Cost Effective Speculation with the Omnipredictor

Arthur Perais, André Seznec

HAL Id: hal-01888884
https://hal.inria.fr/hal-01888884

Cost Effective Speculation with the Omnipredictor

Arthur Perais   André Seznec
INRIA Univ Rennes IRISA CNRS

ABSTRACT

Modern superscalar processors heavily rely on out-of-order and speculative execution to achieve high performance. The conditional branch predictor, the indirect branch predictor and the memory dependency predictor are among the key structures that enable efficient speculative out-of-order execution. Therefore, processors implement these three predictors as distinct hardware components.

In this paper, we propose the *omnipredictor* that predicts conditional branches, memory dependencies and indirect branches at state-of-the-art accuracies without paying the hardware cost of the memory dependency predictor and the indirect jump predictor.

We first show that the TAGE prediction scheme based on global branch history can be used to concurrently predict both branch directions and memory dependencies. Thus, we unify these two predictors within a regular TAGE conditional branch predictor whose prediction is interpreted according to the type of the instruction accessing the predictor. Memory dependency prediction is provided at almost no hardware overhead.

We further show that the TAGE conditional predictor can be used to accurately predict indirect branches through using TAGE entries as pointers to Branch Target Buffer entries. Indirect target prediction can be blended into the conditional predictor along with memory dependency prediction, forming the *omnipredictor*.

1. INTRODUCTION

1.1 The Context: Speculation in All Stages

To maximize performance, out-of-order execution superscalar processors speculate on many aspects of the control- and data-flow graphs. Since the *fetch-to-execute* delay may span several tens of cycles, accurate control-flow speculation is tantamount to performance. Speculative memory access reordering is also key to efficiency, but may lead to memory order violations hence frequent pipeline squashes. Therefore, accurate memory data dependency specualation is also a necessity.

Control-flow Speculation.
The instruction fetch front-end of the processor retrieves blocks of contiguous instructions. The correct address of the subsequent instruction block is computed rather late in the pipeline, often as late as the execution stage. Therefore the instruction fetch engine is in charge of speculatively generating this instruction block address. Modern processors instruction address generator features three predictors, the Return Address Stack (RAS) [1], the conditional branch predictor [2] and the indirect jump predictor [3]. It also features a Branch Target Buffer (BTB) [4] to cache decoded targets for direct conditional and unconditional jumps, and in some designs, to predict targets for indirect jumps and/or returns. Among these structures, the most critical one is the conditional branch direction predictor since conditional branches are the most frequent and can only be resolved at execute time. Significant research effort [2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 4, 16, 17, 18, 19] was dedicated to this class of predictors throughout the past 25 years. State-of-the-art propositions [9, 19] combine the neural branch prediction approach [18] with the TAGE predictor [16]. The BTB is also very important, since a missing target results in a pipeline bubble in the front-end: a miss in the BTB for a direct branch can be resolved at Decode. The RAS addresses very efficiently the predictions of return targets provided an adequate management of its speculative state [20]. Lastly, indirect branches were traditionally predicted through the BTB. However the misprediction rate on this class of branch was shown to be problematic [3], especially given that the targets of such branches are always resolved in the Execute stage. Thus, specific predictors were proposed to handle them.

Data-flow Speculation.
To maximimize performance, out-of-order processors should be aware of the complete data-flow graph of the current instruction window. Unfortunately, memory dependencies cannot be determined precisely until the addresses of inflight memory instructions are resolved in the Execute stage. To tackle this limitation, memory dependency prediction (MDP) was proposed [22]. Ideally, the role of MDP is to determine, for a given load if an older speculative store will write to the loaded address so the load can be marked dependent on the store and wait for it to execute. One of the first documented hardware implementations of MDP can be found in the Alpha 21264 [23]. It categorized loads as either "can issue as soon as register dependencies are satisfied" or

"must wait for all older stores to compute their address". More refined schemes were proposed by Chrysos and Emer [24] and Subramaniam and Loh [25]. However, all these contributions introduce dedicated hardware structures to perform speculation.

Overhead of Speculation.

In summary, modern microarchitectures feature several predictors to speculate on different aspects of the control- and data-flow. These predictors are *paramount* to performance, but represent **a significant amount of hardware** and contribute to power and energy consumption. In a fetch block, each instruction can be a conditional branch branch, a memory load or an indirect branch. To enable high over-all front-end bandwitdth, the overall potential prediction bandwidth of these predictors is largely over-dimensioned on many superscalar processor designs. For instance, on the Compaq EV8 [16] or on the IBM Power 8 [36], the conditional branch predictor generates one conditional branch prediction and one possible branch target per fetched instruction (16 per cycle on EV8, 8 per cycle on Power 8).

1.2 Unified Speculation in a High Performance Microprocessor

Implementing a conditional or/and MDP predictor able to generate a prediction per fetched instruction for a block of 8 contiguous instructions might be considered as waste of silicon and/or energy since such a block of 8 instructions rarely features more than 3 branches (including at most one taken) and more than 3 memory loads or stores.

In this paper, we show that the conditional branch predictor, the memory dependency predictor and the indirect branch predictor can be unified within a single hardware predictor, the *omnipredictor*. **This organization saves on silicon real-estate at close to no performance overhead.**

As a first contribution, we introduce the MDP-TAGE predictor, which leverages the observation that the multivalue prediction counter (in fact a 3-bit field) of the TAGE branch predictor entry [19] can be interpreted as a distance to a store for a load instruction. This allows us to blend the memory dependency predictor into the conditional branch predictor.

Second we show that the TAGE predictor storage structure can also be used to predict indirect branches. The 3-bit field in the TAGE entry can be used as a pointer or an indirect pointer to targets stored in the BTB. Up to N targets (N being the maximum number of instruction in the fetch block) may be reached in a single access on the block-based BTB, with accesses to additional targets requiring an extra access on the BTB.

As a consequence, the conditional branch predictor, the memory dependency predictor and the indirect target predictor can be packed together in the omnipredictor. The *omnipredictor* has the storage structure of the TAGE conditional branch predictor, but predicts branch direction for branches, distance to or existence of the producer store for loads, and pointers to targets stored in

the BTB for indirect jumps. The *omnipredictor* provides close to state-of-the-art indirect target and memory dependency prediction through stealing entries in the conditional branch predictor table and the BTB, while only marginally decreasing the branch prediction accuracy. That is, the *ommipredictor* allows to significantly reduce the overall hardware real-estate associated with speculative execution, removing the stand-alone memory dependency predictor and indirect jump predictor without provisioning additional entries in the under-utilized predictors: TAGE or the BTB.

1.3 Paper organization

For the sake of simplicity, unless mentioned otherwise, we will consider a fixed instruction length ISA such as ARMv8.

The remainder of this paper is organized as follows. Section 2 briefly presents state-of-the-art in branch direction, branch target and memory dependency prediction.

Section 3 presents the structure of the instruction fetch front-end on wide-issue superscalar processors implementing a fixed instruction length ISA on processors such as Compaq EV8 and IBM Power 8; it also points out the prediction bandwidth waste associated with such a structure. Section 4 presents the principle of TAGE-based memory dependency prediction. We also describe how conditional branch prediction and memory dependency prediction can be packed in the same hardware predictor. Section 5 further shows that TAGE associated with the block-BTB of a superscalar processor can also be used to predict indirect targets. Section 6 presents experimental results. Section 7 discusses the *omnipredictor* approach in the context of a variable instruction length ISA. Finally Section 8 provides directions for future research as well as concluding remarks.

2. RELATED WORK

2.1 Conditional Branch Prediction

Branch direction predictor have come a long way since Smith's study on branch prediction strategies in which the PC-indexed bimodal table was introduced [2]. Many schemes were proposed over the last three decades, including *2-level* [5, 6], *gshare* and *gselect* [10], YAGS [15], PPM [13] and GEHL [14]. The current state-of-the-art is embodied by neural based predictors [7, 8, 18] as well as the TAGE predictor [16] and its derived predictors [9, 19].

2.1.1 Neural predictors

Jiménez and Lin introduced the family of neural branch predictors in [18]. The predictor first selects a set of weights by indexing in a PC-indexed table, then, the dot product of the weights and the global branch history is computed. If the value is negative, the branch is not taken, and if it is positive, the value is taken. At update time, the weights are adjusted following the actual outcome by increasing the weight if the corresponding outcome in the global history agrees with the actual

outcome, and by decreasing the weight if not. As we focus on TAGE in this study, we refer the reader to [7, 8, 18] and for more details on neural branch predictors.

2.1.2 TAgged GEometric (TAGE)

In this paper, we only consider a basic TAGE branch predictor (i.e., without extensions) [19], however our *omnipredictor* proposal could be implemented on top of the TAGE component of any TAGE-derived conditional branch predictor [9, 19].

TAGE is a global history predictor featuring several partially tagged tables that are backed by a direct-mapped bimodal predictor. The T partially tagged tables are accessed using T hashes of the branch PC, the global path/branch history.The crux of the TAGE algorithm is that the different lengths of the global branch history used in each hash form a geometric series. Thanks to the geometric series of global histories, TAGE is able to capture correlation between very close as well as very distant branches, while dedicating most of the storage to short histories, where most of the correlation is found. A 1+3 TAGE predictor is depicted in Figure 1.

Prediction All tables are accessed in parallel, and the table using the longest global history that matches provides the prediction. If there is no match, the bimodal base predictor provides the branch prediction. Entries of partially tagged components feature a 3-bit saturating counter representing the prediction, a 2-bit *useful* counter used by the replacement policy, and a partial tag. Bimodal entries only feature a 2-bit saturating counter. We point out that in both cases, the counters can encode more information than just *taken* and *not taken*.

TAGE introduces the notion of *provider* and *alternate* prediction. The *provider* is the regular prediction, while the *alternate* is the prediction that would have been used if the *provider* component had not matched. In some cases, accuracy is higher if the *alternate* prediction is used instead of the *provider*.

Update TAGE may update several entries for a single prediction. If the prediction is correct, both *provider* and *alternate* entries may be updated. If the prediction is wrong, the same applies, but another entry is allocated in a randomly chosen component using a longer global branch history. In particular, each tagged entry features a *useful* counter (u) that decides whether an entry can be replaced during allocation. u bits are periodically reset. We refer the reader to [19] for a more detailed description of how TAGE operates.

2.2 Indirect Target Prediction

Indirect target predictors cannot perform well if they are PC-indexed only, given that one PC will have many targets. As a result, Chang et al. first proposed a gshare-like predictor [3] able to differentiate the different targets of a given static branch using global branch history. This led to a refinement proposed by Driesen and Hozle [26], the cascaded predictor, in which a first PC-indexed
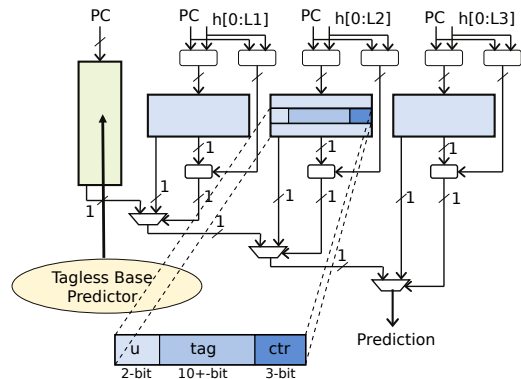

Figure 1: 1+3 TAGE predictor synopsis, from [19].

level is used to filter easy to predict targets, while a second level indexed with a hash of the PC and the global branch/path history.

This latter predictor paved the way for the state-of-the-art ITTAGE predictor [16]. Much like TAGE, ITTAGE features a PC-indexed base predictor (e.g., a BTB) and several partially tagged tables indexed by a hash of the PC and some bits of the global branch history (direction and target). The history lengths used to index the tagged tables follow a geometric series, allowing ITTAGE to capture correlation between very distant branches while dedicating most of its storage to correlation between close branches. ITTAGE operates similarly to TAGE. For instance, recent Intel processors implement a predictor whose accuracy is comparable to the ITTAGE indirect target predictor [21].

Kim et al. propose *Virtual Program Counter* (VPC) to treat indirect branches as a set of conditional branches (one for each target), and use the conditional branch predictor to determine the actual target [28]. That is, several targets using different virtual PCs are stored in the BTB for a single indirect branch. VPC performs several conditional predictions serially until one is found to be taken, and the corresponding target is retrieved from the BTB using the virtual PC.

Lastly, Joao et al. [29] propose to follow paths corresponding to two (or more) targets when an indirect branch is encountered. The targets are selected dynamically and are those followed the most often. This approach requires complex hardware support for out-of-order execution of predicated code.

2.3 Memory dependency Prediction

Memory dependency prediction (MDP) was first proposed by Moshovos [22] to improve scheduling and refined in [24, 25, 30]. Other contributions focused on speculatively bypassing the source of a store to the destination of a corresponding load to increase performance [31, 32, 33, 34].

Figure 2 shows the simulated performance of different MDP schemes implemented in an aggressive out-of-order processor (using the framework later detailed in Section 6.1) relative to blind speculation (i.e., loads never wait for older stores to execute). It is quite clear
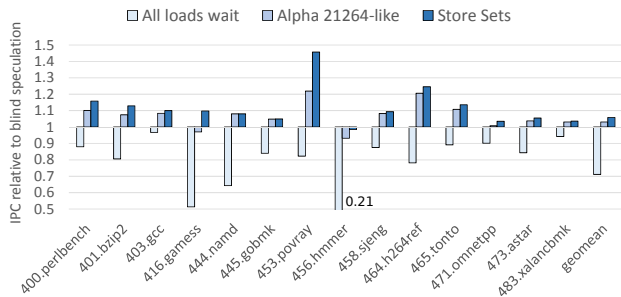
Figure 2: The impact of memory dependency prediction: IPC relative to blind speculation (loads never wait on older stores to execute) for SPECCPU2006 benchmarks sensitive to MDP.

that having all loads wait on all older stores is extremely inefficient as it greatly limits the out-of-order capabilities of the processor. A simple PC-indexed scheme resembling the Alpha 21264's [23] (a bitvector informing if a given load has to wait for all older stores to execute before executing or not) is quite efficient, achieving noticeable ($> 10\%$) speedups over blind speculation in benchmarks featuring many memory ordering violations, but hinders performance in *hmmer*. Finally Store Sets [24] is able to precisely link dynamic loads with dynamic stores. In [25], the performance of Store Vectors is only slightly higher than Store Sets. That is, the advantage of Store Vectors is mostly in reduced complexity. Given that our scheme performs MDP almost for free, we therefore do not consider Store Vectors in this paper.

Regardless, Store Sets requires more storage than the 21264-like predictor: 3.75KB vs. 1KB in our study, assuming 10-bit sequence numbers are stored in the Last Fetched Store Table (LFST) of Store Sets.

3. INSTRUCTION FETCH FRONT-END ON A SUPERSCALAR PROCESSOR

3.1 Next instruction fetch block address generation

In a superscalar processor, blocks of contiguous instructions are fetched in parallel each cycle. At the same time, the address of the next fetch blocks has to be computed/predicted. Accurately predicting the address of the next instruction fetch block is a rather complex process. It involves several distinct operations: 1) computing the fall-through address 2) predicting the targets of all direct branches 3) predicting the directions of all conditional branches 4) predicting the targets of returns 5) predicting the targets of indirect jumps. Finally the final block address is selected depending on all the previous stages and on the instruction decode result. This is illustrated in Figure 3.

This address prediction cannot be performed in a single cycle and therefore spans over several cycles. To hide this delay, a fast predictor predicts the next cache block to fetch each cycle, but is overridden by the complex – but slow – predictor if the fast prediction does not match the complex prediction [11, 17, 23].

3.2 Predicting contiguous instructions

In order to optimize the instruction fetch bandwidth, one has to fetch instruction blocks that may feature several branches with at most one of them being taken. The I-cache access followed by Decode may span several cycles. However, at fetch time, one does not have information on the number and positions of branches within the fetch block. In order to provide conditional branch and target predictions as soon as possible, the simplest solution for fixed instruction length ISA is to predict all the instructions in the block as if they were branches. This is illustrated in Figure 3. Then, the predictions are used for the final next block address selection only if the instructions were effectively branches. This type of design was implemented in the EV8 processor [17] and the IBM Power 8 [36]. In this paper, we assume that the designs of the Branch Target Buffer (BTB) and of the conditional branch predictor follow this approach.

Specifically, at fetch time, the branch predictor and the BTB are queried to provide a prediction for each instruction of the fetch block. A maximum of n consecutive sets of the BTB are accessed using the instruction block address high order bits. The low order bits determine the set associated with each instruction word.

As illustrated by the EV8 predictor [17], a global conditional branch predictor – even a predictor featuring several logic predictor tables like TAGE – can be searched with the same technique. n contiguous predictions are read from the n banks using the index associated with the instruction block address. This index is computed from a hash of the address of instruction block and the global branch history associated with the first instruction in the block. In order to load balance the number of predictions among all prediction words in the prediction lines, the index of the prediction word for instruction i in the block can also also be computed as a hash of the block address and the branch history [17].

Through the remainder of the paper, with the exception of Section 7, we will assume that the conditional branch predictor and the BTB are read at fetch time and provide one prediction per instruction word in the fetch block.

3.3 Wasting the prediction bandwidth

The branch predictor and BTB designs presented above achieve one target prediction and one conditional prediction per instruction word in the fetch block. However, most of these predictions are useless since conditional branches and direct jumps represent only a fraction of the instructions. This leads to a substantial waste of energy in table lookups. Moreover as already pointed out, the memory dependency predictor must also be dimensioned to accommodate up to N loads per cycle. To avoid the waste of prediction bandwidth, one solution could be to limit the number of possible

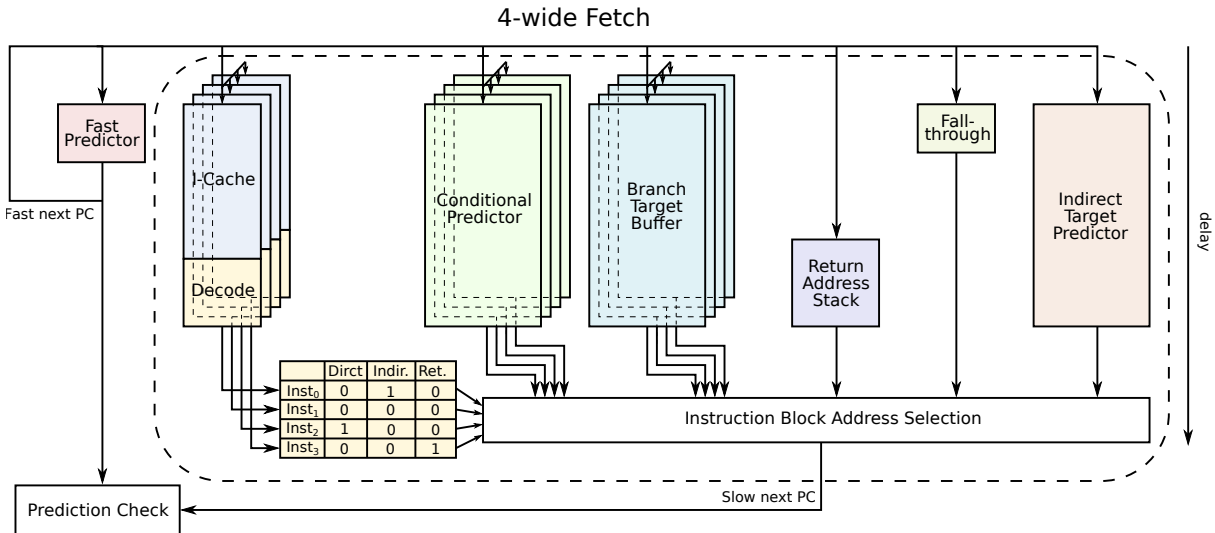|	Dirct	Indir.	Ret.
Inst$_0$	0	1	0
Inst$_1$	0	0	0
Inst$_2$	1	0	0
Inst$_3$	0	0	1

Figure 3: The instruction fetch front end of a superscalar processor.

branches and the number of possible loads per instruction fetch block (e.g., a maximum of 2 branches and 2 loads for 8 instructions). This reduces the instruction fetch bandwidth.

Conversely, the *omnipredictor* increases the usefulness of predictions by giving meaning to conditional branch predictions mapping to non-conditional branches. This information is interpreted just after Decode depending on the instruction type: direct branch, memory access or indirect jump.

3.4 Energy considerations

One can save energy on the predictions through filtering the accesses to the slow next PC predictor and the memory dependence predictor. For instance, one can extend the fast next PC predictor illustrated in Figure 3 with predecode bits, e.g. one predecode bit per instruction to encode the information branch or non-branch. Moreover one can also add extra predecode bits for the presence or absence of a indirect jump, a call or a return in the fetch block. Through using such predecode bits, one can avoid useless reads on the prediction structures. For memory dependence prediction, if delaying the read of the prediction tables after the decode stage does not lengthen the overall front-end pipeline , one can prevent the access to the memory dependence predictor by non-loads instructions.

To limit energy consumption on the *omnipredictor* that combines the branch predictor and the memory dependence predictor in a single structure, one may rely also on predecode bits associated with each instruction in the fast next PC predictor, the predictor tables being only accessed for branch or load instructions.

4. USING TAGE TO PREDICT MEMORY DEPENDENCIES

For a given dynamic load, the role of MDP is to identify the most recent inflight older store that will write

(or already wrote) to the same address, if there is one. This allows to enforce the data dependency through memory when there is one, while maximizing potential for out-of-order execution when there is no dependency. However, MDP encounter both false negatives (load predicted to have no producer but has one) and false positives (load predicted to have a producer but does not) can cost many cycles.

The approach that is generally used for MDP is to train the predictor only when a load is found to have executed before an older store to the same address [23, 24, 25]. Consequently, loads that do depend on an older store but happen to never execute out-of-order with said store do not occupy an entry in the predictor. We use the same approach for the MDP-TAGE predictor. The MDP-TAGE predictor has the same structure as the TAGE predictor, except that only the tagged tables are considered. Upon detecting a memory order violation, a MDP-TAGE tagged entry is allocated for the faulting load if there was no hit at prediction time.

4.1 Forgetting Predictions

To perform well, existing memory dependency predictors [23, 24, 25] are able to forget predictions to mitigate the fact that dependencies may be transient. The TAGE replacement policy already implements a gracefully aging the useful counters with a periodic reset every 512K update [16]. However this period is too long for the memory dependence entries; therefore if a memory dependence entry was predicting a dependence and at execution the data came from the cache (i.e., they were – most likely – not dependent on an older store), we gracefully reset its useful counter with probability $\frac{1}{256}$.

4.2 Linking Loads with Producer Stores

So far, when a load hits in the MDP-TAGE tagged tables, we expect that it should not execute before a particular inflight store, but we do not identify which
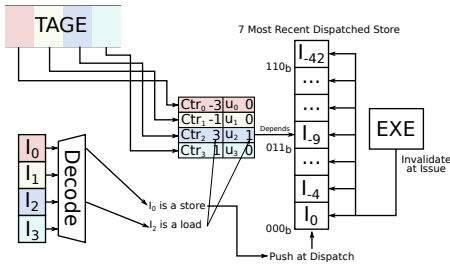
Figure 4: Memory dependency Prediction using only the conditional branch predictor.

one. Linking the load with its producer store can be done in two fashions, that we describe in the following paragraphs.

Coarse Linking.

A first possibility is to implement 21264-like memory dependency prediction i.e., predicting *store-wait* bits. That is, if a load is predicted to have a memory dependency, it will wait for all older stores to execute before executing. Loads that do not hit in MDP-TAGE can issue as soon as their operands become ready. This emulates an Alpha 21264-like predictor that considers path information. Note that this scheme would not require a prediction counter in MDP-TAGE entries.

Precise Linking.

The earlier scheme performs a binary prediction for memory instructions. However, the TAGE conditional branch predictor, as depicted in [19], implements 3-bit prediction counters in tagged tables. For predicting memory dependency, we do not need a prediction counter, but a pointer to identify the precise store on which the data depends. Therefore, with a m-bit field, we can encode 2^m different status for a load that hits in the branch predictor. As our objective is to pack the MDP-TAGE predictor and the TAGE conditional predictor in the same hardware, we will assume $m = 3$.

First, we reserve the counter value 111b for loads that should be marked as dependent on all older stores. Second, we use other counter values to express which precise store the load should be marked dependent on. For instance, if the counter value is 011b, then the load will be marked dependent on the 4th most recently dispatched store. This is achieved by implementing a FIFO of sequence numbers where stores are pushed at Dispatch. Similarly to Store Sets, stores must explicitly check the queue and invalidate themselves when they issue, however, since this is a very small structure (7 entries of 6/7-bit Store Queue identifiers), the cost of doing so is negligible. *Precise Linking* is depicted in Figure 4.

4.3 Combining TAGE and MDP-TAGE

Combining the branch direction and memory dependency in the same physical hardware predictor is straightforward. All the predictions associated with the instruction block are read in parallel from the TAGE tables and the global predictions are computed. After the instruction types are known each prediction is treated according to the instruction type. We refer to this unified predictor as TAGE-MDP-TAGE.

4.4 Towards Speculative Memory Bypassing

The memory dependency prediction scheme can be easily extended to perform speculative memory bypassing through the physical register file [33]. For instance, on an MDP-TAGE hit pointing on the ith youngest store, the target register of the load is renamed to the source register of the ith register store. This speculative memory bypassing would be limited to the window of the seven youngest older stores preceding the current load.

5. PREDICTING INDIRECT BRANCHES

In this section, we present an indirect target prediction (ITP) scheme that leverages the conditional branch prediction infrastructure of a superscalar processor, i.e., the TAGE predictor and the BTB. Then, we unify this indirect target prediction scheme with MDP and conditional prediction within the *omnipredictor* infrastructure. We refer to the unified branch direction and indirect target prediction predictor as the TAGE-IT-BTB predictor.

An indirect branch may have any number of targets, while the BTB features a single target for each static branch. However we can leverage the fact that the BTB is able to speculatively deliver a target per instruction in the fetch block. We use this property to allow the BTB to track several addresses per indirect jump. The TAGE predictor is used to point to these different targets depending on global history lengths.

5.1 Predicting a single target

Many indirect jumps have a single target (or a very dominant target). We accommodate this case as follows. If the BTB entry associated with the PC of the indirect jump hits and there is no hit in the TAGE predictor then the BTB entry is used to predict. We will refer to this BTB entry as the A0 BTB entry (for Access-0).

5.2 Predicting a few targets

Let us assume an 8-wide Fetch stage: 8 targets are pulled from the BTB each cycle. In the meantime, 8 predicted directions are pulled from the branch predictor. At Decode, directions and targets are attributed to decoded instructions when relevant. To provide indirect target prediction capabilities without overhead, we propose that, similarly to memory dependency prediction, and on a hit on the tagged table of TAGE, the value of the 3-bit field (the "prediction counter" for conditional branches) could be used to select one of the 8 targets that were pulled from the BTB, as shown in Figure 5 (the Figure only shows 4 targets assuming a 4-wide Fetch stage). We will refer to these BTB entries as the A1 BTB entries (for Access-1). A tag hit on an A1 BTB entry will be referred to as an A1 BTB hit. Note

that the A0 BTB entry is also an A1 BTB entry, and as such, latency is similar for both types of accesses.

5.3 Branches with a Large Number of Targets

So far, our proposal covers only the indirect branches that feature only a few targets (less or equal than 8). To address this limitation, we propose to reserve one value of the 3-bit field (e.g., 111b) to express that a given indirect branch has many targets. Upon finding out that an indirect branch is in that case at Decode, the instruction fetch front-end till Decode is flushed; the BTB is accessed a second time, using the PC of the indirect branch hashed with the global branch history as an index. This allows to reach many more targets, at the cost of inserting a pipeline bubble and increasing conflict in the BTB.

While the penalty is significant, one has to consider that (i) it remains much smaller than waiting for the target to be resolved at Execute and (ii) Up to seven targets are still reachable using the "fast" prediction scheme. Moreover, while this will most likely increase the number of mis-targets for direct branches, the cost of such a mis-target is much lower than the cost of a mis-target for an indirect branch (e.g., 5 cycles vs. a minimum of 15 cycles in our framework). Therefore, in programs that use indirect branches with many targets (e.g., interpreters), it is preferable to favor indirect targets over direct targets.

We will refer to these BTB entries as the A2 BTB entries (for Access-2). A tag hit on an A2 BTB entry will be referred to as an A2 BTB hit.

In this paper, we found that performing the A2 access using a fixed-length global branch history, similarly to *gshare*, yields good results.

5.4 Summary of Prediction Scenarios

1. TAGE tag miss, but A0 BTB hit: the A0 BTB content is the predicted target.

2. TAGE tag hit, 3-bit field different from 111b, A1 BTB hit in the corresponding entry: the A1 BTB content is the predicted target.

3. TAGE tag hit, 3-bit field is 111b: second access to the BTB with a hash of the PC and the branch history, in case of a tag hit, the A2 BTB content is the predicted target.

4. other cases: no prediction

5.5 Updating the predictor

On an indirect jump, the prediction infrastructure (TAGE and BTB) is updated as follows:

First let us consider the correct prediction scenarios. 1) On an A0 BTB hit (thus a TAGE tag miss), the hysteresis of the BTB entry is strengthened in order to avoid replacement. 2) On an A1/A2 BTB hit (thus a TAGE tag hit), the hysteresis of the BTB entry and the useful field of the TAGE entries are strengthened in order to avoid replacements.
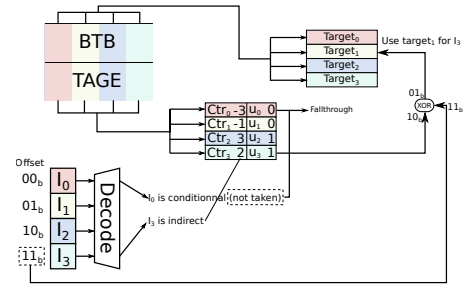

Figure 5: Indirect target prediction using only the conditional branch predictor and the BTB. Case of an A1 hit.

For mispredictions, several scenarios must be considered, but the update process must know, of the 8 targets that were pulled from the BTB (assuming a tagged BTB): 1) Which were the ones already allocated to the indirect branch and 2) Among those, which had the correct target.

1. A0 tag miss and TAGE tag miss: the A0 BTB entry is allocated with the target of the jump. This potentially leads to an A0-hit on the next access to the indirect jump.

2. A1 tag and target match, but TAGE tag miss or pointer mismatch: a new TAGE entry is allocated and its pointer (i.e., 3-bit field) set to the matching A1 BTB entry. If there was a tag hit on TAGE then the matching entry is also updated with the correct pointer.

3. No A1 tag match, and no A2 lookup (TAGE tag miss or pointer $\neq$ 111b): an A1 entry is allocated in the BTB if possible (A0 if not already present in priority, otherwise an A1 entry that has the same target if present, otherwise any A1 entry). If no A1 entry is available for replacement (i.e., all A1 entries are occupied by targets of the indirect branch) then the A2 entry must be allocated. In any case, if the allocated entry is not the A0 BTB entry, a TAGE entry must allocated. In case of TAGE hit, the existing TAGE entry must be updated.

4. In case of A2 BTB mismatch (i.e., TAGE tag hit, 3-bit field = 111b, A2 tag hit, but misprediction), *when not covered by case 2*: if an A1 entry is free, it is allocated, the TAGE entry is updated and a new TAGE entry is allocated. If not, the A2 BTB entry is updated.

5. In case of A2 BTB tag miss (i.e., TAGE tag hit, 3-bit field = 111b, A2 tag miss), *when not covered by case 2*: equivalent to case 4.

Tagless BTB.
So far, we have assumed that the BTB is tagged. However, to limit access time and area (e.g., for a first level BTB), tags may be removed. In that case, TAGE-IT-BTB operates as before, with an exception. Indeed,

at update time, the hardware cannot determine if all A1 entries belong to the indirect jump, therefore, without taking specific action, an A1 entry will always be considered to be free to allocate to the indirect jump. Thus, A2 entries will never be allocated, limiting the number of reachable targets. To remedy this, we can simply use a small pseudo-random number generator to probabilistically allocate an A2 entry on a mis-target.

BTB-less designs.

Processors with low latency I-Cache and decoders may use the actual target resolved at Decode to override the fast predictor for direct branches, as proposed for the Alpha EV8 [17]. In that case, unifying conditional branch and indirect target prediction can be implemented as follows. A table stores several targets per indirect branches e.g., 8. These 8 contiguous entries are read in parallel and the branch predictor provides a pointer. A0- A1- and A2-entries are used the same way as on the TAGE-IT-BTB described above. This removes the need to invest hardware real-estate for a full-fledged indirect predictor such as ITTAGE.

5.6 All together: the Omnipredictor

The TAGE predictor provides a prediction for each instruction. This prediction is not only a binary prediction, but a 3-bit counter accompanied by the usefulness of the prediction. The prediction is the sign of the counter if one wants to predict a direction, but it can be interpreted differently for memory dependencies (distance to the providing store) or pointer to a location for an indirect jump. Packing the three predictors in a single TAGE structure is thus simple: After Decode, each prediction is interpreted according to the type of the instruction.

To allow predicting indirect targets, we use the block-BTB that naturally provides N target words, while most of the instructions are not branches.

Combining branch direction prediction and memory dependency prediction in the same physical hardware predictor is straightforward. All the predictions associated with the instruction block are read in parallel from the TAGE tables and the global predictions are computed. After the instruction types are known (just after Decode or even earlier if pre-decode bits are available), each prediction is treated according to the type of the corresponding instruction.

Although it complicates the predictor update – which is not on the critial path – this organization saves on silicon real-estate and energy by not implementing IT-TAGE tagged tables and a dedicated memory dependency predictor.

6. EXPERIMENTAL RESULTS

6.1 Evaluation Methodology

We evaluate our unified prediction mechanism through cycle-level simulation on the gem5 simulator [38]. Our binaries are compiled for the ARMv8 (Aarch64) ISA. The different pipeline parameters are depicted in Table

1. In particular, we model an aggressive 8-wide pipeline clocked at 4GHz that is on par with recent high performance (e.g., Intel's) microarchitectures. For the configurations assuming an ITTAGE indirect predictor, we assume the baseline predictor, i.e. the table which does not use global history as index, is merfed with the BTB.

We point out that *gem5* does not simulate the 2-level overriding branch prediction scheme depicted in Figure 3, but assumes a single branch predictor level that is able to predict in a single cycle. Given the sizes of the predictors we simulate, this is unrealistic. However, this does not impact the fairness of the performance comparisons we present as all configurations benefit from this optimistic assumption.

We simulated the SPECCPU2006 benchmark suite under a full Linux operating system (3.16.0-rc6). GCC 4.9.3 was used to compile benchmarks, except 416.gamess where GCC 4.7.3 was used. The baseline flags were: *-O3 -static -march=armv8-a -fno-strict-aliasing*. We uniformly gathered 10 checkpoints for each benchmark. For each checkpoint, we simulate the 50M instructions preceding the checkpoint start to warm up the processor's caches and different predictors. Then, we collect statistics for the next 100M committed instructions.

Note that depending on the experiment, we do not report results for all SPECCPU2006 benchmarks. Specifically, we found that only 6 benchmarks are sensitive to the presence of a dedicated indirect target predictor. Similarly, 14 benchmarks are sensitive to the presence of a dedicated memory dependency predictor.

6.2 MDP with TAGE-MDP-TAGE

Figure 6 shows the relative IPC versus blind speculation for our two unified schemes as well as the Alpha 21264-like predictor and Store Sets. Generally speaking, TAGE-MDP-TAGE *Coarse* has the same behavior as the 21264's predictor, with a noticeable improvement in *povray*, hinting that explicit path information is beneficial to memory dependency prediction. By being able to precisely identify producing stores, TAGE-MDP-TAGE *Precise* attains a performance level very close to that of Store Sets, knowing that only the 7 most recent older stores can be reached precisely. Regardless, one has to remember that these numbers are obtained at close to zero-storage overhead, while even the 21264-like predictor requires 1KB of storage. In the remainder of the paper, only TAGE-MDP-TAGE *Precise* is used.

6.3 ITP with TAGE-IT-BTB

Figure 7 pits TAGE-IT-BTB against a very large IT-TAGE predictor. In the SPECCPU2006 suite, we found only 6 benchmarks featuring enough indirect branches to be sensitive to the presence of a dedicated indirect target predictor, *perlbench* being the most impacted. In 4 of those benchmarks, TAGE-IT-BTB performs similarly to ITTAGE, at no storage overhead and even when a 5-cycle bubble is inserted for A2 lookups. In *gcc*, and *povray*, TAGE-IT-BTB actually performs better, which can be explained by the difference in structure (7 tagged components vs. 12 tagged components)

Table 1: Simulator configuration overview. *not pipelined.

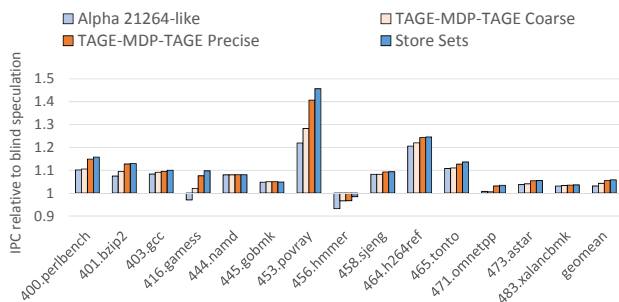| Front End	L1I 8-way 32KB, 1 cycle, 128-entry ITLB 32B fetch buffer, 8-wide fetch (15 cycles min. mis. penalty for indirect and conditional branches) 8-wide decode (5 cycles min. mistarget penalty for direct branches); 8-wide rename
BPred.	<u>Conditional</u>: TAGE 1+12 components [19] 16K(base bi-modal)+**15K/3.75K**(tagged) entry total (**32KB/11KB storage**) <u>Returns</u>: 32-entry RAS [1] <u>Targets</u> (storage in addition to cond. pred.): **Large Baseline** – 2-way 8K-entry BTB (70KB storage) **Large ITTAGE** – ITTAGE 1+7 components: 2-way 8K-entry base (BTB) + 7K-entry history indexed tables [16, 37] (134KB storage) **Large Omnipredictor** – TAGE-IT-BTB predictor + 2-way 8K-entry BTB (70KB-storage) **Small Baseline** – 2-way 2K-entry BTB (17.5 KB storage) **Small ITTAGE** – ITTAGE 1+7 components: 2-way 2K-entry base (BTB) + 7K-entry history indexed tables (82 KB storage) **Small Omnipredictor** – TAGE-IT-BTB predictor + 2-way 2K-entry BTB (17.5KB-storage)
Execution	192-entry ROB, 60-entry IQ unified, 72/48-entry LQ/SQ (Store To Load Forwarding lat. 4 cycles), 235/235 INT/FP pregs 8-issue, 4ALU(1c) including 1Mul(3c) and 1Div(25c*), 3FP(3c) including 1FP-Mul(3c) and 1FPDiv(11c*), 2Ld/Str, 1Str Full bypass; 8-wide retire
MDP	**Baseline:** Blind speculation (loads are predicted to never depend on older stores) **Improvement** – 8K-entry PC-indexed *store-wait* bitvector [23] (**1KB storage**), cleared every 30K access **Store sets** – 2K-SSID/1K LFST Store Sets, not rolled-back on squash [24] (**3.75KB storage**), cleared every 30K access **Omnipredictor** – TAGE-MDP-TAGE predictor (**no storage**), u reset every 512K updates
Caches	L1D 8-way 32KB, 4 cycles load-to-use, 64 MSHRs, 2 load ports, 1 store port, 64-entry DTLB, Stride prefetcher (degree 1) Unified private L2 16-way 256KB, 12 cycles, 64 MSHRs, no port constraints, Stream prefetcher (degree 1) Unified shared L3 24-way 6MB, 21 cycles, 64 MSHRs, no port constraints, Stream prefetcher (degree 1) All caches have 64B lines and LRU replacement
Memory	Dual channel DDR4-2400 (17-17-17), 2 ranks/channel, 8 banks/rank, 8K row-buffer, tREFI 7.8us; Min. Read Lat.: 36 ns. Average: 75 ns.

Figure 6: Using TAGE to store MDP information: IPC relative to blind speculation (loads never wait on older stores to execute) for SPECCPU2006 benchmarks sensitive to MDP. All configurations feature ITTAGE for ITP.
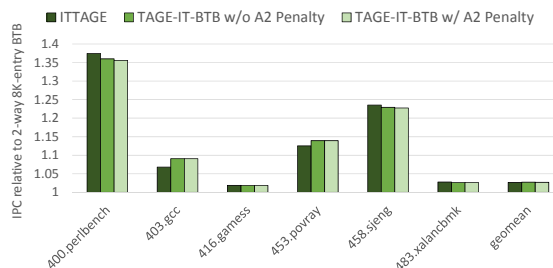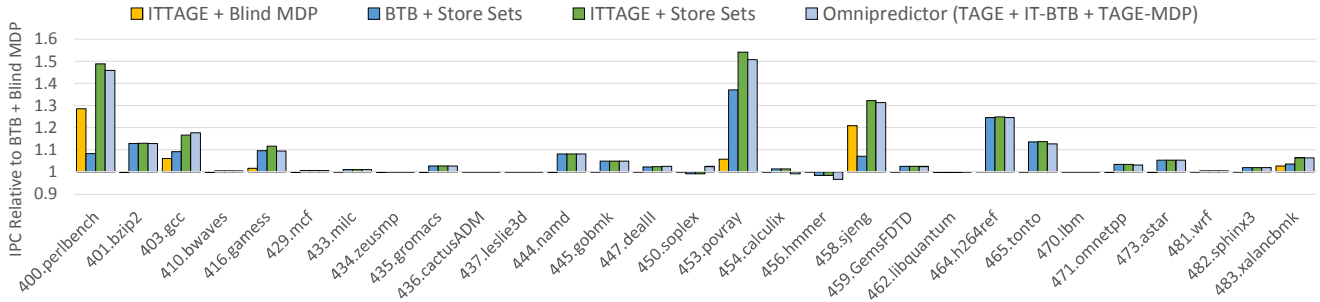

Figure 7: IPC normalized to a 2-way 8K-entry BTB when using a large ITTAGE [16] predictor and a unified scheme, respectively with and without taking the penalty of doing a second BTB read to retrieve the A2 entry. Only SPECCPU2006 benchmarks impacted by indirect target prediction are shown. All configurations feature a Store Sets for MDP.

and history management (different number of bits are pushed in the global history depending on the branch type in ITTAGE [37]).[1] However, in *perlbench* and *sjeng*, TAGE-IT-BTB cannot keep up with the very large ITTAGE predictor, yet is able to improve performance significantly as the gap with ITTAGE is small:

---
[1]Using a very large 50K-entry ITTAGE predictor [37] closes the performance gap.

(a) 15K tagged entries TAGE branch predictor and 2-way 8K-entry BTB
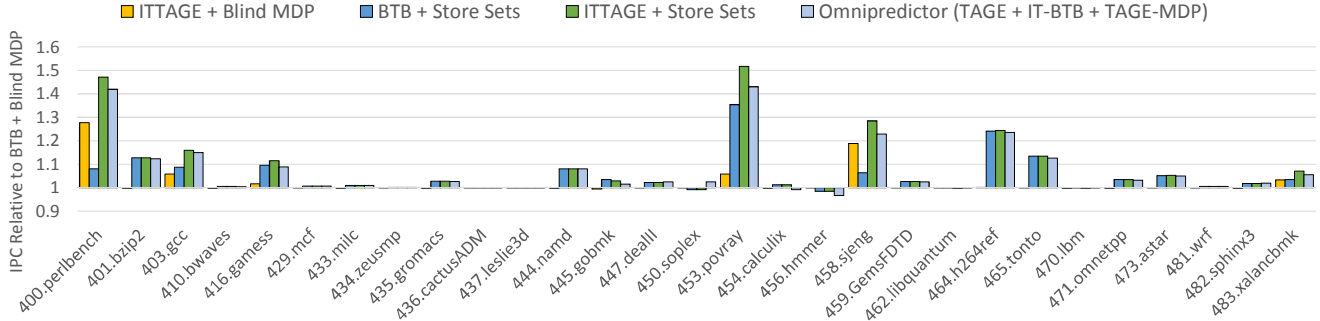
(b) 3.75K tagged entries TAGE branch predictor and 2-way 2K-entry BTB

Figure 8: Performance achieved by the omnipredictor and other ITP/MDP schemes, relative to a baseline with a BTB and blind MDP.

35.5% vs. 37.4% speedup in *perlbench* and 22.7% vs. 23.5% speedup in *sjeng* respectively.

6.4 The Omnipredictor

Performance.

Figure 8a shows the performance improvement brought by several implementations of MDP and ITP over a baseline with blind memory dependency speculation and a BTB only. Clearly, the *omnipredictor* provides performance on par with the state-of-the-art (ITTAGE + Store Sets), although performance is slightly lower in *perlbench*, *gamess*, *povray*, *sjeng* and *tonto*. Nonetheless, one has to consider that these improvements are obtained at close to no storage overhead and no complexity increase in the prediction lookup phase.

On Figure 8b, we considered a much smaller TAGE branch predictor (3.75K tagged entries instead of 15K, but still 12 tagged components, around 11KB) and BTB (2-way 2K-entry instead of 8K-entry, around 17.5KB). Although the trend is vastly similar, the gaps between ITTAGE + Store Sets and the *omnipredictor* are bigger, which is a direct consequence of an increasing competition for the predictor entries. However, in that limited hardware configuration, it is likely that even less storage could be dedicated to discrete memory dependency and indirect target predictors, rendering the *omipredictor* even more appealing.

Impact on Branch Misprediction rate.

Figure 9 shows branch – direction, direct and indirect target – mispredictions per kilo instructions (MPKI), for different ITP schemes, including TAGE-IT-BTB (without TAGE-MDP), and the *omnipredictor*. Specifically, we observe that in *perlbench* and *sjeng*, where TAGE-IT-BTB was not able to perform as well as ITTAGE in Figure 7, the number of indirect target MPKI is slightly higher with TAGE-IT-BTB and the *omnipredictor* than with ITTAGE, although much lower than with a BTB only. Additionally, in *sjeng*, the number of direct target MPKI increases very slightly when using TAGE-IT-BTB and the *omnipredictor*, as a result of indirect branches occupying more BTB entries. Conversely, in *gcc* and *povray*, we can observe that indirect target MPKI is lower with TAGE-IT-BTB, explaining its higher performance in Figure 7.

Overall, given the size of the TAGE predictor, the direction misprediction rate remains mostly unchanged when dedicating TAGE entries to memory dependency and indirect target prediction. Marginal increase can be observed in *gcc*, *gobmk*, *povray*, *sjeng*, and *xalancbmk*.

7. VARIABLE LENGTH ISAS AND LIMITED WIDTH PREDICTORS

So far in this paper, the *omnipredictor* delivers a prediction per instruction word. This exploits the fixed instruction length defined by the ISA, i.e., the PC of
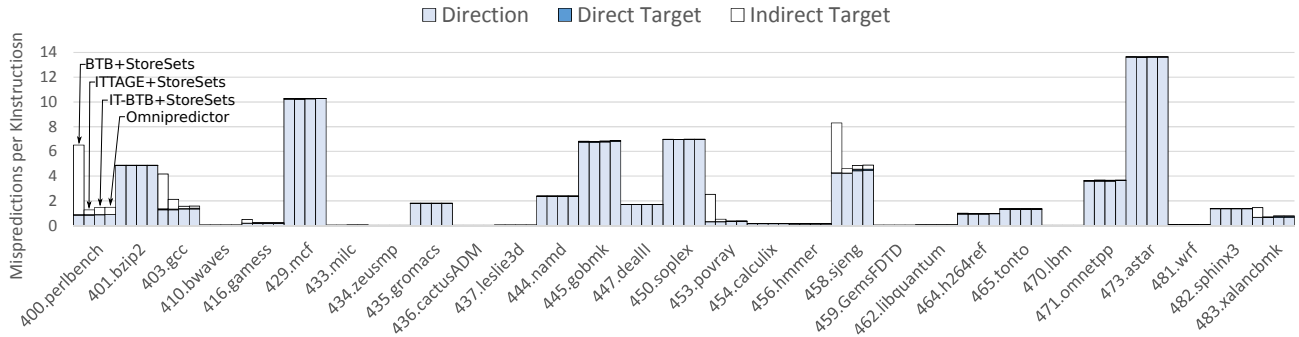
Figure 9: Branch MPKI (mispredictions per kilo instructions) using a 15K-tagged entry TAGE predictor and 2-way 8K-entry BTB. Mispredictions caused by the RAS are not shown as they are extremely rare.

an instruction does not depend on the size of the previous instruction. This is not the case for a variable instruction length ISA such as x86, since the size of the instruction may be as small as one byte, and as large as fifteen bytes. Moreover, even for fixed instruction length ISAs, predicting every instruction as it was a branch may appear as overkilling. However, to maximize instruction fetch bandwidth, one should be able to fetch across several non-taken branches per cycle. Thus, independently of the ISA, the conditional branch predictor and the BTB must be able to deliver several predictions per cycle to achieve fetching across non-taken branches. This can be addressed if the predictors (condition and BTB) deliver multiple predictions for a single instruction address block. However, this constrains the instruction block as the maximum number of non-taken branches per block must be high enough not to be an instruction fetch bandwidth limiter, particularly when resuming from a branch misprediction. A similar and even more stringent constraint comes from the memory dependency predictor that must be able to predict the dependencies for all the load/stores in the instruction fetch block. If two separate structures are used to predict branches and memory dependencies, the instruction fetch block will have to respect the two constraints, a maximum of b branches and a maximum of l loads per block.

The general principles of predicting memory dependencies through a TAGE-like predictor is ISA-agnostic. Therefore the MDP-TAGE predictor can be implemented for a variable length ISA. Such a MDP-TAGE predictor and a TAGE predictor can be packed in a single hardware predictor delivering $b+l$ predictions per cycle instead of b and l respectively. This would allow to relax the constraint on the maximum number of branches and loads per instruction block.

At the same time, predicting indirect targets could also be considered with the TAGE predictor and a block-based BTB. However, the benefit of being able to predict a large number of direct branch targets is unclear with variable instruction length ISA: there is limited instruction fetch bandwidth benefit to be gained from fetching instruction blocks featuring more than 3 consecutive not-taken branches. Thus, the design proposed for the indirect branch predictor in Section 5 could still be adapted, but at the cost of an increase of number of pipeline bubbles in the front end due to a high number of A2 accesses to the BTB.

8. CONCLUSION & FUTURE WORK

Performance on wide-issue superscalar processors is highly dependent on the instruction fetch front end as well as the memory dependency predictor. In the context of fixed instruction length ISAs, some high-end designs implement a conditional branch predictor producing a prediction per fetched instruction. Ideally, the memory dependency predictor should also be able to predict at the same rate. We have shown that a TAGE-like scheme is efficient at predicting memory dependencies. Since an instruction cannot be both a load and a branch, conditional prediction and memory dependency prediction can be packed together in the same hardware predictor. Similarly, modern superscalar processors must also support highly accurate indirect branch prediction. Therefore, we have proposed an indirect branch target predictor that also leverages existing structures: the TAGE predictor as well as the BTB.

Packing the three predictors – conditional branch direction, memory dependency and indirect branch target – in the *omnipredictor* allows to reach a **similar level of performance** as when using three discrete state-of-the-art predictors, but at a much lower hardware cost since memory dependency prediction and indirect branch prediction are performed at **almost zero storage overhead**.

While in our study, we have considered fixed instruction length ISAs, applying the principles of the *omnipredictor* could also be considered for variable instruction length ISAs such as x86.

Future work might consider further unifying other speculation mechanisms, such as criticality prediction [39, 40, 41], hit-miss prediction [42, 43] and the already mentioned speculative memory bypassing [31, 33], in the *omnipredictor*.

9. REFERENCES

[1] D. R. Kaeli and P. G. Emma, "Branch history table prediction of moving target branches due to subroutine returns," in *Proceedings of the International Symposium on Computer Architecture*, pp. 34–42, 1991.

[2] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the International Symposium on Computer Architecture*, pp. 135–148, 1981.

[3] P.-Y. Chang, E. Hao, and Y. Patt, "Target prediction for indirect jumps," in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 274–283, 1997.

[4] J. K. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," in *IEEE Comput. Mag.*, pp. 6–22, 1984.

[5] T.-Y. Yeh and Y. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 124–134, 1992.

[6] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the international symposium on Microarchitecture*, pp. 51–61, 1991.

[7] D. Jiménez, "Fast path-based neural branch prediction," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, dec 2003.

[8] D. Jiménez, "Piecewise linear branch prediction," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, june 2005.

[9] A. Seznec, J. S. Miguel, and J. Albericio, "The inner most loop iteration counter: a new dimension in branch history," in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pp. 347–357, 2015.

[10] S. McFarling, "Combining branch predictors," TN 36, DEC WRL, June 1993.

[11] 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*, pp. 67–76, 2000.

[12] S.-T. Pan, K. So, and J. T. Rahmen, "Improving the accuracy of dynamic branch prediction using branch correlation," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 1992.

[13] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge, "Analysis of branch prediction via data compression," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, 1996.

[14] A. Seznec, "Analysis of the o-geometric history length branch predictor," in *Proceedings of the International Symposium on Computer Architecture*, pp. 394–405, 2005.

[15] A. N. Eden and T. Mudge, "The yags branch prediction scheme," in *Proceedings of the international symposium on Microarchitecture*, pp. 69–77, 1998.

[16] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction Level Parallelism*, vol. 8, pp. 1–23, 2006.

[17] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the alpha EV8 conditional branch predictor," in *Proceedings of the International Symposium on Computer Architecture*, pp. 295–306, 2002.

[18] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the International Symposium on High-Performance Computer Architecture,*, pp. 197–206, 2001.

[19] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of International Symposium on Microarchitecture*, pp. 117–127, 2011.

[20] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Improving prediction for procedure returns with return-address-stack repair mechanisms," in *Proceedings of the International Symposium on Microarchitecture*,

[21] E. Rohou, B. N. Swamy, and A. Seznec, "Branch prediction and the performance of interpreters: Don't trust folklore," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 103–114, 2015.

[22] A. Moshovos, *Memory dependence prediction*. PhD thesis, University of Wisconsin-Madison, 1998.

[23] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[24] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the International Symposium on Computer Architecture*, pp. 142–153, 1998.

[25] S. Subramaniam and G. H. Loh, "Store vectors for scalable memory dependence prediction and scheduling," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, pp. 65–76, 2006.

[26] K. Driesen and U. Hölzle, "The cascaded predictor: Economical and adaptive branch target prediction," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 249–258, 1998.

[27] K. Driesen and U. Hölzle, "Multi-stage cascaded prediction," in *European Conference on Parallel Processing*, pp. 1312–1321.

[28] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, "Virtual program counter (VPC) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1153–1170, 2009.

[29] J. A. Joao, O. Mutlu, H. Kim, R. Agarwal, and Y. N. Patt, "Improving the performance of object-oriented languages with dynamic predication of indirect jumps," in *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 80–90, 2008.

[30] S. Subramaniam and G. H. Loh, "Fire-and-forget: Load/store scheduling with no store queue at all," in *Proceedings of the international symposium on Microarchitecture*, pp. 273–284, 2006.

[31] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *Proceedings of the International symposium on Microarchitecture*, pp. 235–245, 1997.

[32] G. S. Tyson and T. M. Austin, "Improving the accuracy and performance of memory communication through renaming," in *Proceedings of the International Symposium on Microarchitecture*, pp. 218–227, 1997.

[33] T. Sha, M. M. K. Martin, and A. Roth, "NoSQ: Store-load communication without a store queue," in *Proceedings of the International Symposium on Microarchitecture*, pp. 285–296, IEEE Computer Society, 2006.

[34] A. Moshovos and G. S. Sohi, "Read-after-read memory dependence prediction," in *Proceedings of the International Symposium on Microarchitecture*, pp. 177–185, 1999.

[35] H. Bouzguarrou, "US patent 2016/0306632A1 "Branch prediction"," *https://patents.google.com/patent/US20160306632A1/en*, 2015.

[36] B. Sinharoy, J. Van Norstrand, R. Eickemeyer, H. le, J. Leenstra, D. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. Fernsler, "Ibm power8 processor core microarchitecture," vol. 59, 01 2015.

[37] A. Seznec, "A 64-kbytes ittage indirect branch predictor," in *JWAC-2: Championship Branch Prediction*, 2011.

[38] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[39] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, "Dynamic prediction of critical path instructions," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, pp. 185–195, 2001.

[40] E. Tune, D. M. Tullsen, and B. Calder, "Quantifying instruction criticality," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 104–113, 2002.

[41] B. A. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *Proceedings of the International Symposium on Computer Architecture*, pp. 74–85, 2001.

[42] R. E. Kessler, E. J. Mclellan, and D. A. Webb, "The Alpha 21264 microprocessor architecture," in *Proceedings of the International Conference on Computer Design*, pp. 90–95, 1998.

[43] A. Yoaz, R. Erez, M.and Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *Proceedings of the International Symposium on Computer Architecture*, vol. 27, pp. 42–53, 1999.