Storage-Free Memory Dependency Prediction

Arthur Perais and André Seznec
INRIA/IRISA
{arthur.perais,andre.seznec}@inria.fr

Abstract—Memory Dependency Prediction (MDP) is paramount to good out-of-order performance, but decidedly not trivial as a all instances of a given static load may not necessarily depend on all instances of a given static store. As a result, for a given load, MDP should predict the exact store instruction the load depends on, and not only whether it depends on an inflight store or not, i.e., ideally, prediction should not be binary. However, we first argue that given the high degree of sophistication of modern branch predictors, the fact that a given dynamic load depends on an inflight store can be captured using the binary prediction capabilities of the branch predictor, providing coarse MDP at zero storage overhead. Second, by leveraging hysteresis counters, we show that the precise producer store can in fact be identified. This embodiment of MDP yields performance levels that are on par with state-of-the-art, and requires less than 70 additional bits of storage over a baseline without MDP at all.

Index Terms—Memory dependency prediction, branch prediction space-efficiency

1 Introduction & Motivation

Out-of-order microprocessors can be found in many devices, from smartphones to supercomputers. As a result, their behavior is reasonably understood. Out-of-order processors execute instructions as soon as their data dependencies are satisfied, irrespective of program order. Moreover, given the usual presence of several independent dependency chains within sequential programs, OoO processors are generally able to execute several instructions per cycle whereas in-order processor would stall due to the inability to ”lookahead” the instruction stream.

It follows that OoO processors should implement a mechanism to determine if a given instruction has all its data dependencies satisfied. Unfortunately, while register dependencies can be determined easily as they are encoded in the instruction word, memory dependencies cannot be determined precisely until the addresses of inflight memory instructions are resolved. Without support, this can become a major impediment to performance. For instance, a first naive solution is to forbid load from executing until all previous stores have computed their address, at the cost of lowering Instruction Level Parallelism (ILP). A second naive solution is to always allow loads to execute as soon as their register dependencies become satisfied, at the cost of requiring recovery when a load indeed executed before an older store to the same address.

To tackle this limitation, Memory Dependency Prediction (MDP) was proposed [5]. Ideally, the role of MDP is to determine, for a given load, which older 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 hardware implementation of MDP can be found in the Alpha 21264 [4]. 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 [2] and Subramaniam and Loh [13]. However, all these contributions introduce hardware structures to perform speculation.

In parallel to MDP propositions, branch predictors have become more and more sophisticated, e.g., TAGE and perceptron [3], [9]. Branch predictors are also critical to the performance of modern out-of-order processors. Their absence would force the processor to stall on each branch, waiting for it to be resolved without being able to fetch more instructions from memory. Contrarily to state-of-the-art memory dependency predictors, branch predictors are binary predictors, i.e., they only predict whether a branch will be taken or not taken. Nonetheless, binary prediction is sufficient to predict whether a dynamic load has a memory dependency or not, which is already a valuable information. Moreover, because they leverage past history (e.g., global/local branch history) to guide their predictions, modern branch predictors would be able to differentiate instances of a given static instruction that have memory dependencies from those that do not.

Consequently, we advocate for a unified predictor to reduce the overall storage cost of speculation in out-of-order processors. In this paper, we depict the modifications required in the branch predictor to support such unification, as well as two simple mechanisms to link loads and stores if they have been predicted as having a memory dependency by the unified predictor.

2 Related Work

MDP to improve scheduling was first proposed by Moshovos [5] and refined in [2], [12], [13]. Other contributions focused on speculatively bypassing the source of a store to the destination of a corresponding load to increase performance [6], [7], [11], [14].

Figure 1 shows the performance of different MDP schemes implemented in an aggressive out-of-order proces-

Manuscript submitted: 04-Oct-2016. Manuscript accepted: 25-Oct-2016. Final manuscript received: 31-Oct-2016

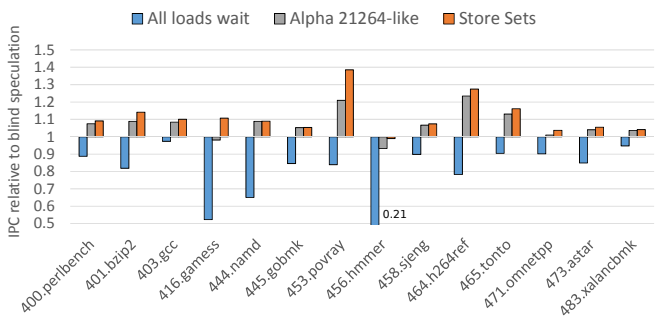

Fig. 1. The impact of memory dependency prediction: IPC relative to blind speculation (loads never wait on older stores to execute) for SPEC2006CPU benchmarks sensitive to MDP.

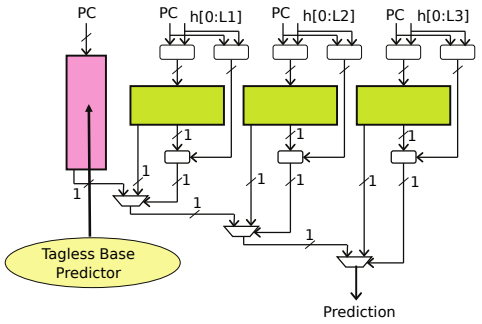

Fig. 2. 1+3 TAGE predictor synopsis, from [9]

sor (later detailed in Table 1) relative to blind speculation (i.e., loads never wait for older stores to execute). It is quite clear 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 [4] (a bitvector informing if a given load has to wait for all older stores to execute before issuing 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 [2] is clearly the best performer as it is able to precisely link dynamic loads with dynamic stores. However, Store Sets is a two level scheme that requires stores to explicitly invalidate themselves in the structure when they issue. In parallel, memory instructions access the structure at Dispatch. Thus, in high-end processors able to compute several store addresses and dispatch several instructions per cycle, pressure on the predictor may be very high. Store Sets also requires more storage than the 21264-like predictor: 3.75KB vs. 1KB in our study, assuming 10-bit sequence numbers are stored in the LFST of Store Sets.

3 A Unified Branch and Memory Dependency Predictor

3.1 The TAGE Branch Predictor

Although recent iterations feature many subcomponents (loop predictor, local component), state-of-the-art branch predictors are built on top of TAGE. As a result, to keep complexity reasonable when discussing our proposition, we only consider a basic TAGE branch predictor (i.e., without extensions) [9].

TAGE is a global history predictor featuring several partially tagged tables that are backed by a direct-mapped bimodal predictor. The N partially tagged tables are accessed using N hashes of the branch PC, the global branch history and the global path history (LSB of branch targets). 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 2.

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. In particular, entries of partially tagged components feature a 3-bit saturating counter representing the prediction, a *useful* bit used by the replacement policy, and a partial tag. Bimodal entries only feature a 2-bit saturating counter.

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 has 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 2-bit *useful* counter (u) that decides whether an entry can be replaced during allocation. u counters are periodically reset. We refer the reader to [9] for a detailed description of how TAGE operates.

3.2 Using TAGE to Predict Memory Dependencies

Upon detecting a memory order violation, a TAGE entry is allocated for the faulting load if there was no hit at prediction time. As a first step, because hitting in a tagged table is sufficient to indicate whether an instruction has a memory dependency or not, the fields in the entry do not need to be explicitly set to specific values. The exception is the u bit which is set to 1, as we use to it control whether a prediction is actually used or not.

3.2.1 *Forgetting Predictions*

Moreover, to perform well, existing memory dependency predictors [2], [4] are able to forget predictions to mitigate the fact that entries are allocated on ordering faults, but not updated when there is no dependency. The same behavior can be emulated by using the u counter to control prediction, i.e., if there is a match and u is not zero, a memory dependency prediction is made. However, if it is 0, no prediction is

made. Since *u* counters are periodically reset, this allows the predictor to forget predictions without changing circuitry. However, the interval at which *u* counters are reset is tuned for branch prediction and is quite high in [9] (reset every 512K update), whereas the interval yielding the best performance for Store Sets (in our framework) is much smaller (clear every 30K dispatched memory instructions).

As a result, in addition to using the already present *u* counters, we propose to monitor load instructions that were predicted as dependent on a previous store during their life in the pipeline. In particular, we record whether they read their data from the SQ or from the cache. At Commit, if data came from the cache (i.e., they were – most likely – not dependent on an older store), the entry that provided the prediction in TAGE is deallocated (the u bit and the tag are reset) with a low probability that we fix to $\frac{1}{256}$, but that could be set dynamically. In other words, we perform both "active" and "passive" update whereas previous schemes preferred "passive" update only.

4 Linking Consuming Loads with Producer Stores

4.1 Imprecise Linking – *Unified Coarse*

A first possibility is to implement 21264-like memory dependency prediction using the TAGE branch predictor to record the *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 TAGE can issue as soon as their operands become ready. This emulates an Alpha 21264-like predictor that considers path information.

4.2 Precise Linking – *Unified Precise*

Our scheme performs a binary prediction for memory instructions. However, TAGE, as depicted in [9], implements 3-bit hysteresis counters in tagged tables. Therefore, we can encode 8 different status for a load that hits in the branch predictor.

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 should be marked dependent on the 4th older store. This is achieved by implementing a 7-entry FIFO of sequence numbers (around 70 bits) 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, the cost of doing so is limited. We also point out that the same can be envisioned with 2-bit hysteresis counters, although more loads would be marked as dependent on all older stores.

5 Evaluation Methodology

5.1 Simulation Infrastructure

We evaluate our unified prediction mechanism through cycle-level simulation on the gem5 simulator [1], using 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.

TABLE 1
Simulator configuration overview. *not pipelined.

Front End	L1I 8-way 32KB, 1 cycle, 128-entry ITLB; 32B fetch buffer, 8-wide fetch; TAGE 1+12 compon. [9] 16K(base)+**15K/3.75K**(tagged) entry total 17 cycles min. mis. penalty; 2-way 8K-entry BTB, 32-entry RAS; 8-wide decode; 8-wide rename
Execution	192-entry ROB, 60-entry IQ unified, 72/48-entry LQ/SQ (STLF lat. 4 cycles), 235/235 INT/FP phys. regs; 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
MemDep	1 – 8K-entry PC-indexed *store-wait* bitvector [4], cleared every 30K access 2 – 2K-SSID/1K LFST Store Sets, not rolled-back on squash [2], cleared every 30K access 3 – TAGE branch predictor, *u* reset every 512K updates. **15K/3.75K** tagged entries.
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.

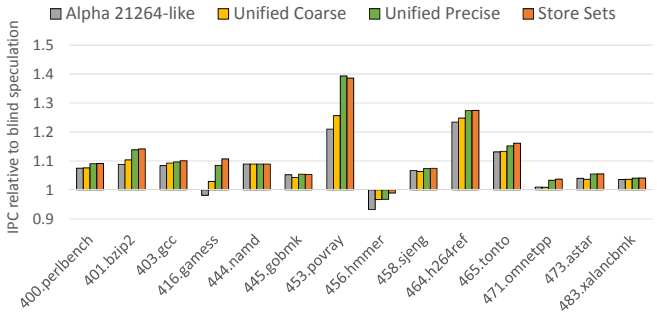

Fig. 3. Using the TAGE branch predictor to store memory dependency information: IPC relative to blind speculation (loads never wait on older stores to execute) for SPEC2006CPU benchmarks sensitive to MDP.

We simulated the SPEC CPU 2006 benchmark suite under a full Linux operating system (3.16.0-rc6). GCC 4.9.3 (Linaro GCC 4.9-2015.01-3) was used to compile benchmarks, except 416.gamess with GCC 4.7.3 (linaro-1.13.1-4.7-2013.01-20130125). The baseline flags were: *-static -march=armv8-a -fno-strict-aliasing*[1]. We uniformly gathered 10 checkpoints for each benchmark. For each checkpoint, we first simulate 50M instructions to warmup the processor's caches and different predictors. Then, we collect statistics for the next 100M committed instructions. Note that we ignore benchmarks for which there is no difference between blind speculation and all the different memory dependency prediction schemes we consider.

6 Experimental Results

Performance Figure 3 shows the relative IPC versus blind speculation for our two unified schemes as well as the

1. 464.h264ref and 482.sphinx3 required the option *-fsigned-char*.

Alpha 21264-like predictor and Store Sets, using the TAGE predictor featuring 15K tagged entries. Generally speaking, *Unified 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, *Unified Precise* reaches the same level of performance as Store Sets and even slightly outperforms it in *povray*. 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. Due to lack of space, we do not show results for a TAGE predictor that has only 3.75K tagged entries (but still a 16K-entry bimodal table), especially as speedups are similar.

Table Occupancy and Impact on Branch Prediction Accuracy Figure 4 shows, for each of the 290 checkpoints, the average percentage of tagged entries containing a memory dependency prediction (sampled every 1K updates), for 15K, and 3.75K tagged entries over 12 components. We observed that 90% of the checkpoints require 18% or less of the tagged tables for MDP in the two configurations (10% or less for the large predictor). Note that the occupancy is computed for tagged tables only, i.e., the 16K entries of the bimodal component of TAGE are all dedicated to branches. Therefore, in most cases, the impact of unifying branch and memory dependency prediction on branch prediction accuracy is very limited. Specifically, for 15K and 3.75K tagged entries respectively, we found that blind speculation has 5.46 and 6.10 average committed (only mispredictions on the correct path are counted) MPKI, Store Sets has 5.45 and 6.09 average committed MPKI, and Unified Precise has 5.45 and 6.12 average committed MPKI.

7 Complexity Increase in the Branch Predictor

Although we do not require additional storage, loads now access and update the branch predictor. However, in the context of fixed-length instruction sets, this can be addressed by reading one prediction for each instruction of the fetch block using a single access, by grouping predictions of contiguous instructions in contiguous predictor entries, as in the Alpha EV8 branch predictor [10]. In fact, modern superscalar RISC processors may already perform branch prediction in this fashion as it is a practical way to predict branches without knowing which instructions are branches, and how many there are. For variable-length instruction set, supporting multiple accesses per cycle is more complex to implement but can be achieved by grouping a statically defined number of predictions in a single entry, as proposed by Perais and Seznec for value prediction [8].

8 Conclusion & Future Work

The refinement of modern branch predictors makes them suitable for predicting outcomes different from "branch taken" and "branch not taken". In this letter, we depicted how the TAGE branch predictor could be adapted to perform memory dependency prediction at almost zero storage overhead, achieving performance on par with Store Sets [2]. Future work should aim to refine the unification of MDP and branch prediction, specifically regarding the TAGE update and allocation policy. Similarly, future work should consider unifying other speculation mechanisms (e.g., criticality-prediction, hit-miss prediction, etc.) with branch prediction.

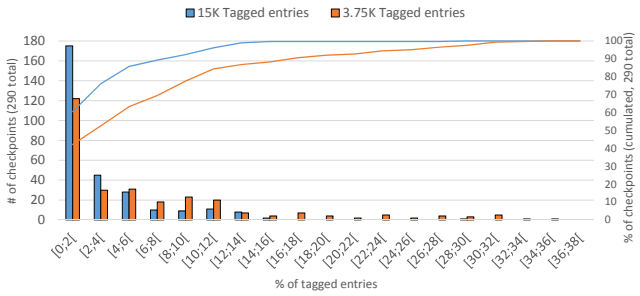

Fig. 4. Percentage of TAGE tagged entries dedicated to MDP (290 checkpoints total).

References

[1] 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*, 39(2):1–7, Aug. 2011.

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

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

[4] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

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

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

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

[8] A. Perais and A. Seznec. Bebop: A cost effective predictor infrastructure for superscalar value prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 13–25, 2015.

[9] A. Seznec. A new case for the tage branch predictor. In *Proceedings of International Symposium on Microarchitecture*, pages 117–127, 2011.

[10] 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*, pages 295–306, 2002.

[11] T. Sha, M. M. Martin, and A. Roth. Nosq: Store-load communication without a store queue. In *Proceedings of the International Symposium on Microarchitecture*, pages 285–296, 2006.

[12] 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*, pages 273–284, 2006.

[13] 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*, pages 65–76, 2006.

[14] 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*, pages 218–227, 1997.