UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1821

Leveraging Existing Microarchitectural Structures to Improve First-Level Caching Efficiency

RICARDO ALVES

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2019

ISSN 1651-6214
ISBN 978-91-513-0681-0
urn:nbn:se:uu:diva-383811

Dissertation presented at Uppsala University to be publicly examined in Sal VIII, Universitetshuset, Biskopsgatan 3, Uppsala, Monday, 26 August 2019 at 09:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Mattan Erez (Department of Electrical & Computer Engineering at The University of Texas at Austin (UTA)).

Abstract

Alves, R. 2019. Leveraging Existing Microarchitectural Structures to Improve First-Level Caching Efficiency. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1821. 42 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0681-0.

Low-latency data access is essential for performance. To achieve this, processors use fast first-level caches combined with out-of-order execution, to decrease and hide memory access latency respectively. While these approaches are effective for performance, they cost significant energy, leading to the development of many techniques that require designers to trade-off performance and efficiency.

Way-prediction and filter caches are two of the most common strategies for improving first-level cache energy efficiency while still minimizing latency. They both have compromises as way-prediction trades off some latency for better energy efficiency, while filter caches trade off some energy efficiency for lower latency. However, these strategies are not mutually exclusive. By borrowing elements from both, and taking into account SRAM memory layout limitations, we proposed a novel MRU-L0 cache that mitigates many of their shortcomings while preserving their benefits. Moreover, while first-level caches are tightly integrated into the cpu pipeline, existing work on these techniques largely ignores the impact they have on instruction scheduling. We show that the variable hit latency introduced by way-misspredictions causes instruction replays of load dependent instruction chains, which hurts performance and efficiency. We study this effect and propose a variable latency cache-hit instruction scheduler, that identifies potential misschedulings, reduces instruction replays, reduces negative performance impact, and further improves cache energy efficiency.

Modern pipelines also employ sophisticated execution strategies to hide memory latency and improve performance. While their primary use is for performance and correctness, they require intermediate storage that can be used as a cache as well. In this work we demonstrate how the store-buffer, paired with the memory dependency predictor, can be used to efficiently cache dirty data; and how the physical register file, paired with a value predictor, can be used to efficiently cache clean data. These strategies not only improve both performance and energy, but do so with no additional storage and minimal additional complexity, since they recycle existing cpu structures to detect reuse, memory ordering violations, and misspeculations.

Keywords: Energy Efficient Caching, Memory Architecture, Single Thread Performance, First-Level Caching, Out-of-Order Pipelines, Instruction Scheduling, Filter-Cache, Way-Prediction, Value-Prediction, Register-Sharing.

Ricardo Alves, Department of Information Technology, Computer Architecture and Computer Communication, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-0681-0
urn:nbn:se:uu:diva-383811 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-383811)

To my parents

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Alves R, Nikoleris N, Kaxiras S, Black-Schaffer D. ”Addressing energy challenges in filter caches”. In IEEE 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) 2017 Oct.

II Alves R, Kaxiras S, Black-Schaffer D. ”Dynamically Disabling Way-prediction to Reduce Instruction Replay”. In IEEE 36th International Conference on Computer Design (ICCD) 2018 Oct.

III Alves R, Kaxiras S, Black-Schaffer D. ”Minimizing Replay under Way-Prediction”. In Tech. Rep. 2019-003. Department of Information Technology (Uppsala University) 2019 May.

IV Alves R, Ros A, Black-Schaffer D, Kaxiras S. ”Filter Caching for Free: The Untapped Potential of the Store-Buffer”. In ACM/IEEE 46th International Symposium on Computer Architecture (ISCA) 2019 Jun.

V Alves R, Kaxiras S, Black-Schaffer D. ”Efficient Temporal and Spatial Load to Load Forwarding”. *(under submission)* 2019.

Reprints were made with permission from the publishers.

Contents

Abstract

Low-latency data access is essential for performance. To achieve this, processors use fast first-level caches combined with out-of-order execution, to decrease and hide memory access latency respectively. While these approaches are effective for performance, they cost significant energy, leading to the development of many techniques that require designers to trade-off performance and efficiency.

Way-prediction and filter caches are two of the most common strategies for improving first-level cache energy efficiency while still minimizing latency. They both have compromises as way-prediction trades off some latency for better energy efficiency, while filter caches trade off some energy efficiency for lower latency. However, these strategies are not mutually exclusive. By borrowing elements from both, and taking into account SRAM memory layout limitations, we proposed a novel MRU-L0 cache that mitigates many of their shortcomings while preserving their benefits. Moreover, while first-level caches are tightly integrated into the cpu pipeline, existing work on these techniques largely ignores the impact they have on instruction scheduling. We show that the variable hit latency introduced by way-misspredictions causes instruction replays of load dependent instruction chains, which hurts performance and efficiency. We study this effect and propose a variable latency cache-hit instruction scheduler, that identifies potential misschedulings, reduces instruction replays, reduces negative performance impact, and further improves cache energy efficiency.

Modern pipelines also employ sophisticated execution strategies to hide memory latency and improve performance. While their primary use is for performance and correctness, they require intermediate storage that can be used as a cache as well. In this work we demonstrate how the store-buffer, paired with the memory dependency predictor, can be used to efficiently cache dirty data; and how the physical register file, paired with a value predictor, can be used to efficiently cache clean data. These strategies not only improve both performance and energy, but do so with no additional storage and minimal additional complexity, since they recycle existing cpu structures to detect reuse, memory ordering violations, and misspeculations.

1. Introduction

Low latency first-level caches are essential for performance. Fast cache designs however can be energy costly, so more sophisticated designs have to be employed to mitigate the energy cost without sacrificing low latency accesses. Two popular classes of strategies to achieve this are way-predictors and filter caches. Way-predictors speculate on what cache way needs to be probed, preserving the same low latency access of a parallel access cache, while using only a fraction of the energy on a correct prediction. On incorrect predictions however, the cache needs be probed twice which increases access latency. Filter caches on the other hand, install data in a smaller buffer, allowing for an even faster access than a parallel access cache. The downside is that, for this strategy to be efficient, programs have to exhibit enough locality to offset the cost of fetching data from L1 and installing it in the filter cache. Otherwise, not only can the performance and energy benefits disappear, but even degrade compared to a parallel access L1. Another important aspect common to both these strategies is that they introduce variable latency L1 hits. Since first-level caches are tightly connected to the cpu pipeline, predictable access latency is required to correctly schedule load dependent instruction chains. Existing work on energy efficient first-level caches largely ignores this aspect in their designs.

Another synergistic strategy to caching, is to try to hide access latency instead of decreasing it, by using out-of-order execution. This approach requires complex structures and buffers that aim to keep the pipeline full for performance, and keep track of the original instruction order for correctness. One such strategy is the use of a store-buffer, that allows the processor to retire stores under a store-miss, freeing resources sooner and avoiding stalls. A second strategy is to guess load addresses and speculatively prefetch the data into the pipeline, effectively achieving a zero *load-to-use* latency for loads, i.e. a zero latency cache. Both these strategies focus on performance, and as such are not as energy efficient as they could be.

This thesis is divided in two parts, (1) analysing how effective first-level cache optimization techniques are and how to to improve them (Section 1.1); and (2) how can we pair complex out-of-order execution structures with existing pipeline buffers to opportunistically use them as a cache with minimal complexity overhead (Section 1.2).

1.1 Energy Efficient L1 Cache

The need for a fast first-level cache leads many L1 designs to read all ways in parallel to avoid the serialization overhead of accessing tags first (Figure 1.1b). However, such parallel readouts are energy inefficient, as only one of the read ways is actually used. Serializing the access and read the tags first followed by only the appropriate way [2] is not ideal since it increases access time (Figure 1.1a). Both filter caches and way-predictors try to improve energy efficiency without increasing access latency.

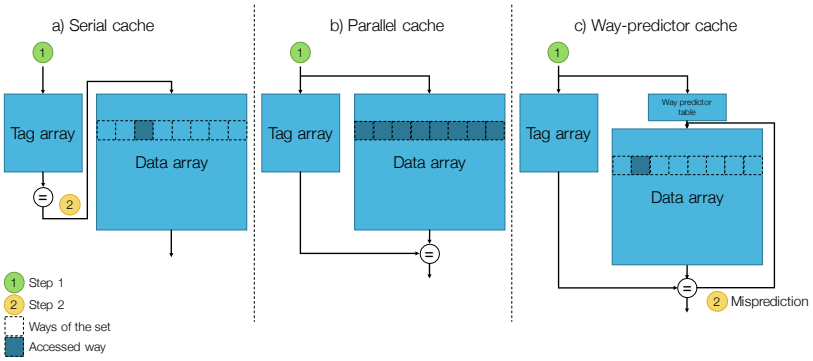

Figure 1.1. Diagram detailing the steps of accessing three different types of caches

Way-predictors reduce dynamic energy consumption by preserving the parallel access of tag and data array, but predicting the correct way and speculatively accessing that single way only (Figure 1.1c). However, on a misprediction, the latency is double that of a parallel cache, as a second access to the data array is required to fetch the correct data. Filter caches on the other hand, add a small direct-mapped L0 to reduce the number of bits read per access, while simultaneously providing shorter access times [7, 1, 4]. However, programs with poor locality will suffer from increased latency and energy due to installing data into the L0 from L1 on each L0 miss.

Filter Cache vs Way-Predictor

In terms of latency the filter cache has an advantage over way-predictors, since hits in the smaller L0 data array, return with lower latency than L1 hits (for example, 1 cycle vs. 2). Both strategies add latency on misprediction/L0 misses compared to a parallel cache, however the filter cache's lower L0 access latency reduces its miss penalty (L0+L1) compared to that of the way-predictor's (2xL1). As a result, a filter cache will perform better than a way-predictor with the same successful-prediction ratio/L0 hit ratio.

Energy-wise, a way-predictor seems to have an advantage over a filter cache, as mispredictions only require two ways to be read contrary to L0 misses that incur the additional cost of reading a whole new line from L1 and writing it

into L0 (since communication between cache levels is done at cache line granularity). For programs with low successful-prediction ratios/L0 hit ratios, the way-predictor will be the most energy efficient technique since the filter cache configuration will cause excessive data installations in L0. For programs with high enough successful-prediction ratios/L0 hit ratios, it is reasonable to assume that the way-predictor and filter cache designs are about as energy efficient since both configurations only read a single way of a set on each access. However, this assumption ignores latency and energy memory layout constraints.

While architects often treat memory arrays as monolithic 2D structures, in reality they are highly optimized hierarchies of smaller sub-arrays. The size of these sub-arrays play a very large role on the latency and energy characteristics of a cache and are a decisive factor when deciding the cache layout. This introduces constraints in caching design that affects how the above caching techniques perform and are often overlooked, such as, the minimum amount of ways a way-predictor cache can read per access. In Paper I we investigate how effective way-prediction and filter caches are at saving energy when considering SRAM layouts constraints, and how filter caches effectiveness can be improved by using a similar searching and replacement policy of a way-predictor cache.

Interaction with the Pipeline

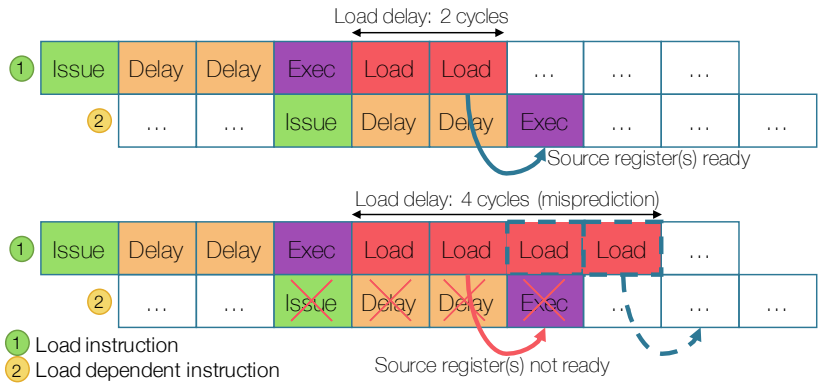

Figure 1.2. Speculatively scheduling on a pipeline with an issue-to-execute delay of 2 cycles. Scheduling of a load-dependent instruction on a correct way-prediction (top) and on a way-misprediction, which increase load latency (bottom).

Due to the complex nature of out-of-order pipelines, most processors today take several cycles from the moment an instruction is selected to execute, to the moment its execution actually starts. This latency is called *issue-to-execute delay* [17]. As a result, to keep the pipeline full, instruction scheduling needs

to happen several cycles before execution, and as such, the scheduler has to speculate when the source operands will be available [6, 12, 24]. Misspeculations will cause incorrect scheduling and force instructions to be replayed (Figure 1.2). While all the first-level cache optimisations mentioned so far improve energy and latency, they also impact hit latency depending on the prediction result. The introduction of non-deterministic L1 hit-latency influences instruction scheduling accuracy, and how that impacts performance has largely been ignored.

In Paper II we explore how prediction affects performance due to variable L1 hit-latency, and how the problem can be mitigated by having a prediction aware scheduler. Paper III extends the study for pipelines with different issue-to-execute delays, identifies replays as more detrimental to performance than higher load latency, and improves on the solution proposed by Paper II.

1.2 Data Caching in the Pipeline

In parallel with the use of caches, modern processors also take advantage of out-of-order execution to hide memory latency, through instruction level parallelism (ILP) and memory level parallelism (MLP). The pipelines of these processors typically use a large *physical registers file* (PRF), *store-queue* (SQ), *store-buffer* (SB), etc, to expose parallelism (for performance) and keep track of the original program order (for correctness).

Store-Queue/Store-Buffer

The store-queue and store-buffer are responsible for keeping track of the original order of store instructions. To maximize the use of available resources, the store-buffer (SB), which holds stores between commit and when they are written to memory, is often unified with the store-queue (SQ), which holds stores from dispatch to commit. The resulting unified SQ/SB allows for better overall utilization of the expensive CAM-FIFO needed and eliminates the cost of moving entries between separate queues.

The SQ/SB purpose is twofold: (1) to keep track of the stores' original order, so that they are committed to memory in that same order, and (2) to forward data to load instructions that address the same memory location of an uncommitted store (store-to-load forwarding), thus guaranteeing that a load always accesses the most recent value. For this reason, the queues have to store not only the addresses, but also the values of the in-flight store instructions. Moreover, they have to be probed on every load for correctness. This in essence, makes the SQ/SB an extra cache level, dedicated to dirty data.

In Paper IV we show how ineffective the SQ/SB is as a cache due to being mostly underutilized. This is the result of an over aggressive write-back policy that prioritises performance over energy efficiency. We propose a third logical

queue, the *Store-Buffer-Cache* (SBC), that extends the SQ/SB and maximises the shared physical structure (that we call S/QBC) hit-ratio. Also, by leveraging the memory dependency predictor, we are able to avoid accessing the L1 for data that is forwarded from the S/QBC, thus improving both performance and collective energy of S/QBC and L1.

Physical Register File

The physical register file (PRF), holds the results of executed instructions, so the data can be correctly forwarded to its consumers. This includes the results of load instructions, and as such, the content of some memory addresses are replicated in both the memory and in the PRF. While not required for correctness, this structure can be used to elide extra accesses to the cache for data that is already available in the pipeline (load-to-load forwarding). This makes it an extra level of cache as well. Existing register sharing techniques [5, 18, 19, 3, 13] aim to extract this locality to improve performance, but have limited success since they have to rely on heuristics such as instruction distance [16] to detect reuse.

A more radical approach to improve performance is to use value prediction (VP) [9, 8, 26, 25, 14, 15] to guess the load memory address and prefetch the data directly into the pipeline [21, 11], thus completely hiding memory latency and achieving a zero *load-to-use* latency. We call this technique, *pipeline prefetching*. While effective, this technique also significantly increases accesses to memory since every predicted load address needs to access the memory twice, once to prefetch with the predicted address, and a second to validate the prediction.

In Paper V, we explore how one can leverage the predicted load addresses, required to do pipeline prefetching, to detect potential data reuse in the PRF. By using addresses instead of instruction distance, we are able to expose more locality than a state-of-art register sharing technique. Also due to using addresses, we are able to take advantage of spatial locality as well as temporal locality, and in some cases, expose more than the total locality exposed by the load queue. By making the observation that memory ordering can be validated without replaying load instructions [20], we are also able to validate the prefetched loads without re-executing them. This allows us to, not only nullify the extra memory requests introduced by prefetching, but actually reduce the total number of memory accesses compared to a non-prefetching pipeline.

1.3 Thesis Structure

This thesis is divided in two parts. In the first part of the thesis (Section 2), we are able to improve on existing first-level caching techniques. In Paper I by improving over filter cache energy efficiency and way-prediction latency, with a hybrid of both that takes into account SRAM cache layout limitations; and in Papers II and III by better integrating these caching techniques into the pipeline with an instruction scheduler that considers the variable hit latency introduced by them.

In the second part of this thesis (Section 3) we improvise extra caches with the existing storage of an out-of-order pipeline, and reduce accesses to the first-level cache. In Paper IV we extend the SQ/SB with a new logical queue that holds dirty data written back to memory, improving SQ/SB hit ratio – a cache for dirty data; and in Paper V, we leverage predicted load addresses to detect reuse and forward clean data from the PRF, improving PRF hit ratio – a cache for clean data.

2. Improving L1 Cache Energy Efficiency

SRAMs are built as matrices of sub-arrays tied together with common data and control (clock) interconnection in the form of an h-tree. Each sub-array is a group of SRAM cells, and the width of the sub-arrays determines the minimum number of bits read per cache access. As the access time and read energy are determined by the size of the sub-array (latency through the decoders, word-lines, bitlines, and sense amps), the smaller we can make the sub-array, the faster and more efficient each access should be. However, this does not factor in the latency introduced by the associated h-tree. For a cache of a given capacity, using smaller sub-arrays requires more of them: the more sub-arrays we have, the larger and slower the h-tree must be to connect them. This means, the faster cache design is not necessarily the one with the smallest sub-array size possible. In fact, a latency optimised 32KB cache design requires 128 bits wide sub-arrays.

2.1 SRAM Cache Layout Characteristics

An L0 cache can reduce energy consumption and access time due to its smaller capacity. A small capacity cache allows the use of small sub-arrays, which read more quickly and activate fewer SRAM cells on each access, without increasing the h-tree latency. Figure 2.1 compares an optimized 32 KB 8-way associative L1 cache (128x128 bit sub-arrays) with a 4KB direct-mapped L0 cache (64x32 bit sub-arrays). The smaller size of the L0's sub-arrays means its minimum readout width is only 32 bits, which is one-fourth of the minimum readout width of the L1's sub-arrays.

However, while L0 accesses are inherently efficient because of their far smaller sub-array's readout width, the filter cache still experiences the energy cost of accessing the larger sub-arrays in L1 on each L0 miss. The way-predictor cache reads one L1 sub-array (128 bits) at a minimum, and on a miss it must read a second. Since accessing a full set requires reading 256 bits (8 words per set), this means that the way-predictor will require roughly half the energy of a parallel cache (not 1/8th as expected from the architectural description) on a correct prediction, and the same energy on a misprediction (not 2/8th).

Since the way-predictor reads out more data than the predicted word, how a set is mapped in the cache (fragmented vs contiguous) will have an impact on energy. Stripping the cache line to group together words of the set with

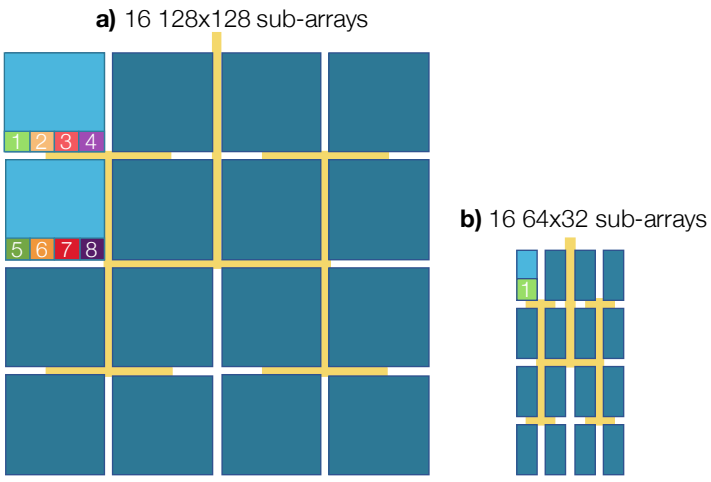

Figure 2.1. Diagram comparing sub-array size and h-tree complexity between a 32KB L1 data array and a direct-mapped 4KB L0 data array

the same index (similarly to a parallel cache shown in Figure 2.1a), the way-predictor might increase its accuracy. Even if the predicted way is wrong, the correct way might be read anyway because it shares the same sub-array. However we did not see a big enough increase in the prediction success ratio to offset the energy overhead of installing new cache lines with a fragmented mapping.

When the sub-array layout is taken into account, we see that the efficiency of the way-predictor is significantly reduced due to the need to read out 128 bits from a full sub-array (half of the sub-arrays of a parallel cache) rather than the architecturally expected 1 way (1/8th of a parallel cache). However, L0s do not suffer from this because their smaller size means they can use smaller sub-arrays with smaller minimum readout widths. Despite looking architecturally as if the L0 is the same structure as a single way of the L1, the need to optimize SRAM designs for the much larger L1 forces the use of larger sub-arrays, which increases minimum readout energy.

2.2 Paper I: Way-Prediction/Filter Cache Hybrid

L0 cache hits are faster and more energy efficient than hits on the L1 cache, due to the L0 smaller size which permits a sub-array layout with smaller readout width. However, the filter cache configuration suffers from the energy overhead of copying data from L1 into L0. To address this, in Paper I we propose the use of a directed map L0 that stores the MRU way of each set and shares the tag array with the L1. We propose MRU as the L0 installation policy for two reasons, it is effective for way-prediction [23] and already available from the

L1's replacement policy. This allows the design to be simple since the same index function can be used to index both the L0's and the L1's data array and no substantial control bits need to be added (no extra bits if the cache already tracks MRU information). The L0 size is derived from the size of 1 L1 way, since the L0 stores the MRU way of each set.

L1 accesses are still relatively energy expensive and L0 misses require the larger L1 to be probed. An access to a parallel L1 is much more energy costly then an L0 access, and a high enough L0 miss ratio could negate the energy benefits of having an L0 at all. This extra cost can be mitigated by performing a serial L1 access on an L0 miss. This guarantees that only the necessary sub-arrays will be activated but increases the access latency and in turn, the L0 miss penalty. Another alternative would be to use a way-predictor on the L1. However, the L0 filters the locality of the accesses making an MRU based predictor ineffective.

MRU-L0

With a better understanding of the how sub-array readout width affect filter caches and way-predictors efficiency, we propose the MRU-L0 cache. This cache uses a direct-mapped L0 to store the MRU way of all the sets in L1 and shares a single tag array between the L1 and L0. This design choice implies two invariants: (1) that the L1 is inclusive in relation to L0, and (2) that the L0 needs to have exactly 1/*n*th the capacity of the L1, *n* being the associativity level of L1. These invariants also entail that the L0 hit ratio will be similar to the success ratio of a way-predictor, since the way stored in the L0 (the MRU way) is the same as the way the way-predictor would speculatively access. Moreover, since the L0 has a copy of the MRU way of all the L1 sets, there is no need for a dedicated L0 tag array; the contents of L0 can be tracked with the same MRU bits in the L1's tag array.

The MRU-L0 cache (Figure 2.2) issues probes to both the tag array (which covers the L0 and L1) as well as the L0 data array in parallel. In case of an L0 hit, the request is satisfied from the L0 array, and the extra tag data is ignored. In a miss, the L1 data array is accessed using the read tag information to read only the required data. In the case of a complete miss (L0 and L1) the L1 data array is not accessed, and the data can be read from the next level in the hierarchy and installed directly into L0. When the line is later evicted from L0 it will be installed in L1.

MRU-L0 vs. Way-Prediction

By probing the L0 data array instead of L1 data array in parallel with the tag array, the critical path is now reduced. As in an L0, this means that the MRU-L0 speeds up the cache access time to a latency of an identically sized filter

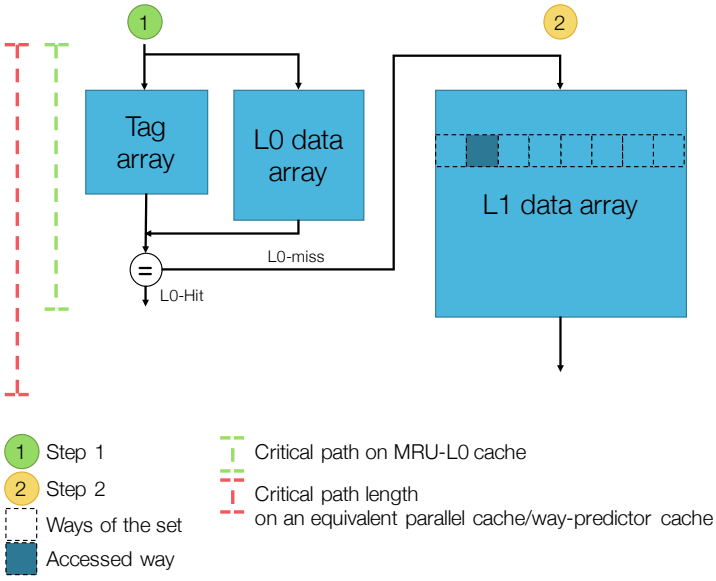

Figure 2.2. Diagram breaking down a memory access on an MRU-L0 cache

cache. On an L0 miss, the L1 data array needs to be probed. This second step is similar to a normal L1 cache access and should have the same latency.

Since the L0 array is smaller than the L1 data array, each access is not only faster but also more energy efficient. On an L0 hit, the energy spent accessing the smaller sub-array(s) of the L0 is significantly less than accessing L1 data array. On an L0 miss, the MRU-L0 is not more energy efficient, but overall (hits+misses energy) it is more efficient than a way-predictor.

MRU-L0 vs. Filter Cache

The MRU-L0 cache accesses the tag array and the L0 in parallel. In case of an L0 miss, since the MRU-L0 cache shares the tag array with L0 and L1, the location of the data in the L1 data array is already known at the time the L0 miss is detected. On the other hand, a filter cache configuration after a L0 miss is detected, needs an extra lookup in the L1 tag array to know where the required data is located. MRU-L0 thus improves L0 miss by reducing miss energy. Similarly to the L0 miss improvement, only a single lookup is required to detect an L1 miss. This improves L1 miss by making its detection faster (only one tag array probe) and less energy costly (no search in L1 required for detecting the miss).

Results

Our results show that the MRU-L0 cache reduces average energy consumption compared to a traditional filter cache by 26%. Compared to a way-predictor, the MRU-L0 cache optimization reduces energy consumption by 2% while improving performance by 6% on average.

2.3 Speculative Scheduling

Way-prediction has been considered an effective way to reduce the dynamic energy of first-level caches since its high prediction accuracy directly reduces dynamic energy (fewer ways probed) while incorrect predictions have minimal impact on performance (only a small latency increase on mispredictions). Figure 2.3 compares the success ratio of an L1 data cache way-predictor [1] with its performance impact (normalized to a parallel access cache). These results make way-predictors look promising, as latency-sensitive benchmarks tend to have a high enough way-prediction success ratio to not suffer from mispredictions, while applications with many mispredictions are not particularly latency-sensitive. However, these results ignore the impact of instruction replay (i.e., they essentially assume a pipeline with a 0-cycle issue-to-execute delay), which is not realistic for out-of-order processors.

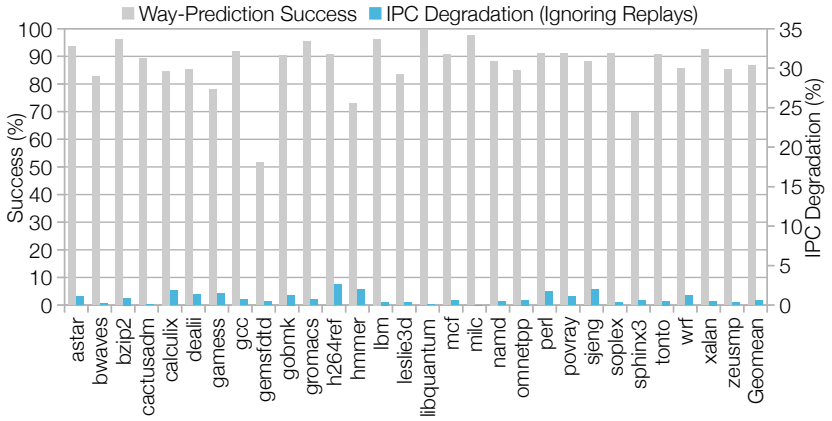

Figure 2.3. Way-prediction success ratio (higher is better) and its effects on performance ignoring replays (IPC degradation, lower is better) compared to a baseline parallel cache.

When the penalty of instruction replay is included, way-mispredictions not only increase the latency of loads, but also affect the issue stage of the pipeline

[1] While MRU-L0 is a more effective strategy than way-predictor, we chose way-prediction to evaluate the effect of variable L1 hit-latency on instruction scheduling and performance. The choice was motivated by the fact that large out-of-order cores, the cores more likely to have large distance between the issue and execution stages, are the less likely to benefit from the MRU-L0 cache performance-wise.

by forcing new instructions to be delayed due to replayed instructions being re-scheduled. This effect is shown in Figure 2.4, where we see how the previous results (Ignoring-Replays) compare to a realistic implementation with a 4 cycle issue-to-execute delay, that optimistically schedules assuming a correct way-prediction (Replay-Optimistic).

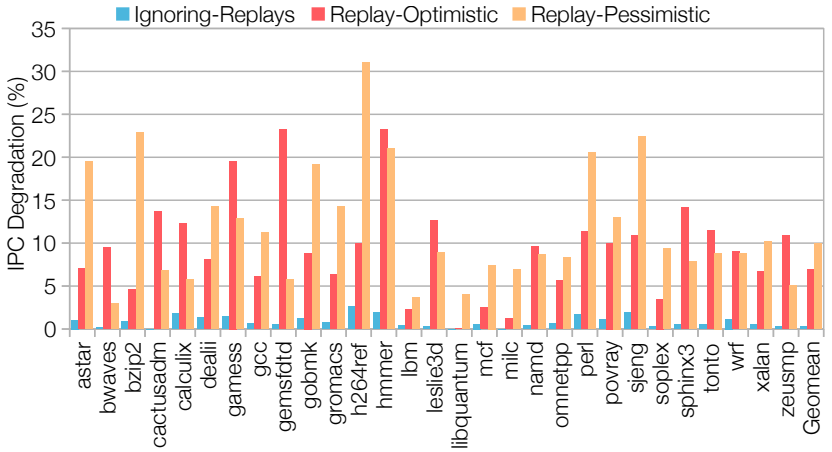

Figure 2.4. Performance impact of including replay costs in way-prediction performance analysis. Performance degradation (over a baseline parallel access cache) for way-prediction with No Replay and for an issue-to-execute-delay of 4 under Optimistic and Pessimistic scheduling (lower is better).

The difference when taking into account replays is substantial. We see that the average IPC degradation for the way-predictor is now 6.9%. This is significantly worse than the 0.5%, the value reported by previous work, when ignoring replays. The difference is particular evident in benchmarks with low way-prediction accuracy, such as *gemsfdtd*. When we ignore replays, these benchmarks are not affected by mispredictions, as they are not sensitive to load latency, but when the cost of replay is included, their high misprediction rates lead to significant performance losses due to instruction replay.

One can eliminate the need for replays by *pessimistically* scheduling dependent instructions assuming way-mispredictions (Figure 2.4, Replay-Pessimistic). This will eliminate the overhead for benchmarks with high way-mispredictions ratios that are not latency-insensitive. However, this hurts benchmarks that are sensitive to load latency such as *h264ref*. For these benchmarks, the way-predictor has a high success ratio, and the benchmarks benefit from the earlier scheduling of load-dependent instructions. Overall, a pessimist scheduling strategy produces even worse results and defeats the purpose of way-prediction.

2.4 Paper II & III: Prediction-Aware Instruction Scheduling

In order to avoid excessive replays without increasing latency for all loads, in Paper II we propose *Selective Way-Prediction*, which learns when the way-predictor is likely or not to mispredict and schedules dependent instructions accordingly. By doing so we can minimize replays without hurting performance of latency sensitive applications. To do so, we add a confidence measuring unit (CMU in Figure 2.5) that follows the same simple strategy proposed for branch predictors [22]: The CMU is a table of saturating counters, indexed by the least significant bits of the load PC, that are incremented/decremented on a correct/incorrect way-prediction. If the count is equal or greater than a threshold, we deem the way-predictor accurate for that load, and thus enable way-prediction and optimistically schedule dependent instructions. Otherwise, the way-prediction is disabled and load-dependent instructions are delayed accordingly.

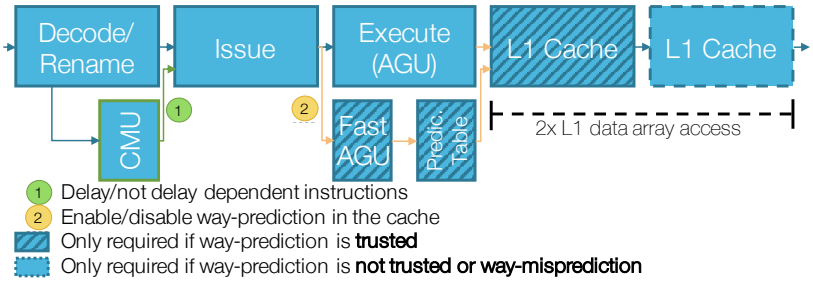

Figure 2.5. An early way-predictor confidence measurement from the confidence measuring unit (CMU) is used to control scheduling of dependent instructions and to enable or disable way-prediction.

Scheduler and Cache Modifications

Depending on the confidence for a particular load, the scheduler makes different scheduling decisions for the load dependent instructions. This information allows the scheduler to take advantage of the low load-to-use latency of high-confidence (correct) way-predictions, while avoiding paying the replay cost of overly-eager scheduling for unreliable way-predictions.

The confidence is also used to dynamically enable/disable way-prediction in the cache to improve energy efficiency. For low-confidence predictions we disable the first (speculative) data access and wait for the tag results. This avoids the extra data array probe on a misprediction, which provides the energy-efficiency of a serial cache for low-confidence loads. Alternatively, a performance-oriented optimization could execute a parallel load on low-confidence predictions to ensure low access latency at the cost of increased energy (simulations

of this approach showed little IPC benefit despite a significant energy increase, even with a perfect confidence prediction).

Biased Selective Way-Prediction

In Paper III we extend the work to include pipelines with different issue-to-execute delays. The results show that the performance impact of variable hit latency is more pronounced on deeper pipelines, as expected, but still significant on shallow ones. Moreover, we identify instruction replay as being more problematic to performance than higher load latency. Given this find, we proposed a simple change that bias the scheduler to be more conservative when guessing and scheduling instructions for correct predictions. This solution increases the amount of incorrectly scheduling instructions for a mispredcition (false negatives) but reduces the amount of incorrectly scheduling instructions for correct prediction (false positives). Despite increasing the total amount of scheduling errors, it reduces replays, which improves performance and cache energy even further.

Results

Overall, the *Biased Selective Way-Prediction* reduces the performance penalty of a standard way-predictor from 6.9% to 2.9% by reducing the percentage of instruction replays from 10% to 2.9%. As a result, the additional dynamic energy of a way-predictor over a serial cache is reduced from 8.5% to 1.9%.

3. Caching Before the First-Level Cache

3.1 Store-Queue and Store-Buffer

A common challenge in out-of-order pipelines is that stores that are ready to commit may be stalled due to cache misses or cache contention. Such delays block the ROB and may stall the pipeline. To allow stores to retire under these conditions, a *store-buffer* (SB) is used to track stores that have committed but have not yet been written back to memory. When entries in the SQ have been committed, they are moved to the SB until they are written back to memory (typically to the L1).

The store-queue (SQ) and store-buffer (SB) are generally implemented in a unified physical structure called the SQ/SB. The unified approach means that the distinction between entries in the SQ and SB is purely logical: stores that are not yet committed are in the SQ portion, and stores that are committed but not yet written to memory are in the SB portion. This allows for a more efficient implementation, as there is no need to copy between separate buffers on commit (moves simply require changing head/tail pointers as all moves are in store order) and either the SQ or the SB size can increase up to the maximum capacity. The higher utilization by sharing capacity between the SQ/SB is important as the structure is implemented as a FIFO, but requires CAM access (to allow searches by address for later loads). As a result, the cost of this structures is high, but it must be large enough to handle bursts of write misses that would otherwise stall the processor.

Too avoid increasing load latency, loads probe the SQ/SB and L1 cache in parallel. If the address matches a store in the SQ/SB (i.e. a SQ/SB hit), the data is forwarded from the youngest store that matches the address, and the in-flight L1 cache request is ignored. In addition, since L1 caches are generally physically tagged, the load address has to be translated, requiring a parallel access to the TLB as well.

Under Utilising the Store-Buffer

We do not hit much on the SQ/SB despite installing all dirty data into it. The relatively small size of the SQ/SB, combined with its aggressive eviction policy (designed to keep it as empty as possible to avoid stalls) results in a low utilization and a low hit ratio. Figure 3.1 demonstrates this low-utilization for a 56-entry SQ/SB across the SPEC2006 benchmarks. While the SQ/SB is highly-utilized (>80% full) at some point in all benchmarks, the majority of

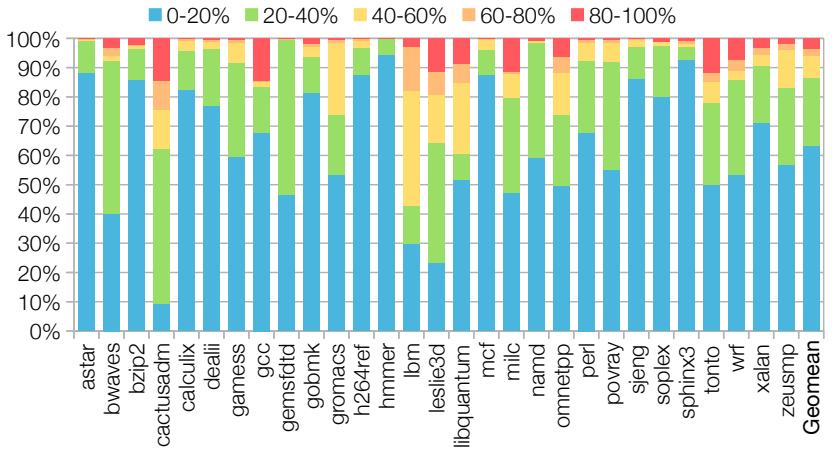

Figure 3.1. SQ/SB occupancy across benchmarks. All benchmarks have high SQ/SB utilization (>80%) at some point, while the majority of the time it experiences low utilization (<40%).

the time the buffer remains largely empty (<40% full). Indeed, the average benchmark uses 20% or less of the SQ/SB for 62% of its execution, and 40% or less for 85% of its execution.

This low utilization translates into a low hit-ratio. Figure 3.2 shows the percentage of loads that receive their data from the Standard SQ/SB (aggressively writing back data to L1) with a Skylake-like 56-entry unified SQ/SB. While some benchmarks such as *cactusadm* have substantial SQ/SB hit ratios, most applications have hit ratios around or below 10%, and the overall SPEC2006 average is only 8.1%. This is not surprising given the SB goal to be as empty as possible (i.e. to avoid pipelines stalls), but is very low for an effective cache.

The low hit ratio suggests that (1) programs are unlikely to benefit from the lower latency of data forwarded by the SQ/SB as the majority of the loads experience the longer L1 latency on SQ/SB misses, and that (2) today's approach of probing both the SQ/SB and the L1/TLB in parallel is reasonable, as most data requests will miss in the SQ/SB and have to access the L1/TLB anyway. With this SQ/SB hit ratio, serializing SQ/SB→L1/TLB accesses (to avoid accessing the L1 on SQ/SB hits) would increase the latency for the 92% of the accesses that miss in the SQ/SB and only provide an energy benefit for the 8% that hit.

To explore the potential of a SQ/SB cache, we implemented an Optimal SQ/SB (that delays writes from the SB to the L1 as long as possible without hurting performance), and the picture is different. Figure 3.2 shows that with the Optimal SQ/SB, some benchmarks have a substantial SQ/SB hit ratio, going as high as 46.6%, an increases of 4.3x over the Standard SQ/SB. On average, the hit ratio increase to 18.4% from 8.2%, a 2.3x improvement.

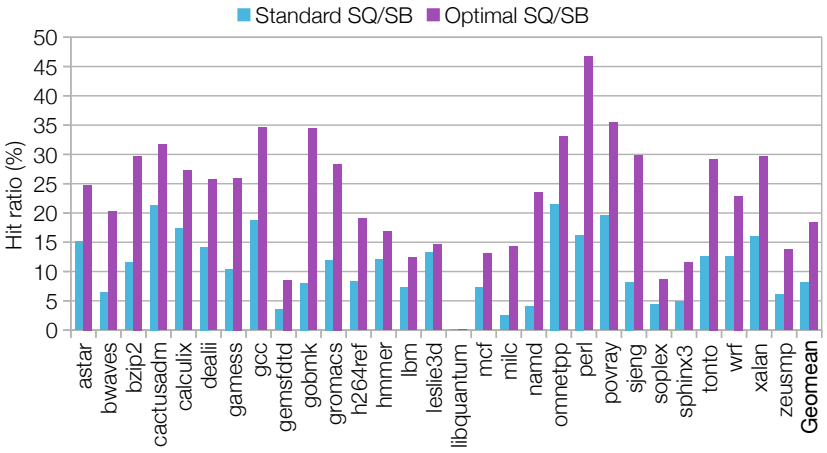

Figure 3.2. Percentage of loads that hit in the SQ/SB for: 1) A Standard SQ/SB that aggressively writes out to memory, and 2) an Optimal SQ/SB that maximizes hits by keeping values around exactly as long as possible without stalling the processor.

3.2 Paper IV: Store-Buffer as a Cache

While delaying stores in the store-buffer can increase the buffer's hit-ratio substantially, it can also degrade performance due to stalls when the SB fills from bursts of writes. In paper IV, instead of delaying stores in the SB to increase hits, stores are written to the L1 as soon as possible as a normal SB so not to hurt performance. However, a *copy* of the store is kept in a third logical queue, called *Store-Buffer-Cache* (SBC). This means that there are now three queues that share the same physical buffer, and the distinction between them is logical. The unified buffer now holds 3 types of stores, uncommitted (SQ), committed and not written to L1 (SB), and committed and written to L1 (SBC). We call this new unified structure (SQ/SB/SBC), S/QBC.

The S/QBC is able to keep the results of stores as long as possible as evictions only happen when a new entry is needed. Since the stores in the SBC were already written back to memory, evictions can be done silently and immediately whenever space is needed for new writes in the SQ (only a queue pointer update is required). This maximizes hits in the S/QBC, by keeping data around as long as there is space, without causing extra cpu stalls due to lack of available space for new stores.

To make this new buffer an effective cache, we also need to avoid accessing the a higher cache level when the data is available in the buffer, which means, the buffer needs to keep its stores coherent and deal with translations from virtual to physical addresses.

Store-Buffer-Cache Coherence

We handle coherence by taking advantage of the fact that stores are treated differently from loads in the coherence domain. Specifically, a store that is written to the L1 implies that the local L1 has *ownership* of the data block because its data is *dirty*. As a result, L1 invalidations or evictions of *clean* cache lines are *irrelevant* to the SBC, and can be ignored. However, if we lose ownership of a cache line, either through an invalidation, an eviction, or simply because of a read request from another core that forces the local cache line to downgrade to `SHARED` (and become clean), then we are no longer able to detect that the coherence actions on that line affect our SBC. In this case, one of our own prior stores has been affected by a coherence action, and if a corresponding clean copy exists in the SBC it must be stale.

Since we now have restricted the cases where we must react, we can relax the specificity of our reaction: instead of invalidating specific data in the SBC (which would require an energy expensive associative search) we can just bulk-flush all cached data in the SBC for these more restricted circumstances. In our unified S/QBC, such a bulk-flush simply requires moving the head SBC pointer to coincide with the head SB pointer. To avoid unnecessary flushes even further, we employ a coloring mechanism to track epochs of dirty data, allowing us to only flush when data from the current epoch is invalidated. Depending on how many colors we use, we need more bits (per cache line) to track the current color. To avoid this overhead as much as possible, we propose a mechanism to bulk change colors in the L1, to approximate an infinite number of colors and only requiring the addition of an extra bit to each cache line.

Store-Buffer-Cache Address Translation

A translation from virtual to physical address is required to detect possible incorrect forwardings due to synonyms, even on S/QBC hits. We can elide the TLB access on S/QBC hits since the load queue (LQ), SQ, SB and SBC hold both virtual and physical addresses [10]. A load that matches a virtual address from the SQ, SB or SBC, can copy the same physical address of the matching store entry as well: virtual-to-physical mapping is one-to-one. One-to-many mappings (homonyms) are avoided by the operating system.

A load hit on an SBC entry whose physical address has not yet been retrieved requires only a single TLB access to translate both the earlier store and the later load. This eliminates the need for a second TLB access for the load. The S/QBC can therefore ensure that all load hits will have correct physical addresses, even though they will not cause TLB accesses.

Predicting S/QBC Hits

The most straightforward way to have the S/QBC improve energy efficiency, is to serialize the access to S/QBC and the L1. However, even taking advantage of all locality available in the S/QBC, such approach would increase the latency of more than 80% of the loads. To avoid this loss of performance, while retaining the energy savings, we need to predict whether a load will hit in the S/QBC so that we can choose to disable the L1 probe, without incurring a latency penalty for loads that do not hit.

Fortunately this problem can be trivially solved by using the same memory dependency predictor required to effectively perform *speculative memory bypassing*. The predictor needs to predict if a load instruction will have its data forwarded from the SQ/SB, so the instruction scheduler issues instructions in an order that does not violate memory ordering. The predictor is able to correctly predict S/QBC hits and misses with a 93.6% accuracy, and therefore we can use it to enable/disable the parallel L1 probe.

Results

Our results show that the S/QBC is able to take advantage of the existing storage capacity of the SQ/SB to reduce dynamic L1 and TLB energy by 11.8% with no performance impact (indeed, a marginal 1.5% improvement). The overall design has essentially no overhead (0.2% additional L1 storage for one additional dirty bit per line and one additional S/QBC tail pointer) and does not increase data movement energy (moving entries from the SB to the SBC is only a logical pointer update).

3.3 Pipeline Prefetching

A pipeline prefetcher, uses a value predictor (VP) to guess the address of a load instead of the value itself, *prefetches* the value from L1, and feeds it directly to the instruction that consumes it. This effectively makes the predicted load have a zero *load-to-use* latency. Validation of the prediction remains similar to traditional value predictors and requires the memory to be accessed again. This means that an address predictor requires two memory accesses per load, one to prefetch the data and a second to validate the prediction, which negatively impacts pressure to the L1 ports and energy. Figure 3.3 shows the percentage of loads predicted and executed twice, i.e. the percent increase in memory accesses compared to a traditional non-prefetching pipeline. Since value prefetch will be most successful when as many predictions as possible can be made, this advocates for techniques to increase prediction coverage, which in turn will aggravate the problem of pressure to the L1 ports and eventually hurt performance.

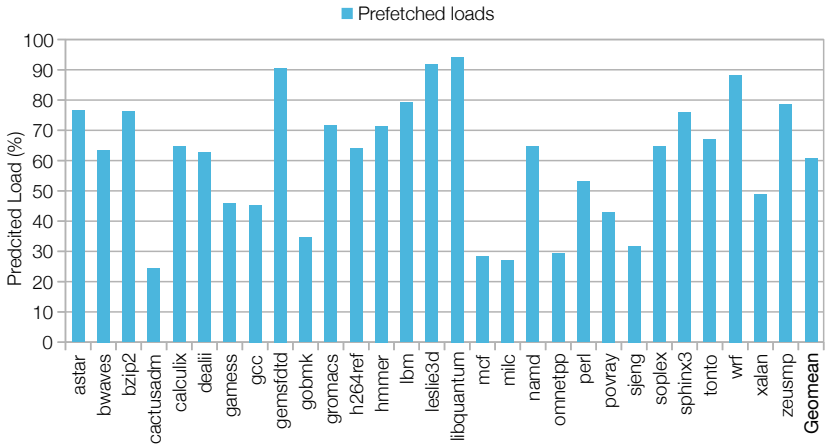

Figure 3.3. Ratio of loads that a stride address load predictor covers and that a pipeline prefecther [21] would have to replay.

Reuse in the Physical Register File

Alternatively, one can take advantage of *register sharing* techniques to perform load-to-load forwarding using the physical register file. This has the potential to speed up memory accesses since data is forwarded using the same physical register, and improve energy since only a single memory access needs to be made to validate the forwarded data. This strategy has the advantage of not only achieving the same zero latency load as a pipeline prefetcher, but it does so while also reducing the accesses to memory. However, for this strategy to be successful, highly-effective reuse prediction is required. Unfortunately existing reuse techniques based on *instruction distance prediction* [16] are unable to be accurate enough, and can exploit less than half of the load-to-load forwarding potential that exists in the loads present in the load queue at any given time (Figure 3.4).

While pipeline prefecth is an effective IPC improvement technique, it brings added pressure to the cache and register file since it requires accessing both structures on prefetch and validation. Register sharing can help mitigate the problem by sharing registers instead of prefetching and only accessing the cache for validation [16]. However, proposed approaches that use instruction-distance have limited ability to take advantage of the reuse present in the register file and address prediction schemes suffer from significant increases in L1 pressure by having to validate based on values.

3.4 Paper V: Physical Register File as a Cache

In Paper V we propose new register reuse technique that leverages the fact the predicted loads have their address available as early as in the fetch stage (from the address predictor), and as such, it is possible to have the physical register

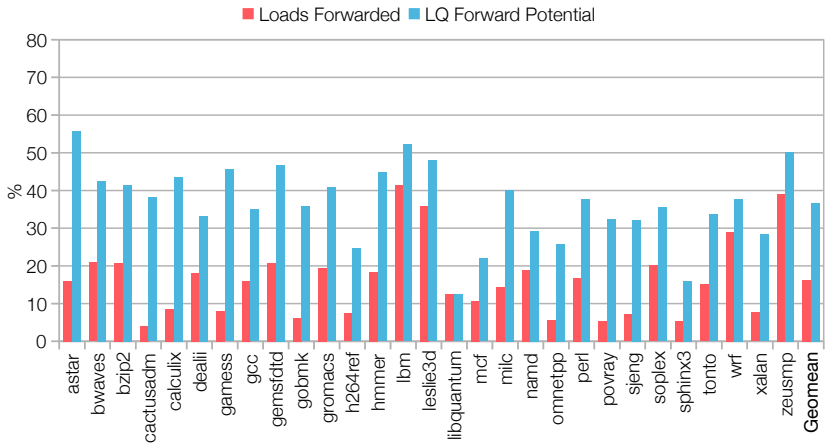

Figure 3.4. Ratio of loads that are forwarded using instruction distance to detect reuse, compared to total reuse potential available in the LQ.

file behave as a cache for this subset of load instructions. That is, we propose to detect data load-to-load reuse based on predicted addresses instead of instruction distance. This allows us to have better accuracy at detecting reuse (avoiding accessing the L1 for the data) and validate forwarding by comparing the predicted address to the generated one (avoiding accessing the L1 for validation).

To effectively use the PRF as a cache we need to address two issues: we need to be able to 1) *find* the prefetched data and 2) *forward* it efficiently without having to re-access the cache. This is achieved via a table that establishes a connection between the address name space and the register name space, and maps the reuse of prefetched data between load instructions.

Existing register reuse techniques suffer from limited coverage of the total potential for load-to-load reuse and require accessing the cache to validate correctness. The key to solve this problem, is how to have addresses early enough in the pipeline, i.e. before executing the loads themselves.

Finding and Forwarding Prefetched Data via PRF

To detect register reuse and perform load-to-load forwarding through the register file, we propose the introduction of a speculative register translation table (the *Address Tag, Register Tag* (AT-RT) table) to identify load-to-load reuse *based on addresses*, and associate those addresses to the register tags of the PRF that holds the data. This table is in essence a set of address tags and register tags for the data in the PRF that was loaded earlier from memory. The AT-RT is direct-mapped using the address as index and returns the PRF entry that corresponds to that address. Each of its entries holds a register tag, a reference counter for the number of instructions that are sharing it, a valid bit, and

an address tag. The table is indexed using the predicted address for each load, since this is provided by the load value predictor in the fetch stage.

Forwarding Validation

Our approach relies on two types of predictions: (1) the load address, and (2) whether the load-to-load forwarding violates memory ordering, i.e, if there are conflicting stores between the loads. The first prediction can be easily validated when the load's address is finally generated. Validation compares the actual address to the predicted address, which only requires an access to the load queue (where the predicted address is stored).

Validating memory order is more involved and needs to be separated into external and internal violations. External ones come in the form of memory coherence invalidation requests. Although complex to solve, this type of ordering violations is present in all multi-core systems, and as such, the cpu already has mechanisms to detect the violating loads [20], rollback the values of the shared registers [16] and replay the infringing instruction chains [17].

In order to deal with internal memory ordering problems, we need to detect memory ordering violations. Our approach is independent of the memory dependency predictor, but for this work we chose to implement SVW (*Store Vulnerability Window* [20]) as it minimises secondary accesses to the L1 when validating memory ordering. This mechanism relies on *store sequence numbers* (SseqN) to detect if a load is in the vulnerability window of a store and only loads that are in the vulnerability windows need to be replayed.

Exposing Spatial Locality via the PRF

A key function of caches is that they expose spatial locality as well as temporal locality. Fetching neighbouring data on an access, minimises later accesses to higher cache levels and thus improves energy efficiency of accessing memory. However, reuse techniques only take advantage of temporal locality, i.e. only forward data between loads that address the same exact memory location. Is not clear how one would expose spatial locality with an heuristic (such as instruction distance) so it is not surprising existing register reuse techniques do not attempt to do this. Since we use addresses to detect reuse however, how to expose spatial locality becomes a more manageable problem. Despite this, the problem is still not trivial, since it is not obvious what is the best data granularity ("cache line size") that should be fetched from L1 on each access and installed in the PRF, nor how can this data be mapped and forwarded without unnecessarily increasing register pressure.

We address this issue by fetching 128 bits on each access and modifying the AT-RT table so that each entry now tracks the group of registers that contain any of the 128-bit block. The 128-bit block was selected since this is the min-

imum amount of bits we have to read on each L1 cache access (see Section 2). Fetching more bits would only be necessary if there was a performance benefit of having the data closer to the cpu. This is not the case since the pipeline prefetcher (the primary reason to have a value predictor) will guarantee that the data is available as soon as it is required. This optimization aims to improve energy efficiency and reducing the number of redundant accesses to the same sub-array is the ideal approach.

Each table entry tracks multiple registers depending on the load size, e.g., 8, 4, 2 or 1 registers for load sizes of 1, 4, 8 or 16 (or more) bytes respectively. This way, loads can have their data forwarded from the PRF even if no load has addressed the same exact memory location before, increasing forwarding potential.

The downside of this approach is that we could be installing extra data that will not be used (extra writes and pressure in the PRF). We can circumvent this problem, since load addresses are available speculatively in the front-end of the cpu. One can use those address to detect coalescing opportunities and only install that exact data from the 128-bit sub-array read-out into the PRF. Given the small prefetch range (128 bits), the time between rename and load write-back is enough to find most loads (also address predicted) that map to the same 128-bit block. This, while not optimizing reuse potential, will remove superfluous data inhalation in the PRF.

Results

Our proposed AT-RT address based register reuse technique improves data forwarding through the register file by 2.2x on average compared to a state-of-art instruction-distance register sharing strategy. Moreover, it not only eliminates the extra load instructions associated with pipeline-prefetching, but it reduces the total number of load instructions L1 accesses by 36%, without compromising any performance benefit associated with prefetching loads to the pipeline.

4. Summary

Fast and energy efficient access to memory are fundamental for performance. Filter caches and way-predictors are common approaches to improve L1 energy efficiency and decrease access latency. Out-of-order cpu pipelines aim to hide memory latency instead, with the use of store-buffer that allows stores to retire even under store misses and pipeline prefetching, that can achieve a effective zero load-to-use latency to memory. Both of these techniques leave potential on the floor: L1 co-optimization with layout and replay, and pipelines co-optimization of speculative state storage and caching.

In the first part of this thesis we investigated how to improve first-level caching techniques (Paper I), and how to integrate them effectively with an out-of-order pipeline (Papers II and III). In the second part of the thesis we investigated how to re-purpose storage in the pipeline to use as a cache, namely the store-buffer (Paper IV) and the physical register file (Paper V). The summary of our contributions is:

Paper I: MRU-L0

Problem	Compared to a parallel access cache, Filter caches improve performance but sacrifice energy efficiency, while way-predictors improve energy efficiency but sacrifice performance. These strategies also large ignore energy and timing constraints imposed by SRAM.
Solution	MRU-L0: adds an extra buffer that stores the MRU way of each set, that is more a efficient structure since it is banked for the size of 1 way only.
Results	MRU-L0 cache is as performant as a filter cache (low latency access) and as energy efficient as a way-predictor (same prediction accuracy).

Papers II & III: Selective way-prediction

Problem	Way-prediction introduces variable L1 hit-latency. This causes instruction misscheduling and replays which significantly impact performance beyond previously reported results.
Solution	Measure way-prediction accuracy and delay scheduling of instructions dependent on untrustworthy predictions.
Results	Reduced performance penalty associated with instruction replays, and improved way-prediction energy efficiency.

Papers IV: Store-Buffer-Cache

Problem	SQ/SB holds copies of dirty data and needs to be probed on every access just as a cache. However, the over-aggressive write-back policy limits its effectiveness as a cache (i.e. reducing latency and energy of memory accesses).
Solution	By extending the SQ/SB with an extra logical buffer (store-buffer-cache), we can increase the buffer hit-ratio of all three queues (S/QBC) without compromising its original function as a way to hide store-misses and prevent pipeline stalls. To do so, we need to integrate the S/QBC with coherence and predict when we will hit to avoid the L1/TLB probes.
Results	Improved SQ/SB(/SBC) hit-ratio, reduced accesses to L1, improved performance and energy efficiency of S/QBC+L1.

Papers V: Address based, PRF register sharing

Problem	Pipeline prefetchers improve performance but increase pressure in L1 cache by replaying all prefetched loads. PRF reuse techniques can help mitigate this problem but they struggle to detect reuse with instruction distance.
Solution	Use the value predictor, required by the pipeline prefetcher to predict addresses, and use those addresses to detect reuse in the PRF instead of distances. Since we only deal with load instructions, we can also validate prefetched data without re-executing loads.
Results	Improved reuse of data in PRF and reduced bandwidth pressure in the cache due to prefetching into the pipeline.

5. Svensk Sammanfattning

Snabb åtkomst av data är nödvändig för prestanda. För att uppnå detta, måste datorprocessorer göra bruk av snabba cacheminnen på förstanivå för att kunna göra minnesåtkomst snabbare, detta ofta i samband med oordnad exekvering för att förmildra minnespassiv tid vid minnesåtkomster. Samtidigt som dessa tekniker effektivt uppnår prestanda, har de å andra sidan en synnerligen hög energikostnad, vilket har lett till flertalet framtagna tekniker som låter formgivare att byta prestanda mot energieffektivitet.

Teknikerna ”way prediction” och filter-cacher är två av de vanligaste strategierna för att förbättra effektiviteten hos cacheminnen på förstanivå, samtidigt som man minimerar minnespassiv tid. Bägge av dem avgör mellan olika kompromisser: ”way prediction” byter viss minnespassiv tid mot högre energieffektivitet, medan filter-cacher byter viss energieffektivitet mot lägre minnespassiv tid. Dessa tekniker är dock inte ömsesidigt uteslutande. Genom att låna vissa element från bägge och genom att ta med begränsningar inom SRAM-minnesutformning i beräkningarna, presenterar vi en ny MRU-L0 cache som mildrar flera av deras nackdelar och samtidigt drar nytta av fördelarna. Dessutom: cacheminnen på förstanivå är nära sammankopplade med CPU-pipelinen, men relaterat arbete har mestadels ignorerat de effekter som dessa har för schemaläggning av instruktioner. Vi visar att den varierande minnespassiva tidsåtgången vid cache-träffar orsakar omkörningar av de instruktionskedjor som beror på läsinstruktioner, vilka i sin tur påverkar prestanda och effektivitet negativt. Vi studerar denna effekt och föreslår en instruktionsschemaläggare för varierande minnespassiv tid vid cache-träffar som identifierar eventuellt felaktiga schemaläggningar; som minskar omkörningar av instruktioner; som minskar negativ inverkan på prestanda; och som vidare förhöjer cachernas energieffektivitet.

Moderna CPU-pipelinor gör även bruk av sofistikerade strategier för exekvering för att för att förmildra minnespassiv tid vid minnesåtkomster och förbättra prestanda. Medan deras huvudsakliga användningsområde är till för prestanda och korrekthet, så behöver de mellanliggande lagringsutrymme som också kan användas som en cache. I detta arbete uppvisar vi hur lagringsbufferten, när den paras ihop med minnesberoendeprediktorn, kan användas till att effektivt cachea skrivdata och hur den fysiska registerfilen, ihopparad med en värdesprediktor, kan användas till att effektivt cachea läsdata. Dessa strategier förbättrar inte endast prestanda och energiåtgång, men gör detta även utan ökat krav på lagringsutrymme och med minimal extra komplexitet, eftersom de återanvänder redan existerande CPU-strukturer för att detektera instruktionsåteranvändning, överträdelser av minnesordning och felspekulationer.

6. Acknowledgments

This thesis would not be possible without the help of many people. As a gesture of gratitude, I would like to briefly write some kind words about them. Please understand that my better skills do not include poetic writing. But while these words might not be beautiful, I promise they are honest.

First, I want to thank my two advisors, David Black-Schaffer and Stefanos Kaxiras. Thank you for your guidance and continuous support, during the good and bad times alike. You were both great teachers and I learned a lot from you – it truly was a growing and humbling experience to have worked with you both.

I also want to thank all the people from the UART group. You definitely made the last few years at polacksbacken a fantastic experience. I would like to single out my current and past office-mates, Gustaf Borgström, Per Ekemark, Hassan Muhammad, Mihail Popov, Nikos Nikoleris, Moncef Mechri and Germán Ceballos. It was a pleasure to come to work every day and that was in large part because of you all.

Quero agradecer também familiares e amigos de Portugal. Um especial obrigado para os meus pais, que como sempre, me apoiaram e encorajaram em mais um enorme desafio da minha vida – um humilde obrigado para amortizar mais um pouco uma dívida que nunca vou conseguir saldar.

Finally, I want to thank my partner Ikki. It was a tough journey but you were always patient, loving, and supporting. You are wonderful and I consider myself very privileged to have you in my life.

References

[1] Nikolaos Bellas, Ibrahim Hajj, and Constantine Polychronopoulos. Using dynamic cache management techniques to reduce energy in a high-performance processor. In *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, pages 64–69. IEEE, 1999.

[2] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, et al. Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor. *Digital Technical Journal*, 7(1):0, 1995.

[3] B Fahs, T Rafacz, SJ Patel, and SS Lumetta. Continuous optimization. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 86–97. IEEE, 2005.

[4] Roberto Giorgi and Paolo Bennati. Reducing leakage in power-saving capable caches for embedded systems by using a filter cache. In *Proceedings of the 2007 workshop on MEmory performance: DEaling with Applications, systems and architecture*, pages 97–104. ACM, 2007.

[5] Stephan Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 216–225. IEEE, 1998.

[6] Ilhyun Kim and Mikko H Lipasti. Understanding scheduling replay schemes. In *Software, IEE Proceedings-*, pages 198–209. IEEE, 2004.

[7] Johnson Kin, Munish Gupta, and William H Mangione-Smith. The filter cache: an energy efficient memory structure. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 184–193. IEEE Computer Society, 1997.

[8] MH LIPASTI. Value locality and load value prediction. In *Proc. of 7th Int'l Conf. on Architectural Support for Programming Languages and Operatings Systems, 1996*, 1996.

[9] Mikko H Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 226–237. IEEE Computer Society, 1996.

[10] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. Coatcheck: Verifying memory ordering at the hardware-os interface. *ACM SIGOPS Operating Systems Review*, 50(2):233–247, 2016.

[11] Lois Orosa, Rodolfo Azevedo, and Onur Mutlu. Avpp: Address-first value-next predictor with value prefetching for improving the efficiency of load value prediction. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):49, 2018.

[12] Subbarao Palacharla, Norman P Jouppi, and James E Smith. *Complexity-effective superscalar processors*, volume 25. ACM, 1997.

[13] Arthur Perais, Fernando A Endo, and André Seznec. Register sharing for equality prediction. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 4. IEEE Press, 2016.

[14] Arthur Perais and André Seznec. Eole: Paving the way for an effective implementation of value prediction. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 481–492. IEEE, 2014.

[15] Arthur Perais and André Seznec. Bebop: A cost effective predictor infrastructure for superscalar value prediction. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–25. IEEE, 2015.

[16] Arthur Perais and André Seznec. Cost effective physical register sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 694–706. IEEE, 2016.

[17] Arthur Perais, André Seznec, Pierre Michaud, Andreas Sembrant, and Erik Hagersten. Cost-effective speculative scheduling in high performance processors. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 247–259. IEEE, 2015.

[18] Vlad Petric, Anne Bracy, and Amir Roth. Three extensions to register integration. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.*, pages 37–47. IEEE, 2002.

[19] Vlad Petric, Tingting Sha, and Amir Roth. Reno: a rename-based instruction optimizer. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 98–109. IEEE, 2005.

[20] A Roth. Store vulnerability window (svw): re-execution filtering for enhanced load optimization. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 458–468. IEEE, 2005.

[21] Rami Sheikh, Harold W Cain, and Raguram Damodaran. Load value prediction via path-based address prediction: avoiding mispredictions due to conflicting stores. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–435. ACM, 2017.

[22] James E Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[23] Kimming So and Rudolph N. Rechtschaffen. Cache operations by mru change. *IEEE Trans. Computers*, 37(6):700–709, 1988.

[24] Jared Stark, Mary D Brown, and Yale N Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 57–66. ACM, 2000.

[25] Nathan Tuck and Dean M Tullsen. Multithreaded value prediction. In *11th International Symposium on High-Performance Computer Architecture*, pages 5–15. IEEE, 2005.

[26] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 281–290. IEEE Computer Society, 1997.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1821

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-383811

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2019