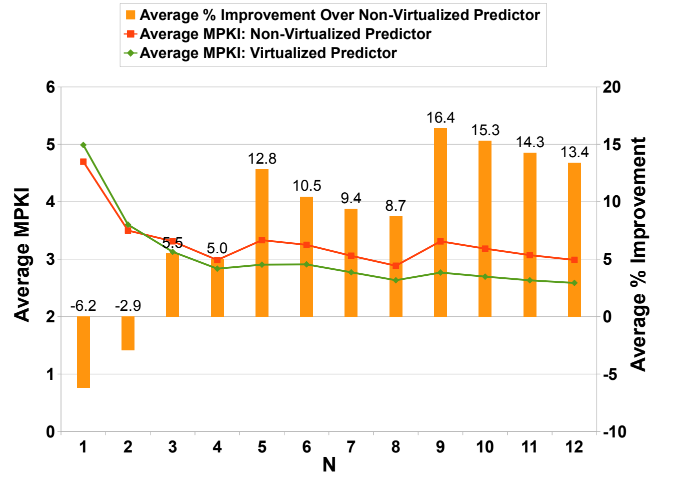


Fig. 7: Accuracy Improvement for Virtualization of Delay-Constrained Configurations

instruction block and page sizes. The two dashed plots show the percentage improvement in MPKI for the two different IBS values over (approximatley 1%-3%). In the remainder of this paper, we use a page size of 32 entries and an IBS of 256 B.

### *B. Virtualized Design with Realistic Virtual Table Latency*

Figure 7 shows the average MPKI of the virtualized and non-virtualized predictors as a function of $N$ for the delay-constrained configurations. For these results, the number of entries in the tagged tables varies as a function of $N$ according to Table I. Therefore, unlike previous sections, the MPKI curves are not a straight function of $N$. For the virtualized design, the L2 table is kept constant at 32 K entries in all cases. Its access latency is assumed to be 20 cycles. The bars show the percentage improvement in MPKI through virtualization over the baseline. For $N = 1$, virtualizing the predictor results in a worse MPKI. At this point, the virtual table size (32 K entries) is only twice the size of the dedicated table (16 K entries). The same reasoning applies for $N = 2$, where the virtual table is only four times larger than the dedicated table. Starting at $N = 3$, we see positive improvement in accuracy. This suggests that in our scheme, the virtual table has to be relatively

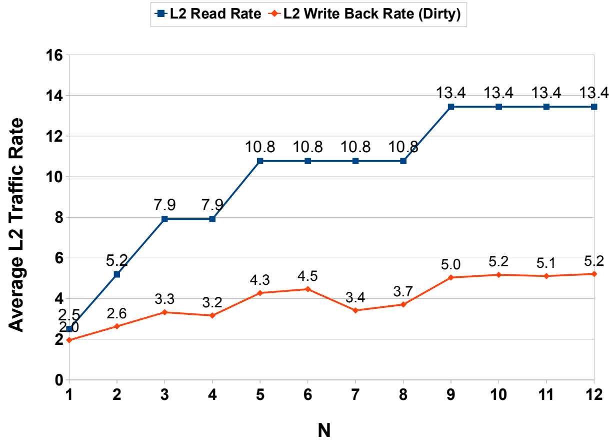


Fig. 8: Effect of Virtualization on L2 Cache Traffic

larger than the dedicated table to compensate for losses due to page size constraints. The best accuracy for the baseline predictor is achieved at $N = 8$. For this point, virtualization can improve accuracy by 8.7%. Alternatively, a virtualized design with five tagged tables can be used to achieve the same accuracy, reducing the dedicated storage of the predictor by 12 KB, 25% of the original dedicated predictor size.

To observe the effect of virtualization on the L2 traffic, Figure 8 shows the L2 cache read and write overhead rates (shown as percentages) for each $N$. These rates are measured as the number of L2 reads or writes per conditional branch. L2 writes occur only for dirty pages, and as a result are fewer than L2 reads. The figure shows that the maximum read and write overhead rates are 13.4% and 5.2% both at $N = 12$.

## VI. Related Work

There have been numerous works on branch direction prediction. We limit our attention to few closely related works due to space limitations. Delay-sensitive hierarchical branch predictors [1] share the same goal as our work: to allow the use of a larger more accurate predictor without increasing prediction delay to more than a single cycle. While they were introduced in the context of allowing predictors to scale to future technologies, they could potentially be used to hide the latency of large predictor tables. Three schemes are proposed for overcoming delay: a caching approach, an overriding approach, and a cascading lookahead approach applied to the gshare predictor. The last two schemes could be considered as alternatives to virtualization. However, in both cases prediction accuracy depends on the accuracy of both a fast and less accurate predictor and that of a larger more accurate one. These schemes could benefit from using a virtualized predictor (which provides predictions in a single cycle) as the first-level predictor. Unlike a virtualized design, these schemes require large dedicated budgets for implementing the two predictor levels.

Virtualization has been applied to the EXplicit dynamic branch predictor with ACTive updates (EXACT) [6]. EXACT employs a different approach to improving branch prediction by using information about load and store instructions that may affect branch outcomes. It requires that such instructions update the branch predictor directly. Two new structures are introduced which are in charge of keeping track of the effect of load and store instructions. Virtualization is applied to the latter structure, the *Active Update* unit, which takes the address of a store instruction as index and outputs the PC of the branch affected by the store, as well as the effects of the change. Since active updates are tolerant of 400+ cycles of latency, a straightforward application of PV is used on this unit. In our scheme, virtualization is applied to the branch predictor unit itself, and is thus orthogonal to the EXACT predictor.

## VII. Conclusions and Limitations

We have proposed a method for improving the accuracy or reducing the dedicated storage cost of branch direction prediction. Our method virtualizes one of the tables in the TAGE predictor and stores a much larger virtual table in the L2 cache. Through the introduction of a paging scheme, we split a monolithic virtual table into a collection of sub-tables, each assigned to a different portion of the instruction memory space. As a result, locality is introduced in the branch predictor access stream, allowing it to tolerate the increased latency incurred due to virtualization. Experimental results showed that the virtualized predictor can either be used to increase accuracy by 8.7% or to reduce dedicated storage by 25%.

We have used functional simulation to evaluate the accuracy of the proposed branch predictor designs. This is typical of branch prediction studies. We used instruction count to approximate processor cycles and to emulate delay of the predictor L2 table. Cycle-accurate simulation would allow us to determine the impact of our design on the L2 cache traffic, as well as the effect of the improved accuracy on the overall processor performance. Furthermore, it would allow us to determine predictor L2 table sizes which benefit prediction accuracy without contending for the L2 cache data.

## References


[1] D. A. Jiménez, S. W. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 33. New York, NY, USA: ACM, 2000, pp. 67–76.

[2] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi, "Predictor virtualization," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 157–167.

[3] I. Burcea and A. Moshovos, "Phantom-btb: a virtualized branch target buffer design," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '09. New York, NY, USA: ACM, 2009, pp. 313–324.

[4] A. Seznec, "The l-tage branch predictor," *Journal of Instruction Level Parallelism*, vol. 9, April 2007.

[5] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," HP Laboratories, Tech. Rep., April 2009.

[6] M. Al-Otoom, E. Forbes, and E. Rotenberg, "Exact: explicit dynamic-branch prediction with active updates," in *Proceedings of the 7th ACM international conference on Computing frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 165–176.